AFRL-RY-WP-TR-2015-0017

PREVENTING EXPLOITS AGAINST SOFTWARE OF UNCERTAIN PROVENANCE (PEASOUP)

David Melski

GrammaTech, Inc.

MAY 2015
Final Report

Approved for public release; distribution unlimited.

See additional restrictions described on inside pages

STINFO COPY

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

AFRL-RY-WP-TR-2015-0017 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//
KENNETH LITTLEJOHN, Program Manager
Avionics Vulnerability Mitigation Branch

//Signature//
DAVID G. HAGSTROM, Chief
Avionics Vulnerability Mitigation Branch

//Signature//
TODD A. KASTLE, Chief
Spectrum Warfare Division

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE ***(DD-MM-YY)*** May 2015	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 26 August 2010 – 30 November 2013

4. TITLE AND SUBTITLE
PREVENTING EXPLOITS AGAINST SOFTWARE OF UNCERTAIN PROVENANCE (PEASOUP)

5a. CONTRACT NUMBER
FA8650-10-C-7025

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
David Melski

5d. PROJECT NUMBER
OthAF

5e. TASK NUMBER
RY

5f. WORK UNIT NUMBER
Y0LG

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
GrammaTech, Inc.
531 Esty Street
Ithaca, NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Intelligence Advanced Research Project Activity (IARPA)
Office of the Director of National Intelligence (ODNI)
Washington, DC 20511

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYWA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2015-0017

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
©2014 GrammaTech, Inc. The U.S. Government is joint author of the work and has the right to use, modify, reproduce, release, perform, display or disclose the work. PAO Case Number 88ABW-2015-2491, Clearance Date 20 May 2015. Report contains color.

14. ABSTRACT

We describe the results of the research and development of PEASOUP (Preventing Exploits Against Software of Uncertain Provenance), a technology that enables the safe execution of software executables. PEASOUP provides the following capabilities: prevents exploits of number-handling weaknesses and memory-safety weaknesses; prevents OS command injections, OS command argument injections, SQL injections, and denial-of-service exploits based on inducing a null-pointer dereference; and prevents any exploit based on arc-injection or code-injection, regardless of the type of vulnerability targeted for attack. PEASOUP also offers experimental protection against exploit of many concurrency and resource drain vulnerabilities, including: file-system Time-Of-Check-to-Time-Of-Use (TOCTOU) vulnerabilities, use of non-reentrant functions in signal handlers, deadlock vulnerabilities, atomicity violations, memory leaks, and file-handle leaks.

The PEASOUP effort advanced the state-of-the-art in automatic machine-code analysis, diversification, confinement, and remediation. Specific results include: a technique for preventing command injection attacks inspired by DNA Shotgun Sequencing, a technique that often allows server programs to remain operational after an attempted null-pointer dereference, improved integer-error analyses, improved protections for heap- and stack-allocated memory, novel techniques for analyzing file input types, and a superior design for a software dynamic translator that prevents attacks against the translator.

15. SUBJECT TERMS
software security, automatic binary repair, automatic binary hardening, exploit prevention

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	258	Kenneth Littlejohn
Unclassified	Unclassified	Unclassified			**19b. TELEPHONE NUMBER** *(Include Area Code)* N/A

Abstract

We describe the results of the research and development of PEASOUP (Preventing Exploits Against Software of Uncertain Provenance), a technology that enables the safe execution of software executables. PEASOUP provides the following capabilities: prevents exploits of number-handling weaknesses and memory-safety weaknesses; prevents OS command injections, OS command argument injections, SQL injections, and denial-of-service exploits based on inducing a null-pointer dereference; and prevents any exploit based on arc-injection or code-injection, regardless of the type of vulnerability targeted for attack. PEASOUP also offers experimental protection against exploit of many concurrency and resource drain vulnerabilities, including: file-system Time-Of-Check-to-Time-Of-Use (TOCTOU) vulnerabilities, use of non-reentrant functions in signal handlers, deadlock vulnerabilities, atomicity violations, memory leaks, and file-handle leaks.

The PEASOUP effort advanced the state-of-the-art in automatic machine-code analysis, diversification, confinement, and remediation. Specific results include: a technique for preventing command injection attacks that was inspired by DNA Shotgun Sequencing, a technique that often allows server programs to remain operational even after an attempted null-pointer dereference, improved integer-error analyses and protections that apply to large programs with a low false positive rate, improved protections for heap- and stack-allocated memory, novel techniques for analyzing file input types, and a superior design for a software dynamic translator that prevents attacks against the translator.

TABLE OF CONTENTS

SECTION **PAGE**

List of Figures

List of Tables

1.0 Summary

We describe the results of the development of PEASOUP (Preventing Exploits Against Software of Uncertain Provenance), a technology that enables the safe execution of software executables of uncertain provenance. PEASOUP prevents exploits of number-handling weaknesses and memory-safety weaknesses in Software of Uncertain Provenance (SOUP). In addition, PEASOUP prevents any exploit based on arc-injection or code-injection, regardless of the type of vulnerability targeted for attack.

PEASOUP advanced the state-of-the-art in automatic program analysis, diversification, confinement, and remediation. In the remainder of this section, we describe the results of research in each areas.

Analysis. The PEASOUP analyzer uses a novel combination of precise run-time analyses [105] with recent techniques for generating high-coverage test suites [48, 97]. The analyzer has the following components:

1. *Test-case Generation:* concolic execution is used to analyze the subject program (in binary form) and generate a test suite.
2. *Input Classification:* various run-time monitoring tools are used to classify the program inputs as ‘bad’ inputs that cause the program to crash and ‘good’ inputs that appear to function normally. During Phase 1, we demonstrated that even simple run-time monitors are sometimes sufficient for this task.
3. *Intermediate Representation Recovery:* PEASOUP applies a combination of static analysis and dynamic analysis to recover an *intermediate representation* (IR) of the subject program. The IR holds facts about the syntactic and semantic structure of the subject program, such as the location of instructions, functions, and data-object layout. The design of the IR Recovery module is novel in that (i) it can leverage the classification of inputs and (ii) it can provide feedback to the test-case generation and input classification modules in order to improve overall analysis results.
4. *Variant Generation:* PEASOUP use the recovered IR to generate many different variants of the subject program. The goal of variant generation is to remove vulnerabilities and introduce diversity. PEASOUP uses a novel design for representing program variants. Each variant is stored as a set of rules that rewrite the original program into the variant. The rules are applied dynamically using *software dynamic translation*. This architecture allows PEASOUP to have the advantages of both *static* and *dynamic* rewriting approaches. Specifically, PEASOUP computes the desired variant offline, leveraging the results of exhaustive analyses, but apply the changes online. The application of program transformations on-demand means that PEASOUP automatically tolerates certain types of errors in the recovered IR, such as misidentifying data as code.
5. *Variant Validation:* Automatic program analysis and transformation is frequently impossible because any flaw in IR recovery can lead to incorrect program transformation and undesired changes in program behavior. PEASOUP solves this problem in two ways: (i) by leveraging the inherent robustness of just-in-time application of transformations, as described above and (ii) validating each of the generated program variants using the classified test inputs. In order for PEASOUP to consider a program variant acceptable, it requires that test-case generation succeeded in generating a high-quality test suite and that the variant has the same observable behavior as the original program on the ‘good’

inputs. Variants are also considered more robust if their behavior differs from the original program on ‘bad’ inputs.

We demonstrated that the utility of this approach to binary analysis. For example, we demonstrated that we could safely transformation the layout of stack-allocated data despite flaws in the IR recovery algorithm for identifying the boundaries of objects on the stack. The behavioral-equivalence testing discarded those variants that relied on misidentified stack boundaries.

We developed a novel analysis for determining the layout of data in a binary. This analysis has been transitioned to GrammaTech’s commercial tool, CodeSonar.

Diversification. We developed a novel diversification technique called *Instruction-Location Randomization* (ILR) that works by relocating 99.7% of the instructions in a program. Many approaches to diversification suffer from low-entropy or complicated estimates of the measure of introduced entropy based on assumptions that may not hold for some executables. ILR advances the state of the art in program diversification because it does not suffer from these shortcomings: it is simple to show that 99.7% of instructions can be relocated to any one of 2^{31} addresses (on a 32-bit machine). This represents 3.5 *orders of magnitude* improvement over some of the most common, successful diversification techniques. ILR makes any type of arc-injection attack infeasible, including attacks based on Return-Oriented Programming (ROP). Furthermore, ILR still has the desirable properties of existing diversification techniques: it has low overhead and is easy to deploy.

In addition to ILR, we demonstrated that PEASOUP can safely perform *Stack-Layout Transformation* (SLX) on software binaries. Previous approaches to randomizing the layout of the stack required access to a program’s source code. We believe that part of the promise of PEASOUP is that it feasible to take transformations that currently require source code and apply directly to binaries.

Confinement. PEASOUP incorporates many advances in confinement techniques, including:

- A novel dual-process architecture for Software Dynamic Translators. We demonstrated the feasibility of exploiting SDTs. Our dual-process architecture prevents these exploits.
- Novel memory-error protection technologies. To our knowledge, PEASOUP is the first and only existing technique that can successfully mitigate the infamous Heartbleed vulnerability (in some circmustances).
- A novel technique for defending against command-injection attacks. Our technique is loosely inspired by a technique called *DNA Shotgun Sequencing* that is used by biologists to sequence DNA. The technique opens a new area of research we call *positive taint inference.* We demonstrated the effectiveness of our technique for preventing OS command injections.

Remediation. Finally, PEASOUP demonstrated advances in the state-of-the-art for automatic program remediation. For example, the padding introduced by PEASOUP’s diversification techniques allowed real-world applications to continue correct execution when they were provided malicious inputs.

The remainder of this report provides details on each of these results.

2.0 Introduction

We describe the results of the research and development of PEASOUP, a technology that enables the safe execution of software executables. PEASOUP (Preventing Exploits Against Software of Uncertain Provenance) provides the following capabilities: prevents exploits number-handling weaknesses and memory-safety weaknesses; prevents OS command injections, OS command argument injections, SQL injections, and denial-of-service exploits based on inducing a null-pointer dereference; and prevents any exploit based on arc-injection or code-injection, regardless of the type of vulnerability targeted for attack. PEASOUP also offers experimental protection against exploit of many concurrency and resource drain vulnerabilities, including: file-system Time-Of-Check-to-Time-Of-Use (TOCTOU) vulnerabilities, use of non-reentrant functions in signal handlers, deadlock vulnerabilities, atomicity violations, memory leaks, and file-handle leaks.

The PEASOUP project was broken into three phases, with funding for subsequent phases dependent on performance in earlier phases. Each phase focused on providing protection for two new classes of vulnerabilities, as well as improving the degree of protection for the vulnerability classes addressed in earlier phases. Phase 1 of the PEASOUP project provided protection against exploits of improper handling of fixed-width computations and memory safety errors. The protections implemented in Phase 1 also provide partial protection against use of tainted data, input validation errors, and command injection exploits. Phase 2 provided protection against OS command injection, OS-command argument injection, SQL injection, and denial-of-service via a null-pointer dereference. Phase 3 provided partial protection against concurrency vulnerabilities and resource drains.

PEASOUP combined new approaches to analysis, confinement and diversity in order to break through the limitations of existing practice as follows:

- *Analysis:* leading (source-code) analysis tools find only a small fraction of CWE test cases, even when used in combination. PEASOUP developed advances in IR recovery that enabled more effective defensive transformations.
- *Confinement:* current confinement techniques only protect a single run of the program and operate in fail-stop or fail-oblivious modes, leaving the program vulnerable to Denial-of-Service (DOS) attacks or open to unpredictable behavior. PEASOUP advances recent techniques for automatically patching vulnerabilities by generating patches for vulnerabilities prior to deployment. Automatic testing of patches ensures that unpredictable behavior is avoided.
- *Diversification:* current diversification techniques do not provide guarantees: often program variants have the same vulnerabilities, albeit in different places. While PEASOUP cannot change this fundamental limitation of diversification, it employs high entropy, machine-code diversification techniques to make certain attacks infeasible, e.g., by mutating the passing of function parameters to prevent arc attacks.

PEASOUP provides defense-in-depth by combining a collection of defenses, including:

- *Automatic fault removal.* PEASOUP uses an (offline) analysis phase to discover vulnerabilities, generate one or more strategies to compensate for the underlying fault, and automatically test each candidate strategy. An online *execution manager* applies remediation strategies.

- *Instruction-level confinement.* The execution manager uses *Software Dynamic Translation* to efficiently monitor the execution at the granularity of individual instructions.
- *High entropy diversification.* The analysis phase generates many variants of the program, and the execution manager randomly selects a variant at run time. The high degree of differences in the variants makes it impractical to construct an exploit that works on more than one variant.
- *Hardware-enforced confinement.* The hardware's page protection mechanism is used to prevent tampering of the execution monitor. This provides an outer shell around the executable to help confine damage in the unlikely event that the other defenses fail.

2.1 Innovation Goals for the Proposed Research

The PEASOUP project sought to advance the state-of-the-art in many different areas:

- *Improved test-case generation for executables*. Recent automatic test-case generation techniques have demonstrated the capability to 1) discover faults in large programs [97], 2) achieve high coverage and discover faults in mature, well-tested programs [48], and 3) achieve specific coverage metrics in analysis of machine code [125]. PEASOUP sought to advance techniques for generating high coverage test suites based on analysis of machine code [125].

- *Improved IR recovery.* Optimizing compilers build an *Intermediate Representation* (IR) of the program to guide optimization. The final step of compilation generates machine code from the IR. *IR recovery* reverses this process by constructing an IR based on analysis of the machine code. PEASOUP advancep the state-of-the-art in IR recovery, particularly for the key problem of data delineation.
- *Improved techniques for automated fault removal.* Recent results have demonstrated the feasibility of automatically patching vulnerabilities in running software [160]. PEASOUP improved on [160] in the following ways:
  - PEASOUP discovers faults and generates remediation strategies *prior* to deployment. This approach prevents zero-day attacks and reduces the possibility of denial-of-service.
  - PEASOUP automatically tests each remediation strategy without user feedback. This reduces the potential for unpredictable behavior.
- *Secure Software Dynamic Translation (SDT).* SDT is an efficient technique for monitoring an executable at the instruction level and modifying its runtime behavior. Prior to PEASOUP, all SDT implementations that we are aware of live in the address space of the translated program. We demonstrated that this architecture is subject to attack. Furthermore, under PEASOUP we developed a novel architecture that runs the translator in a separate process from the translated process, allowing the translator to be protected. We demonstrated that this approach provides additional security with minimal additional runtime overhead.

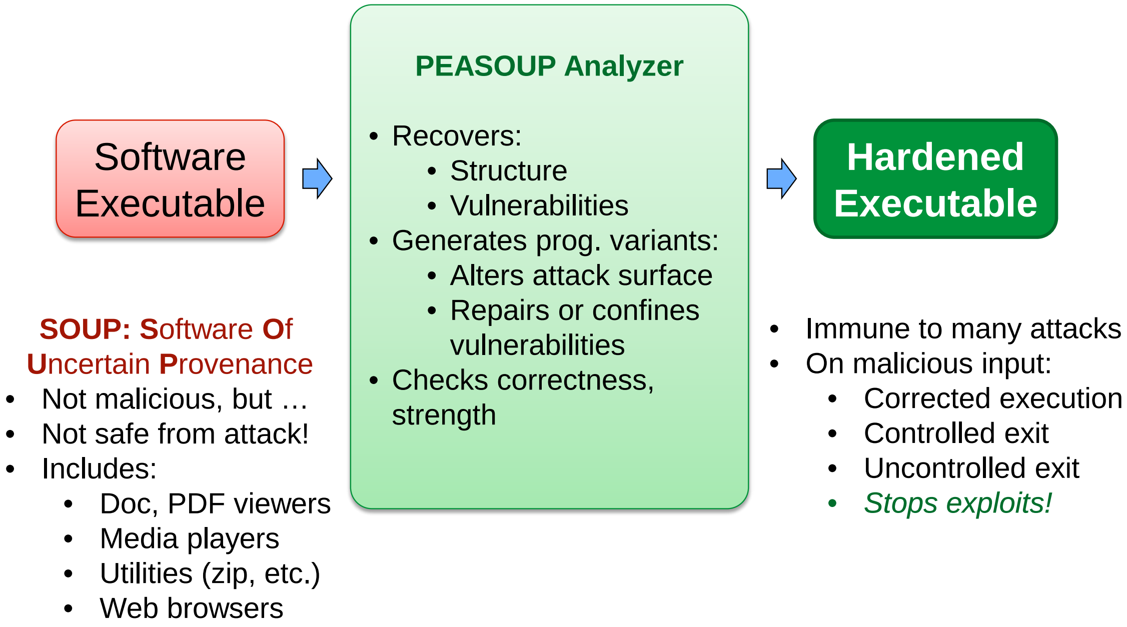


Figure 1. Use of PEASOUP to Create Hardened Executables

- *Dynamic application of high-entropy diversification techniques to machine code.* Previously, such techniques required source code, but our improvements to IR recovery and our ability to test variants enableed these techniques for executables and enable novel techniques.

2.2 Summary of the Products and Transferable Technology

Conceptually, the PEASOUP analyzer takes as input executables from *Software Of Uncertain Provenance* (SOUP) and produces hardened versions of those executables, called *peasoupified* executables (see Figure 1). Using PEASOUP consists of two steps: first, the analyzer must be run to create the peasoupified versions of the executables; then the peasoupified executables must be installed in place of the original executables. The peasoupified executables can be run on different machines than the analyzers. The analyzer requires a few third-party tools to run, while the peasoupified executables are stand-alone.

The current platform requirements for PEASOUP are as follows:

Usage Requirements		
	PEASOUP Analyzer	**Protected Executable**
Platform Dependences	Ubuntu, x86-32 and -64; No runtime code generation in SOUP; Prelim. Support for CentOS, REHL	Ubuntu, x86-32 and -64; No runtime code generation in SOUP; Prelim. Support for CentOS, REHL
Software Dependences	Commercial: IDA Pro 6.5 Requires some open source, e.g. Postrgres, g++, nasm.	None: protected exe is self-contained; [Optional] relocate protected executable with slight modifications to run script.
Installation	Manual: IDA Pro 6.5 Scripted: Download and configuration of all other software	None: protected exe is self-contained; [Optional] relocate protected executable with slight modifications to run script.
Running	Invoke analyze script	Invoke protected executable

As described below, PEASOUP stores its IR in a relational database. Currently, when PEASOUP creates a peasoupified binary, it extracts all of the necessary information from the database and stores it in a directory alongside of the peasoupified executable. This simplifies distribution of the peasoupified binary.

We envision many possible scenarios for the deployment of PEASOUP. In the first scenario, an IT department would use PEASOUP to provide a curated list of software for end users. Many IT shops already configure their private cloud infrastructure to update applications on their Linux boxes from an update server managed by the IT department. For each request for a new application or an update to a deployed application, they vet the application or update and configure their update server to deploy the application according to the enterprises security requirements.

Under this scenario, when there is a new (version of) a package to deploy, the IT department would download the package, extract the executables, peasoupify them, and build a new package file with the peasoupified executables. They would then use the existing package distribution mechanism to deploy the peasoupified executables. Under this scenario, only the administrator needs to be able to peasoupify executables.

Another scenario allows end-users to peasoupify their own applications: they would run the default package management system and allow it to install executables as necessary. Prior to running an application, the user would replace any untrusted executables with peasoupified versions.

PEASOUP was developed for the purpose of protecting executables, but it has other possible applications. An *error amplification* technique increases the likelihood of detecting errors. A software flaw can go unnoticed if all of a developer's test inputs that exercise the flaw happen to pass, for example, because the flaw happens to not cause incorrect outputs for the test inputs. If a developer peasoupifies their application, then there is a chance that PEASOUP will flag an error, even if it does not affect the program's output. In this way, PEASOUP amplifies the error signal, allowing the developer to find and fix the error prior to release.

PEASOUP can also be used to classify program inputs as potentially malicious. Given a collection of files of a certain type (e.g., PDF) some of which may be malicious and some of which may be benign, a user can create a peasoupified version of a viewer for the files (e.g., a peasoupified PDF viewer) and run all of the files through the peasoupified viewer. Malicious inputs are likely to cause different behavior when run under PEASOUP than when run on the unprotected version of the program. Furthermore, PEASOUP may directly detect malicious actions induced by malicious inputs. This use of PEASOUP for *malicious input classification* could also be performed at the perimeter of a network, e.g., so that potentially malicious files can be identified and quarantined instead of delivered to end users.

One barrier to the eventual deployment of PEASOUP is the potential for undesired *Altered Functionality* (AF) where the behavior of a peasoupified incorrectly differs from the behavior of the original. AF is particularly problematic for PEASOUP when used directly for program protection. With error amplification and malicious input classification, the tolerance for altered functionality is somewhat higher, possibly simplifying these uses of the technology.

2.3 Use of Third-Party COTS Products

The PEASOUP analyzer depends on two mature and well-respected commercial software packages:

- **The IDA Pro Disassembler and Debugger**, a Commercial Off-The-Shelf (COTS) product of Hex-Rays SA. IDA Pro will not be delivered. It is available for purchase from Hex-Rays. At the Government's request, GrammaTech will loan up to three licenses for IDA Pro executables to sites needing them for the purpose of evaluating the delivered prototypes.
- **Yices: An SMT Solver**, a COTS product of SRI International. Yices will not be delivered with the research results prototype. It is available for free download from SRI International. At the Government's request, GrammaTech will loan up to three licenses for Yices executables to sites needing them for the purpose of evaluating the delivered prototypes, albeit this will just be a convenience given that Yices is available for download from SRI.

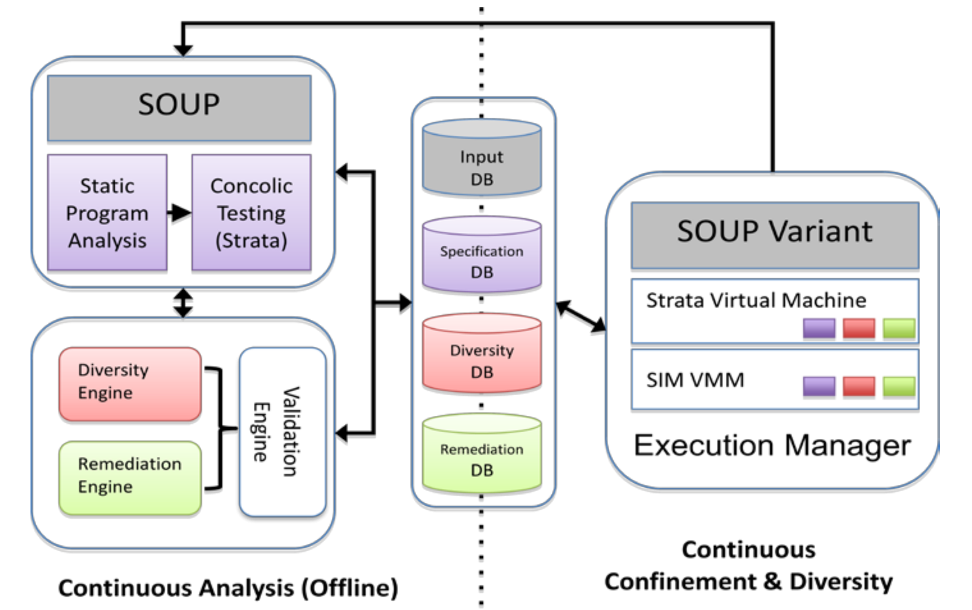


Figure 2. The PEASOUP Architecture

Widespread deployment of the research results prototype will require licenses for both IDA Pro and Yices, but it is inappropriate to budget for these licenses prior to deployment.

In addition, PEASOUP makes use of several open-source packages, both for building and deploying PEASOUP. These include: postgres, beaengine, binutils 2.19, diablo_toolchain, elfio, boost, config.guess, libdwarf, libelf, and scons. As these tools are open source, their use should not affect deployment or usage of PEASOUP.

2.4 Overview of the Technical Approach and Plan

As stated above, the goal of PEASOUP was the protection of specific classes of vulnerabilities that may be present in Software Of Uncertain Provenance (SOUP). To meet that goal, we present the PEASOUP architecture (see Figure 2). In the remainder of this section, we describe the components of PEASOUP at the highest level of components: the *analyzer*, the *execution manager*, and the *Intermediate Representation Database* (IRDB).

2.4.1 The (Offline) Analyzer

The analyzer must be run before the SOUP is sanctioned for use. The analyzer may be re-invoked if confinement detects an attack against a previously unknown vulnerability. The analyzer has many responsibilities, shared between four sub-modules as described in the following subsections.

IR Recovery and Vulnerability Detection Module. IR recovery (analysis of the structure of the SOUP) and vulnerability detection are at the heart of the analyzer. The recovered IR must be precise enough to support low-overhead confinement and high-entropy diversification. Most vulnerabilities in SOUP should be discovered by the analyzer offline, before deployment. As stated above PEASOUP achieves these goals through a combination of advanced techniques for generating high-coverage test suites and precise (and potentially expensive) run-time analyses.

Remediation Strategist. Techniques that are failure-stop or failure-oblivious still allow denial of service attacks. To avoid DOS attacks, and entropy-exhausting derandomizing attacks, PEASOUP employs various remediation strategies for the vulnerabilities that it detects. The remediation strategist is responsible for analyzing each detected vulnerability and developing one or more strategies for eliminating or at least ameliorating the vulnerability. In terms of the fault taxonomy, the strategist may opt for repair of the fault or for a recovery plan.

Variant Generator. The variant generator is responsible for generating many different variants of the SOUP for use in diversification. It relies heavily on the recovered IR to generate a set of variants with high entropy. The variant generator uses many different diversification techniques.

Validation Module. Both automatic remediation and variant selection come with the risk of modifying the program behavior in unintended ways. The generation of remediation strategies and program variants will be designed to limit this risk, but it cannot be eliminated. The validation module ensures that unintended changes do not occur. It consists of two components:

1. *Behavior Equivalence Detector (BED).* The BED module uses automatic test-case generation to test if two program variants are equivalent *on 'normal' executions* (e.g., that do not exercise a vulnerability). We expect remediation to change program behavior on an input that exploits a vulnerability. Similarly, the purpose of diversification is to make exploits have unpredictable behavior (*e.g.*, crash instead of taking the control).

2. *Test-Suite Evaluation Technology (TSET).* The purpose of the TSET component is to ensure that BED is working correctly. TSET works by generating mutations of a program that are expected to alter the program's behavior, and checking that BED detects the mutants.

2.4.2 The Execution Manager

PEASOUP uses the execution manager to run SOUP. Its responsibilities include:

- Monitoring the execution of the SOUP for exploits of vulnerabilities that were not discovered offline, or were found but could not be remedied.

- Modifying the execution stream of the SOUP to implement proven remediation strategies. This may include insertion or modification of instructions in the SOUP and/or speculative execution and roll-back of dangerous sections of code. (Remediation strategies are available prior to execution. They are applied at runtime for convenience: the execution manager is based on technology (software dynamic translation) that makes it both simple and efficient to modify the dynamic execution.)

- Implementing run-time diversity by selecting from the pool of pre-generated variants.

- Feeding information about any detected attacks back to the analyzer for potential refinement of the IR and remediation strategies.

The execution manager is implemented using a novel approach to Software Dynamic Translation (SDT). SDT is a technique for modifying the execution stream of a running process. A software dynamic translator does not execute the instructions of the program directly. Instead, it copies each instruction, on demand, and executes the (possibly modified) copy. SDT is extremely well suited to the needs of PEASOUP. The translator touches every instruction during the execution, giving it the opportunity to insert very fine-grained monitoring code and to implement arbitrary patches for remediation of vulnerabilities. PEASOUP is built on the Strata SDT developed at UVA for software dynamic translation in PEASOUP [181].

A drawback of SDT is its vulnerability to attack. Strata could be vulnerable due to a false negative in protection of the vulnerabilities that PEASOUP handles, or due to a vulnerability that PEASOUP does not handle. To protect Strata, we developed a novel dual-process architecture for software dynamic translation: the translator runs in a separate process from the translated SOUP process. Both processes share most of their memory, but with different page permissions (that are enforced by the operating system and the hardware). The SOUP has no direct access to the translator code or data and only has read/execute access to the translated code cache. When new code must be translated, a secure *Remote Procedure Call* (RPC) is issued from the SOUP to the translator.

2.4.3 The Intermediate Representation Database

The final component of PEASOUP is the Intermediate Representation Database (IRDB). The analyzer and the execution manager communicate through the IRDB. The IRDB records facts about the program, including the structure of the SOUP (types, layout, and other IR), locations of known vulnerabilities, remediation strategies, program variants, and a test suite with high coverage.

2.5 Objectives, Scientific Relevance, Technical Approach and Expected Significance

This section expands on the overview given in Section 2.3, as follows: Section 2.5.1 describes the technology that is leveraged in building PEASOUP. Section 2.5.2 describes how that technology is assembled to create the PEASOUP components.

2.5.1 Technology Leveraged in PEASOUP

PEASOUP was designed to leverage a strong, existing technology base. This section gives an overview of the technology that we intended to use in PEASOUP. We have noted the cases where the actual implementation deviated from the planned use of existing technology; this happened in response to research results during performance of the project.

2.5.1.1 Automatic Generation of High-Coverage Test Suites

For smaller programs, PEASOUP's offline analyzer leverages a capability to automatically generate test suites with high program coverage. The analyzer uses this capability to drive the dynamic analyses for IR recovery, vulnerability detection, and equivalence testing. To implement the capability, PEASOUP builds on several advanced techniques developed prior to the beginning of the contract. An important technique for generating comprehensive test suites is *concolic execution* [35, 48, 58, 96, 185], which seeks to achieve high program coverage by generating inputs that force a program to follow each possible execution path. While achieving complete path coverage is impossible (at least, impractical) for any interesting program, several concolic-execution techniques achieve high coverage in reasonable time [35, 48, 58, 97, 136, 137]. The fact that these techniques have been successful in finding bugs in very large executables [97], as well as in very mature, well-tested executables [48], supports their use in PEASOUP.

Unfortunately, we believe there are limitations to the use of automated test-case generation for analyzing larger programs. Limitations in the test-case generation technology mean that often it fails to generate a high-coverage test suite, especially for larger programs. Even supposing that this limitation could be overcome, a large coverage test suite may include an impractical number of tests. Simply running a test suite that is large enough to generate good coverage may be impractical. This does not mean that symbolic execution cannot play a role in a tool like PEASOUP. However, future efforts should focus on either incorporating all testing as part of the test-case generation, so that the test suite does not have to be run repeatedly and separately, or aggressively leverage the parallelism of running many tests in parallel.

In the remainder of this section, we elaborate on the principles behind concolic execution and related techniques, as well as the existing technology that we will serve as our starting point.

```
bool bounds_check(unsigned int x) {
    x = x + 100;      // (1)
    if( x < 1000 )    // (2)
        return true;  // (3)
    return false;     // (4)
}
```

Figure 3. Concolic Example

Concolic execution combines *conc*rete execution (used in dynamic analysis) with symb*olic* execution (used in static analysis) [185]. It has also been described as *directed automated random testing* [96], *whitebox fuzzing* [97], and symbolic execution [35, 58]. The technique starts by generating a random input for the program (or module) under test. The concrete (actual) execution of the program is observed on the random input. As the concrete execution proceeds, a set of symbolic constraints is recorded that summarizes what must be true for execution to proceed along the observed path. At the end of the execution, any solution to the constraints should provide an input that would cause the program to follow the same path, execute the same sequence of instructions, and decide all branches in the same way. By modifying the constraints before they are solved, it is possible to get an input that follows a similar, but slightly different path.

Consider the function in Figure 3: let execution begin on (random) input x=201,056. The execution begins at statement (1) in the concrete (actual) state x=201,056, and the symbolic state is true (*i.e.*, unconstrained). After statement (1), the concrete state is x=201,156, and the symbolic state is described by a single constraint $x_1 = x_{in}+100$. Concrete execution proceeds to (2), which evaluates to false, causing execution to proceed to (4). The concrete state remains unchanged during these transitions, but the symbolic state at (4) is updated by conjoining the constraint for the decision of the branch: $x_1 = x_{in}+100 \wedge x_1 \geq 1000$.

At this point, the execution ends. Note that any solution to the symbolic constraints that describe the symbolic state, also known as *path constraints*, would cause execution to follow the same path, and that the initial input is one solution to the path constraints. Concolic execution proceeds to search for an input that will cause a different choice at the last (and in this case only) branch. The constraint from this branch is negated, giving the constraints: $x_1 = x_{in}+100 \wedge x_1 < 1000$.

An SMT solver is called to generate a solution to the path constraints, for example x_{in}=750, and a new round of concolic execution is performed using the chosen solution as inputs. For this example, two rounds are sufficient to achieve complete path coverage, and the process terminates. Note that traditional fuzzing (using random inputs) has a very low probability ($1000/2^{32}$) of generating an input that exercises line (3).

The effects of system calls and pieces of OS state that we do not want to consider as inputs (e.g., because we cannot control them or in order to make the analysis more efficient), can be "borrowed" from the concrete state [96]. This may cause the concolic engine to miss some execution paths that are possible (because it over-tightens the symbolic constraints to the specific concrete execution it has witnessed). However, it will always be able to make forward progress and does not get stuck with expensive (or even undecidable) analysis problems. In effect, concolic execution trades off the strengths and weaknesses of dynamic and static analysis against one another.

As part of a Phase II SBIR on "Deobfuscating Tools for the Validation and Verification of Tamper-proofed Software" GrammaTech is developing a concolic-execution engine for x86 machine code. The salient feature of our concolic-execution engine is the use of GrammaTech's framework for software dynamic translation (gtSDT) in order to monitor actual program executions and to obtain concrete run-time values of the processor's registers and memory locations. One task in this project was to adapt GrammaTech's concolic-execution engine to work with Strata, instead of gtSDT.

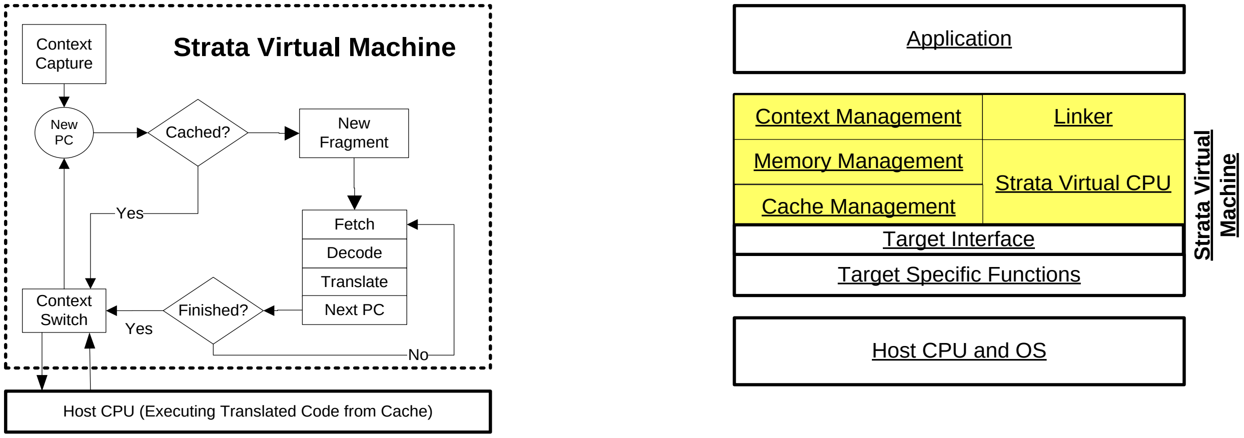
The DASH algorithm uses an abstract program representation to drive the concolic-execution engine [40]. In each round, the algorithm first looks for a path in the abstract program representation that leads to an error state (or a security violation). It then uses a step similar to concolic execution to grow from a previously observed execution towards the abstract error path it has found. If it fails to expand any of the observed paths, DASH refines the abstract program representation to eliminate the abstract path. The algorithm iterates until one of the following three results is reached:

1. An input is discovered that triggers one of the errors that DASH was searching for.
2. Abstraction refinement results in an abstract program representation with no error paths. In this case, the representation serves as proof that the program is error free.
3. The algorithm times out. The algorithm still produces a test suite with good coverage.

Our colleagues at the University of Wisconsin have demonstrated that the DASH algorithm can be applied directly to machine code [125, 203]. They have also demonstrated that it is possible to modify the DASH algorithm to achieve test suites with useful coverage properties. Unfortunately, the use of the DASH algorithm on machine code has poor scalability. A research goal for PEASOUP is to determine if a simplified version of DASH can be used to improve the coverage of generated test suites.

Static Machine-Code Analysis. PEASOUP also leverages static analysis, where appropriate. In Phase 1, the static analysis is implemented as an IDA Pro plugin. In future versions of PEASOUP, we may leverage more advanced static analyses [36, 37, 126, 170].

2.5.1.2 Strata: a Retargetable Software Dynamic Translator



(a) Strata operates by fetching, decoding, and translating an application's instructions into a translation cache.

(b) The Strata VM sits between the application and the host CPU

Figure 4. Strata Architecture

The Execution Manager requires the ability to monitor and in some cases to modify the execution stream of the protected executable at the granularity of individual instructions. Software Dynamic Translation (SDT) is a technology that enables software malleability at the instruction level by providing facilities for run-time monitoring and code modification. SDT can affect an executing program by injecting new code, modifying existing code, or controlling the execution of the program in arbitrary ways. To facilitate SDT research and development of innovative SDT applications such as next generation software field certification technology, the University of Virginia constructed a portable, extensible SDT infrastructure called Strata. As shown in Figure 4, Strata is organized as a process-level virtual machine that mediates execution of an application's instructions. Strata was designed and implemented using object-oriented principles with target independent and dependent services that can be easily reconfigured and retargeted for new applications and computing platforms [181].

Strata dynamically loads a binary application and mediates application execution by examining and possibly translating an application's instructions before they execute on the host CPU. Translated application instructions are held in a Strata-managed code cache called the fragment cache. Once a fragment finishes execution, the Strata VM captures and saves the application context (e.g., PC, condition codes, registers, etc.). Following context capture, Strata processes the next application instruction. If a translation for this instruction has been cached, a context switch restores the application context and begins executing cached translated instructions on the host CPU. Otherwise, the instruction is translated (and, possibly, instrumented) and the translation is placed into the fragment cache and is executed on the host CPU.

For the Execution Manager, the key point is that Strata examines each instruction in the program and can insert instrumentation code to enforce desired security properties. For example, if a security-critical system call should only be made with its arguments having certain value ranges, as observed during acceptance testing, then instrumentation code could be dynamically inserted at each call site to check the values of arguments before allowing the call to be made.

Because Strata uses the host CPU to execute instructions, it is very efficient. Figure 5 shows Strata's performance (normalized to native execution) for the SPEC2000 benchmark suite on an

AMD Opteron 244 machine running in IA32 mode. The figure shows that applications executing under control of Strata run only 2.2% slower on average than when run natively [109]. For some applications, 179.art for example, Strata even yields small performance improvements over native execution due to improved spatial code locality.

The performance of the Strata virtual machine is important for achieving the performance goals of PEASOUP. In previous DARPA-funded work UVA demonstrated the practicality of running what would seem at first glance to be expensive operations, *e.g.*, encrypting/decrypting binaries, at less than 10% overhead [112]. The Strata VM was the subject of two independent red team exercises as part of DARPA Self-Regenerative Systems program during which no major flaws of were found.

2.5.1.3 Secure In-VM Monitoring

Strata enforces security properties on a program from within the program's application space. This structure allows Strata to enforce arbitrary policies while achieving high efficiency. However, for most security researchers it commits a cardinal sin: it exposes the security mechanism itself (Strata) to attack. For this reason, many security tools are designed to reside in a completely separate virtual machine. This location promises them isolation from the application vulnerabilities and even vulnerabilities in the OS, but it introduces a high performance overhead. For PEASOUP, we need both (a) the fine granularity and high efficiency offered by Strata and (b) the equivalent of VM-based isolation for Strata itself. In order to achieve these goals, we proposed to integrate Strata with a technique known as Secure In-VM Monitoring (SIM) developed at the Georgia Tech [191]. As described above, we eventually developed an architecture that put the translator into a separate process, rather than a separate VM.

As mentioned above, performance overhead is the main drawback of external security tools that reside in a security VM rather than in the desktop OS. External security monitors are activated when specific security-related events occur in the OS. Generally, hooks can activate event handlers of the security monitor via mediation by the hypervisor. In addition, hypervisor intervention is also required for virtual machine introspection (VMI), when data needs to be read and written from the OS's address space. The overhead of external security tools is thus primarily due to the change in privilege levels that occurs while switching back and forth between the kernel-level and the hypervisor-level.

Georgia Tech developed the Secure In-VM Monitoring (SIM) approach to address these performance issues while keeping the security benefits of having the security tool external to the desktop OS. Their design meets two requirements. The first requirement is fast invocation (i.e., invoking handlers residing in the monitor should not involve any privilege level changes). The second requirement is that reading and writing data from the OS should occur at native speed.

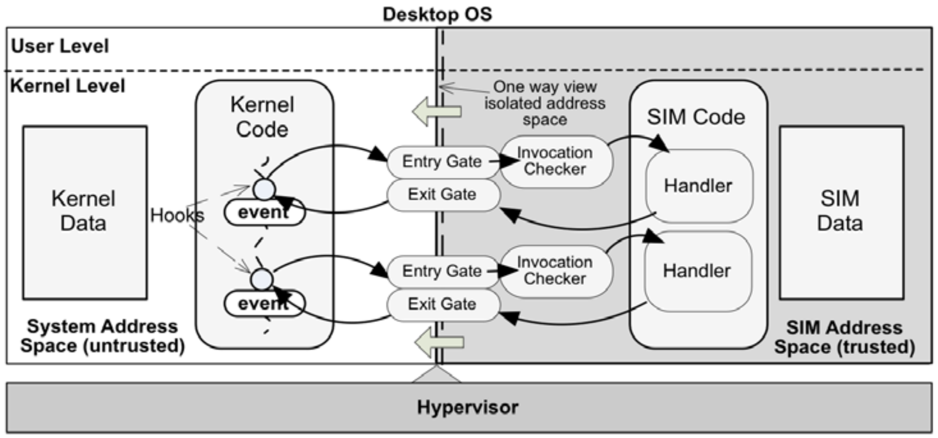


Figure 6. Architecture of SIM

The overall design of SIM is shown in Figure 6. The key idea of SIM is to introduce a separate hypervisor-protected virtual address space called the SIM address space in the guest VM containing the user's OS environment. This protected address space is used for the security monitor. It exists in parallel to the virtual address spaces being utilized by the OS, which we will refer to as the system address spaces. The SIM address space includes the security monitor's code (SIM code) and data (SIM data). The entire system address space is mapped into the SIM address space, but not vice versa. This mapping means that the security monitor can view the address space of the OS, but no code executing in the OS can view the security monitor's address space. In order to securely transfer execution between the desktop OS and the monitor inside SIM, a number of entry gates and exit gates are generated. By utilizing the hardware features in Intel VT-x, these gates allow switching between the OS and the monitor inside SIM without having the hypervisor activated. Moreover, the memory mapping approach allows reading and writing from the OS's address space at native speeds, satisfying the performance requirements.

The SIM approach provides security similar to placing monitors in an external security VM. First, the entry and exit gates are the only means of transferring execution between the system address space and the SIM address space. By ensuring that the pages containing the gates are the only pages containing execute privileges on both address spaces, we enforce that there is no other way to transfer between the address spaces. Hooks are placed in the kernel before specific events that transfer control to corresponding gates. The entry gate has an invocation checker module that verifies whether the gates are invoked by valid hooks. Finally, by having all the system address space contents that are mapped into the SIM address space as non-executable, we ensure that no code in the untrusted user's OS can alter behavior of the security monitor in SIM. Therefore, isolation, secure invocation and integrity of monitoring behavior are satisfied by design.

2.5.2 Components of PEASOUP

Section 2.5.1 presented the technologies that will be used in PEASOUP. This section describes how those 'ingredients' are assembled in PEASOUP.

2.5.2.1 Offline Analyzer

The offline analyzer can leverage high-coverage test suites (see Section 2.5.1.1). This is used for IR recovery, vulnerability detection, and validation.

IR Recovery and Vulnerability Detection Module. Our approach to IR recovery and vulnerability detection combines static analysis, dynamic analysis, and automatic generation of test suites. At the beginning of the project, we envisioned the use of dynamic analysis and test-case generation as critical. However, we discovered static-analysis techniques that were much more scalable and robust.

The one drawback of dynamic analyses is that they only provide results for the executions they observe. Our attempted use of (automatically generated) high-coverage test suites was meant to overcome this shortcoming. The high coverage would have allowed us to recover a high precision IR.

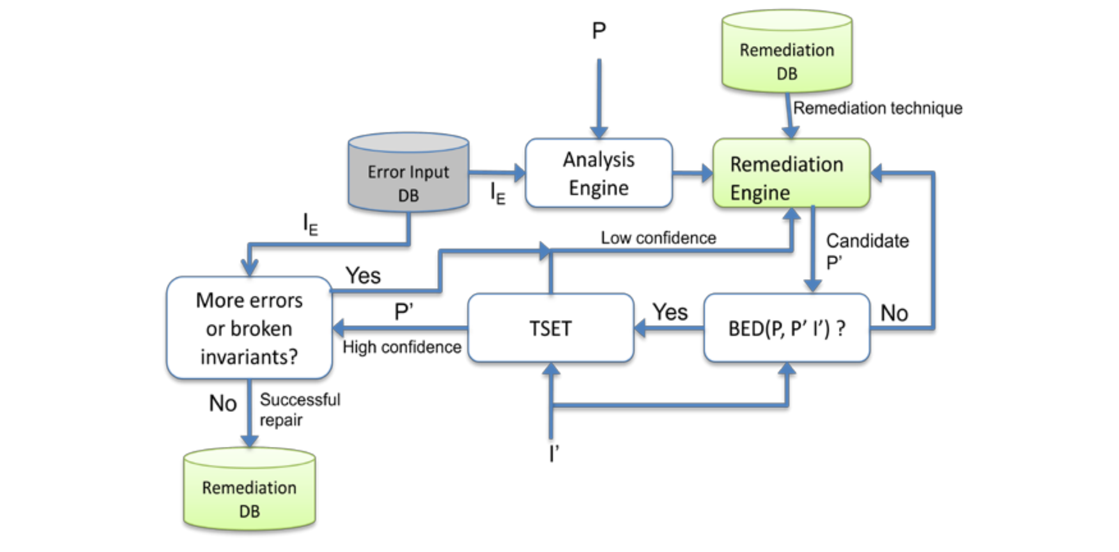


Figure 7. The Remediation Strategist

Remediation Strategist. As explained above, identifying a vulnerability or detecting an attack are not sufficient. In order to do better than failure-stop or failure-oblivious, it is important to discover vulnerabilities offline and prepare a remediation. To this end, PEASOUP uses the *remediation strategist* illustrated in Figure 7. As the figure shows, the remediation strategist contains a database of candidate mechanisms based on either repairing the underlying fault or invoking a recovery plan. The candidates are applied and evaluated offline to determine which mechanisms yield a high confidence that the program semantics are preserved.

The remediation strategist works by analyzing the program using inputs that have caused the binary program, P, to fail. Ideally, remediation strategies are carefully constructed in such a way that they are likely to fix the vulnerability without causing a deviation from the program's specified behavior. Several recent research results have demonstrated the viability of discovering and applying remediation strategies that meet these criteria [47, 69, 160, 171]. However, PEASOUP does more than rely on heuristics and careful design to ensure the safety of its candidate remediation strategies. Each remediation mechanism is evaluated to ensure that normal behavior is preserved (using the BED module described below) with high confidence (using the TSET module described below).

The candidate remediation database holds possible remediation candidates to be applied when a program vulnerability is detected. There is a large variety of approaches to remediation. The exact approach taken will depend on the vulnerability that is discovered and the information available in the recovered IR. Some example approaches include:

- Insert sanity checks, such as checks for buffer overflows, and discard actions that are likely to lead to unexpected behavior [160].
- Generate and return an "error code" from the function that contains the error [193].
- Find error handling code in or near the failure point and invoke it [193].
- Return an error code from known system calls (`mmap`) or functions (such as `malloc`).
- Filter inputs that are likely to exploit a vulnerability [47, 69]. Alternatively, allow suspicious input to be processed, but use VM checkpointing and careful monitoring to rewind and discard the input if an attack is detected.
- Ignore the instruction that generates the error [171].

Prior work has shown many of these techniques to have applicability and some success in patching a program in the face of program faults. Consequently, they make good remediation candidates. An innovation of the PEASOUP architecture is the way in which the candidates are evaluated.

Variant Generator. The variant generator uses the recovered IR to heuristically generate program variants that are validated by the validation module.

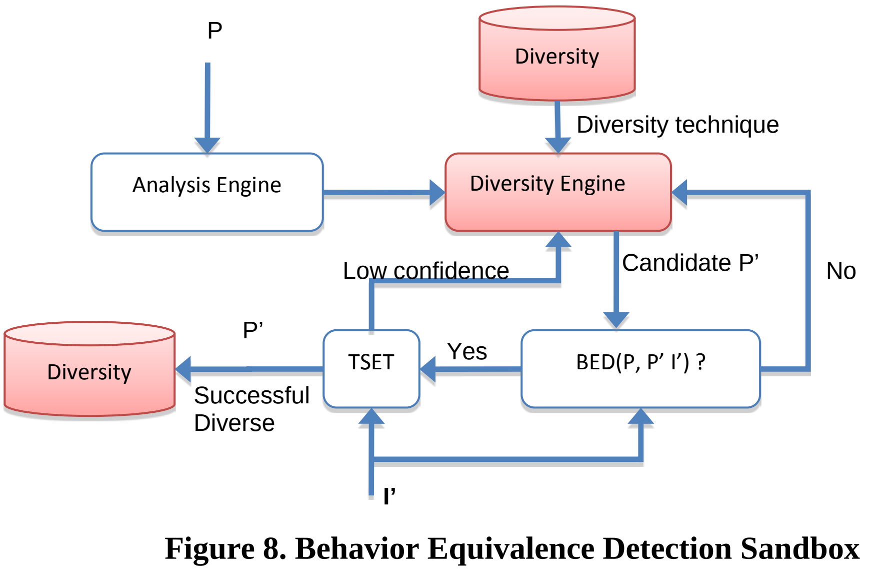


Figure 8. Behavior Equivalence Detection Sandbox

Validation Module. Our approaches to remediation and diversification require validation of the remedies and the program variants. For example, our approach to diversification uses heuristic transformations guided by the recovered IR to generate program variants. Often, these heuristic transformations produce functionally-equivalent (on normal program input) versions of the program. Sometimes, however, the assumptions made by the heuristics turn out to be invalid and the diversity transformations fail. Instead of rejecting the heuristic-approach, we use a Behavior Equivalence Detection (BED) module to determine whether the heuristic generated a useful program variant and Test-Suite Evaluation Technology (TSET) to provide high confidence that the transformations indeed are correct. If the BED module or TSET module report that the diversity transformation is broken or has low confidence in being correct, the variant can be rejected. In this way, we believe the PEASOUP architecture can provide high-entropy diversity for much of a program, even when only the binary version of that program is available.

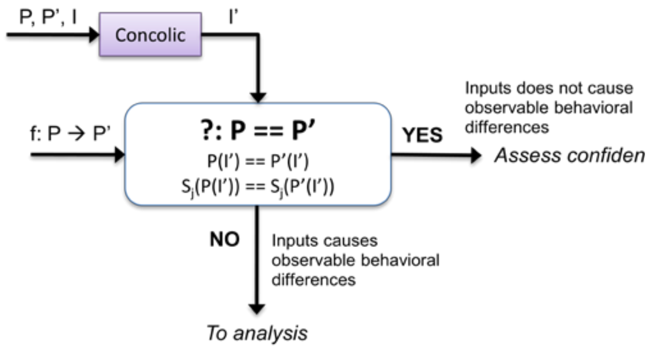

P, P', I
Concolic
I'
f: P → P'
?: P == P'
P(I') == P'(I')
S_j(P(I')) == S_j(P'(I'))
YES
Inputs does not cause observable behavioral differences
Assess confiden
NO
Inputs causes observable behavioral differences
To analysis

Figure 9. BED Module

The following sections describe the BED module and TSET modules in more detail.

Behavior Equivalence Detection. The Behavior Equivalence Detection (BED) module uses concolic testing to determine if two versions of a program (often, one version is the original unprotected SOUP binary, and another is a diversity variant in consideration for use online) behave the same for a broad class of program inputs that we believe to be "normal" inputs (labeled I' in the figure below). We assume that the original program acts normally when heavy-weight, precise detectors (such as MEDS, taint-tracking, etc.) do not detect any abnormal behavior for the given program input. In some cases, it may be feasible and appropriate to simply run both versions of the program and ensure that the program generates the same output for each input in the test suite (P(I')==P'(I') (see Figure 9).

In other cases though, a more thorough comparison is necessary. In such cases, it may be desirable to run both variants (P and P' in Figure 9) in lock-step and maintain a mapping between the state of the original version to the variant. If we detect any variability between the two variants while maintaining the state mapping, we know that the variants are behaving differently. For example, none of our diversification techniques should affect control-flow decisions. So, a difference in the control-flow decisions between P and P' when run on identical inputs indicates an unacceptable change in behavior, even though different control-flow decisions may not always be lead to different outputs [167, 168]. This functionality was not implemented during Phase 1, but will be investigated as necessary in later phases of the project.

Ultimately, the output of the BED module is a yes-or-no answer. Yes indicates that the variants behave the same, while no indicates that P and P' definitely compute different functions. To assess the confidence of *yes* answers, we rely on the TSET module, described below.

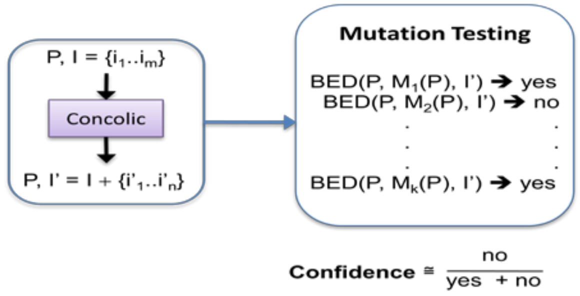


Figure 10. Test-Suite Evaluation

Test-Suite Evaluation Technology. Ultimately, the BED module's ability to detect differences depends on the input set's ability to distinguish two similar programs from one another. If we wrongly assume that BED has a high ability to distinguish such programs, our proposed diversity transformations will likely break the program and create many false positives. Since a false positive will invoke the remediation module to determine if a patch is necessary, avoiding false positives will be very important. Consequently, we *must* have high confidence that the BED module is capable of detecting incorrect transformations and repairs.

To combat this problem, PEASOUP uses Test-Suite Evaluation Technology (TSET). TSET is designed to determine whether concolic testing can adequately detect program differences. The simplest version of TSET evaluates the test suite simply based on the coverage achieved by the test suite. A more advanced approach will use a variation of mutation testing; that is, we will programmatically insert errors (M_i in Figure 10, where $0<i<N$ in the figure below) into a program (P in the figure below) and determine whether the test suite is capable of detecting those errors. If the test suite is capable of detecting errors in the program, we have higher confidence that it can be used to detect programmatic differences in the BED module.

Evaluating Remediation Candidates. Given the variety of possible patches to a program, we need a mechanism to determine (prior to deployment) patches that maintain the program's functionality without introducing security vulnerabilities. To achieve this goal we thoroughly test the revised version to behave the same on all normal inputs that we can derive from advanced coverage testing. If the patched program produces the same output for all test cases that the original program does under a "normal" execution (i.e., no confinement policies are violated), then we need to consider the anomalous inputs sets. For those inputs, previous technology is not capable of providing any confidence that a particular patch is suitable for a particular vulnerability. Our approach differs in that we test the patches on anomalous inputs and evaluate if the program is behaving properly.

To accomplish this, the program needs to:

- Not produce further confinement errors after the patch code has executed.
- Keep the program's data and control flow invariants intact to provide higher confidence that the patch does not open a program to new vulnerabilities.

- Continue to properly process input; i.e. not enter erroneous states nor enter infinite loops.

PEASOUP uses the TSET and BED modules to help address the equivalence (modulo the remediation) of the original program and the patched program with high confidence. Since the testing of the patched program occurs offline, we can use expensive confinement policies (such as MEDS) to ensure that the program's behavior past the patch point is not anomalous.

2.5.2.2 Execution Manager

As described above, the execution manager is responsible for application of remediation strategies, selection of program variants, and confinement of exploits against undetected or unhandled vulnerabilities. By combining software dynamic translation (see Section 2.5.1.1) with a multi-process translation architecture (see Section 2.5.1.3) we can meet these requirements with low overhead. The application of remedies and selection of program variants is relatively straightforward with software dynamic translation. More details are provided in Section 3.3.5. In the remainder of this section, we describe some examples of the confinement techniques that we believe will be most beneficial.

2.6 Related Research

Discussion of related research is located throughout this report. Perhaps the most relevant paper is the result of the Applications Community project led by MIT [160]. This research is similar to PEASOUP in that it uses confinement techniques to identify faults and dynamic analysis to develop patches that remove the faults. MITs results are particularly important because they demonstrate that automatic removal of faults can be effective and safe, in some circumstances. However, [160] is limited in several ways that PEASOUP is not: (i) it only detects a small class of faults; (ii) faults are detected solely by online confinement, leaving them vulnerable to zero-day attacks; and (iii) the quality of patches is determined by user feedback.

In addition to addressing the shortcoming listed above, the PEASOUP project also aimed to advance the state-of-the art in program analysis, automatic fault remediation and diversification techniques.

Traditional static analysis attempts to explore all possible program behaviors to identify potential vulnerabilities. Generally, traditional static analyses have shortcomings: (i) applicability to narrow classes of vulnerabilities, (ii) high computational cost, and (iii) high false-positive rates. New lightweight analysis tools, e.g., GrammaTech's CodeSonar, exhibit scalability and false-positive rates that make the tools more useful for real software but these attributes are achieved at the cost of high false-negative rates (missed vulnerabilities), which are not acceptable for automatic protection. PEASOUP demonstrates that carefully designed static analyses can be very effective for automatically recovering IR that enables defenses of important classes of vulnerabilities, including those resulting from number-handling and command-injection weaknesses.

Dynamic program analysis observes runtime behavior and identifies any errors that occur. Relevant examples of such analyses include Control Flow Integrity (CFI) [24], Data Flow Integrity (DFI) [69], dynamic taint analysis [115], and program shepherding [122]. Typically, these techniques first perform an offline analysis to learn the "normal" behavior of the program, and then monitor each run of the program for deviations. The offline step often requires program source code. Some monitoring techniques require modifications to the OS. Overheads imposed by most dynamic analyses are prohibitive: some techniques even rely on special-purpose hardware to reduce the overheads [227]. Finally, dynamic analyses do not adapt to new exploits. The proposed approach advances the state of the art in dynamic detection of vulnerabilities as follows: (i) it operates directly on stripped executables; (ii) it requires no OS modifications or special-purpose hardware; (iii) it has low overheads; and (iv) it uses a dual-process architecture [191] to ensure security of the monitor.

Concolic Execution. As previously discussed, c*oncolic execution* (or *whitebox fuzzing*) [49, 96] uses analysis to derive inputs that exercise execution paths that have not been explored before. Concolic-execution-based tools, such as SAGE [97] and KLEE [48], have been shown to scale to real-world programs and were used to find critical vulnerabilities. In our approach, concolic execution was tightly integrated with run-time program monitoring, with a goal of improving vulnerability detection. Unfortunately, we were not able to generate sufficiently high-coverage test suites for medium and large programs. While concolic-execution-based equivalence testing has been proposed before [48], diversification and patching require relaxed criteria, namely, equivalence testing modulo “normal” behavior. We are unaware of existing techniques for this kind of testing.

Fault remediation. The original focus of work on fault tolerance and remediation was to prevent software from failing due to regular programming errors, but recently the security community has expanded the focus to include withstanding malicious attacks. The proposed techniques range from *software rejuvenation* [151], in which the program (typically, a server) is occasionally restarted to counter the effect of software aging (e.g., the cumulative effect of memory leaks, unreleased file locks, and file-descriptor leaks), to *slipstream execution* [118], in which a shadow copy of a program is executed in parallel with the main copy and the redundancy is used to withstand transient hardware faults. Recently, an approach that combines redundancy and rejuvenation has been proposed to counter cyber attacks [223] Vigilante [69]—a system for network worm containment—employs a different approach to fault remediation. Vigilante analyzes compromised applications (located on a honeypot server) to derive input filters to be deployed for the actual installations of the application.

Recently, language-based approaches that rely on the analysis of an application have been proposed. M. Rinard, M. Ernst, and B. Demsky developed an approach for inferring application-data-structure invariants, detecting their violations, and performing online repairs necessary to re-establish the invariants [77-79]. An independent red-teaming effort has shown that their approach was able to successfully withstand real-world attacks. Sidiroglou et al. proposed a recovery technique that relies on *rescue points*—application code for handling programmer-anticipated failures. In their tool ASSURE, the execution is restored to an appropriate rescue point after a fault is detected [193].

Our work built on the above ideas. A key innovation of our approach was the offline evaluation of the possible remediation approaches for a given program vulnerability.

Diversification. Data Diversity is a technique for creating software systems that can tolerate faults during execution [26]. The concept is to execute multiple, identical copies of a program with each copy using different but equivalent data. Because the data read by the copies is different, the copies execute differently, and in some cases, a copy may not be subject to any given fault. The outputs of the copies are supplied to a voter, and the most common output is selected for use. It has been noted that diversity can be used to thwart the software monoculture problem; that is, diverse variants deployed across different installations can keep one attack from being successful against all sites [88]. Current diversity techniques require compiler-tool-chain support, or suffer from low entropy. Our use of data diversity will address the limitations by providing high-entropy diversity using only static analysis and testing of binary code.

2.7 Project Contributors

GrammaTech is the Prime on the PEASOUP effort. GrammaTech brought a wealth of experience and relevant technology to the project based on its research into machine code analysis starting in 2002 [38, 38]. In particular, GrammaTech provided Grace, the concolic execution engine that served as the starting point for automatic test-case generation in PEASOUP.

The Univ. of Virginia (UVA), Georgia Institute of Technology, and Raytheon were all subcontractors on the PEASOUP effort. UVA and Georgia Tech provided crucial technology for PEASOUP. UVA's contributions started with Strata and MEDS, mature tools for software dynamic translation and runtime detection of memory errors, respectively. UVA also brought extensive experience in building security tools to help develop all of the core technologies in PEASOUP.

Georgia Tech's contributions began with Secure In-VM Monitoring (SIM), an advanced technique for secure, low-cost monitoring of guest domains in a virtual machine. Georgia Tech applied its technology to securing the components of PEASOUP from attack, introducing diversity to the application-OS interface, and implementing recovery techniques.

Raytheon brought its extensive experience in successfully integrating large complicated software systems to the PEASOUP project. In particular, Raytheon provided integration testing to ensure that all of the components developed by various team members work together. Raytheon is also focused on finding real-world test platforms for PEASOUP and ensuring that PEASOUP can be successfully transitioned at the end of Phase III.

In addition to the above contributors, the PEASOUP effort benefited from the ongoing collaboration between GrammaTech and the University of Wisconsin on machine-code analysis. In particular, Wisconsin was heavily funded to continue its research into machine-code analysis and GrammaTech had access to any extensions that Wisconsin made to its technology.

2.8 Summary of Statement of Work Tasks

This section contains a copy of the SOW tasks, as amended during the course of the project. Notice, the SOW task items are numbered according to the original SOW, not the section numbers of this document.

2.8.1 Phase 1 Tasks

For Phase 1, the contractor shall orient the research towards initially handling vulnerabilities classes C1, C5, and C7 (buffer overflow/underflow). The contractor shall meet the required performance goals for at least two vulnerability classes, but will retain the flexibility, if needed or strategically advantageous, to replace the anticipated vulnerability classes with others from the list specified in the BAA. The goals of Phase 1 are to successfully process 85% of executables, render 75% of the vulnerabilities in two classes unexploitable, with no more than 20% runtime overhead. The contractor shall document the findings of the effort. The remainder of this section describes the following tasks for Phase 1.

SOW Task 3.1.1 Analysis.

SOW Task 3.1.1.1 Tools for providing "ground truth" IR. The contractor shall develop or enhance existing tools for creating an accurate representation of the program's Intermediate Representation **(**IR) based on the output of compilation tools. This will serve two purposes: (a) allow work to start on components that require IR before the IR recovery module is complete; and (b) provide a baseline for evaluating the quality of recovered IR.

SOW Task 3.1.1.2 IR Recovery and Vulnerability Detection Module. The contractor shall develop an IR recovery and vulnerability module that will include the following subtasks:

SOW Task 3.1.1.2.1 The contractor shall design the format of the IR and vulnerability database.

SOW Task 3.1.1.2.2 The contractor shall develop a tool for automatic test-case generation for Linux executables by combining a concolic execution engine with a software dynamic translation tool, such as Strata. The tool should achieve 90% instruction coverage.

SOW Task 3.1.1.2.3 The contractor shall integrate the automatic test-case generator with the Memory Error
Detection System (MEDS) and enhance functionality to provide the necessary IR recovery (95% agreement with ground truth) and vulnerability detection (75% of vulnerabilities detected).

SOW Task 3.1.1.3 The contractor shall develop prototypes of the Behavior Equivalence Detection (BED) and Test-Suite Evaluation Technology (TSET) components.

SOW Task 3.1.1.4 The contractor shall develop improved techniques for generating high coverage test suites. The contactor shall investigate extensions to the concolic engine and the use of more advanced techniques, such as adaptations of the DASH algorithm.

SOW Task 3.1.2 Confinement.

SOW Task 3.1.2.1 The contractor shall develop technology for fine-grained confinement of Phase 1 vulnerabilities. The contractor shall enhance the chosen software dynamic translation tool to provide confinement of the Phase 1 vulnerabilities based on the information in the recovered IR. The goal is to achieve low overhead (less than 15%) with high recall (greater than 80% of exploits prevented).

SOW Task 3.1.2.2 The contractor shall perform migration of the Secure In-VM Monitoring (SIM) to protect a guest Linux domain. The SIM prototype currently protects guest domains running Windows. Under this task, the contractor will adapt SIM to protect Linux domains.

SOW Task 3.1.2.3 The contractor shall integrate the dynamic translator (e.g. Strata) with SIM. The contractor shall adapt SIM to protect the translation technology in the Software Dynamic Translation *(*SDT).

SOW Task 3.1.3 Diversification.

SOW Task 3.1.3.1 The contractor shall develop diversification transformations and the ability to select program variants at run time. The diversification transformations will be performed by the variant generator and will be guided by the IR. Variants will be selected at runtime in the Execution.

SOW Task 3.1.3.2 The contractor shall investigate techniques for leveraging the Virtual Machine (VM) to support diversification. Possible techniques include changing the system call mapping, so that injected code cannot reliably invoke system calls.

SOW Task 3.1.4 Remediation. During Phase 1, work on automatic remediation shall be limited to the development of efficient Virtual Machine Monitor (VMM)-based check pointing techniques.

SOW Task 3.1.5 Integration, evaluation, and transition.

SOW Task 3.1.5. 1 Architecture development and interface definition. The contractor shall develop an overall architecture and maintain the definition of the interfaces between major software components.

SOW Task 3.1.5.2 Software development support. The contactor shall provide support and continually determine the state of software during development. The contractor shall perform nightly regression tests, evaluate test coverage and track bugs in technology software. The contractor shall perform and maintain software Configuration Management (CM) throughout the phase. The contractor shall define and establish the CM process required and shall provide mastery of the test corpus. The contractor shall document and describe vulnerabilities and exploits found in the test corpus. The contractor shall extend the test corpus to include exploits not provided in primary test corpus.

SOW Task 3.1.5.3 System integration. The contractor shall perform system integration for PEASOUP and identify a lead technology integrator throughout the agile software development process. The contractor shall plan and execute integration in increments in concert with the development of software components.

SOW Task 3.1.5.4 Metrics Testing. The contractor shall periodically evaluate technology performance with respect to vulnerabilities and metrics set forth in the BAA and with respect to capabilities expected at the time of evaluation, and shall report test results to component developers. An evaluation shall occur prior to each program milestone/waypoint after the kickoff meeting.

SOW Task 3.1.5.5 Extended Analysis, Confinement, Diversification, and Remediation. The contractor shall investigate further techniques for analysis, confinement, diversification, and remediation specifically intended to address the following types of test results from independent testing and evaluation, namely, tests in which a test program was not processed successfully, tests in which a test input was not prevented from exploiting a test program, and tests in which the prototype technology altered the intended benign functionality of a test program in unexpected ways.

SOW Task 3.1.5.6 Extended Testing. The contractor shall support independent testing and evaluation as required, including on-site technical support and prototype interoperability with Government test automation software.

2.8.2 Phase 2 Tasks

In Phase 2, the contractor shall orient their efforts towards further refining techniques for handling the vulnerability classes selected in Phase 1. The contractor shall incorporate protections against two additional vulnerability classes (C5–command injection and C8–null pointer dereference); PEASOUP will be evaluated on its ability to prevent exploits of vulnerabilities in these classes during Independent Phase 2 independent test and evaluation. The contractor shall begin development of protections for vulnerabilities in additional vulnerability classes that will be (independently) evaluated in Phase 3, beyond those included in the Phase 2 independent test and evaluation. As in Phase 1, the contractor shall be allowed to retain the flexibility to replace or expand the anticipated vulnerability classes. The contractor shall continue to expand on Phase 1 results and to significantly ramp up their efforts on remediation. The goals of Phase 2 are to render 90% of the Phase 1 vulnerabilities unexploitable and 80% of an additional two vulnerability classes that are evaluated during the Phase 2 independent test and evaluation, with a runtime overhead of no more than 15%. The contractor shall document the findings of the effort. The remainder of this section describes the following tasks for Phase 2.

SOW Task 3.2.1 Analysis.

SOW Task 3.2.1.1 The contractor shall continue to enhance IR recovery and vulnerability detection initiated in Phase 1.

SOW Task 3.2.1.2 The contractor shall continue to enhance the test-case generation initiated in Phase 1.

SOW Task 3.2.2 Confinement.

SOW Task 3.2.2.1 The contractor shall continue to enhance fine-grained confinement initiated in Phase 1.

SOW Task 3.2.2.2 The contractor shall continue to enhance BED and TSET initiated in Phase 1.

SOW Task 3.2.2.3 The contractor shall develop and make enhancements to SIM. Tasking under this effort shall include: developing enhanced protection for the monitoring code generated by Strata; removing any residual vulnerabilities in the SIM prototype, including reliance on guest OS; and developing techniques to dynamically alter the degree of monitoring performed by the SIM based on security conditions.

SOW Task 3.2.3 Diversification.
The contractor shall continue to enhance diversification initiated in Phase 1.

SOW Task 3.2.4 Remediation.

SOW Task 3.2.4.1 The contractor shall continue work on automatic remediation initiated in Phase I.

SOW Task 3.2.4.2 The contractor shall develop a database for storing remediation candidates.

SOW Task 3.2.4.3 The contractor shall develop a Remediation Strategist component and enhance the Execution Manager with capability to select and apply remedies.

SOW Task 3.2.5 Integration, evaluation, and transition.
The contractor shall continue to perform system integration, evaluation, and transition activities for Phase 2. Tasking shall include: proving updates to the system architecture; maintaining interface definitions between major software components; managing CM, performing system integration, conducting regression testing, performing metrics testing, and documenting/describing vulnerabilities and exploits.

SOW Task 3.2.5.1 The contractor shall begin to port PEASOUP to run on an x86-64 platform (OS and processor) and to protect x86-64 binary executables.

SOW Task 3.2.6 Support for Independent Testing and Evaluation. The contractor shall support Phase 2 independent testing and evaluation as required, including on-site technical support and prototype interoperability with Government test automation software.

2.8.3 Phase 3 Tasks

In this phase, the contractor shall incorporate protections against two additional vulnerability classes (C2–tainted data and C3–error handling) and continue to refine the vulnerability classes selected for Phases 1 and 2. As in the previous phases, the contractor shall retain the flexibility to replace or expand the anticipated vulnerability classes with others. In addition to achieving the final metrics required for Phase 3, the contractor shall also ensure that all developed technology is ready for transition to end users. The contractor shall provide additional capabilities in Phase 3 as required, such as support for a 64-bit Operating System (OS). The goals of Phase 3 are to render 95% of the Phase 1 and Phase 2 vulnerabilities unexploitable and 90% of an additional two vulnerability classes, with a runtime overhead of no more than 10%. The contractor shall document the findings of the overall effort.

SOW Task 3.3.1 Analysis.

SOW Task 3.3.1.1 The contractor shall continue to further enhance IR recovery and vulnerability detection from Phase 2.

SOW Task 3.3.1.2 The contractor shall continue to further enhance test-case generation from Phase 2.

SOW Task 3.3.2 Confinement.

SOW Task 3.3.2.1 The contractor shall continue to further enhance fine-grained confinement from Phase 2.

SOW Task 3.3.2.2 The contractor shall enhance VMM-based confinement.

SOW Task 3.3.3 Diversification.
The contractor shall continue to enhance diversification from Phase 2.

SOW Task 3.3.4 Remediation.
The contractor shall continue to enhance automatic remediation from Phase 2.

SOW Task 3.3.5 Integration, evaluation, and transition.
The contractor shall continue to perform system integration, evaluation, and transition activities required for Phase 3. Tasking shall include: proving final updates to the system architecture; maintaining interface definitions between major software components; managing and performing CM, performing system integration, conducting regression testing, performing metrics testing, and documenting/describing vulnerabilities and exploits.

SOW Task 3.3.5.1 The contractor shall complete the port of PEASOUP to run on an x86-64 platform (OS and processor) and to protect x86-64 binary executables.

SOW Task 3.3.6 Support for Independent Testing and Evaluation. The contractor shall support Phase 3 independent testing and evaluation as required, including on-site technical support and prototype interoperability with Government test automation software.

2.8.4 Management Tasks

SOW Task 4.1 Program Management.
The contractor shall exercise administrative and financial management functions during the course of this effort, such as scheduling of activities and milestones; describing status; outlining contractor activity and progress toward accomplishment of objectives; and documenting in detail the work performed and the results of the effort including technological breakthroughs. The contractor shall facilitate Air Force access to internal data developed in compliance with contractual tasks. The contractor shall provide technical and financial status reporting, including participation in scheduled Principal Investigator (PI) meetings and providing project summary information as required.

SOW Task 4.2 Deliverables.
SOW Task 4.2.1 The contractor shall deliver all reporting requirements in accordance with the Contract Data Requirements List (CDRL), Exhibit A to the contract.

SOW Task 4.2.2 The contractor shall deliver all developed prototype software and corresponding source code as identified in the project tasking. Delivery of the software shall be inclusive of software source code and executable.

2.9 Outline of Remainder of Report

This report is organized according to the guidelines provided in ANSI/NISO Z39.18-2005. Section 3.0 describes the methods, assumptions, and procedures used to develop and evaluate PEASOUP. Section 4.0 presents the results of the evaluation and discusses their significance. Section 5.0 provides concluding remarks. Sections 6.0 and 0 list references and abbreviations, respectively.

3.0 Methods, Assumptions, and Procedures

We built a prototype of the PEASOUP system and evaluated it in many different ways, including: (i) measuring its ability to protect programs with known vulnerabilities, (ii) measuring its ability to protect programs with vulnerabilities seeded by a third party, (iii) measuring its effect on the attack surface of the programs in the SPEC 2010 benchmark suite, and (iv) measuring the accuracy of recovered IR against 'ground-truth' IR for a suite of common utility programs.

In the remainder of this section, we provide more details on the construction of the PEASOUP and the experiments we used to evaluate it. Section 3.1 describes the approaches taken to experimental evaluation of PEASOUP. Section 3.2 briefly describes the platform used to develop and test PEASOUP and any assumptions used in the experimental evaluation of PEASOUP. Section 3.3 provides additional details on the core components that are combined in PEASOUP to implement program analysis, transformation, and protection. Sections 3.4–3.9 describe defenses against exploits that target number-handling errors (C1), resource drains, command injection, memory-safety errors, and null-pointer errors, respectively. Section 3.10 presents generic defenses that prevent a variety of exploits.

3.1 Evaluation Metrics and Methodology

The STONESOUP project defined three metrics for the project:

- The percent of programs that are successfully processed. As originally envisioned, this was the percent of programs that PEASOUP declared successfully processed. However, PEASOUP attempts to process almost all programs, so over the course of the project, this metric was effectively replaced with a measure of *Altered Functionality* (AF), or the percent of peasoupified programs that no longer worked correctly. In contrast to the original metric, which should be to process 85% or more of programs, the AF measure should ideally be zero.
- The percent of vulnerabilities (in targeted vulnerability classes) that are *Rendered Unexploitable*, which we abbreviate as the *RUE* metric. The targeted RUE varied over the three phases of the program.
- The percent overhead incurred by protection(s). The targeted overhead was 10%.

The project included an independent evaluation to collect metrics.

The project metrics are designed to evaluate the STONESOUP technologies as a whole. In addition to participating in the independent evaluation, we also focused on gathering metrics related to the performance of individual components and techniques used in PEASOUP. In particular, we were interested in evaluating the following aspects of PEASOUP:

- *Effectiveness of IR Recovery*. One of the important innovations in PEASOUP is its approach to IR recovery from unadorned executables. In order to better understand the effectiveness of our IR recovery, we developed techniques for approximating "ground truth" IR based on analysis of compilation artifacts.
- *Effectiveness of Automatic Test-Case Generation*. PEASOUP is designed to be robust if individual modules behave poorly. Effective test-case generation allows PEASOUP to be more aggressive in the protections it applies. We sought metrics to help us measure our progress in the development of the test-case generation module.

- *Effectiveness of Individual Protections.* In some cases, defensive techniques used in PEASOUP provided defenses beyond what are required for the program, and therefore beyond what was evaluated during the independent evaluation. We sought to develop more detailed metrics in these cases. Often, these more detailed metrics provide the best possible evaluation for IR recovery: a recovered IR is only valuable in so far as it can be used in defensive transformations. Measuring recovered IR against ground-truth IR lacks this context and can be misleading for this reason.

In the remainder of this section, we provide some details on our approach to evaluation of the PEASOUP technology. More details on evaluation techniques are provided in individual sections describing those technologies.

3.1.1 Preliminary Phase 1 Test and Evaluation (December 2011)

All of the STONESOUP projects were subjected to an independent test and evaluation of their Phase 1 prototypes. The Mitre Corporation developed a suite of tests for each of the weakness classes that were defended by STONESOUP technologies. In the case of PEASOUP, this included tests suites for integer-handling weaknesses and for memory-handling weaknesses. In the remainder of this section, we provide details of how the independent test and evaluation of PEASOUP was conducted. Section 4.1 describes our understanding of the preliminary results of the Phase 1 test and evaluation, although we have not been privy to the final results.

3.1.1.1 Attendees

The evaluation team consisted of two Mitre employees. The PEASOUP team included two representatives from the GrammaTech team, three from the Raytheon team, and one from the University of Virginia team.

3.1.1.2 Setup

Raytheon hosted the PEASOUP Test & Evaluation at their Arlington, VA facility. Prior to Mitre arriving, Raytheon set up all of the computers in the large conference room and performed a complete regression test. Raytheon configured the test network with no single points of failure (backup DHCP server, Network switch, XenCenter computer) and had all equipment on Uninterruptable Power Supplies.

Raytheon developed scripts to control the Xen Virtual Machines that were used in the T&E. One set of tools performed VM snapshots, resets, shutdowns and reboots. Another set of scripts allow us to deploy new versions of PEASOUP and perform configuration changes on all VMs concurrently. These tools were all used at the T&E and all worked as designed.

Raytheon also developed scripts to configure a computer to run PEASOUP and to build an installable PEASOUP tar file. This allows for easy installation of PEASOUP on all Virtual Machines.

On the first day of the T&E, Mitre set up their servers (laptops) and performed several dry run tests. We also engineered and implemented the reset mechanism. The actual tests were conducted on December 14-15, 2011. December 16th was reserved for contingency and was not used.

3.1.1.3 Testing Parameters

There were 4 test “manifests” (collections of test cases). The four manifests were:

1. Number Handling – Engineered (nh-eng) with 10 tests cases.
2. Buffer Overflow – Engineered (bo-eng) with 209 tests cases.
3. Number handling – Real World (nh-rw) with 3 tests cases.
4. Buffer Overflow – Real World (bo-rw) with 2 tests cases.

However, we quickly noted that the nh-rw contained all the tests programs with the same test inputs that bo-rw contained. Once noting this, we eliminated bo-rw from our testing to speed up the testing process.

There were two configurations of PEASOUP tested:

1. Standard - all PEASOUP protections
2. Partial-C1 - PEASOUP with some (but not all) integer-handling checks disabled.

We added the second configuration after determining that some of the integer-handling checks were overly protective and breaking correct functionality under the standard configuration. For completeness, we ran all tests under both configurations.

3.1.1.4 Test and Test Input Descriptions

Most tests had "good" (non-malicious) and "bad" (malicious) inputs. Below is a description of the inputs used:

1. We did not have time to examine the inputs for nh-eng and bo-eng in detail.
2. Nh-rw 3 programs
    a. Testcase ngircd – Ngircd, an IRC server, had 2 inputs, one good and one bad. The good input (sent from a perl-based client) allowed a user to log into the IRC server and send 3 messages. The bad input was designed to exploit a flaw in the program which allowed a user to login with an invalid user name.
    b. Testcase bzip2 – There were several good inputs that asked bzip to compress and decompress files. One bad compressed input caused bzip to issue a segmentation fault.
    c. Tinyproxy – We were asked to protect a tinyproxy server. The server was configured to accept client connections, and forward them to a remote server. One (allegedly) good input was provided that downloaded an html file from the remote server, and forwarded that file to the client. (According to PEASOUP, the provided input to tinyproxy triggered a double-free error. We believe PEASOUP was probably correct in this assessment, in which case, the input is arguably not benign.)
3. Bo-rw had2 programs, ngircd and bzip2 using the same inputs as for nh-rw.

3.1.1.5 Process Notes

For the most part, T&E went very smoothly. Of course, there were a few hiccups.

3.1.1.5.1 **What Worked Well**

- Raytheon’s configuration of XenController and the large number of VMs were very useful in being able to parallelize our testing efforts. In particular, the scripts to reset, revert, and copy files to/from multiple VMs were invaluable.
- Raytheon’s configuration of isolated network, UPSes, etc. worked flawlessly.
- Mitre, GrammaTech, Raytheon, and UVa worked well together to diagnose and mitigate issues.
- In an unplanned, but spectacular, coincidence, we had 2 machines connected to the closed network. These machines allowed us to diagnose issues and log results without interfering with running tests.
- Mitre allowed us to run two PEASOUP configurations since we had time.
- Even on-site examination of the test results was very informative.
- Mitre’s flexibility in allowing us to run tests multiple times, with multiple configurations, and manual intervention when analysis or execution was hung was greatly appreciated and necessary to generate valid and meaningful T&E results.
- The majority of the protections (HeapRand,P1/Pn, ILR, PC Confinement, and function call monitoring) did not seem to produce many/any false positive messages. The one exception was the checks on individual integer operations.

3.1.1.5.2 **Hiccups**

- PEASOUP’s logging mechanism attempted to log the command that was executed. Unfortunately, at least in some instances the logging mechanism attempted to interpret the message to be logged. This resulted in error messages in stderr for the execute script. Mitre will need to adjust for this in their scoring. Such messages can be identify as being reported from “ps_tne_log.py.” (We can provide more details on this issue, if useful.)
- PEASOUP introduced significant startup delays, sometimes as much as 5 minutes. These delays are an artifact of a poorly coded mechanism that could be addressed in a few days. We considered attempting to address this issue during T&E, but decided the risk was not worth the reward. This issue (introduced in the last 2 weeks of PEASOUP development) completely invalidates all timing results from T&E.
- Several test cases had race conditions between the starting of the server and a client’s attempt to connect. This issue was significantly exacerbated by PEASOUP’s long startup delay. In particular, tinyproxy and ngIRCd test cases needed to have an artificial delay added so that we could be assured that the server was ready to accept connections before the client attempted its connect.
- VM resets generally were quite efficient and resulted in few problems. However, in at least one case (ngIRCd) we noted that a previous test case execution resulted in a process being left running, which invalidated the results from subsequent runs. Full resets between each execute step would have helped.

- Mitre recorded the start and end time of the “execute” phase on the Test Harness machines. Our test harness had to parse the command, and performed significant logging. While not excessively time consuming, accurate overhead measurement requires that we not time this logging, etc. For example, our logging invoked python at least 10 times for each executable. This clearly can dominate the time for short-running executables.
- It took us some time to diagnose a performance issue in the test manager that limited the number of test harnesses we could run in parallel. Ultimately, the test manager was able to handle up to 21 test harnesses running simultaneously, although this was still short of the 40 test harnesses we ran simultaneously with the dry-run release of the test manager.
- Stage 1 (test validation) was integrated with stage 2 (PEASOUP evaluation). In some instances (mostly the real world tests), this caused significant problems. If stage 1 fails, we need to triage the issue immediately and not wait until stage 2 (which can be lengthy) finishes. This issue resulted in tinyproxy not working until the very last moment.
- The test cases for the client/server applications were particularly challenging to get working correctly. A dry-run example that uses the client/server model would be useful.

3.1.2 Final Phase 1 Test and Evaluation (April, 2012)

The results yielded by test and evaluation conducted in December 2011 (which we referred to above as preliminary test and evaluation) were deemed inconclusive by the project managers and a decision was made to conduct a second round of test and evaluation in April, at the end of a three month extension to the Phase 1.

In March 2012, we received the results of the December T&E and the programs that were used to test PEASOUP during the December T&E. Unfortunately, the inputs for the December tests were incomplete and not clearly marked. Nevertheless, we treated any report of “altered functionality” or “not rendered unexploitable” in the December report as a report of a critical bug in PEASOUP. Given the inputs and programs we were given, we did our best to reproduce and fix the reported failures in PEASOUP. We set up regression tests based on the test programs and inputs from the December T&E. The regression tests were very useful for identifying bugs and shortcomings in PEASOUP and testing fixes.

3.1.2.1 Attendees

The evaluation team consisted of two Mitre employees. The PEASOUP team included two representatives from the GrammaTech team, three from the Raytheon team, and one from the University of Virginia team.

3.1.2.2 Setup

In preparation to the T&E, Raytheon accepted two early deliveries of the Mitre Test Manager and Test Harness. These tools were downloaded, installed and tested, with feedback and problem reports sent back to Mitre.

Additionally, Raytheon set up a farm of 40 Virtual Machines (VMs) for conducting tests and implemented scripts for verifying that every machine was configured identically and for correcting any differences. They also developed tools to control and monitor the VMs during testing and tools to archive the T&E tests, inputs and results.

Raytheon hosted the Phase 1 Extension Test & Evaluation Activity at our Arlington, VA facility, on April 18-20, 2012. The T&E was held in one of our smaller conference rooms, but still with enough power and A/C for the necessary equipment. Equipment setup and checkout was conducted on April 16, with the final version of PEASOUP being installed on April 17. The actual T&E went fairly smoothly. The only technical problem we had was that the reset manager based on Mitre’s ruby script did not scale up to 40 VMs – we ended up writing a replacement in Java that worked well.

3.1.2.3 Testing Parameters

The testing parameters for April T&E where similar to those for the December T&E described in Section 3.1.1.3.

3.1.2.4 Tests and Test Inputs

Mitre team significantly extended the set of engineered tests used in the December T&E as well as the set of good and bad inputs for each test.

Process Notes

The April T&E went much smoother than the December T&E. In particular, we were able to leave T&E with a copy of all of the tests, the ability to rerun the tests for ourselves, the automated score that was computed for PEASOUP, and the ability to rescore modified versions of PEASOUP. This was immensely valuable. As is to be expected with a large set of tests that attempts to be comprehensive, there were some issues with some of the tests. The following table shows the changes that we recommended in the automatic scoring of the tests.

Category	Test Case ID	Test Case Name	CWE	Original Result	Recommended Modification	Rationale
BO	TC_2667	TC_C_785_base1_linux	785	Still Exploitable	Rendered Unexploitable	The test system misclassified these in two ways: first, they do not detect the case where the (integer) error is corrected and the path name is not truncated; second, the bad outputs can be left over from a previous run (e.g., during stage 1), so a controlled exit was not detected as rendering unexploitable. The first issue is what arose during the early runs. The later issue arose on the later runs.
BO	TC_2668	TC_C_785_base2_linux				
BO	TC_2657	TC_C_785_v937_linux				
BO	TC_2658	TC_C_785_v961_linux				
BO	TC_2659	TC_C_785_v981_linux				
BO	TC_2660	TC_C_785_v988_linux				
BO	TC_2665	TC_C_785_v1003_linux	785	Still Exploitable	Mark input 1189 as good	None of the 785 tests actually contain a 785
BO	TC_2664	TC_C_785_v1023_linux				

BO	TC_2663	TC_C_785_v1061_linux				vulnerability where the buffer is too small to hold a path name. Instead, they arbitrarily truncate the path name inside of a buffer of sufficient size. Usually, the truncation is driven by a number - handling error. The bad inputs for these tests do not lead to a number handling error.
BO	TC_2662	TC_C_785_v897_linux				
BO	TC_2666	TC_C_785_v945_linux				

BO	TC_2205	TC_C_126_1043_linux	126	Still Exploitable	Mark input 1067 as good	CWE 126 is reading past the end of a buffer. These test cases do not read past the end of a buffer. They may read past the end of the data that has been initialized in a buffer, but not past the end of the buffer.
BO	TC_2206	TC_C_126_1076_linux				
BO	TC_2207	TC_C_126_1088_linux				
BO	TC_2208	TC_C_126_1110_linux				
BO	TC_2210	TC_C_126_1118_linux				
BO	TC_2211	TC_C_126_1127_linux				

BO	TC_2425	TC_C_170_v1025_linux	170	Still Exploitable	Mark inputs 1542 and 1548 as good	The bad input does not actually cause the null terminator to be overwritten. There is no occurrence of a 170 vulnerability (or any vulnerability, that we can see), on these inputs.
BO	TC_2434	TC_C_170_v1040_linux				
BO	TC_2426	TC_C_170_v1059_linux				
BO	TC_2427	TC_C_170_v1101_linux				
BO	TC_2428	TC_C_170_v896_linux				
BO	TC_2429	TC_C_170_v915_linux				
BO	TC_2430	TC_C_170_v958_linux				
BO	TC_2431	TC_C_170_v970_linux				
BO	TC_2432	TC_C_170_v985_linux				
BO	TC_2433	TC_C_170_v995_linux				

BO	TC_2424	TC_C_170_base	170	Still Exploitable	Mark test as invalid	All have inputs 1602 and 1604 as bad inputs. These bad inputs cause a 20-character string buffer to be filled without putting in a null terminator before the end of the buffer. PEASOUP's preferred approach to fixing this kind of error is to grow the buffer (i.e., by inserting padding at the end of the buffer) and making sure that a null terminator occurs in the padding. In
BO	TC_2423	TC_C_170_v905				
BO	TC_2419	TC_C_170_980				
BO	TC_2420	TC_C_170_v983				
BO	TC_2421	TC_C_170_v986				

						fact, we believe we did this for some subset of the above tests (at least one, maybe all of them). This allows the 20 characters that were read into the buffer to be read out, but does not allow any sensitive data to escape. The issue is that our fix is indistinguishable from what the unaltered program does, because there is already a buffer (of padding) immediately following vulnerable buffer, and it already has a null terminator placed at its beginning. This was presumably done to limit the amount of extra data that was dumped when reading out from the buffer. The net effect is that only a single byte beyond the end of the buffer is read --- the null terminator at the beginning of the next buffer. The exploit apparently is the reading of this null terminator (which could be sensitive in some scenarios, although they are a bit specialized). Again, it is indistinguishable whether the null terminator inserted by our fix (which would not be a "sensitive program value") was read, or the original null terminator was read.
BO	TC_2591	TC_C_416_v916_linux	416	Still Exploitable	Mark bad input as good	This program does not actually have a use-after-free vulnerability. The call to free has been commented out.
BO	TC_2580	TC_C_416_v1030_linux	416	Still	Rendered	We fixed these programs

BO	TC_2597	TC_C_416_v951_linux		Exploitable	Unexploitable	by delaying the effects of the call to free. The testing system is unable to distinguish a repaired program from one that is still vulnerable.
BO	TC_2616	TC_C_761_1011_linux	761	Altered Fn.	Mark input 1393 as bad	The io-pair 1393 triggers the weakness (free of a pointer not at the start of the buffer), so PEASOUP was correct to alter functionality by issuing a controlled exit. In fact, io-pair 1393 looks almost identical to (correctly labeled, bad) io-pair 1960, except that 1960 seems to have an extra environment input (that is not read by the test program?). We also issue a controlled exit for 1960.
BO	TC_2619	TC_C_761_v1094_linux				
BO	TC_2611	TC_C_761_v927_linux				
BO	TC_2614	TC_C_761_v982_linux				
RW	TC_1006	ngircd-81_linux	191	Still Exploitable	Invalid Test, or Passing	We have been using ngircd for regression ever since the first T&E. I am confident we are not altering the behavior (in any disallowed way), and we should be passing it --- as we did during the first T&E. The test system is showing us as 'altering behavior' due to a race condition in the test script between ngircd reaching the point it is accepting connections and launching the coprocess that attempts a connection. Technically, we changed the timing characteristics of ngircd by incurring a 5-10 second startup delay. This is the only way in which we "altered behavior" of ngircd, and it is allowed, at least

						implicitly, under the solicitation and the ROE. Note: the same (or similar) race condition had to be fixed during the first T&E. We did not have time to complete a fix, this time (our first try failed).

NH	TC_3002	TC_C_191_v501	191	Altered Fn.	Mark input 1584 as bad	This program is a variant of recaman. We diagnosed the issue and sent a detailed description on 4/19/2012. In brief, the issue is as follows: input 1584 (labeled good) causes exactly the same integer error as input 1582 (labeled bad) and PEASOUP handled both of them in exactly the same way. However, input 1584 happens to not cause an infinite loop, which is what the test is looking for. This is because the overflow calculation happens to result in the value 0. Note, it appears that the particular overflow operation is not the one marked as the trigger in comments.

Additionally, we discovered an issue that affects many of the tests for CWE 127. CWE 127 is a buffer under-read. Most of the tests for CWE 127 are based on a variant of Tiny FTPD. On a bad input (request for a missing file), the program under-reads a buffer and sends a string of A's from the previous buffer (a stand-in for sensitive information). We believe that we are successfully inserting padding in front of the under-read buffer, so that instead of sending the sensitive string, the under-read results in sending a string of zeros (0x0). I.e., the exploit, READ APPLICATION DATA, is prevented.

However, the io-pair 996 says to check the contents of the file d_errdata.bin. The client seems to write a long string of the digit '0' (0x30) in d_errdata.bin no matter *what* is sent from the test program. Thus, we're getting "still exploitable" despite the fact that the exploit was prevented.

We expect that for most of these, we would not have passed during T&E — but that we do now, with the improved analysis for identifying buffer boundaries. Even so, it seems that these tests are invalid. We expect this affects all of the 127 tests that use io-pair 996:

TC_2220	TC_C_127_base1_linux
TC_2221	TC_C_127_v1001_linux
TC_2222	TC_C_127_v1031_linux
TC_2223	TC_C_127_v1035_linux
TC_2224	TC_C_127_v1041_linux
TC_2225	TC_C_127_v1042_linux
TC_2231	TC_C_127_v1075_linux
TC_2226	TC_C_127_v1080_linux
TC_2234	TC_C_127_v1089_linux
TC_2232	TC_C_127_v1128_linux
TC_2227	TC_C_127_v884_linux
TC_2228	TC_C_127_v892_linux
TC_2233	TC_C_127_v941_linux
TC_2229	TC_C_127_v996_linux

The remaining 127 tests are valid, as far as we know.

We also found issues with many of the tests for CWE 126. In this case, there were many different issues, as follows:

- TC_2212/TC_C_126_v921_linux: the buffer allocated is big enough, therefore input 1067 should be marked as good
- TC_2204/TC_C_126_v1012_linux: test comparison relies on the overflowed area being filled with 0s. However, when PEASOUP breaks apart the buffer, the likely values in the padded area are also 0s. The test cannot distinguish between working and non-working defense and should be invalid.
- TC_2209/TC_C_126_v1112_linux: the test is invalid. The client program only seems to take into account the first 6 bytes (corresponding to the string "Begin." and writes out all 41s (A's) to the output file. Even when we change the source program so that the buffer is NOT overflowed, the client writes out the same output file. Furthermore, we peasoupified the program, verified that we changed the content of the targeted buffer, and yet, the client still writes out the same output file.
- TC_C_2212/TC_C_126_v921_linux: same problem as v1112 (we did not run the test, only looked at test case).
- TC_C_2213/TC_C_126_v946_linux: same problem as v1112 (we did not run the test, only looked at test case).
- TC_C_2216/TC_C_126_v984_linux: same problem as v1112 (we did not run the test, only looked at test case).

- TC_C_2215/TC_C_126_v975_linux: same problem as v1112 (we did not run the test, only looked at test case).
- TC_C_2214/TC_C_126_v956_linux: the test has no overflows. The value passed in by the client is used to set the variable returnBufSize. However, sensitiveBuffSize is the variable used but it is set to 516 (i.e., no overflows). Input 1067 should be marked as good for this test case.
- TC_C_2217/TC_C_126_v997_linux: the test has no overflows. However, a comment in the test source says “if file request is smaller than block size, then data returned could be sensitive data.” However, CWE 126 is a buffer overread, which is not the case here. Also, the test program does not put any "sensitive data" in the residual part of the buffer, and therefore, no "sensitive data" can be leaked for this example. The test should be invalid.

3.1.3 Phase 2 Test and Evaluation

All of the STONESOUP projects were subjected to an independent test and evaluation of their Phase 2 prototypes. The Mitre Corporation developed a suite of tests for each of the weakness classes that were defended by STONESOUP technologies. In the case of PEASOUP, this included tests suites for integer-handling, memory-handling, command-injection, and null-pointer dereference weaknesses. In the remainder of this section, we provide details of how the independent test and evaluation of PEASOUP was conducted. Section 4.2 describes our understanding of the results of the Phase 2 test and evaluation.

The Phase 2 independent Test and Evaluation (T&E) was run significantly differently from the Phase 1 T&E. The tests were all run on a private cloud at Mitre’s facilities in Bedford, MA. GrammaTech and Raytheon personnel installed PEASOUP on a VM during a visit to Bedford from July 1–3, 2013. While three days were allocated for the installation of performer technology, we only required half a day to successfully install PEASOUP.

Our quick installation allowed us to test PEASOUP against of the phase 2 programs and test the entire testing system (called STEW) when our VM was cloned onto multiple instances. This allowed us to discover some anticipated issues with the ways the tests were built that were causing PEASOUP to fail. For example, the tests for SQL injection would load the SQL library using *dlopen* and call the SQL routines using *dlsym*. To our knowledge, no real-world program uses an SQL library this way. We were able to adjust PEASOUP to handle this usage pattern.

After we completed the installation of PEASOUP and debugging of some issues, our team left Mitre. Mitre ran the T&E after our team left.

3.1.4 Phase 3 Test and Evaluation

The Phase 3 T&E was led by TASC, Inc. In contrast to previous T&E, all of the tests were run using Amazon’s S3 cloud services. PEASOUP struggled during the Phase 3 T&E and we had limited resources to evaluate the results. Some of our limited results are described in Section 4.3.

3.1.5 Component Test and Evaluation

The independent test and evaluation focused on the ability of PEASOUP to defend against weaknesses. Many of the PEASOUP components are interesting in their own right. In this section, we briefly describe the measurements that we gathered for individual components of PEASOUP. We divide our metrics into three broad categories: performance of automatic test-case generation, fidelity of recovered IR, and effectiveness of individual transformations performed by PEASOUP. In many cases, the effectiveness of a given transformation provides an additional measurement on the fidelity of the recovered IR.

3.1.5.1 Test Case Generation Metrics

As described above, PEASOUP uses automatically generated test cases for IR recovery and for vetting of candidate programs that are diversified and have vulnerabilities patched. The quality of the generated test suites impacts the effectiveness of many of PEASOUP's defensive transformations. A test suite of poor quality will mean that fewer defensive transformations can be applied, and the overall level of protection will be lower.

In general, higher quality test suites achieve higher *coverage* of the test program. Here, coverage can be defined in many different ways, with two popular measures being *instruction coverage* and *branch coverage*. Instruction coverage refers to the percentage of a program's instructions that are executed at least once when the program is run on all of the tests in the suite. Similarly, branch coverage refers to the percentage of possible outcomes (e.g., true or false) that are observed at program branch points when the program is run on the test suite.

It is unclear what coverage metric is most important in PEASOUP. For example, in the case of Stack-Layout Randomization (SLR), the most useful coverage metric may measure just the percentage of stack-accessing instructions that are executed. Only stack-accessing instructions can interact with SLR. We focused primarily on measuring instruction coverage of our generated test suites, as higher instruction coverage often implies high coverage of other program aspects.

One significant drawback of measuring test-suite quality using coverage metrics is that (a) it is impossible to achieve 100% coverage for most programs and (b) it is (theoretically) impossible to determine the maximum achievable coverage in an automated way. This holds no matter what type of coverage metric is used.

Achieving 100% coverage is often not possible because most programs contain *dead code*, or instructions that can never be executed, no matter what input is provided to the program. There are many different reasons that dead code may be present in a program. Often, the linker will pull in library code that is not (and cannot be) used by the subject program. For example, consider a classic "hello world" program:

```
int main() {
    printf("Hello world!\n"); return 0;
}
```

When this program is built (using static linking), the linker must pull in the code for the printf routine from the standard library. Traditionally, linkers pull in entire object files at once. Thus, even though the program only uses printf, the linker will add all of the functions in the same object file as printf to the final executable. Furthermore, for this program, most of the code in the printf function is dead. The printf function is one of the most complex in the standard library, and can understand very complicated format strings. The hello-world uses none of this functionality, but the relevant code is included, nevertheless.

In our experiments, we discovered a program where roughly 75% of the code appears to be dead (unable to be executed on any input). For this program, 25% is the maximum achievable (instruction) coverage. For other programs, the percentage of dead code is obviously much lower. Unfortunately, determining the percentage of dead code is an undecidable problem.

The dead-code problem makes it difficult to interpret the meaning of coverage metrics achieved by our generated test suites. Nevertheless, we used instruction coverage to determine if the quality of our generated test suites was improving (higher coverage was achieved) as we made improvements to the Grace concolic engine. In Phase II, we would like to find ways to approximate the maximum achievable coverage, e.g., by manually generating high-coverage test suites for some programs. This would allow us to appropriately scale the measurements achieved by Grace on those programs.

3.1.5.2 Ground-Truth IR Measurement Tools

During this reporting period we completed the initial ground-truth IR comparison tool (gtir_compare) for comparing with ground-truth IR from DVT and DWARF. The tool may need to be extended as new types of IR become important. The tool compares the following aspects of the IR:

1. Procedure entry points (DVT and DWARF);
2. Procedure boundaries (DWARF);
3. Instruction boundaries and sizes (DVT);
4. Data boundaries and sizes (DVT and DWARF);
5. Stack variable boundaries and sizes (DWARF);
6. Symbolic references to statics (DVT).

We also fixed some problems with erroneous ground-truth information from DVT related to string literals. This is described in Section 3.1.5.2.4.

Finally, we applied this tool on the coreutils suite of programs to compare ground-truth IR against the IR computed by CodeSurfer/SWYX. The results are described later in Section 4.6.

3.1.5.2.1 **Comparing Procedure Boundaries**

DWARF includes information about procedure boundaries; that is, for each procedure it gives ranges of effective addresses that comprise the procedure. We added a facility to extract this information from the DWARF format, store it in our ground-truth IR database, and then to compare it against the address ranges covered by the same procedure in the CodeSurfer/SWYX-computed IR.

3.1.5.2.2 **Comparing Symbolic References to Statics**

In order to perform static rewriting of binaries, it is important that all symbolic references are captured by the IR. This is probably one of the most challenging problems to address when analyzing stripped binaries, as while it is difficult to distinguish addresses from numbers, it is even more difficult to determine which object the address is referencing.

We distinguish symbolic references into static, stack, and heap. References to stack generally occur as an offset from a register, usually ebp or esp, while references to heap generally occur as an offset from a register that has been assigned the return value from a call to malloc. References to statics, on the other hand, occur simply as raw numbers, and are in general indistinguishable from “true numbers”. Note that our use of the term “statics” here refers to objects in the binary image’s data — in the .data, .text, or .bss section — and encompasses the notions of globals, file statics, and function statics in C source code.

While it is important to accurately recover stack references, it is currently excluded from gtir_compare because such ground truth information is not available either via DVT nor DWARF. Ground truth heap references are also unavailable, though it is less important for our needs.

DVT recovers symbolic references to statics by parsing the assembly listing generated by cc1. Symbolic operands that look like “foo” or “bar + 4” are parsed and recorded as a pair <symbol_name, offset>.[1] Note that the “symbol_name” component is redundant if we also maintain a symbol table: from the binary we can recover the operand’s actual value, so given the equation operand_value = symbol_value + offset, we only need to know the offset to precisely capture the nature of the symbolic operand. However, we use symbol_name for sanity checking, which turns out to be important in the face of a link-time string-literal optimization; this is discussed later.

3.1.5.2.3 **DWARF Static Variable Sizes**

We also added code to extract sizes of static variables for DWARF; this was a small addition to existing code that extracted such information for stack variables only. The sizes are computed by traversing DWARF’s type information.

3.1.5.2.4 **DVT String Literals**

DVT obtains information about code and data objects by parsing the assembly file generated by the compiler, but this information is laid out relative to sections in the object files. For DVT to accurately determine the layout of static code and data in the final executable, it must understand the linking process. It does so by doping the object file with symbols that tell it how each section of each object file is mapped in the final executable. This approach assumes that the linker will keep entire sections intact, i.e., that it will not tear apart sections or merge them.

Unfortunately, the current version of gcc does not satisfy this assumption. In particular, the linker performs an optimization whereby string literals from different compilation units are shared. Without changing the behavior of DVT, this effectively means that the DVT IR will include certain string data objects that are just plain incorrect.

We address this by cross-checking at the names of symbols recorded at data references against the names of data objects. Consider an example compilation unit foo.o which contains a string literal labeled S1 at address 123 off of its data section, and an instruction “move eax, S1” at address 456 off of its text section. Suppose the linker appears to place foo.o at address 10000 within the final executable. DVT will have recorded a data item named S1 associated with address 10123, and a symbolic reference for the instruction at 10456 to “S1 + 0”. However, when the linker optimization kicks in, the string literal may end up at a totally different address 10789. When we examine the final executable, the instruction at 10456 will have a numeric operand 10789 which should evaluate to the same value as the symbolic expression S1 + 0.

[1] DVT also parses a second kind of symbolic operand, of the form “foo – bar”, but we exclude these from the current ground-truth IR comparison.

However, we will find that this conflicts with the data record stating that S1 is at 10123. When we detect such an inconsistency, we delete the data record for S1 as spurious, and attempt to place it at 10789.

Note that gcc's string literal optimization shares not only string literals that are identical to each other, but also common-suffix substrings. For example, if a program has string literals S1 = "foobar" and S2 = "bar", S2 could be mapped to S1+3. We account for such cases in our adjustment phase; in so doing, we find that the program contains many symbolic references with non-zero offsets. These are particularly important for a good IR recovery algorithm to get right, but also challenging to do so.

3.1.5.3 Diversification Components

Automatic program diversification lowers the probability of a successful exploit, but it does not completely remove the threat. We evaluated many of our diversification techniques based on how effectively they were able to introduce entropy, and hence how effective they are likely to be at stopping an exploit.

3.2 Platform and Environment Assumptions

We believe that the techniques used in PEASOUP are largely platform independent, and would be equally effective if applied on multiple platforms. However, resources prevented us from testing this hypothesis empirically. PEASOUP currently protects applications for 32- and 64-bit Ubuntu 12.04 LTS. We have also done some preliminary testing of PEASOUP on CentOS and REHL. The results of those tests were positive and we do not anticipate

3.3 Core Technologies

PEASOUP assembles many different components to provide analysis and protection. As described previously, at the top level, PEASOUP consists of three components:

- The *Intermediate Representation Database (IRDB)* is the central repository for all information that PEASOUP learns about a subject program. In addition to containing facts about the structure the subject program and the locations of vulnerabilities in the program, it contains all the necessary information to protect the subject program during execution.
- The *analyzer* is run offline, prior to deployment of the SOUP. It populates the IRDB. The analyzer uses automatic test case generation, static and dynamic analysis techniques, runtime fault detectors, and empirical evaluation to learn how to best protect the subject program.
- The *execution monitor* is responsible for preventing exploits of the vulnerabilities in the SOUP during runtime. It makes use of the information in the IRDB to alter the instruction stream at runtime and prevent exploits.

This section provides details on how the individual components of PEASOUP are implemented:

- Section 3.3.1 provides details on the design and implementation of the IRDB.
- Section 3.3.2 describes Grace, the concolic engine used in PEASOUP for automatic input generation.

- Section 3.3.3 discusses how PEASOUP replays the inputs generated by Grace in other stages of the analysis.
- Section 3.3.4 describes the static analyzer used by PEASOUP to recover information about the structure of the subject program; information recovered by the static analyzer is later vetted using dynamic analysis.
- Section 3.3.5 presents our work on recovering the data layout used by a program, specifically, the data layout of stack activation records.
- Section 3.3.6 provides an overview of PEASOUP's approach to dynamic program rewriting.
- Section 3.3.7 describes our investigation into efficient checkpointing techniques that we considered using for PEASOUP.

3.3.1 Intermediate Representation Database (IRDB)

Conceptually, the IRDB sits between the analyzer and the execution monitor. In fact, the execution monitor does not talk directly to the IRDB, because it is difficult to ensure a reliable communication mechanism from the SOUP. Instead, the data needed by the execution monitor is dumped into a SPRI file that can easily be read by Strata as it executed the SOUP (see Section 3.3.5).

We have developed and implemented a database schema in Postgres for managing and coordinating the information flow between various components of the PEASOUP toolchain. Our initial schema keeps track of programs, potential program variants, program library dependencies, instructions, functions and addresses. One key feature of our schema is that every program is tagged with a *doip id* that identifies its originating analysis. This is helpful for establishing the confidence of individual IR facts. Ultimately, all IR facts can be validated using BED, but using heuristics based on doip id are important for improving performance.

Below, we give pseudo SQL for generating the tables. The last set of tables is not quite SQL syntax. Those tables are per variant, and we substitute #PROGNAME# with the proper variant name.

```
CREATE TABLE doip
(
  doip_id         SERIAL PRIMARY KEY,
  confidence      integer,
  tool_name       text,
  comment         text
);

CREATE TABLE variant_info
(
  schema_version_id             integer DEFAULT 1,
  variant_id                    SERIAL PRIMARY KEY,
  name                          text    NOT NULL CHECK (name <> ''),
  orig_variant_id               integer DEFAULT -1,
  address_table_name            text,
  function_table_name           text,
  instruction_table_name        text,
  doip_id                       integer DEFAULT -1
);
```

```
CREATE TABLE file_info
(
  file_id          SERIAL PRIMARY KEY,
  url              text NOT NULL CHECK (url <> ''),
  hash             text,
  arch             text,
  type             text DEFAULT 'ELF-Static',
  elfoid           OID,
  doip_id          integer DEFAULT -1
);

CREATE TABLE variant_dependency
(
  variant_id       integer REFERENCES variant_info,
  file_id          integer REFERENCES file_info,
  doip_id          integer DEFAULT -1
);

--------------------------------------------------------------------
This portion of the Schema defines the Peasoup Tables
that are per variant.
--------------------------------------------------------------------

CREATE TABLE #PROGNAME#_address
(
  address_id            SERIAL PRIMARY KEY,
  file_id               integer REFERENCES file_info,
  vaddress_offset       integer,
  doip_id               integer DEFAULT -1
);

CREATE TABLE #PROGNAME#_function
(
  function_id           SERIAL PRIMARY KEY,
  entry_point_id        integer
  name                  text,
  stack_frame_size      integer,
  out_args_region_size  integer,
  use_frame_pointer     boolean,
  doip_id               integer DEFAULT -1
);
```

```
CREATE TABLE #PROGNAME#_instruction
(
  instruction_id            SERIAL PRIMARY KEY,
  address_id                integer REFERENCES #PROGNAME#_address,
  parent_function_id        integer,
  orig_address_id           integer,
  fallthrough_address_id    integer DEFAULT -1,
  target_address_id         integer DEFAULT -1,
  data                      bytea,
  callback                  text,
  comment                   text,
  ind_target_address_id     integer DEFAULT -1,
  doip_id                   integer DEFAULT -1
);
```

3.3.2 Input Generation: The Grace Concolic Execution Engine

Grace is a tool for automated generation of test suites that achieve high coverage of program code. Grace operates directly on a program binary (i.e., it does not require the source code to be deployed) and generates a collection of program inputs that can be fed to a program with the use of a separate utility called *input replayer* (see section 3.3.3 for detailed description). Currently, Grace supports several types of inputs—command line arguments, basic file input, and network input. As we continue developing Grace, we will add support for other, more esoteric, types of inputs, such as environment variables, time queries, file permissions, etc.

Grace plays an important role in the offline (analysis) phase of PEASOUP. The inputs generated by Grace are used to:

- **Recover intermediate representation of a program:** PEASOUP uses run-time error detectors to monitor program behaviors induced by the test suite. The detectors are used to categorize the inputs as ‘good’ and ‘bad.’ Once the inputs are categorized, they can be used with dynamic analyzers and machine learning to recover facts (or IR) about the structure and semantics of the program.
- **Validate program transformations that make vulnerabilities unexploitable:** The input suites are also used to compare program behavior before and after defensive transformations are applied to ensure that the program has not been broken by the transformation. This is described in more detail in the section on BED, the Behavior Equivalence Detector. The primary purpose of BED is to validate the program IR, although it also serves as an extra check on the correctness of the program transformations.

Grace is based on concolic execution—a technique for white-box test generation that uses a combination of concrete execution and symbolic execution [49, 49, 58, 58, 96, 96]. Concolic execution starts by running a program on a seed input—either randomly generated or provided by the user. In addition to maintaining a regular (concrete) program state, concolic execution propagates a symbolic state, which expresses every value in a program symbolically as a function of program inputs. During execution the concolic execution collects a set of symbolic constraints that characterize inputs that cause program to execute the observed path. At every branch point in the path, concolic attempts to generate an input that will make execution “stray” from the observed path. To do that, it feeds the slightly-augmented set of collected constraints to an off-the-shelf decision procedure. If the constraints are unsatisfiable, then no such input exists. Otherwise, concolic execution uses the model produced by the decision procedure to derive such an input. The derived input is added into the work queue. After the program run is over, the next input is selected from the work queue and the process is repeated. Overall, concolic execution continues until either the work queue becomes empty indicating that all possible program paths have been explored (typically, this is only possible for small, synthetic benchmarks) or until the analysis resources run out (e.g., the time allocated for input generation expires).

There are several existing tools that are based on concolic execution: KLEE [48], SAGE [97], MACE [35, 58], and S2E [57], to name a few. Before committing to using Grace in this project we have evaluated these tools[2] and found them lacking in several important ways for our purposes. We formulated the following requirements for the PEASOUP test-generation tool:

- **Input replay.** The tool must be able to produce inputs that can be replayed in a setting that is as close to native execution as possible. This allows us to employ existing off-the-shelf run-time error detectors, such as for instance Valgrind MemCheck [32], to detect vulnerabilities in programs.
- **Vulnerability coverage.** As we described above, concolic execution targets the coverage of *explicit* conditional branches in the program. However, most program errors and vulnerabilities are *implicit* in the program—the analysis must perform additional checks and sometimes even propagate additional information in order to expose them. For instance, to expose a buffer overrun, the tool must propagate the lengths of the buffers that program manipulates and check that the offset used to access a buffer does not exceed its length. In majority of cases, the vulnerabilities are there because the explicit error checks are missing from the program. The test-generation tool must introduce these error conditions into a program and attempt to generate inputs that satisfy them.
- **Easy deployment.** All design decisions in PEASOUP are evaluated on the basis of the impact they will have on eventual deployment. Ease of deployment is not always determinative; achieving and surpassing program metrics is more important. In the case of Grace, deployment considerations led us to prefer *application-level virtualization* for managing concrete executions. Other tools use *system-level virtualization* [58, 128, 210] or *emulation* [48]. These approaches have been shown to be very effective, but we believe that application-level virtualization offers some advantages regarding deployment.
- **Operating system and library modeling.** Typical software makes heavy use of libraries and interacts with the operating system to perform I/O. Operating system code and library code is typically more arcane and is harder to reason about than the user code. Additionally, the execution of OS code is harder to monitor—it requires either modifications to the kernel or whole-system emulation which is expensive. At the same time, OS code and library code is typically more robust and less likely to have exploitable vulnerabilities. The tool as much as possible should avoid performing symbolic execution of system code and library code and use concise models of it instead.

[2] We evaluated the tools that were publically available. These included KLEE and S2E. SAGE is an internal Microsoft tool which we could not obtain.

- **No manual input seeding.** It has been observed that concolic execution achieves higher coverage faster if it is seeded with an input that takes it “deep” into the program. When started with a random seed, concolic execution spends extra time learning the correct file format before it is able to get past the input file parser. In contrast, when seeded with properly formatted file, concolic execution bypasses the parser immediately and starts generating inputs that exercise core functionality of a program. Some concolic execution tools, notably SAGE [95, 97], expect users to provide a set of seed inputs for the analysis (e.g., a small set of well-formed PDF files for the Acrobat reader or a small movie file for a media player). However, in the context of STONESOUP, the users of protected software are not obligated to find the meaningful seed files for the software they plan to use. Thus, a suitable input-generation tool should try to deduce automatically the file formats that software accepts as input and construct or obtain the suitable seed inputs to speed up the exploration.

We engineered Grace to address the above requirements. Grace relies on native execution rather than emulation for performing concrete program runs. To monitor and control concrete program execution Grace uses software dynamic translation (SDT) framework, Strata [181, 181] from University of Virginia, described in Section 2.5.1.2. Using native execution makes Grace easy to deploy—there is no need to adapt and maintain an emulator. Additionally, it makes Grace’s analysis more precise—the analyzed program runs directly on the target platform and Grace observes all of the execution nuances. Grace relies on library and system call models to avoid the exploration of shared library and OS code[3]. Also, Grace employs a number of heuristics for scheduling inputs and for automatically deriving meaningful seed inputs in order to boost the exploration of a program.

In the following, subsections we will describe several novel techniques that we investigated and prototyped in Grace during Phase 1. Section 3.3.2.1 extensions to concolic execution that make it more effective in finding vulnerabilities. Section 3.3.2.2 describes our techniques for modeling library calls. Finally, Section 3.3.2.3 describes heuristics that Grace uses to sort input work queue and to generate good seed inputs.

3.3.2.1 Vulnerability Coverage

As we mentioned before, in traditional concolic execution, input generation is triggered only for *explicit* conditional branches in the program. This allows concolic execution to achieve good program coverage; however, it does not provide a guarantee that generated tests will expose actual program errors and vulnerabilities. For instance, a generated input may exercise a path on which a buffer overrun is possible, but is not triggered, and so goes unnoticed. It is arguable that the majority of vulnerabilities in software are due to missing explicit error checks.

In order to detect errors an analysis must perform the corresponding error checks implicitly. Some error checks are trivial—for instance, to detect a division by zero error, the analysis just needs to check whether the divisor is zero prior to the division operation. Some are more complex and require additional information to be inferred and propagated—for instance, to detect a stack-smashing attack, the analysis needs to infer the stack layout.

[3] In this project, we consider common library code (as e.g., standard C library) and operating system code to be trusted.

In the context of concolic execution, the implicit error-checking conditions must be used in conjunction with the constraints obtained from explicit branches to generate inputs that trigger the corresponding errors. In Grace, we implemented a general mechanism for introducing such error-checking conditions to which we refer as *virtual asserts*. Virtual asserts are implicit conditions that are attached to sensitive operations, such as integer division and memory accesses. The virtual assert for a division operation checks that the value of a divisor is not zero, and the virtual assert for a memory access checks that the accessed address is not null, and that the access is within bounds. Additionally, we experimented with assigning virtual asserts that check for integer overflow to arithmetic operations. When Grace encounters a virtual assert, it treats it as an explicit program branch—that is, it checks how it evaluates in the concrete run, and attempts to generate an input that will make its concrete value flip.

An additional challenge with vulnerability coverage is that it may not always be enough to trigger a vulnerability to detect it. For instance, consider a stack-based buffer overrun that overwrites several memory words after the buffer, but does not do enough damage to alter the execution of a program (i.e., the return address on the stack is not smashed and no important variables are clobbered). A run-time analysis may dismiss such an overrun as being benign, even though a malicious exploit for it may be crafted. Ideally, we want our input-generation tool to produce inputs that cause the vulnerabilities to be detected unequivocally. E.g., the tool should try to produce an input that does not just override the buffer, but that overrides the entire frame and smashes the return address. We are in process of investigating the techniques for doing this in Grace. We envision a tight integration between run-time error detector and Grace—the detector will communicate the error conditions it checks to Grace to be used in input generation.

3.3.2.2 Library Modeling

Typically, software does not encompass all the functionality required to achieve its goals. It makes use of standard libraries and interacts with the operating system to perform certain types of operations—for instance, to read and write files, to send data over a network, and to perform commonly used data manipulations (e.g., to copy buffers and zero-terminated strings). Compared to the general third-party software, an operating system is typically more robust and less likely to contain vulnerabilities. Also, the user has more control over its selection and installation. Standard libraries often come along with the operating system distribution and are linked dynamically to the third-party software. In this light, there are multiple advantages to using concise models of standard library functions and system calls rather than analyzing them directly[4]:

- The concolic execution is much more effective because it does not have to re-explore library calls and system calls for each their invocation. Library code is typically large and more complex than the user's code so its analysis generally takes longer and tends to be less precise.
- The models are used to remap the physical input/output operations onto the logical structure of the input that the decision procedure is able to reason about. That is, for

[4] Note that this does not preclude the tool from detecting vulnerabilities that involve library code (e.g., a buffer overrun caused by `strcpy` function). The models for the involved library functions will capture their semantics in a sufficient detail for the tool to detect such vulnerabilities.

instance, the model for the read system call will link the physical bytes in a file to the symbolic array that is used to model the contents of the file in the logical encoding of the trace. Later, when reifying the input from the decision procedure model, Grace will remap the data in the symbolic array back to physical contents of the file.

- The models can capture the semantics of the corresponding calls at a higher level than the direct concolic execution the function yields. Concolic execution observes only a single path through a function where as the model can capture the effect of multiple paths simultaneously. This again allows a concolic-execution tool to avoid some expensive, but not particularly interesting analysis. We explain this in more detail below.

During Phase 1 we put a significant amount efforts focused on Grace into designing effective ways to model standard library functions. We found that it is important to be able to capture the semantics of a function at a sufficiently high level—i.e., make the model represent symbolically multiple possible execution paths through the program, especially for functions that involve loops. We will illustrate this with a simple function that computes the length of a zero-terminated string (in standard C library, the function is called strlen). Consider the following program:

```
int main (void)
{
    FILE * f;
    char buf[1024];
    …
    fgets(buf, 1024, f);
    …
    int len = strlen(buf);
    if (len == 512) {
        …
    }
}
```

The implementation of strlen consists of a sentinel-search loop that looks for a zero terminator. When the body of strlen is concolically executed, a tool will construct a sequence of input strings—a string of length zero, a string of length one, a string of length two, up until a string of length 512 is generated and the condition in the if statement is satisfied—taking the total of 512 iterations to trigger that branch. What's even worse, concolic execution will not stop after that. but will continue generating additional inputs that contain longer and longer input strings, up to the 1024 limit imposed by the fgets function. This large number of "uninteresting" inputs bogs down the exploration of the program and dilutes the generated test suite. In contrast, a good model for strlen will link symbolically the return value of the strlen function with the values stored in the buffer (i.e., it will capture that the byte in position len is zero, while the bytes that precede it in the buffer are non-zero). With such a model, concolic execution needs only two iterations to cover all the paths in the program.

One major challenge is how to encode the symbolic relationships of the kind illustrated above logically. Ultimately, we need the logic for expressing models to be decidable. Moreover, we would like the decision procedure query times to be very fast—preferably taking no more than a fraction of a second in majority of cases. This requirement excludes logical features like universal quantification and transitive closure that could be used to encode the semantics of such loops naturally.

The approach we have taken in Grace is to pick a bound up to which we model the behavior symbolically. The bound is selected heuristically for each invocation of a function—for instance, for the `strlen` we will pick the bound to be the larger of some predefined lower bound (128 in the current implementation) and twice the concrete length of the parameter string. Fixing the length up to which the model is symbolic allows us to enumerate the constraints for individual bytes in a single quantifier-free propositional formula, which can be decided efficiently. To ensure the completeness of exploration, Grace also attempts to produce an input where the input string exceeds the length of the buffer—on the concolic run induced by that input, the bound will have a larger value causing a larger portion of the string to be modeled symbolically.

The next challenge that we had to address is how to express the bounded semantics of a function. We found that there are two viable possibilities:

Compute outputs from inputs. In this encoding, the model constructs logical expressions that compute function outputs from its inputs. Going back to our `strlen` example, the return value can be computed as follows (let *`B`* denote the symbolic bound, and *`ite`* denote an “if then else” operation):

```
rv ← ite(buf[0] == 0, 0, ite(buf[1] == 0, 1, ite(…., ite(buf[B] == 0, B, B+1)…)))
```

Constrain outputs based on inputs. In this encoding, the model allocates a fresh logical variable for each output and imposes constraints on them so that they can only take proper values. Again, looking at the `strlen` example—rv is introduced as a fresh logical variable and the following constraints are placed on it. For each *`i`* from *`0`* up to *`B`*:

```
                (rv == i) ⇒ (buf[i] == 0)   ∧   (buf[i] == 0) ⇒ (rv ≤ i)
```

The first part of each constraint makes sure that if the length of the string is equal to the corresponding number, the byte in the buffer at that position is zero. The second portion ensures that there are no zeros in the buffer up to the length of a string.

In our experiments we found that the latter encoding is much more efficient in practice, at least with the decision procedure we employed in Grace (Yices 1.0.29). This is most likely due to the fact that it avoids the nesting that is inherent to former approach—instead, each byte is constrained independently.

3.3.2.3 Input Sorting and Seeding

Other important aspects of concolic execution that we have investigated in Phase 1 are:

- With what input to seed the concolic execution?
- In which order to concolically execute generated inputs?

Theoretically, concolic execution can kick off exploration with any input—it can be a randomly generated input or an “empty” input, which feeds no data to a program. However, it has been shown in practice that concolic execution performs much better when it is seeded with a well-formed input that makes program do something meaningful—such as a PDF file for a PDF reader or a movie file for media player. The reason for this is that the front end of a typical program performs some amount of input parsing and validation, and it takes time for concolic execution to learn the correct input formats so that it can generate inputs that successfully pass the validation phase and trigger some deeper, more interesting program behaviors.

Supplying the seed files for the analysis is not generally a big hassle for the end user—it is generally easy to find some well-formed input files for the majority of third-party software. However, in the context of STONESOUP (and, particularly, in the context of STONESOUP testing and evaluation), it was postulated that the tool must operate fully automatically with no help or hints from the user. Thus, in Grace, we implemented several heuristics for automating the process of finding good seed inputs.

Heuristic 1: Use existing files. When Grace concolically executes a program, the program does not see and cannot access directly the actual file system on the host. Grace intercepts all file-system accesses and redirects them to a sandbox. This allows Grace to create, and feed to a program, actual input files without worrying of overriding some important system information (e.g., `/etc/passwd` file). Originally, Grace started with an empty sandbox. If it learned that a program made an attempt to open a file, it created an empty one for the next concolic iteration (that is, Grace generated an input that includes an empty file with the corresponding name).

While this strategy generally works, it may take Grace a lot of time to learn the contents of the file that are well formed and trigger meaningful behaviors of a program. This is especially bad when a program reads a system configuration file, for instance `/etc/hosts`—a network configuration file, which we can assume to be trusted. In such case, instead of generating network inputs, Grace will spend majority of its time on generating meaningful configuration files—which is, generally, a bad idea because the contents of those files are, typically, dictated by some external considerations (e.g., the host's IP address and its DNS address) and, thus, generating a configuration file that will make system execute properly is next to impossible.

To alleviate this problem, we augmented Grace to check, whenever an analyzed program attempts to open a file, whether the corresponding file exists within a real file system. If it does, it is copied into the sandbox and fed to the program. We found that this strategy allows Grace to achieve greater program coverage faster.

Heuristic 2: Use the `file` utility to learn file formats. The file utility recognizes many different standard file formats by looks at a few "magic" bytes in a file—a format signature. For instance, a PDF file starts with characters "PDF-", followed by the version number of the format, and a bzip2-compressed file starts with "BZh". Incidentally, most programs that operate on files of specific format typically start file processing by checking format signature and discarding the file if it has wrong format. Thus, concolic execution will generate files that contain the right format signatures (but arbitrary data) fairly early in the analysis. In Grace, we experimented with the following heuristic: whenever a generated input contains a non-empty file, we will run that file through the `file` utility, to learn whether program expects a file of some known format. If the format is known—we can pick valid file of that type (e.g., obtain a file of that type on the internet) and use that file on a subsequent concolic run. We have not yet implemented this heuristic in its entirety, but our initial experience looks promising.

The second question that we posed above is in what order a concolic-execution tool should use inputs it generated. The easiest approach is to execute the inputs in "first-come, first-served" fashion. This is the default exploration strategy in Grace. However, this approach has significant drawbacks. Consider a program that does input validation: most of the inputs generated by concolic execution initially will fail some of the checks and will trigger some error-printing functionality in the program, but will not reach into the core of the program. There will be some inputs that have been generated based on the validation checks that penetrate deeper into the program, but—in a first-come, first-serve input ordering they will be swamped by “useless” inputs that fail validation right away. Thus, most concolic execution tools, Grace included, implement some prioritization scheme for inputs—inputs with higher priority are executed first.

There are many possible heuristics for prioritizing inputs. The one we experimented with was based on increase in coverage achieved by each input—the more instructions that were not previously seen are executed by a program, the higher the priority of the corresponding input. To obtain the instruction coverage, we extended the replayer tool we describe in Section 3.3.3. The main challenge posed by this approach is that incremental coverage is not a stable metric—it depends on the order in which inputs are executed. An execution of an input potentially changes the overall coverage and, thus, the priorities of other inputs that are queued up for execution. Ideally, on each iteration, the tool must scan the entire input queue and pick the input that gives the largest increase in coverage. However, this is very expensive. Thus, instead, we implement a less precise, but more efficient prioritization scheme in which we assign priorities based on increase in coverage with respect to the coverage obtained by previously generated inputs. Once the priority is assigned it does not change.

Our experiments have shown that the above strategy of ordering inputs makes program exploration significantly faster for some benchmarks. However, there is a price to pay—measuring the coverage for each generated input takes time. We are still in process of evaluating the involved trade-offs. We believe that the machinery for gathering coverage data can be made much more efficient, thereby tilting the balance in favor of sorting the work queue.

3.3.3 Input Replayer

The inputs generated by Grace (or other input generation tools) are only useful to PEASOUP if they can be fed into the subject program while the program is observed by runtime monitors and analysis tools. We call the process of running the subject program on a generated input, “replaying” the input and call tools for replaying inputs, “replayers.” Replaying of inputs is ubiquitous throughout PEASOUP:

1. Each generated input is replayed by Grace as it performs concolic execution of that input. Grace also replays generated inputs to determine the (relative) code coverage achieved by each input.
2. Each generated input is replayed with runtime security monitors (e.g., MEDS) in order to classify inputs as ‘good’ and ‘bad.’
3. Classified inputs are replayed with dynamic analysis tools during IR Recovery.
4. Inputs are replayed by BED in order to test behavioral equivalence between program variants and the (original) subject program.
5. Inputs may be replayed by TSET in order to test the effectiveness of the test suite, and establish the confidence of BED.

It is important that the machinery for feeding the inputs to the subject program does not interfere with any machinery needed for runtime monitoring during the replay. For example, the replayer should not interfere with gathering code coverage information, or measuring security vulnerabilities. It is also important that replaying the input does not inadvertently damage the host platform in any way. For example, during concolic execution of the utility ‘kill’, we found that concolic execution would generate inputs that caused replay to terminate useful (test) processes. We refer to the above properties as the *transparency* and *safety* of an input replayer. In the remainder of this section, we describe the input replayer we developed during Phase 1.

A single input generated by Grace may contain any number of command line arguments, files and network connections. These pieces of data are encapsulated within a JSON-formatted file that the input replayer can feed into a subject program or program variant. The replayer captures output from the subject program in the form of streams (stdout and stderr), files, or network transmissions.

Like Grace, file and network activity is sandboxed to avoid affecting the file system and ensure safe execution. A program attempting to open the file ‘/etc/hosts’, for example, will be redirected to a mirror of the file hierarchy containing a file of that name, such as ‘/home/user/replay_sandbox/etc/hosts’. The actual contents of the mirrored file are determined by the data generated by Grace and stored in the JSON file. Similarly, network I/O is redirected to a Unix domain socket fed with the data contained in the input file.

The replayer’s core is implemented using two approaches, selectable at runtime. The ptrace-based replayer uses the ptrace system call to monitor events in the subject program such as the opening of files. The Strata-based replayer uses the software dynamic translation engine to monitor the execution of code blocks, matching addresses with entry points for known file and network operation symbols (open, fopen, socket, etc.). In either case, when the I/O operation is detected it is redirected to a sandbox as described earlier.

In summary, replay of inputs works as follows: (1) the subject program is run under the replayer; (2) the replayer monitors the subject for events that open an input resource (such as a file) or would potentially cause an unsafe system access; (3) when an input resource is about to be opened, the replayer creates a mock object or network stream containing the intended contents for that resource and redirects the subject to read from the mocked resource, instead; and (4) unsafe system calls are skipped.

We have predominantly used the Strata-based replayer, as it is faster and allows for collecting instruction coverage. This coverage data can be used by Grace to determine the improvement in coverage that inputs in successive generations produce, using that improvement to sort inputs within Grace’s work queue.

3.3.4 STARS Static Analyzer

Several of the defenses and transformations provided in the PEASOUP project rely upon information produced by a static analysis of the program binary being protected. A static analyzer for program binaries was developed in prior software security projects, and extended for the needs of the PEASOUP project. This static analyzer, called STARS (STatic Analyzer for Reliability and Security), will be described in sufficient detail to explain its role in the PEASOUP Phase 1 project and its potential for future phases of the project.

3.3.4.1 IDA Pro Disassembler

STARS is implemented as a plug-in to the popular IDA Pro disassembler. IDA Pro is the leading commercial disassembler, targeting more than 40 computing platforms [59]. The IDA SDK (Software Developer’s Kit) permits the development of C++ language plug-in modules that will use data structures constructed by IDA Pro in its disassembly of a program binary. These data structures include important information concerning instructions (e.g. a list of operands, which operands are read and which are written), functions (e.g. starting and ending instruction addresses, function name, cross-reference links to functions that call this function), and data segments (e.g. starting addresses and sizes of data variables). The information provided by IDA Pro is the starting point for the security-specific analyses provided by STARS.

3.3.4.2 STARS Plug-in Architecture

STARS uses the information provided by IDA Pro to begin its analyses of the program to be protected. The following subsections describe the design of STARS in detail. Note that STARS is currently implemented for x86/Linux program binaries, but can be ported to any IDA Pro target in the future.

3.3.4.3 Code discovery.

A common problem in disassembly of a program binary is the discrimination of code addresses from data addresses. Data can be interleaved with code to varying degrees in different architectures; e.g. it is common for some constant data to appear in the code segments of a binary. Because the opcode definitions in most architectures are dense rather than sparse, almost any bit pattern could appear to be a valid opcode, making a data byte appear to be the beginning of an instruction.

There are two primary design approaches for disassemblers: *recursive descent* and *linear scan*. A recursive descent disassembler starts at the program entry point, defined in the binary file header, and follows control flow (jumps and calls) to continually discover new code addresses. Because the program will not jump to data addresses, this helps avoid false identification of data as code. The pitfall in this approach is that some code addresses are only reached by indirect call instructions that can be difficult to analyze statically. A linear scan disassembler begins with the program entry point and moves linearly through the binary, applying some heuristics and backtracking to detect and fix problems of false identification of code as data. This approach detects code that is only reached by indirect calls, but the heuristics do not always fix the false identification problems.

To make the code discovery process even more precise and sound, STARS parses the output from the Linux `objdump` disassembler, which is a linear scan disassembler, and compares its code discovery to the code addresses found by IDA Pro, which is a recursive descent disassembler. STARS applies heuristics to determine which code addresses discovered by `objdump` seem to be valid, based on analysis of factors such as whether the code addresses will flow back to the existing code addresses, whether the code patterns fit the known function prologue and epilogue patterns of compilers, etc. Valid code addresses that were missed by IDA Pro are then sent to IDA Pro for re-analysis, integrating them into the program database that is internal to IDA Pro. This integration causes IDA Pro to produce the information about the newly discovered code that will be used later by STARS.

3.3.4.3.1 **Auditing IDA Pro information.**

In addition to improving the code discovery results from IDA Pro, experience has shown that certain code patterns and even certain opcodes cause problems in IDA Pro. STARS performs certain audits of the information provided by IDA Pro to fix up control flow cross references, fix operand lists with certain operands not marked as being written to for a few opcodes, etc. After the code discovery and auditing phases are complete, STARS has reliable information from IDA Pro to use in building its own object-oriented description of the complete program.

3.3.4.3.2 **Building a class hierarchy for the program.**

STARS builds an object-oriented class hierarchy describing the program, with the levels of the class hierarchy being `Program, Function, Basic Block,` and `Instruction`. A `Program` consists of a collection of functions, plus data segments from the binary. A `Function` consists of a collection of `Basic Block`s, plus a description of the stack frame data layout (local variables, incoming and outgoing arguments, saved return address, and saved registers) and identification of stack allocation and de-allocation instructions. The `Basic Block`s are determined by STARS while making an initial pass over the instruction information provided by IDA Pro, including control-flow cross-references to identify jump targets. A `Basic Block` is a collection of `Instructions`, plus data sets produced by the data flow analyses described in the next subsection. An `Instruction` encapsulates the address, size, disassembly text, operand list and opcode information provided by IDA Pro, which is augmented by extensive STARS analyses to identify attributes of interest to our security analyses, e.g. whether the instruction has a memory operation that could be aliased to unknown memory addresses, whether the instruction is a form of no-op inserted by a compiler for various purposes such as code alignment, etc. After building the basic `Instruction` class information, STARS produces sets of defined and used (e.g. written and read) operands, called DEF and USE sets, as well as a register transfer list (RTL) that describes the operation of the instruction in binary tree form. The interior nodes of the RTL tree are operators, while the leaf nodes are operands such as registers or memory locations. The RTLs, DEF sets, and USE sets are the basic components upon which the data flow analyses of the next subsection will operate.

3.3.4.3.3 **Data flow analyses**

The key to performing security-specific static analyses in STARS is the data flow analysis phase. STARS first performs LVA (Live Variable Analysis), in which four sets are built for each `Basic Block`, identifying which registers and memory locations are Killed (i.e. written to), Upward Exposed (i.e. read within the block before being written to), LiveOut (i.e. the value upon exit from the basic block will be read before being written over), and LiveIn (i.e. the value upon entry to the block is either Upward Exposed or passes through the block untouched and is LiveOut) [3]. These LVA sets are useful in their own right, but are particularly needed in the next data flow analysis.

The major data flow analysis is the construction of a *fully-pruned SSA (Static Single Assignment) form* of the program. An SSA form defines which writes to registers or memory locations correspond to which later reads from those registers or memory locations, implicitly creating a complete set of def-use chains. In order to avoid extraneous def-use chains, *the fully pruned SSA form* construction algorithm makes use of the LVA sets to avoid creation of SSA chains for variables that are no longer live at the relevant program points [127]. The result is a set of SSA def-use chains that are sound and precise, paving the way for analysis of the data types of each chain.

3.3.4.3.4 **Type inferences and annotations**

STARS uses the SSA chains to infer data types using a simplified type system, in which each value in the program is one of the types POINTER, PTROFFSET (e.g. a difference between two

POINTER addresses), or NUMERIC (e.g. all non-POINTER data, whether it is integer, floating point, character string, code address, or boolean). This simplified set of types was developed for the MEDS (Memory Error Detection System) project, in which pointers are tracked dynamically during program execution to ensure that memory writes cannot occur outside of the intended memory referent for the pointer used in the memory write [105]. The NUMERIC type has subtypes such as STRING and CODEPOINTER, and the POINTER type has subtypes such as STACKPTR, HEAPPTR, and GLOBALPTR for different memory allocation types. The STARS type lattice has a top value of UNINIT and a bottom value of UNKNOWN (meaning that a mixture of NUMERIC and POINTER or PTROFFSET uses were detected). The type inference process begins with certain provable inferences based on opcodes and operands (e.g. a memory write through `[ebx]` implies that register `ebx` holds a POINTER at this point in the program). STARS iterates over SSA chains, propagating the type of a DEF to all USEs in the chain, and inferring the type of the DEF if all USEs have had a consistent type determined.

The results of STARS analyses are emitted into an annotations file, which can be read by other tools (e.g., a run-time security monitor based on Strata). The annotations include hints on how to monitor for memory overwriting errors, the basic locations of functions, stack frame layout info, and stack allocation and de-allocation instruction locations. The set of annotations emitted by STARS was enhanced during the PEASOUP project to support the defenses implemented in Phase 1 of PEASOUP, as described next.

3.3.4.4 PEASOUP extensions to STARS

Annotations from STARS assisted in three different aspects of the PEASOUP defenses. First, existing annotations concerning the stack layout, stack allocation and de-allocation instruction locations, function locations and sizes, and instruction locations and sizes were deemed sufficient to support the ILR (Instruction Layout Randomization) transformation. ILR can be supported even more extensively in the future by improvements to STARS, as discussed in the next subsection.

Second, stack frame layout randomization was supported by improving the precision of the identification of sub-regions within each stack frame, such as the outgoings arguments area, the local variables area, and the saved registers area. In addition, new annotations indicate that certain functions had IDA Pro analyses that were incomplete and their stack frames might not be properly partitioned in the STARS annotations, in which case the stack frame layout should not be re-arranged or randomized because software errors (i.e. PEASOUP false positives) might result.

Third, significant analyses were added to STARS to track the bit width and signedness of registers and stack locations. The bit width is obtained from IDA Pro information. The signedness is inferred from certain opcodes (e.g. a sign-extended load from memory indicates that both the memory location and the target register are signed, while a zero-extended load would indicate unsigned operands). Signedness and bit width are also inferred for the return register of numerous standard library functions (e.g. `atol()` returns a signed long integer). STARS then propagates signedness and bit width along def-use SSA chains. For any opcode that could produce a truncation, overflow, or signedness error, annotations were emitted to inform other PEASOUP tools how to instrument the binary to detect the potential integer errors at run time. These annotations indicate the bit width and signedness to be tested.

3.3.5 Data Delineation Analysis (DDA)

One of the memory protections we have used in PEASOUP is Stack Layout Transformation (SLX). The SLX protection relocates and pads stack objects to prevent attacks that are based on overwriting stack data (e.g., return addresses, function pointers, and local variables). The effectiveness of SLX depends upon inferring accurate stack layout data—a challenging problem for a stripped binary[5]. If SLX misses some of the stack-object boundaries, then some of the objects will not be adequately protected. If, on the other hand, SLX mistakenly breaks up a stack object that the program expects to be contiguous, then the transformation will likely break the correct functionality of a program. Thus, we needed a robust and precise analysis for identifying boundaries of stack objects.

We have considered a number of existing approaches and found that none of them provided an effective and complete solution to our problem. Existing techniques for inferring object boundaries tend to fall short in one of two ways. Some heuristic-based approaches, e.g., IDA Pro, often break large objects into pieces [74]. This can result in false alarms if used for bounds checking, and may even disrupt program functionality if used for program transformation. Other approaches, e.g., [36, 166], assume that a program is memory safe, and thus derive bounds that include the potential overruns. The resulting information is unhelpful for buffer-overrun detection.

To meet the needs of the SLX transformation, we have designed a novel, heuristic-based analysis for inferring locations and sizes of stack objects in an arbitrary stripped executable. We call our analysis Data Delineation Analysis (DDA)[6]. We use the term “object” to refer to any top-level datum, such as an array, structure, or variable, regardless of whether or not the binary was created from an object-oriented language. DDA first finds a set of object boundaries that is largely a superset of the desired result, and then it systematically refines this set by eliminating boundaries that fall within larger aggregate objects, such as structure instances and arrays. Insofar as the initial set of boundaries provides an over-approximation to the ground truth, our goal is to eliminate as many boundaries that cause false-positive buffer-overrun warnings as we can; at the same time, we want to retain as many boundaries that enable overrun detection as possible.

For the identification of aggregate objects, we use a novel technique that we call Parameter-Offset Analysis (POA). POA operates by identifying and symbolically propagating possible “base” pointers to objects. For each base pointer, the set of constant offsets used in memory dereferences is collected to estimate the extent of the object pointed to by the base pointer. That is, if the analysis sees an instruction “`mov eax, [ebx+128]`” and the symbolic value of `ebx` is base pointer P_{base}, the analysis infers that the instruction is performing a field access and concludes that the object pointed to by P_{base} is at least 132 bytes long (the memory access reads four bytes).

To avoid the pitfall of over-approximating object extent based on unsafe memory accesses, we rely on the intuition that buffer overruns are more likely to be in loops where the address or offset changes each iteration. Therefore, our analysis only makes use of offsets that appear as syntactic constants in the program text.

In the subsequent sections we will present the individual pieces of the analysis and describe how they fit together. A detailed description of the analysis can also be found in [98].

[5] Even when the symbolic and/or debug information is available, not all of the stack-object boundaries are immediately available. A complier typically allocates some amount of auxiliary storage on the stack (for e.g., register spills, temporary variables, canaries, etc.) the information about which is not retained in the debugging information.

[6] The analysis infers boundaries for both stack objects and global objects, though we did not get a chance to use the information about global objects in the context of PEASOUP.

3.3.5.1 Initial Boundary Identification

The first step of DDA is to identify a superset of object boundaries. This set is later refined by removing the spurious boundaries that partition the aggregate objects inferred by the Parameter Offset Analysis. To build this set, our analysis collects all constant stack offsets that appear in the body of a function. That is, the analysis scans each function's instructions looking for memory dereferences of the form `[reg+c]`, where `reg` is the stack or frame pointer (`esp` or `ebp` on x86) and `c` is a constant. For each such access, we add an object boundary derived from c. This approach is close in spirit to the heuristic employed by IDA Pro.

3.3.5.2 Parameter Offset Analysis

The problem with using constant stack offsets as the set of object boundaries is that a compiler typically computes all offsets within an activation record—including offsets of internal structure fields and array elements—as constant offsets from the beginning of the frame. We illustrate this with the example shown below.

```
struct S {
    int a;
    char b[50];
    int c;
};

void bar(struct S * s)
{
    s->a = s->c + 5;
}

void foo()
{
    struct S s;
    s.b[0] = 'a';
    bar(&s);
}
```

```
080483b4 <bar>:
 80483b4:   push   ebp
 80483b5:   mov    ebp, esp
 80483b7:   mov    eax, DWORD PTR [ebp+0x8]
 80483ba:   mov    eax, DWORD PTR [eax+0x38]
 80483bd:   lea    edx, [eax+0x5]
 80483c0:   mov    eax, DWORD PTR [ebp+0x8]
 80483c3:   mov    DWORD PTR [eax], edx
 80483c5:   pop    ebp
 80483c6:   ret

080483c7 <foo>:
 80483c7:   push   ebp
 80483c8:   mov    ebp, esp
 80483ca:   sub    esp, 0x44
 80483cd:   mov    BYTE PTR [ebp-0x38], 0x61
 80483d1:   lea    eax, [ebp-0x3c]
 80483d4:   mov    DWORD PTR [esp], eax
 80483d7:   call   80483b4 <bar>
 80483dc:   leave
 80483dd:   ret
```

Note that access to a local structure field "`s.b[0]`" in `foo` (at effective address `0x80453cd`) is translated into a single constant offset from the procedure frame (`ebp-0x38`). Thus, using the constant stack offset heuristic from previous section will make the analysis flag the address of `s.b` as the boundary of an object, breaking up the instance of `struct S`. If we were to transform the layout of `foo`'s activation record based on this information, we would have broken the program because function `bar` expects the instance of `S` to be a single, contiguous object.

The Parameter Offset Analysis (POA) recognizes such compound objects as the instance of struct S in the example above by detecting base object addresses, and for each identified base address, collecting a set of constant offsets that are relative to that base address. The collected constant offsets allow our analysis to estimate the extent of identified objects. The primary source of base addresses is the set of stack offsets that are passed as the parameters to procedures. In our example, the stack address `ebp-0x3c` is passed as a parameter to `bar` (at `0x80483d7`), thus DDA learns that there may be an aggregate object at offset -60 from `ebp`.

Additionally, base object addresses are extracted from the following operations:

- Loading of a stack-based address into a register,
- Using a stack-based address as a source or destination of a hardware loop instruction, such as REP STOS on x86.

The sets of constant offsets that are relative to the object base addresses are collected by computing for each function parameter the set of constant offsets that are dereferenced based from that parameter. For the function bar in the example above, the analysis derives the following set of offsets: 0-3 (field `a` access at `0x80483c3`, and 54-57 (field `c` access at `0x80483ba`). Thus, the object passed by reference via the first parameter of `bar` is expected to be at least 58 bytes long.

The DDA analysis then combines the information that a stack object at offset ebp-0x3c is passed as the first argument to bar, which expects objects that are at least 58 bytes long, and learns that the constant stack offset ebp-0x38 is not a true boundary and must be removed from the set of object boundaries.

We implemented the DDA analysis as follows:

- The analysis is static: it picks up program representation (control-flow graphs, call graph, instructions) from the Intermediate Representation Database (IRDB) that is populated by other PEASOUP analyses.

- The analysis is interprocedural and compositional: a summary of each function is computed in separation; summaries of the callees are used to build up the summary of the caller. We wanted to avoid fixed point computation for efficiency, so we eliminated recursion by removing back edges from the call graph of the program. The augmented call graph is sorted in reverse topological order and an intraprocedural analysis is applied to each function.

- An intraprocedural analysis performs symbolic execution of a procedure. The symbolic state is used to identify memory dereferences that are based on parameters and that have scalar offsets. Again, to avoid fix point computations, we perform the analysis on a spanning tree of control-flow graph of a procedure.

There are two major simplifications that allowed us to make the analysis scalable:

- Ignoring recursion

- Analyzing spanning trees of control flow graphs (CFGs) rather than analyzing full CFGs.

While, in general, these simplifications may negatively affect the precision of the analysis, we believe that our choice is justified: modeling loops precisely seems to be of little value for the analysis that targets dereferences with constant offsets. The experimental evaluation of the DDA analysis indicates that precision loses that are due to these simplifications are negligible.

Collapsing overlapping objects. Some stack objects that the Parameter Offset analysis identifies may overlap. Consider the example below:

```
struct S {
    int a;
    int b;
    int c;
};

struct T {
    int u;
    int v;
    struct S s;
};
```

```
void foo(struct S * s) {
    … s->a … s->b … s->c…;
}

void bar(struct T * t) {
    … t->u … t->v …;
    … t->s.a;
}

void foobar() {
    struct T t;
    foo(&t.s);
    bar(&t);
}
```

The Parameter Offset analysis will determine that functions `foo` and `bar` both dereference offsets [0, 11] from the passed-in `struct` pointers. Let us assume that, in function `foobar`, the local variable `t` is allocated at frame offset 24. The Parameter Offset analysis will identify two local objects in `foobar`—both 12-byte long—one starting at offset 24 (an offset passed to bar) and one starting at 32 (the offset passed to foo). Note that the two objects overlap, leading to an inconsistent frame view.

We extended the analysis to perform a post-processing step that collapses together the overlapping objects. Our assumption is that objects that overlap are likely to be the parts of one large object and should be kept together by transformations of SLX kind. Our implementation, does not simply collapse the objects together, but rather constructs a tree of objects trying to infer the structure of the containing object (i.e., a form of type inference). However, we do not yet use this extra information in the overall Object Delineation analysis.

The above approach is conservative and may lead to extra false negatives, though we have not yet observed any. As our study of DWARF debug information showed, GCC may reuse stack memory for objects with disjoint live ranges (e.g., the variables from different lexical scopes). Our extended analysis will collapse such objects together, even though, potentially, such objects could be untangled and relocated separately by the SLX-like transformation for additional protection. If this issue poses problems, we will address it by taking live ranges into consideration.

Summarizing system calls. We use the Parameter Offset analysis to automatically generate models for a number of standard libraries (about 70 libraries found in /lib/i386-linux-gnu on 32-bit Ubuntu 12.04). The models are serialized on disk as text files and are read in by the Parameter Offset analysis as needed. Reusing standard library models allows us to reduce both the analysis time and used memory—standard libraries are typically large. Our attempts to apply the Parameter Offset analysis to some of the T&E base programs along with the standard libraries that they include routinely ran out of memory.

This modeling approach worked well for many of the library functions. However, some library functions are implemented as thin wrappers on top of Linux system calls. One particular example of such function is `_xfstat64`, which is used extensively in coreutils and often manipulates stack allocated instances of `stat64` structure. To handle such functions, we added the infrastructure for linking the manually-constructed models for system calls and implemented models for a few dozens of system calls that appear throughout libc implementation.

3.3.5.3 DDA Implementation and Integration with SLX

We implemented the Data Delineation Analysis on 32-bit Ubuntu 12.04 and integrated it with the stack-layout transformation infrastructure (SLX). SLX employs a hierarchy of sources that include constant offsets from a stack pointer and analysis results from STARS. When aggressive transformations fail, SLX backs off to a more conservative transformation and tries to validate again. We modified SLX to retrieve the boundaries inferred by the DDA and use them as the starting point for the speculative refinement of object boundaries used for the layout transformation. This setup allowed us to perform initial experiments on using DDA results for program transformation. We describe the experimental results in Section 4.4.5.

In the course of Phase 3, we have ported DDA implementation to work on 64-bit Ubuntu and to analyze 64-bit binaries. The work primarily involved generalizing the DDA code to cover the specifics of 64-bit platform:

- *Register-based parameter passing:* x64 uses primarily registers for parameter passing, whereas x86 primarily passed parameters via stack. We had to extend the analysis to support that.
- *Red zones:* x64 ABI allows the compiler to not "allocate" stack space for leaf functions. That is, locals and temps are stored above (at lower addresses than) the top of the stack (the `rsp` register). Since we compute stack boundaries relative to to the top of the stack, we had to extend the analysis to correctly handle negative offsets.
- *System call handling:* we had to adjust our handling of system calls which was primarily x86 specific to also support x64 system call convention.
- *Library summarization:* we automatically generated models for the x64 version of libc (`/lib/x86_64-linux-gnu/libc-2.15.so`). This was sufficient for our initial experiments, but given more time, we would want to generate models for all of the libraries under /lib/x86_64-linux-gnu.

In the process of porting we have discovered and fixed several issues in our specifications for the syntax and semantics of the x86_64 instruction set.

Unfortunately, we were not able to complete the SLX integration of DDA analysis prior to the start of Phase 3 Test and Evaluation.

3.3.6 Dynamic Rewriting.

Figure 11 shows the high-level architecture of the off-line or redeployment portion of PEASOUP. PEASOUP consists of a static analyzer, called STARS [63], that disassembles x86 binaries, performs extensive static analysis of the binary, and then stores the results of the analysis along with the binary in a persistent store called the IRDB (Intermediate Representation Database). The IRDB is the repository for all information known or determined about a binary. A PEASOUP analysis and transformation phase uses information in the IRDB to create new versions of a binary, called variants, where various armoring transformations and remediation policies have been applied. Rather than statically rewrite the binary, PEASOUP produces programs, called *Sprockets*, that are used by a software dynamic translation system to transform the original binary into the corresponding variant at run time. Sprocket programs are specified via the Sprocket Program Rewriting Interface (SPRI).

To ensure that the variants produced by PEASOUP run appropriately, they are then "vetted" by a tool called BED (Behavior Equivalence Detection). BED runs each variant using a test suite (if available) to ensure that the variant produces the same output as the original binary. In addition, BED uses a fault injector to inject faults into the application to determine the effectiveness of the remediation policies generated by PEASOUP.

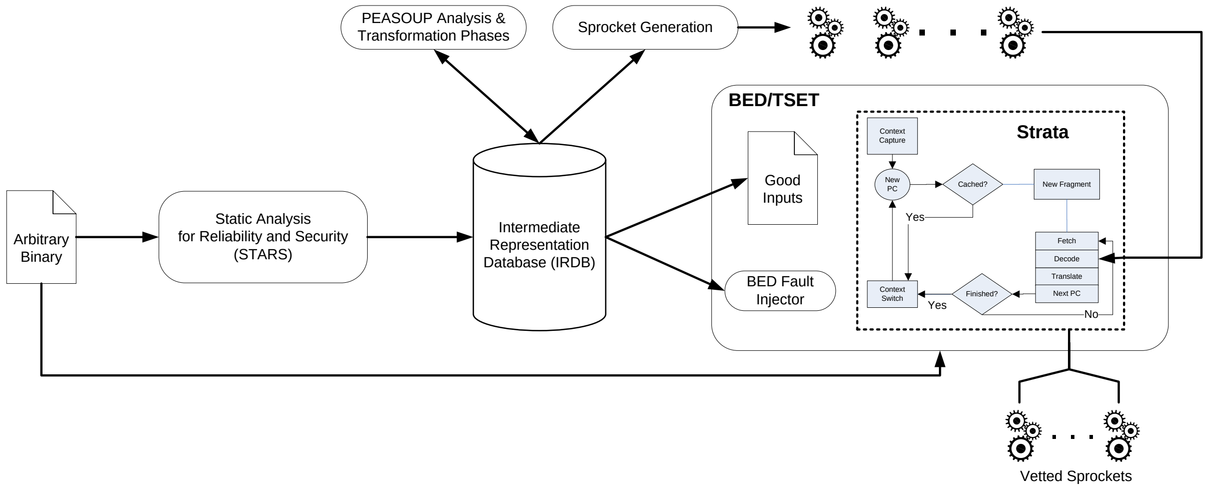


Figure 11. PEASOUP Architecture: Offline Generation of Sprockets Programs

Figure 12 shows a deployed binary that is protected by PEASOUP. The vetted Sprocket programs are applied to the binary by the software dynamic translator, Strata. PEASOUP has the ability to dynamically select from the set of Sprocket programs to effect temporal change in the protections that are applied.

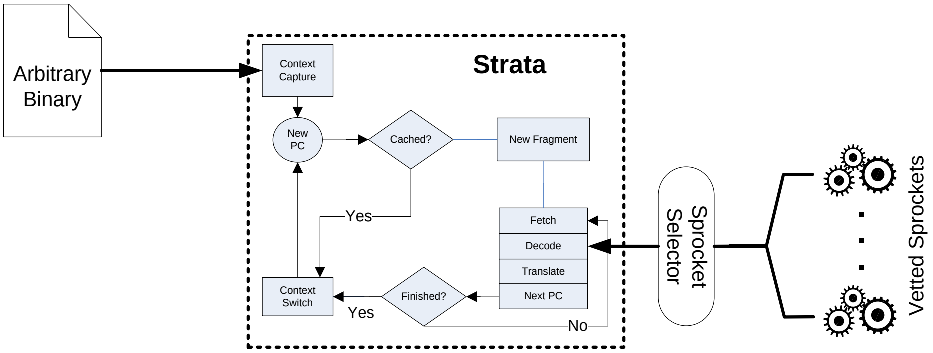


Figure 12. PEASOUP Architecture: Online Selection of Sprocket Programs

Section 3.3.6.1 discusses the generation of Sprocket programs via the Sprocket Program Rewriting Interface (SPRI) and how dynamic binary rewriting solves many problems associated with static binary rewriting techniques to effect significant change in the binary.

3.3.6.1 Sprockets and SPRI

Sprocket programs are specified via the Sprocket Program Rewriting Interface (SPRI). SPRI defines simple rewriting rules that come in two forms. The first form, the redirect form, transfers control to a specified target address (lines 1 and 3 in Figure 13).

The second form, the instruction definition form, indicates that there is an instruction at a particular location (line 2).

The net effect of applying the SPRI rules shown in Figure 13 is to rewrite the instruction `sub esp, 20` instruction at address `0x8000` to be `sub esp, 40`. The stack layout transformation described in Section 3.8.2 uses such rules to transform stack frame allocations.

```
Original program fragment:
(a) 0x8000 sub esp, 20
Rewrite rule:
(1) 0x8000 -> 0xFF00
(2) 0xFF00 ** sub esp, 40
(3) 0xFF01 -> 0x8001
```

Figure 13. Sprocket Rewrite Rule to Change Stack Frame Allocation.

Together these two types of rules provide the foundation for building a wide range of Sprocket programs. The example shown illustrates the equivalent of a small patch that modifies only 1 instruction. At the other end of the scale, transformations such as ILR (Instruction Location Randomization) seek to rewrite all instructions in a binary.

Despite its conceptual simplicity, manually writing Sprockets in SPRI would be a tedious and error-prone process. Instead, Sprocket developers encode their transformations using a high-level C/C++ API to manage the creation and deletion of program variants, and to manipulate program state, e.g. to insert, delete, or replace instructions and re-route control flow. The API transparently interacts with the IRDB to commit any changes. With this architecture, the composition of Sprockets is naturally performed by chaining together transformations: one Sprocket encodes its transformation in the IRDB, the next Sprocket then takes as input the new database state, and then effect its own transformations. PEASOUP will then automatically generate SPRI rules for any program variants by essentially performing a ``smart diff'' between the IRDB representation of a variant against the IRDB representation of the original binary.

3.3.7 Efficient Checkpointing for Remediation

Another major task for Phase 1 was to develop efficient VMM-based checkpointing technique for automatic remediation. Besides remediation, checkpointing-based fast virtual machine provision is also attractive for solving the scalability problem we are facing in offline analysis.

Checkpoint for Remediation

PEASOUP has a powerful offline analysis, but due to the limitation of offline analysis (complete and soundness) and the vulnerability class coverage, not all vulnerabilities can be fixed at offline phase. Currently, when unfixed vulnerability is exploited, our execution manager will detect this and in most cases, perform a control exit. However, ideally we would like to have an efficient remediation mechanism so our execution manager can restore the execution to a known good state. In PEASOUP, we plan to use checkpointing technique to build our remediation infrastructure.

Scalability Problem

During offline analysis, Grace performs concolic analysis on input SOUP to generate high coverage test suite. One problem with Grace now is it does not scale well, so analyzing an input could take a long time due to the path explosion problem. One promising approach to improve the scalability is Hybrid Concolic Testing [136]:

> *From the initial program state, hybrid concolic testing starts by performing random testing to improve coverage. When random testing saturates, that is, does not produce any new coverage points after running some predetermined number of steps, the algorithm automatically switches to concolic execution from the current program state to perform an exhaustive bounded depth search for an uncovered coverage point.*

By leveraging checkpointing technique, the performance of *hybrid concolic test* could be further improved based on the following observations:

1. Since the concolic execution only begins after some predetermined number of steps, the performance could be improved if we could checkpoint the test program before switching to concolic execution thus avoiding re-execution the same steps over and over again;
2. After checkpointing current status, on current multi-core system, we could explore different paths in parallel at the same time.

3.3.7.1 Related Work

Mainstream virtual machine monitors like VMware, Xen, KVM and VirtualBox all have snapshot (checkpointing) support. Snapshot allows user to save the current state of a virtual machine and revert to this state if something goes wrong. Therefore it has already been used for remediation. The problem is, the performance of the snapshot mechanisms used in these systems are not very good enough for PEASOUP: it usually takes tens of seconds to several minutes to save or restore a snapshot. And a large portion of time is spent on I/O, i.e. writing or reading the state to non-volatile storage like hard drive whose throughput is usually not very high.

To solve this problem, researchers form Georgia Tech have proposed a copy-on-write (COW) based technique[7] to reduce the VM's downtime and the performance overhead incurred by other forms of VM checkpointing. That is, instead of halting the VM and taking a whole memory dump, this technique will use COW to protect the memory states at the snapshot time. So it won't impose a long downtime.

In Cloud computing and honeyfarm environment, another critical requirement is the ability to provide a virtual machine as fast as possible. To meet this requirement, researchers have proposed several solutions. Among these solutions, Potemkin[8] and SnowFlock[9] are the representative ones. Their basic idea is similar, that is used a mechanism that is very similar to *NIX fork, so they call this technique VM fork. By using this technique, they can create many task-demanding and transitory VMs in seconds.

[7] Michael H. Sun and Douglas M. Blough, ***Fast, Lightweight Virtual Machine Checkpointing***, Technical Report, Georgia Institute of Technology, 2010.

[8] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. 2005. ***Scalability, fidelity, and containment in the potemkin virtual honeyfarm***. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (SOSP '05)

[9] Horacio Andres Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. ***SnowFlock: rapid virtual machine cloning for cloud computing***. In *Proceedings of the 4th ACM European conference on Computer systems* (EuroSys '09).

3.3.7.2 Design

Since we will use KVM as the VMM, our design is based on KVM's existing snapshot mechanism.

Checkpointing

Because KVM already has the functionality to create checkpointing (savevm) and revert to a checkpointing (loadvm), we plan to leverage these existing functionalities to build our checkpointing mechanism by:

1. Adding a new hypercall to let Strata request a checkpointing creation (reverting);
2. Modifying KVM to trap this and notify the user mode controller (QEMU);
3. Modifying QEMU to handle this event and calling the corresponding function.

To keep the performance overhead low, especially keeps the downtime low, we plan to COW based checkpointing technique similar to *Fast, Lightweight Virtual Machine Checkpointing* (Xen based).

VM fork

To checkpoint the target program for Grace, there are two basic approaches: one is *nix system fork based and the other is virtual machine based. Since Grace also relies on filesystem and networking, VM-based solution (i.e. VM fork) is better. More specifically, we plan to add following functionality:

- Add a new hypercall to let Grace clone a VM;
- When invoked, the VMM will
    - Checkpoint current analyzing environment;
    - Create multiple copies of the analyzing environment and let Grace choose which ones to execute;
    - And if possible, execute the select the copies in parallel.

3.3.7.3 Implementation

This section presents our primitive implementation.

Checkpointing

As discussed above, the ideal way to implement checkpointing is COW. The QCOW2 virtual disk format already supports COW and the Linux system also supports memory COW. However, on the other hand, as KVM kernel part is tightly integrated with the Linux kernel, even duplicating the kernel VCPU state without COW is not easy. Therefore, to develop the prototype quickly, we decided not to use memory COW, and just to reuse KVM's existing checkpointing mechanism.

But to solve the latency imposed by I/O thus emulate the result of COW, we modified KVM's checkpointing mechanism, so instead of checkpointing the memory image to hard disk, it will save the image in a ramfile (file completely stored in memory). Although this strategy is not spatial efficient and saving a whole memory image requires more time than COW individual page, we think as it is appropriate for an engineering proof-of-concept implementation, because 1) memory is very cheap now and 2) the memory used by a VM is not large in our configuration.

VM fork

As of checkpointing, since the parent VM and the child VM share a large portion of system states (memory and disk), the ideal way to implement VM-fork is using COW mechanism. But as implementing a COW time consuming, our current strategy is: when the parent VM wants to fork a child, we save its states into a checkpointing, then we create a new VM as the child VM and after the child VM is booted, it loads the saved states from the checkpointing.

To implement this strategy, we reuse a large portion of KVM's existing snapshot mechanism and only made a small change:

1. We modified the savevm function so the snapshot can be saved to a dedicated checkpoint file. To simplify configuration, recent KVM implementation saves the snapshot to a virtual disk image instead of a separated file (like VMware does). However, this is bad for us because we cannot share the created snapshot. So we modified KVM to let it save the snapshot to the given checkpoint file (a dedicated virtual disk image).
2. We added another load checkpoint function to let the child load the saved snapshot. The reason is, when creating the child VM, we cannot let it share the virtual disk image with its parent VM. So we create a new virtual disk linked to the parent's one. As a side effect, we can no longer access the saved disk snapshot (the checkpoint file only contains CPU, memory and other device states). But the original loadvm function will fail when it cannot find the snapshot for the disk.

To improve the performance, we put the snapshot file in a tmpfs or fast SSD and also put the swap file on the SSD.

In general, the workflow of forking a VM in this implementation is:

1. The parent VM calls savevm to create a snapshot which will be saved in an in-memory checkpoint file;
2. After the snapshot is created, the VMM creates a new virtual disk for the child;
3. The VMM creates a new instance of the VM and resumes the execution of the parent VM;
4. When the child VM is booted, the VMM loads the saved snapshot from the checkpoint file.

3.3.7.4 Future Improvement

This section describes the updated design of our fast checkpoint/VM fork technique.

3.3.7.4.1 **Recent advances in checkpointing techniques**

On VEE 2011, there are two papers on checkpointing techniques that both leverage page cache.

The first one[10] is based on the observation that a large portion of the memory used by modern OS is for page cache. Since page cache can be reconstructed from virtual disk file, it is unnecessary to save those pages as part of a checkpoint, thus improving both speed and space efficiency. The technique behind this is automatically and transparently tracking I/O operations of the guest to the external storage and maintains a list of memory pages whose contents are duplicated on non-volatile storage.

The second one[11] proposed a fast warm-reboot technique to reduce the performance downgrade caused by rebooting.

Inspired by these two papers, we proposed following mechanism to further improve our checkpointing technique.

3.3.7.4.2 **Page cache combining**

The intuition behind this is, for KVM, both the host Linux system and the guest VM has page cache for the same content of virtual disk, thus one copy could be eliminated to improve the virtual I/O performance. One simple way to eliminate the duplication is to use cache=none option to disable host page cache. However, this has two major drawbacks. First, the performance of a qcow2 format virtual disk is terrible when host page cache is disabled. Raw format has good or better performance when host page cache is disabled, but it does not support snapshot. Second, letting each virtual machine use its own page cache prevents common content being shared. Therefore, we propose combining the host and guest page cache. Later we will see that this may also benefit checkpointing.

The combination could be implemented by leveraging the transparent I/O tracking technique mentioned above. That is, when the guest OS issues an I/O operation to read (write) a block from (to) disk and allocate a page cache for it, instead of going through the traditional handling path (i.e. notifying QEMU, parsing virtual disk format, issuing corresponding virtual disk operation and putting the content in guest's memory), the host kernel can directly map the host page cache for the virtual disk file block into the guest address space. To handle a remapping event generated inside the guest, we can modify the virtio paravirtualized block device driver to indicate KVM to unmap the host page cache.

If we can further put the virtual disk parsing function into the kernel, we can also save context switch between the host kernel and QEMU, e.g. by using raw format virtual disk. To solve the problem of checkpoint support for raw virtual disk format, we can simply disable page cache flush for virtual disk file on the host once the guest has been checkpointed.

[OPTIONAL] We can also share virtual disk page cache between different VMs based on a COW manner. This multiplexing can be handled by adding another label to the page cache.

[10] Eunbyung Park, Bernhard Egger and Jaejin Lee, *Fast and Space Efficient Virtual Machine Checkpointing*, VEE 2011.

[11] Kenichi Kourai, *Fast and Correct Performance Recovery of Operating Systems Using a Virtual Machine Monitor*. VEE 2011.

3.3.7.4.3 **Updated Design**

Leveraging this technique, we update our design as:

Checkpointing

When checkpointing a VM, only two things will be save at the checkpoint time: the VCPU context and the top level entry of the secondary translation structure (EPT/NPT). Then the whole guest's address space and associated host page cache will be COW protected. A new checkpointing data structure is created to track further changes. Things need to be tracked includes guest pages and host page caches that are modified.

Now a reverting means restore the saved VCPU context and the top level secondary translation structure (since old pages are still there, there is no need to change the translation structure), and the free all the pages that are COW. But committing a checkpoint is more complicated because we need to walk the saved translation structure to free saved pages. However, if we track pages being saved instead of pages being modified, then committing is optimized as simple freeing all the saved pages but reverting will become complicated. Or maybe we could track both things.

To save memory, when a new checkpoint is created, the new one will be linked to the old one. So when reverting the system, we can just begin with the change set associated with that checkpoint and discard all the changes made by itself and its descendants.

VM fork

Forking will still be handled by passing the saved checkpoint to the child, but this time, only the VCPU context and the top level secondary translation structure will be passed. The whole guest physical address space will also be COW protected, but tracking is not needed.

3.4 C1: Number-Handling Errors

3.4.1 Confinement of Incorrect Number-Handling Weaknesses

3.4.1.1 Integer Offline Analysis

PEASOUP uses STARS to analyze the binary and emit annotations for instructions that may result in an integer weakness. Annotations include the address of the instruction being instrument, bit-width information as well as sign information. The table below summarizes each annotation type:

Address	Annotation Type	Bit Width	Sign	Description
<address>	OVERFLOW, UNDERFLOW	8, 16, 32	SIGNED, UNSIGNED, UNKNOWN	Annotation emitted for x86 instructions that may overflow or underflow
<address>	SIGNEDNESS	8, 16, 32	SIGNED	Annotation emitted for x86 instructions that may result in a sign conversion error
<address>	TRUNCATION	8, 16, 32	SIGNED, UNSIGNED, UNKNOWN	Annotation emitted for x86 instructions that may result in a loss of information

In addition, PEASOUP also identifies critical functions such as those manipulating memory, strings and buffers. Overflowing variables that are directly or indirectly used as parameters to such functions is a common attack vector. We monitor access to these functions and verify that critical parameters fall within a reasonable range.

3.4.1.2 Integer Instrumentation

Instrumenting binaries to detect integer weaknesses is performed via an Integer Transform sprocket. For each annotation generated by STARS, the Integer Transform sprocket adds instrumentation to check for the occurrence of a realized integer weaknesses, e.g. an overflow condition.

For example, STARS might emit the following annotation for an multiply instruction at address 0x8000:

```
0x8000   INSTR CHECK OVERFLOW UNSIGNED 32 EAX
```

Assuming that the instruction at 0x8000 is the following:

```
0x8000    imul eax, ebx
          <nextIntruction>
```

The Integer Transform sprocket will then specify the following instrumentation:

```
0x8000    imul eax, ebx
          jno <nextInstruction>    # jump not overflo
          call integerOverflowHandler
          <nextInstruction>
```

In this example, PEASOUP can test x86 condition codes directly. If the multiply instruction does not overflow, then the cost of the instrumentation is the additional jno instruction. Other annotation types may result in slightly more complicated instrumentation code.

Note that sprocket developers specify the instrumentation code needed via a high-level C++ interface. The actual generation of the SPRI rewriting rule for this transformation, and the safe composition with potentially other rewriting rules, is automatically performed by the PEASOUP toolchain.

3.4.1.3 Benign Weakness Detection

Instrumenting all instructions that might lead to an integer weakness would likely disrupt the normal behavior of most programs as it is well-known that benign integer weaknesses abound in any programs of significance. Recently, researchers have confirmed this observation for integer overflows [218].

Common constructs such as hash functions, media decoders and encoders, encryption and decryption routines, modulo arithmetic, random number generators, to name but a few, all make deliberate use of integer operations that are instances of integer weaknesses. Further, compilers may generate code that exhibits integer weaknesses even though none were present or intended at the source code level. In addition, the C/C++ standards that define the semantics of integer operation are on the one hand complicated to understand, even for experienced programmers, and on the other hand, the standards are ill-defined. As a result, integer weaknesses are common.

Distinguishing security-critical integer weaknesses from benign weaknesses is therefore necessary. PEASOUP identifies benign weaknesses using a two-step algorithm:

(1) In step 1, PEASOUP instruments the program to detect integer weaknesses such as integer overflows, underflows, signedness errors, and truncation. PEASOUP then replays all inputs generated and categorized as benign by the GRACE concolic engine. Instructions that result in an integer warning are categorized as benign.

(2) In step 2, PEASOUP *only* instruments instructions that are not marked as benign.

If aggressive detection of integer weaknesses is the goal, then PEASOUP can be configured to instrument all instructions that were not marked as benign, even those that were not executed during replay. The disadvantage of aggressive detection is the increased probability of false positives, i.e., raising an error and potentially interfering with the normal continued operation of a binary. Alternatively, PEASOUP can also be configured to instrument only instructions that were covered by the inputs generated by the GRACE concolic engine. While this policy reduces false positives, it may also increase false negatives.

3.4.1.4 Saturating Arithmetic Policy

PEASOUP can be configured to effect various policies whenever an integer weakness is detected via the instrumentation described previously. The saturating arithmetic policy seeks to ensure that arithmetic operations stay within a fixed range. If an operation results in a value that exceeds the minimum or maximum value allowed, then this policy would replace this value with the corresponding minimum or maximum value. PEASOUP uses the bit-width and sign information contained in an annotation to determine the value to use for the saturation policy. For example, if an operation results in a negative number and is assigned to an unsigned quantity, then this quantity would be saturated to the value 0.

3.4.1.5 Final Integer Transformation

The current PEASOUP configuration is to implement the following set of policies.

Integer weakness type	**Policy**
Abuse of API	Emit error message and perform a controlled exit
Benign weakness	Emit warning message with no program alteration
Non-benign weaknesses	Remediate using saturating arithmetic, continue execution

These policies were selected to aggressively detect integer weaknesses. For a real-world deployment, these policies would need to be adjusted to be less aggressive so as to not raise false alarms when non-attack inputs. We are currently in the process of using TSET (Test Suite Evaluation Technology) to instrument only instructions that are covered by Grace, the PEASOUP high-coverage input generator.

3.4.1.7 Evaluation

We have evaluated the number handling framework using both synthetic programs to test each type of potential weakness as well as using real programs. The table below summarizes are results.

CWE	Test Examples	Attack Vector	Benign Weakness (Offline)	Online Detection	Result	Remediation
196	smartfuzz example	Integer sign conversion error ➜ unreachable code		✓	✓	Saturating arithmetic, continued execution
191	recaman, countlines	Integer sign conversion error ➜ negative buffer indexing		✓	✓	Saturating arithmetic, continued execution
121, 680	solitaire encryption	Integer error ➜ buffer overflow ➜ overwrite return address ➜ crash		✓	✓	Saturating arithmetic, continued execution
189, 680	bzip2	Integer vulnerability + overflow ➜ crash	✓	✓	✗	Detected integer weakness. * Should have identified as benign.
119, 189	3 html utilities	None known	✓	✓	✓	2 benign weaknesses detected;1 weakness compensated with saturating arithmetic, continued execution

These tests cover a wide range of CWE. As expected, our aggressive configuration was able to detect integer weaknesses for all these tests. However, for bzip2, the aggressive policy also resulted in applying a remediation action, thereby disrupting the normal functionality of bzip2. We hypothesize that given additional time Grace would have generated inputs to cover the remediated instruction. It would then have been classified as a benign integer weakness and would not have resulted in the remediation policy being applied.

For the html utilities test provided by MITRE, PEASOUP detected two benign weaknesses, and one non-benign weakness. The saturating arithmetic remediation policy resulted in the application emitting a parser warning (the utilities parse HTML) with the final output being the same.

We are in the process of tuning our default configuration to reduce potential false positives (as was the case with bzip2). Specifically, we believe that the following changes would result in a policy that would achieve the “sweet spot” for real-world deployments:

- increase the coverage of our test input suite by both using file seeding and by increasing the time allotted for test input generation
- use TSET to instrument only instructions that are covered by our test suite

These policies would reduce false positive rates as PEASOUP would apply remediation policies only to instructions covered during replay. The extent to which false negatives is reduced would be proportional to the time allotted for the test input generation phase.

3.5 C4: Resource Drains

We classify resource drains along several dimensions:

1. The type of resource that is drained, which can include memory, disk, file descriptors, sockets, and CPU.
2. The cause of the drain, which can include repeatedly allocating small ammounts, or allocating too much all at once.
3. The behavior of the resource drain. We identify four categories:
    a. **Leaks** occur when all references to the resource is lost before it is released.
    b. **Drags** occur when references still exist, but no possible continuation from the current execution still will result in a use the resource.
    c. **Over-ownership** occur when an execution simply allocates too much of a resource, even if the execution will technically use them eventually.
    d. **Overallocation** occus when too much of a resource is allocated all at once.

Different defenses are appropriate depending on the classification of the different resources. We explored a two stage approach for handling resource drains. The first stage was designed to detect and defend the first time an attack against a resource occurred. During this stage, PEASOUP would execute the following steps:

1. Deploy a resource monitor on vulnerable resources (e.g., loops that could be controlled by an attacker).
2. Deploy an anomaly detector to deter unusual drains on the resource (e.g., not leaving an inner loop after a long time.
3. When an anomaly is detected, attempt to generate a signature for the malicious behavior that caused the resource drain. Then kill the thread causing the drain.
4. Run a conservative garbage collector to attempt to reclaim any drained resources.
5. If conservative collection failed to reclaim enough of the resource, restart the process.

The second stage was designed to use signatures gathered during the first stage. As in the first stage, it would use a resource monitor and an anomaly detector. However, it would also monitor for the signature of the attack. If the signature was again detected, it would start to insert execution delays to slow the rate of resource consumption, long before the resource drain would affect other connections.

We completed partial implementations of the above strategy for file handles, heap memory, and infinite loops. Unfortunately, we were not able to complete integration with the rest of PEASOUP.

3.6 C5: Command Injection

The material in this section was also published in EDCC'14 [150].

Software weaknesses that lead to OS Command Injection Attacks are the #2 entry in MITRE's 2011 CWE/SANS list of Top 25 Most Dangerous Software Errors [60]. The high ranking makes intuitive sense: attackers that compromise an application can issue arbitrary commands to the underlying operating system, as if they were the owner of the application. The potential damages are especially catastrophic when the targeted applications are network-facing servers running with high privileges, e.g., file servers, mail servers, routers and even security appliances [1], CVE-2007-3572}, [7], [12].

In recent years, various taint-tracking techniques have been developed to thwart command injection attacks in general [129], [46], [165], [120], [155], [162], [149], [100], [56]. These techniques typically work by tracking the flow of data from an external source as it propagates through a program to a securitysensitive operation, such as network input flowing to a database command. Prior to issuing a security-sensitive operation, a command is first checked against its taint markings to ensure that critical parts of the command are not tainted. Modern taint trackers provide fine-grained resolution and keep track of taint markings at the level of individual characters. The resulting accuracy leads to few false alarms (false positives) and few undetected attacks (false negatives) [149], [162], [90], [221], [103], [101], [56].

Unfortunately, taint-tracking techniques are not practical for software binaries as keeping track of taint markings incurs high run-time overhead. Even state-of-the-art optimized taint trackers exhibit performance overhead between 50% and 200% [46]. In this section, we use OS command injection attacks for *program binaries* as a motivating example. PEASOUP also employs this technique to defend against SQL, XPATH, and LDAP injections. Our solution solution has the following characteristics[12]:

- Operates on binaries. The technique should operate directly on binaries, without requiring access to source code. In many deployment scenarios, source code will not be available, e.g., due to intellectual property protection measures, binary distribution, or use of legacy code.
- Easy deployment. The technique should be easy to apply and deploy. For example, it should not require the installation of a custom interpreter or significant changes in software development processes [149], [162], [90].
- Low/no overhead. A protected binary should incur very low overhead (< 1%) in attack-free mode.
- Low rates of missed attacks and altered functionality. The technique should be effective at stopping attacks but it should not modify the functionality of protected binaries under normal operation.

The design landscape for run-time, taint-based defensive techniques is summarized in the following table:

[12] These attributes were inspired by address space layout randomization, a security technique widely deployed across major commodity operating systems [2].

	negative taint	**positive taint**
taint tracking	Haldar ’05 [100] Newsome ’05 [148] Nguyen-Tuong ’05 [149] Pietraszek ’06 [162] Futoransky ’07 [90] Qin ’06 [165], Xu ’06 [221] Chin ’09 [56] Bosman ’11 [46] Papagiannis ’11 [155]	Halfond ’06 [103] Halfond ’08 [101]
taint inference	Sekar ’09 [184]	S_3

In one dimension, the focus is on keeping track of either untrusted data (*negative taint*) or trusted data (*positive taint*). In the other dimension, taint markings are derived either from tracking the flow of data through a program (*taint tracking*), or by inference (*taint inference*). S^3 investigates a combination of positive tainting and taint inference, the previously unexplored quadrant in this design space.

To meet our design goals, S^3 draws inspiration from *taint inference*, a technique described by Sekar [184]. Instead of instrumenting programs to keep track of the propagation of taint markings, Sekar’s technique simply infers taint marking by correlating inputs to substrings in security-critical operations using an approximate string matching algorithm. By obviating the need to propagate taint, taint inference achieves low overhead. Sekar’s taint inference technique relies on two main assumptions: (1) the accurate identification of external input data, and (2) external data is mostly used verbatim when used in a command. These assumptions hold true for most web applications, but not for binary programs. For example, consider a server that uses various forms of data encoding or proprietary protocols to read input, and possibly uses shared memory to communicate with other programs. In this case, it is difficult and expensive to identify and monitor sources of input. Furthermore, if the input is encrypted or encoded, as is often the case with servers that use SSL, inferring taint markings based on the program’s input becomes impossible.

S^3 captures the primary benefit of taint inference, i.e., low overhead, but uses positive tainting to obviate the needs of identifying sources of external data or relying on a readily observable correspondence between external input and critical commands.

The primary contributions of this paper are:

- We identify *positive taint inference*, a previously unexplored design point in the landscape of taint-based dynamic techniques.
- We demonstrate a realization of *positive taint inference* that we call software DNA shotgun sequencing (S^3). S^3 forgoes in-depth and expensive program monitoring to infer taint markings.
- We highlight weaknesses in taint-based detections of OS command injection attacks, which motivates the need for better program specifications.
- We present and evaluate a working prototype of S^3, which effectively thwarts OS command injection attacks. S^3 has essentially no performance overhead and operates on

binary programs, making it a practical, deployable solution to OS command injection attacks.

3.6.1 Threat Model

Before describing S^3 in detail, it is necessary to understand the threat model of OS command injections that positive taint inference addresses.

The threat model assumes software is intended to be benign, but also that it likely contains flaws. The program, when run, reads untrusted user input possibly from many sources such as files, environment variables, shared memory, or network sockets. The input is used to create commands that are issued to the OS. Most inputs to the program are benign and cause the OS command to behave as intended by the programmer, but malicious inputs may exploit the program flaw to violate the security policy intended for the OS command. An OS command injection occurs when attackercontrolled inputs change the programmer-intended syntactic structure of a command [202], [184]. Further, the program may be performance sensitive, and cannot tolerate high run-time overhead.

This threat model includes the common "remote attacker" model where malicious input is specified over the network to a server-type program. However, it also includes privilege escalation attacks where a local user attempts to gain additional privileges (such as root access), by providing a malicious command line, environment variable, etc. to a program.

3.6.2 Software DNA Shotgun Sequencing: High-Level Overview

Software DNA Shotgun Sequencing (S^3) is a technique inspired by genetic research [216]. In genetics, DNA shotgun sequencing breaks up very long DNA strands into short snippets, operates on (e.g., sequences) the snippets, and then recombines the results. Software DNA Shotgun Sequencing is similar in that we extract string fragments from a program, operate on them, and then later recombine them to validate some aspect of the program. Based on this idea, we invented the S^3 technique described in this paper. S^3 can thwart OS command injection attacks by matching the program's DNA fragments to the commands it attempts to issue. If commands cannot be matched, S^3 assumes that the DNA that has been injected into the program is potentially dangerous.

S^3 differs from traditional taint-based techniques in two fundamental ways:

- S^3 does not seek to propagate taint markings as a program executes. Instead, it uses *taint inference,* a concept introduced by Sekar [184].
- However, in contrast to Sekar, S^3 infers taint markings for trusted data instead of untrusted data. The emphasis on trusted data is referred to as *positive tainting* and was developed by Halfond et al. [103], [101].

S^3 combines taint inference and positive tainting. We use the term *positive taint inference* to distinguish our work from Sekar's negative taint inference technique.

3.6.2.1 S^3 Architecture

S^3 consists of five major compoments. The goal of the **DNA Fragment Extraction** component is to extract string literals, i.e, DNA fragments, from the binary and its associated libraries. This analysis is done once, prior to program execution, and the analysis time is not counted against the run-time overhead.

The **Command Interception** component intercepts security-critical commands so that they can be vetted.

The **Positive Taint Inference** component determines which characters in the intercepted command should be trusted by matching the command against the extracted DNA string fragments. Any unmatched character is deemed untrusted. Combining DNA fragments native to the protected binary to infer taint is a novel form of taint inference and one of the key contributions of the S^3 architecture.

The **Command Parsing** component parses the intercepted command to identify critical tokens and keywords.

The **Attack Detection** component combines the output of the Positive Taint Inference and Command Parsing component to determine whether an attack has occurred. A command is deemed an attack if a critical token or keyword is marked as untrusted.

Upon attack detection, S^3 either rejects the command outright and returns an error code, or it alters the command before passing it on to the operating system. The current prototype uses a simple form of error virtualization that simulates a failed command invocation by substituting an error code in place of the actual command [194], [192].

To illustrate how S^3 works, we use the following vulnerable program as a working example:

```
char *path = "/bin"; int main(int argc, char** argv) {
  char cmd[100];
  snprintf(cmd, 100,
           "%s/cat %s", path, argv[1]);
  system(cmd);
}
```

3.6.2.2 Example with Benign Input

```
/bin/cat README
BBBBBBBBUUUUUUU
CCCCCCCC-------
```

(a)

```
/bin/cat README; rm -fr *
BBBBBBBBUUUUUUUUUUUUUUUUU
CCCCCCCC-------C-CC-CCC--
               * ** ***
```

(b)

Figure 14. Sample command, with S^3's "blessed" and "critical" markings

When the program in the working example is passed a benign input such as "README", the resulting command is shown in the first line of Figure 14a. The Positive Taint Inference component annotates each character in the command (B denotes that the character is trusted or *blessed*, U denotes untrusted), as denoted by the second line of the figure. In this case, /bin/cat is trusted as it matches the composition of the DNA fragments "/bin" and "/cat" extracted in the offline DNA Fragment Extraction process. The Command Parsing component identifies critical tokens and keywords (C denotes critical), as shown in the third line of the figure. Lastly, the Attack Detection component takes as input the intercepted commands along with all annotations, and marks any critical command that is not blessed. Since all critical commands are blessed, the command is determined to be legitimate and is allowed to execute.

3.6.2.3 Example with Attack Input

Consider the malicious input README; rm -fr * that seeks to recursively delete user files. The resulting command is shown in Figure 14b. Like the other example, the Positive Taint Inference component annotates each character in the command. Again, only /bin/cat matches the extracted fragments. The Command Parsing component identifies critical tokens and keywords, like before, except that this time the semicolon, rm command, and the -fr flags are also detected, as shown on line 3. Lastly, the Attack Detection component is invoked, and detects that there are critical command characters that are untrusted (shown with asterisks on line 4 of Figure 14b). Since S^3 has detected the attack, an appropriate remediation technique can be applied. The program can be shut down or /bin/cat the command can be blocked or sanitized before allowing it to be passed to the operating system.

3.6.3 Software DNA Shotgun Sequencing: Detailed Overview

While the S^3 architecture is generic, e.g., it could be applied to web applications, we present details and discuss challenges encountered as we map S^3 into a practical instantiation to defeat OS command injection attacks for binary programs.

3.6.3.1 DNA Fragment Extraction

The accuracy of the fragment extraction process is crucial. If fragments are missed, valid commands might be flagged as injections. If extra fragments are extracted, malicious command injections might not be flagged (See Section VI-B for further discussion).

1) String extraction: Extracting string fragments from binary programs is more difficult than it first appears. Our first attempt used the Linux program strings, which linearly scans a binary program and extracts null-terminated sequences of ASCII characters that have a length larger than a given threshold. Unfortunately, short strings are sometimes important. Consider this C++ snippet:

```
string q = "rm ";
q += "-f ";
q += filename;
system(q.c_str());
```

which creates and executes an OS command.

Using strings, the threshold needs to be sufficiently low to find short strings. Unfortunately, low thresholds tend to yield lots of garbage strings, which affects accuracy. Furthermore, compilers use many optimizations that can make strings harder to detect. For example, to initialize a string on the stack, a compiler might use a sequence of store instructions:

```
mov [esp+28], 0x2d206d72 # "rm -"
mov [esp+32], 0x00002066 # "f \0\0"
```

Each move stores four bytes onto the stack, ultimately creating the proper null-terminated string. Other compiler idioms may complicate accurately finding all strings, as well. We have seen examples of the compiler inlining some standard library functions that have constant operands, such as memcpy(dst,"rm -f ", 6). This optimization yields inlined constants much like the previous string initialization example. Lastly, strings reports all strings in the executable file, which can include debug information, shared library names, compiler-version identifiers, etc. As these types of strings cannot be used to form OS commands, they should be excluded from consideration.

To deal with these issues, we use static analysis of the program to derive the string fragments. The static analysis starts by fully disassembling the program into a database which holds each instruction in the program, indexable by address, function, and control flow information. We use a hybrid linearscan disassembler and recursive-descent disassembler to ensure we get good coverage of all instructions, as described by Hiser, et al. [107], [106].

```
3:   Usage: spamass-milter -p socket
     [-b|-B bucket] [-d xx[,yy...]]
     [-D host]
11:  popen: failed(%s). Will not send a copy to spambucket
33:  recipients; spamc gets default username
34:  ). Will not send a copy to spambucket
46:  Could not extract score from <%s> 56: error. could not replace
     body.
57:  Could not extract score from <
173: popen failed(
273: hX
274: hx
279: h8
281: h@
282: h(
287: ><
286: @+
288: Z
304: 0
305: 8
306: @
307: (
308: "
309: .
310: /
311: :
312: ’
313: _
314: >
315: ‘
```

Figure 15. Sample fragments manually extracted from SpamAssassin Milter Plugin (28 shown out of 315 fragments total)

After disassembly is complete, the instructions are scanned for accesses or creation of string values. We analyze each instruction’s immediate operands and apply three heuristics to identify string fragments:

- Check if the immediate value holds the address of a program location and the location is the beginning of a sequence of printable characters or one printable character terminated by a null byte.
- Check immediate values to see if they contain a string fragment. Attempt to combine immediate values of sequential instructions to form one string fragment. This heuristic handles the case of strings constructed via sequential store instructions, as described in the previous example.

- Check immediate values for PIC-relative addressing that might point to a string in PIC code.

Finally, we check for other string fragments or string pointers in data sections.

2) Post-processing of DNA Fragments: Programs compiled from C or C++ often contain statements that use format specifiers, e.g. %d, %f, %s, %x. We split such fragments into their constituent sub-fragments using the format specifiers as delimiters. A fragment such as

```
"/bin/rm -f \%s; /bin/touch \%s"
```

would be split into the sub-fragments "/bin/rm -f ", and "; /bin/touch". As the analysis cannot be sure that such fragments are used as format strings, the original fragment as well as the sub-fragments are retained in the list of signatures.

Figure 15 shows a representative sampling of the DNA fragments from the Spam Assassin program. The length of the fragments range from 1-111 characters. Because of the %s specifier, Fragment 11 expands into sub-fragments 34 and 173. Likewise, fragment 46 expands into fragment 57 and 314. Fragments 273–282 are likely spurious and result from the inherent imprecision of static analysis on binaries.

Astute readers will notice the short fragments that contain potentially dangerous shell metacharacters (fragments 306–315), or short fragments that could be composed in an attack (fragments 273-288). In Sections IV-E and VI, we discuss how the S^3 policies deal with short and potentially dangerous fragments.

3.6.3.2 Command Interception

For binaries derived from C/C++, commands are typically encapsulated in an Application Programming Interface (API) and accessed via dynamically-linked shared libraries.

The S^3 prototype leverages standard library interposition facilities to transparently intercept and wrap function calls to the underlying operating system. S^3 intercepts the system, popen, rcmd, and exec family of functions. Other functions could obviously be intercepted as well, but we have identified these as the primary candidates for OS command injection.

3.6.3.3 Command Parsing

This component is responsible for identifying the security critical parts of a command. For OS commands, the critical parts consist of command names, options, delimiters, and the setting of environment variables.

The S^3 prototype uses a simple, combined lexical analyzer and parser. The parser is careful to identify special characters which could indicate the start of a new command (such as the semicolon character), match quotation marks and parentheses, etc. Ideally, one would use a full, formally-verified lexical analyzer and distinct parser to detect keywords, etc. However, it is impossible due to the nature of the shell language (bash in our case). Consider this command:

```
echo Touching ${file}; touch `foobar`
```

What are the “correct” lexical analysis and parse for this command? The answer depends on the value of the file variable and the output of the foobar executable. If file is set to a single quote character and foobar returns the same thing, then there is exactly one command, echo. Since

<commandName>
;[\s]<commandName>
$(<commandName>
||[\s]<commandName>
&&[\s]<commandName>
‘<commandName>
<environmentVar>[\s]=
-<optionFlags>
--<optionFlags>

Figure 16. Attack detection policies using the same fragment origin policy ([\s] denotes an optional whitespace).

```
/bin/cat README; rm -fr *
BBBBBBBBBUUUUUUUUUUUUUUUU
CCCCCCCCC      C CC CCC
```

Figure 17. Overlapping policies to detect attacks.

variables are expanded and sub-processes are executed before the command is parsed, the correct parse cannot be determined *a priori*. Under most circumstances, though, such odd substitutions are not the case.

For the purposes of detecting OS command injections, we need to know the possible places where a command could be invoked. Our simple parser assumes that the structure of the command is not changed by the results of executing subcommands. In the case of our simple example, the parser marks the command like so:

```
echo Touching ${file}; touch `foobar`
CCCC          CCCCCCCC CCCCC CCCCCCCC
```

where `C` indicates that a critical command character exists at the given location.

3.6.3.4 Positive Taint Inference

Conceptually, the Positive Taint Inference component infers which portions of the command come from within the program, and which ones come from external sources. To accomplish this step, it checks each substring in the command to determine if that location is within the set of DNA fragments. This pseudocode illustrates the process:

```
for each DNA fragment, f
  for each position, i, in the command
    l=len(f) i
    if f==command[i .. i+l-1]
      mark_blessed(command[i .. i+l-1]);
```

Conceptually, this algorithm could be quite expensive, $O(n^3)$ where $n = max(len(sig), \#sigs, len(command))$. In practice, though, we use a move-to-front heuristic to organize the DNA fragments required to trust commands and exit the outermost loop when enough of the command is trusted to verify its safety. Further, each command and each DNA fragment is typically short, on the orders of tens or hundreds of characters. These simple observations and adjustments dramatically reduce the time necessary to make the inference.

3.6.3.5 Attack Detection

Attack detection consists of scanning the command for any character that has been marked as untrusted by the Positive Taint Inference component and critical by the Command Parsing component. In addition, we impose the constraints shown in Figure 16 that command names, shell metacharacters used for starting subcommands and their associated command names, option flags, and environment variable names must come from a single DNA fragment (*same fragment origin policy*).

This policy helps to compensate for the case when a short, critical token, such as a semi-colon or a quotation mark, is present in the set of DNA fragments. Such fragments allow attackers great latitude to create strings that append new commands, as in "; rm -rf". Unfortunately, these DNA fragments cannot simply be discarded, because many programs do use such fragments to terminate their commands. However, it appears uncommon for a program to use such fragments to introduce a new command, so we disallow this behavior entirely.

Figure 17 illustrates how these policies provide overlapping means to detect attacks. The core policy of checking for untrusted critical characters (shown in boldface red) is augmented with the same fragment origin policy (shown with rectangles). Note that rm is covered by three separate policies. Thus, even if ; and rm were somehow both extracted as fragments, the attack would still be detected correctly.

When no attack is detected, S^3 passes the command to the operating system to execute. However, if an attack is detected, S^3 does not pass through the command, but can enact any one of a variety of remediation responses, such as shutting down the program, warning the user and asking for permission to continue, or logging the attack. For the prototype described in this paper, we chose to return an error code as if the library call had failed. This policy makes sense in many cases, as well-written programs are designed to gracefully handle error conditions.

3.6.4 Related Work

We focus our discussion on software-based, run-time defensive techniques.

3.6.4.1 Taint tracking

1) Taint tracking in Managed Runtimes: Livshits provides an extensive review of dynamic taint tracking projects [129]. Most projects use a form of negative taint tracking, i.e., these projects keep track of external (untrusted) data as it flows through a program, and check whether such data is used in a security-sensitive operation [100], [149], [162], [221], [101], [101], [56]. The notable exception is the WASP project by Halfond et al. which uses positive taint tracking to keep track of internal (trusted) data [101]. The primary tradeoff is that positive taint tracking favors false positives (breaking application functionality) whereas negative taint tracking favors false negatives (missing attacks). Halfond advocates the use of positive taint tracking as it provides a more conservative security posture.

Unfortunately, both positive and negative taint tracking are seldom used in practice. Perl and Ruby are the only two major languages that we are aware of that provide support for dynamic taint tracking out of the box [21], [18]. Several projects modified the PHP run-time engine to support taint tracking at the level of individual characters [149], [162], [90]. To avoid modifying the PHP run-time engine, PHP Aspis applies source code transformations selectively to only parts of a web application. This scheme maps well to extensible applications whose core is well-maintained but where the quality of thirdparty plugins is unknown [155], [23]. Despite the selective application of taint markings, Aspis incurs high overhead of 2.2X on Wordpress. Haldar et al. provided coarse-grained taint tracking for Java strings [100]. Chin and Wagner implemented taint tracking at the level of characters for Java [56]. In general, fine-grained approaches to taint tracking result in higher precision and fewer false positives.

2) *Taint-tracking for Binaries:* The overhead numbers reported for the systems highlighted below are illustrative of the rapid rate of progress in reducing the overhead of taint-tracking techniques on binaries. However, they should not be directly compared to one another as the benchmarks and hardware used vary across these projects.

TaintCheck, one of the early pioneering projects for using taint tracking to detect memory-overwriting attacks on binaries incurred overhead as high as 37X for CPU-bound applications, and from 2.5X to 25X for I/O bound workloads for a typical web server [148]. TaintCheck was built on top of Valgrind, a flexible but relatively slow dynamic binary rewriter [145]. The LIFT project achieved overhead of 3.6X on several SPEC INT2000 benchmarks and 6.2% on server applications [165]. The order of magnitude improvement resulted from several optimizations, including using a more efficient binary rewriter, eliminating instrumentation on provably safe code paths, coalescing checks and reducing the overhead of context switching between the application code and the dynamic binary rewriter. Bosman et al. reported overhead of 2.4X for SPEC INT2006 and 1.5X-3X for real-world applications using an emulator custom-built for taint analysis [46]. Unlike the previous approaches, Saxena et al. use static rewriting as the mechanism for instrumenting binary code [179]. They reported average overhead of 1.95X on several CPU-intensive SPEC95 INT benchmarks. Dytan [63] and libdft [120] incorporate years of experiences with taint tracking to provide easily customizable and generic taint analysis frameworks.

Despite steady and impressive progress in improving the performance of taint-tracking techniques, our stringent overhead requirements (< 1% on binaries [140]) led us to bypass taint-tracking techniques altogether.

3.6.4.2 Taint Inference

S^3 was heavily influenced by Sekar's taint inference technique for protecting web applications against command injection attacks [184]. Sekar's insight of establishing taint markings by correlating inputs to observable commands obviated the need for taint tracking and was the key to enabling practical performance. Instead of inferring taint markings for untrusted data, S^3 seeks to infer trusted data used in OS commands. To highlight this fundamental difference, we view S^3 as an embodiment of *positive* taint inference, in contrast to Sekar's use of *negative* taint inference.

3.6.4.3 Model-based Approaches

Christensen et al. perform static analysis to model possible string values at any point in a Java program [61]. The model extracted represents an over-approximation of the programmers' intended specification for benign commands. The AMNESIA project leverages these models to detect and prevent SQL injection attacks [102]. We believe that AMNESIA can be extended to cover OS command injections. The overhead reported on a set of Java web applications was negligible.

String analysis for binaries is much more challenging as binary code does not retain as much type information as Java byte code. Christodorescu et al. modeled strings for x86 binaries, though the precision of the analysis is limited by the lack of interprocedural analysis [62]. Sophisticated memory-analysis techniques such as Value-Set Analysis (VSA) could also be applied to string extraction [169]. However, it seems likely that string extraction requires abstract domains that are designed for reasoning about strings. VSA uses an abstract domain based on reduced interval congruences which is excellent for reasoning about (strided increments of) pointer values, but likely to lead to imprecise representation of string values.

By extracting and allowing for the arbitrary combination of string fragments, S^3 makes a conscious tradeoff between model complexity and model accuracy. S^3 combines a very simple (but over-approximated) string model with additional policies based on the origin of string fragments for its attack detection policies.

3.7 C6: Concurrency Errors

Concurrency errors are bugs that manifest (usually rarely) when certain thread or process scheduling conditions are met. For our category of STONESOUP, binaries, the T&E team determined that there were 17 CWEs that pertain to concurrency errors. We then grouped these into four broad categories:

- File system TOCTOU vulnerabilities
- Deadlocks
- Signal handler errors
- Atomicity violations

We designed and implemented four detection and mitigation approaches to cover these categories. There were two CWEs (412 and 765) which do not fall neatly into any of these categories. Below we discuss why this is the case, and possible mitigations for these CWEs. We were unable to complete a solution for CWE-412, and CWE-765 was only partially covered by our solutions.

Table 1 shows all the CWEs in scope for C6, and lists which category each CWE goes in.

Table 1: Categorization of Concurrency-Related CWEs

Category	CWE	Description
Other	412	Unrestricted Externally Accessible Lock
	765	Multiple Unlocks of a Critical Resource
Atomicity Violation	367	Time of Check Time of Use (TOCTOU) Race Condition
	414	Missing Lock Check
	543	Use of Singleton Pattern Without Synchronization in a Multithreaded Context
	567	Unsynchronized Access to Shared Data in a Multithreaded Context
	609	Double Check Locking
	663	Use of a Non-reentrant Function in a Concurrent Context
	820	Missing Synchronization
	821	Incorrect Synchronization
File System TOCTOU	363	Race Condition Enabling Link Following
Signal Handling	479	Signal Handler Use of a Non-reentrant Function
	828	Signal Handler with Functionality that is not Asynchronous-Safe
	831	Signal Handler Function Associated with Multiple Signals
Deadlock	764	Multiple Lock of a Critical Resource
	832	Unlock of a Resource that is not Locked
	833	Deadlock

In each of the following sections we describe some relevant background, our design and implementation, and our evaluation of each component of PEASOUP's concurrency mitigations.

3.7.1 Unhandled CWEs

3.7.1.1 CWE-765: Multiple Unlocks of a Critical resource

We handle the mutex case (also, CWE-832) with the deadlock tool. The case for counted resources, such as semaphores, is more complex. Semaphores can either be used to count up (e.g., for signaling units of work available between threads) or count down (e.g., for limiting the number of connections to a shared resource). In the latter case, there is a semaphore initialized with a non-zero value C, and it should never exceed the value C. If one of the threads does a multiple unlock then the value will exceed C, and we may be able to detect this. The other case, counting up, is essentially using a semaphore as a signaling mechanism. Most code is robust to extra signals; if not, it is not clear how to infer that a sent signal is "extra."

We did not implement any functionality to handle this CWE.

3.7.1.2 CWE-412: Unrestricted Externally Accessible Lock

An unrestricted externally accessible lock is one whose state (locked or unlocked) can be controlled by an attacker. This category of attack is most relevant to server software, where denial of service is a possibility. In many ways this functionality fits into the "Resource Drains" categorization better than concurrency errors. The issue is not that concurrent access is causing errors, but that a resource is being hijacked by an untrusted user or process.

We did not implement any functionality to handle this CWE. Adapting the resource drains approach by tracking suspicious callstacks that lead to a lock that is held for a significant amount of time, may be a viable solution.

3.7.2 File System TOCTOU

3.7.2.1 Background

File system Time of Check to Time of Use (TOCTOU) vulnerabilities occur when two operations reference the same file system path, with the intention that the underlying file system object (e.g., inode on Linux) is also the same. This is ultimately a name resolution vulnerability [208], as the vulnerability manifests when the same name resolves to different objects. Table 2 shows a simple example.

Table 2: Code snippet demonstrating a potential file system TOCTOU vulnerability.

```
1   // Open the file
2   int fd = open(“myfile.txt”, “w”);
3   // Write something
4   write(fd, mydata, mydata_size);
5   // Give everyone full permissions on the file
6   chmod(“myfile.txt”, 511);
7   // Close the file
8   close(fd);
```

In this example “myfile.txt” is opened, data is written to it, and then permissions on the file are changed so that any user on the system can read, write, or execute it. Clearly the intention is that the permissions are changed on the same object that was opened and written. However, there is a race condition between lines 2 and 6, where an external actor can change the file system object that “myfile.txt” refers to. A common attack would be to change “myfile.txt” to be a symbol link to a sensitive file that most users cannot access (like */etc/shadow*). If this symbolic link can be created after line 2 executes and before line 6 executes, all users will be able to read and change the sensitive file.

In the standard terminology the first operation is referred to as the “check” and the second operation is referred to as the “use”. In our example, *open*() is the check operation and *chmod*() is the use operation.

Unlike many other vulnerability classes, file system TOCTOU vulnerabilities are highly dependent on the system environment. The code snippet in Table 2 could be perfectly safe if it is executed in a protected sub-directory on the file system; the attacker will be unable to create a symbolic link in that case.

Statically identifying potential file system TOCTOU vulnerabilities is feasible; for example, CodeSonar[13] can flag such bugs in both C/C++ source and x86 binaries. However the False Positive (or “Don’t Care”) rate can be high, because many of the races will be unexploitable given the actual environment the program will run in.

[13] CodeSonar is GrammaTech’s commercial bug finding tool; CodeSonar is not a part of the PEASOUP tool chain.

For this reason, we focused on purely dynamic approaches. One approach we looked into was the use of the "safefile" API described by Kupsch and Miller [124]. Their approach validates the file system objects that each component of the point resolves to. However, there are two fundamental issues with using their approach in PEASOUP:

1. Their validation requires some notion of "trusted path". A prerequisite for using this approach is then to provide a table listing the trusted paths for a given program.
2. They require that the last component of a file path is *not* a symbolic link. This assumption breaks many existing programs/configurations.

In general, the "safefile" API is applicable to new development (i.e., you design your software to make use of it) rather than as a vulnerability mitigation tool applied after the fact.

3.7.2.2 Our Approach

Our approach is dynamic: we implemented a library which gets injected into the application at runtime, and hooks POSIX file system API calls. We track metadata about these calls (e.g., the filename to inode mapping that was used) and use it to either fail the function call or to mitigate the race by avoiding name resolution altogether.

We separated the Linux file system time-of-check to time-of-use (TOCTOU) vulnerabilities into four categories:

1. ***Privilege-droppable*** pairs: access()/open() (and creat(), etc.) vulnerabilities in setuid programs.
2. ***FD-checkable*** pairs: there exists a safe file descriptor version of the use operation, and the check operation generates a file descriptor.
3. ***Double-checkable*** pairs: the check operation is a stat() and the use operation generates a file descriptor (e.g., open()).
4. ***Other*** TOCTOU pairs

We used a similar breakdown of TOCTOU pairs as in [214], but we expanded the pairs to include modern and less portable functions that were not in [214]. Currently we have a breakdown of 222 unique pairs. [214] claimed 224, but when we reproduced their sets we discovered they were counting some pairs multiple times. Of these potential TOCTOU pairs, many are probably unexploitable or unlikely to be found in many programs. For example, the pair (*chmod*(), *chmod*()) is unlikely to be exploitable by itself – it requires some assumptions prior, since the "check" operation is really only checking that the program has permissions to perform the *chmod*(), and that the given file system node exists. I.e. the vulnerability in such a case is not a TOCTOU, it is most likely a missing permissions check.

Below we provide some brief examples of our current TOCTOU mitigations.

Privilege-droppable

The *access* function checks whether the calling process can access a file using the programs *real* user ID and group ID. It is usually paired with a call to *open*, to open the file if the permissions check succeeds. Since both functions use the file name (path really), there is a TOCTOU race here.

There have been attempts in the literature to devise probabilistic solutions to this problem (namely [76] and [205]), but also attacks that successfully *always* win races against them ([45] and [50]).

Linux and other modern POSIX systems support saved set-user-IDs, which means there is a simple work-around for this problem: drop the privilege to the real UID/GUID before calling *open* and then re-elevate privileges.

We recognize the pair (*access*(), *open*()), and prior to the *open*() operation we drop the process privilege and then re-escalate the privilege upon completion. This approach is not portable across all Unix variants, but does work for all the modern Unix-like operating systems. The Linux manual page for *access*() recommends this approach in lieu of using the *access*() function.

FD-checkable

We recognize that given the pair (*creat*(), *chmod*()), we can save the file descriptor from *creat*() and rewrite the *chmod*() to be an *fchmod*() at runtime.

Double-checkable

Any pair that starts with *stat*() and ends with a file descriptor creation, like (*stat*(), *open*()), can be "double checked" by doing an additional *fstat*(). The inode and device of the *stat*()/*fstat*() calls are then compared, and the *open*() call is forced to fail if they mismatch.

Other Pairs

Many of the remaining pairs are either not exploitable (like the *chmod(),chmod()* example given above) or at least seem less common based on many of the bugs referenced in the literature [214] [208].

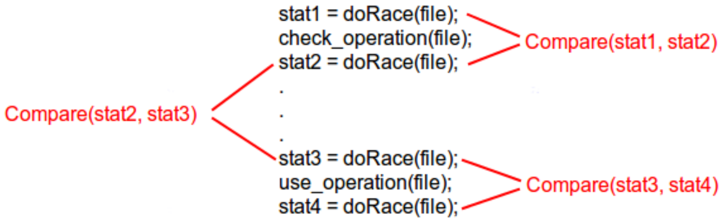
For mitigating races on these pairs, we take inspiration from the work done on portably preventing races between the *access*(),*open*() TOCTOU pair. As mentioned above, we have a mostly portable solution that deterministically prevents *access*(),*open*() races. However, the k-race approach from [76] can be extended to apply to any TOCTOU pair, not just *access*(),*open*(). Furthermore, the attack against this approach that was described in [45] is no longer deterministic on modern Linux kernels.

Determistically attacking *k*-races requires generating a file-maze: a huge set of nested directories, where the last component of the path is a symlink to another set of nested directories. Requiring a file system operation to go through a file maze slows it down noticeably. In order to get determinism, the attacking process monitors the *access time* attribute on the file mazes. However, modern Linux kernels mount file systems with *noatime* by default, meaning the *access time* attribute is not updated. Even if you manually mount a file system with *atime* enabled, the kernel only updates *access time* at very large intervals. The deterministic attack against *k*-race is no longer possible.

It is still possible to probabilistically attack a *k*-race-protected TOCTOU pair, and file system mazes help increase the odds. However, that probability drops drastically as *k* increases.

The original *k*-race approach was specific to *access*()/*open*() pairs. It performed extra calls to *fstat*() on the file descriptor associated with the *open*() result. The inode and device of the resulting "struct stat" object were then compared for each of *k* iterations; any mismatch resulted in failure. This approach requires that the ‘use’ operation in the TOCTOU pair generate a file descriptor, and so is not directly applicable to all pairs.

We extend *k*-race to all TOCTOU pairs by performing 4**k* races, as shown in Figure 18.



```
stat1 = doRace(file);
check_operation(file);
stat2 = doRace(file);
.
.
.
stat3 = doRace(file);
use_operation(file);
stat4 = doRace(file);
```

Figure 18: Extending k-race to arbitrary check/use TOCTOU pairs.

The function *doRace*() takes a file system pathname and returns a stat object containing inode, device, and other metadata information. *doRace*() performs *k* calls to *stat*() and compares the results: if any of the results are different, the function fails, and we force the file system operation (either the check or use operation) to return a failure. We then compare the results *between* *k*-races as well, since we don’t have a file descriptor that guarantees that we are calling *stat*() on the same file system object that the check or use operation are using.

Leaving out any of the four *k*-races leaves a hole where an attacker can possibly win a single race to exploit the TOCTOU. The only time we leave out some of the *k*-races is when one of the operations generates a file descriptor, where we can simply use *fstat*() instead of a *k*-race.

Note that we are not comparing for strict equality of all file object metadata properties. The check or the use operation may mutate some of those properties, and we take that into account in our comparison.

3.7.3 Deadlocks

3.7.3.1 Background

Approaches in the literature for preventing or mitigating deadlock mostly fall into two categories:

1. Statically re-ordering locks in a way that is provably deadlock-free, e.g. Gadara [212]. This is not always a completely automatic process.
2. Dynamically detecting a deadlock and preventing it from occurring again. Dimmunix [117] is the only example we are aware of for this type of approach.

A dynamic approach is more in line with the rest of the PEASOUP tool-chain, and Dimmunix has the advantage of being completely automated. Dimmunix is a system that maintains a resource allocation graph (RAG) at runtime to detect deadlock. It remembers which function call stacks, and in what order, lead to deadlock and tries to avoid those orderings in the future. A resource allocation graph is only applicable for tracking *boolean resources*, such a mutexes. It does not work for counted resources like semaphores.

The majority of deadlock detection/prevention approaches deal with *resource deadlocks*, such as deadlocks when acquiring mutexes. The distributed database literature also deals with *communication deadlocks*, which are deadlocks that can occur due with message passing systems. The difference is that in a resource deadlock the processes/threads (entities) are waiting to acquire a resource that another entity holds. In a communication deadlock, an entity is waiting for a message that will never come.

Traditionally, the operating systems literature has dealt with resource deadlocks. The classical definition of a resource deadlock is that it requires four conditions to hold [195]:

1. Mutual exclusion: the resource(s) in question cannot be simultaneously acquired by two threads.
2. Hold and wait: the threads must hold at least one resource and be willing to wait indefinitely for another resource.
3. No preemption: no thread can preempt a resource from another thread.
4. Circular wait: each thread involved in the deadlock must be waiting for a resource held by another thread in the deadlock, in a circular fashion.

We briefly looked into handling communication deadlocks, specifically for two common synchronization idioms: conditional variables (e.g., pthread_cond_wait()) and semaphores that count up and are used for signaling. However, as we describe in the next section, we only implemented support for resource deadlocks.

3.7.3.2 Our Approach

Our approach is dynamic: we implemented a library which gets injected into the application at runtime, and hooks pthreads synchronization operations. Our approach is very similar to Dimmunix in that it tries to remember deadlocks that it has seen and avoid them the next time. However, our approach improves upon Dimmunix in a few key ways:

- We use a variant of the Banker's Algorithm instead of a resource allocation graph, to support counted resources.
- We monitor the deadlock state *within each application thread*, instead of externally. This allows us to mitigate the deadlock on the first time we see it, without terminating the program.
- We use dynamic semaphores to prevent a deadlock from re-occurring, which guarantees that the same deadlock will not occur twice, and also (we believe) prevents some of the starvation problems that Dimmunix could suffer.

We hook all pthreads and POSIX synchronization APIs, and track resource requests, allocations, and releases. For example, consider the sequence shown below.

Thread 1	Thread 2
`pthread_mutex_lock(m1); // acquires`	
	`pthread_mutex_lock(m1); // waits`
`pthread_mutex_unlock(m1)`	
	`pthread_mutex_unlock(m1);`

Thread 1 *requests* one element of lock m1 (since it is a mutex, there is only one element) and no one else holds it, so it successfully *allocates* m1. Thread 2 then *requests* m1 and blocks. Thread 1 then *releases* m1, allowing Thread 2 to unblock and *allocate* m1, and then finally Thread 2 *releases* m1.

We track the current request, allocate, and release operations for each thread and resource (e.g., mutex) in the process. We then use the deadlock detection algorithm based on the Banker's Algorithm [64] to determine if the process is in a deadlocked state. When a deadlock is detected we force the operation to fail gracefully. For example, if the last operation that occurred to cause deadlock a pthread_mutex_lock(), we will return a non-zero (error) value from this function. Ideally, the process would expect potential errors from these synchronization APIs and handle them properly. In practice, very few open source programs that we have looked at do handle errors from pthread properly. For this reason, we additionally implement deadlock avoidance, which lets us protect against known (previously seen) deadlocks.

We avoid known deadlocks by relying on information learned in the online "learning" phase (where we are run with a few good and bad inputs). In this phase we detect deadlocks and save pertinent information about the order and the callstacks[14] of operations that led to the deadlock.

For example, Figure 19 shows the simplest deadlock scenario. There are two mutexes, *A* and *B*. Two threads acquire these locks in different orders, and if the acquisitions overlap in time (i.e., T1 gets A before T2 asks for A, and T2 gets B before T1 asks for A), deadlock occurs.

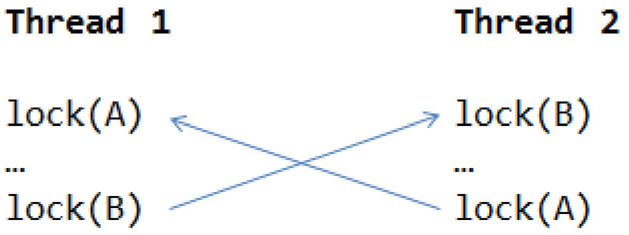


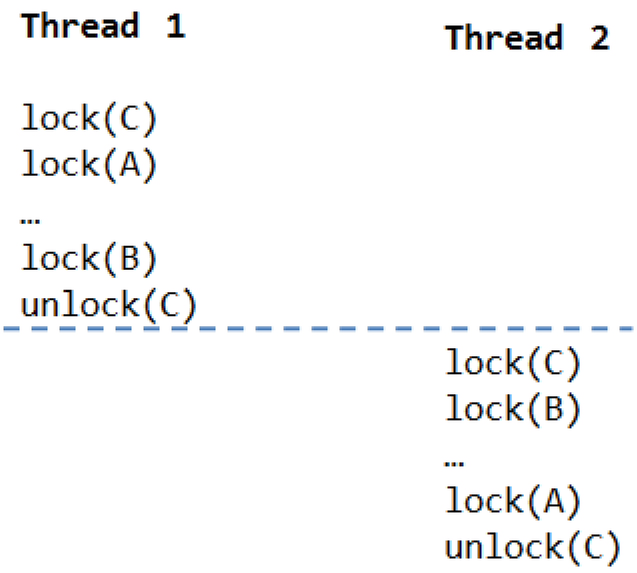
Figure 19: Simple deadlock scenario

The high-level idea of our mitigation is that we can prevent such deadlocks by introducing another mutex *only for this scenario*. We will save the calling context of the lock operations and their orders, and when we see execution which matches that pattern we will dynamically enable the new lock (called *C* in our example). This saved information is a *deadlock trace*. Figure 20 below shows how adding an additional lock for just this scenario creates mutual exclusion between the potentially poorly ordered locking operations. This approach is similar to Dimmunix [117], except that we explicitly use dynamic locks instead of the thread suspension approach Dimmunix used.

There are a variety of choices for what to include in deadlock traces. On one hand we can track very coarse information, such as the calling context and thread of each lock operation involved in the deadlock. This coarseness has the advantage of protecting against variations on a deadlock that we haven't previously seen, but the disadvantage of potentially slowing the program down if the involved calling contexts are very frequently used. This trade-off can be skewed the other

[14] We actually just save a hash of the last *K* functions on the callstack. By default *K*=5.

direction by adding more information to the trace, such as intermediate unlock() operations or other common system calls and their contexts.



```
Thread 1              Thread 2

lock(C)
lock(A)
...
lock(B)
unlock(C)
- - - - - - - - - - - - - - - - - - - -
                      lock(C)
                      lock(B)
                      ...
                      lock(A)
                      unlock(C)
```

Figure 20: Potential deadlock mitigated by an additional lock

Note that we do not use dynamic mutexes in our implementation; instead we use counting-down semaphores. When we have a deadlock with *N* threads involved, they form a chain of operations: each thread is in a hold-and-wait state. In order to prevent such an *N* thread deadlock we only need to break one link of the chain. Our avoidance strategy generalizes to this *N* thread scenario by making use of a count-down semaphore initialized to *N-1*. This allows *N-1* of the problematic threads to obtain whatever resources they want, but prevents the *N*th thread from doing so simultaneously, thus avoiding the deadlock.

Our deadlock tool also handles CWE-764 (Multiple Locks of a Critical Resource) and partially handles CWE-765 (Multiple Unlocks of a Critical Resource). CWE-764 simply manifests as a single-threaded deadlock, so our normal algorithm applies. CWE-765 is handled for binary resources and semaphores that are used to count-down. Whenever we detect a deadlock and force a lock() operation to fail we are in the scenario where an extra unlock could occur, so we must handle it. We do so by tracking the failing lock() operation and watching for a subsequent unnecessary unlock, which we ignore.

3.7.4 Signal Handler Errors

3.7.4.1 Background

Coding errors related to signal handlers often result in deadlocks or memory errors, but can sometimes result in more subtle application-specific problems. The associated CWEs (479, 828, and 831) attempt to describe some common causes of signal handler problems. Certain static analysis tools, like CodeSonar, attempt to flag problems like data races that might occur inside of signal handlers. We did not find any dynamic approaches to signal handler errors in the literature.

3.7.4.2 Our Approach

We use a primarily dynamic approach to mitigating signal handler errors. Note that many signal handler issues will manifest in ways that other PEASOUP mitigations already handle, e.g. memory errors. However it is also important to realize that dynamic approaches to other errors can be thwarted by signals if they get interrupted during mitigation activities.

All three signal handling error CWEs are handled by a simple concept we're calling signal buffering. We have a core component that hooks all signal handlers, and exposes two functions in an API: *buffer_signals()* and *flush_signals()*. When *buffer_signals()* is called all signals sent after that point will be put on a queue instead of being handled by the processes handlers. When *flush_signals()* is subsequently called, all those queued signals will get sent to the appropriate threads.

This general mechanism allows us to mitigate the three CWEs in the following ways:

- CWE-479, "Signal Handle Use of a Non-reentrant Function": hook common non-reentrant POSIX functions and buffer signals around the original functionality. E.g., printf() buffers signals before calling the actual printf() implementation, so no deadlock can occur.
- CWE-828, "Signal Handler with Functionality that is not Asynchronous-Safe": we statically identify atomic sets of variables that may be accessed from other threads and the signal handler. We then surround accesses to these variables by *buffer_signals()* and *flush_signals()* to prevent atomicity violations.[15]
- CWE-831, "Signal Handler Function Associated with Multiple Signals": since we are hooking all signal handlers we can dynamically identify when multiple signals share a handler. These handlers are then wrapped by *buffer_signals()* and *flush_signals()* to prevent adverse interactions between the handler and itself.

There are certain signals that we cannot handle in this way, such as SIGSEGV and SIGFPE. These signals are caused by faulting instructions which are re-executed after the handler completes. Buffering these signals would result in an infinite loop.

In addition to buffering the signals themselves, we need to buffer changes to signal handlers. For example, if the program does something that causes signals to be buffered and then calls either *sigaction*() or *signal*() to change the handler for that signal, we need to make sure that signals are processed by the correct handler.

Consider the following sequence of operations in a program:

[15] We were unable to complete implementation of this functionality for the PEASOUP deliverable.

```
sigaction(1, handlerA)
send signal 1           // Handled by handlerA
sigaction(1, handlerB)
send signal 1           // Handled by handlerB
sigaction(1, handlerA)
send signal 1           // Handler by handlerA
```

We will get the function call sequence {*handlerA*(), *handlerB*(), *handlerA*()} when executing this program. Now consider the same program where we are currently buffering signals to mitigate potential vulnerabilities.

```
buffer_signals()
sigaction(1, handlerA)
send signal 1           // Put on queue
sigaction(1, handlerB)
send signal 1           // Put on queue
sigaction(1, handlerA)
send signal 1           // Put on queue
flush_signals()         // Send the three signals
                        // in the same order.
```

If we only buffer the signals and not the changes to the handlers, *flush_signals()* will use the last handler (*handlerA*) for all three signals. This can cause altered functionality in certain programs, such as gdb.

The correct thing to do is put the signal handler change information on the queue, so that when we flush the queue we process the changes in signal handlers, and get the correct function call sequence {*handlerA*(), *handlerB*(), *handlerA*()}.

As mentioned above, one of the ways that we handle CWE-479 ("Signal Handle Use of a Non-reentrant Function") is by signal buffering around common non-reentrant POSIX functions. This CWE can also manifest with user-written functions that use mutexes and are called from signal handlers. The deadlock tool can handle these situations, *assuming that it does not get interrupted by a signal* while it is detecting or mitigating deadlocks.

For this reason, we ended up integrating the deadlock and signal handling concurrency components. This integration was motivated by the example illustrated in Figure 21.

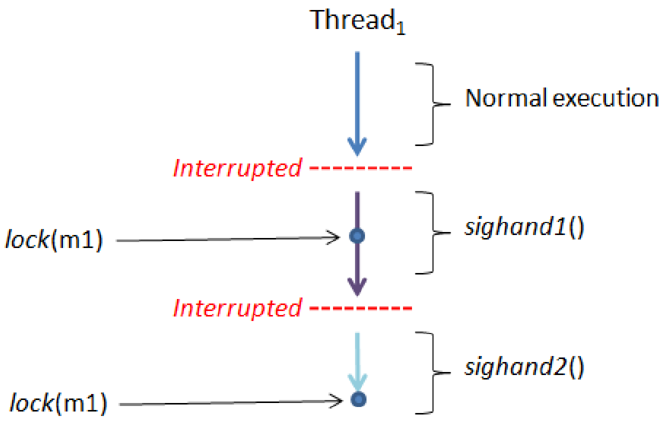


Figure 21: Deadlock with a single thread and interrupt handlers.

As Figure 21 shows, there is a single thread in this scenario. That thread gets interrupted by a signal, and *sighand1*() starts executing. Then that signal handler gets interrupted by another signals, and *sighand2*() starts executing. Both *sighand1*() and *sighand2*() lock mutex *m1*, so a deadlock ensues. This bug falls under both CWE-833 (Deadlock) and CWE-479 (Signal Handler Use of a Non-reentrant Function).

Now consider the same scenario with the deadlock detection involved. When the operation *lock(m1)* occurs, the deadlock tool intercepts it and does some extra bookkeeping needed for deadlock detection. If the deadlock tool's bookkeeping is interrupted by the second signal, it will be unable to prevent this deadlock. This is illustrated below in Figure 22.

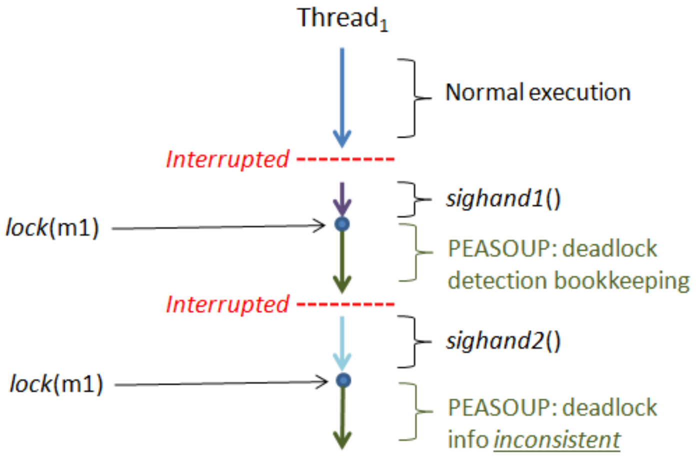


Figure 22: Deadlock with signal handlers; PEASOUP's deadlock detection gets interrupted

The solution to this problem is to have the deadlock detection and signal buffering integrated. Whenever the deadlock detection needs to intercept a *lock()* operation it turns on signal buffering so that it cannot be interrupted.

3.7.5 Atomicity Violations

3.7.5.1 Background

Many of the CWEs associated with concurrency errors are due to improper or missing mutual exclusion for code that is executed in parallel. These types of errors are typically characterized as either data races or atomicity violations.

A data race occurs when two concurrency threads access the same memory where one access is a write and there is no explicit mechanism to prevent “simultaneous” access. An atomicity violation mostly subsumes the concept of a data race. An atomicity violation occurs if a section of code that was intended to be serial is interleaved in a way that causes a bug or alters functionality in an undesirable manner.

An atomicity violation may be a data race on a single memory location (variable), or it may be a bad interleaving between multiple memory locations. In the latter case, these memory locations are said to form an atomic set [104]. Memory locations in an atomic set that are read or written together must be done so atomically. This is a slightly fuzzy definition because it does not specify the scope of what “read or written together” means.

Dynamic data race detectors attempt to detect races at runtime by monitoring execution, such as in ERASER [178] and FastTrack [86]. These tools require high-coverage program traces to get good results, e.g., when attempting to find bugs during a product’s testing cycle. Further, these approaches only detect data races, which can be both intentional (e.g. in lock-free algorithms) and innocuous. Data race detection tools that use the lock-set approach (such as ERASER) tend to suffer from high false positive rates, where-as happens-before based race detectors are more conservative.

Atomicity violations are more difficult to detect than data races, because the definition is not as straight-forward. An atomicity violation is defined in terms of intended serialized execution; determining the programmer’s intention is difficult. Most of the automatic atomicity violation detection tools described in the literature only handle single variable violations [133], [157], [55]. There are four problematic scenarios when considering single variables, as shown in Figure 23.

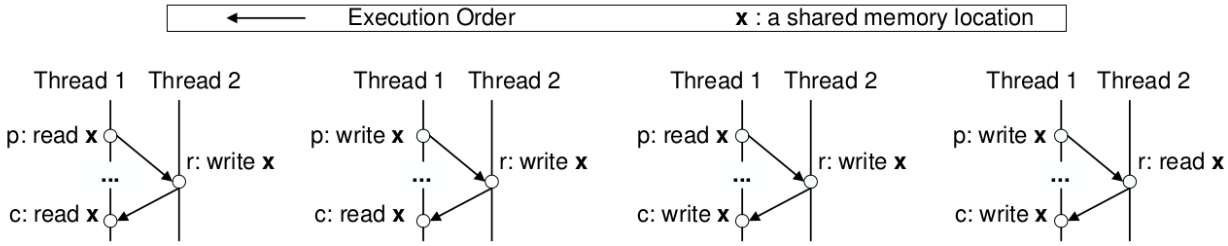


Figure 23: Problematic thread interleaving for single-variable atomicity violations (diagram is from [157], though much of the literature references these same scenarios).

Vaziri et. al. [206] expanded the set of problematic scenarios to 14 when dealing with multiple variable atomic sets (sets of variables that should always be updated together, atomically), but did not present an approach to automatically find such sets. In [131] Lu et. al. present MUVI, a tool for inferring correlations between variables, and describe how to extend existing atomicity violation detectors to handle multiple variables. SVD [219] handles multiple variables by heuristically identifying regions of code that should be serializable. This is in contrast to the previously mentioned approaches, which attempt to enforce atomicity for all uses of correlated variables.

Handling atomicity violations between multiple variables is important for both detection of real-world bugs (34% of non-deadlock concurrency bugs surveyed in [132] depended on multiple variable accesses) and coverage of concurrency CWEs (just one example is CWE-609 "Double Check Locking").

Detecting atomicity violations is necessary but not sufficient. We need to prevent the *exploitation* of atomicity violations. To this end we examined the following approaches for inclusion in PEASOUP:

1. Detection of suspected atomicity violations, with a limited number of rollbacks for recovery (similar to the approach of ConAir [228], but expanded to detect more general violations).
2. Opportunistically increasing the serialization of code regions that access shared variables.
3. Opportunistically increasing memory order determinism
4. Using dynamic race detection between variables in an atomic set, and protecting those sets dynamically.

Ultimately we decided on approach #4 (which we call Avert), and generating a design that we will outline in the next section. However, we ran out of time to complete implementation of Avert, and ultimately implemented some minimal thread schedule diversification to lower exploitability of atomicity violations.

3.7.5.2 Our Approach

3.7.5.2.1 **Avert**

Avert uses dynamic race detection to find possible atomicity violations, and then enables instrumentation that protects the inferred atomic region with a mutex. It starts by statically identifying sets of related memory accesses that should be grouped into an atomic set. It then instruments the binary to watch accesses to these memory locations, and feeds this information into a dynamic data race detector. It also instruments the binary with mutex lock/unlock operations surrounded the identified memory accesses. These lock/unlock operations are disabled by default; when a race is detected they become enabled.

We use data race detection because there exist relatively fast techniques for dynamically detecting data races. False positives can hopefully be filtered using a set of heuristics that identify adhoc synchronizations and lock-free code idioms.

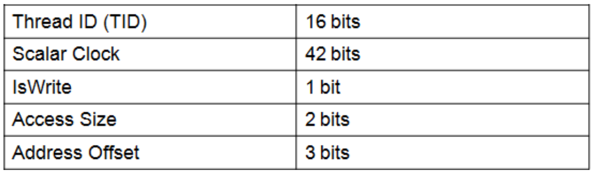
Avert is based on a happens-before race detector similar to what is implemented in ThreadSanitizer2 (TSAN2)[16]. TSAN2's happens-before algorithm is based on Fasttrack [86], which uses a set of optimizations to achieve better performance than most vector-clock-based algorithms.

TSAN2 shadows all of memory with state that tracks the last 4 access operations. This means that for every 8 bytes the process references, TSAN2 uses another 32 bytes overhead to track

[16] https://code.google.com/p/data-race-test/wiki/ThreadSanitizerAlgorithm

reads and writes to that 8 byte address. It requires that executables have been compiled as position independent executables (PIE) in order to perform this shadowing.

Every time a read or a write to an address *A* occurs, a new entry is inserted into the table at shadow(*A*). If the table is full then a previous entry is randomly chosen and overwritten. Figure 24 shows the information kept in each row of the table.



Thread ID (TID)	16 bits
Scalar Clock	42 bits
IsWrite	1 bit
Access Size	2 bits
Address Offset	3 bits

Figure 24: Data kept in shadow memory entries.

TSAN2 keeps information about the last 4 accesses to each address. If there exists two entries that are from different threads, one of them is a write, and there is no happens-before relationship between the events, then a data race is reported.

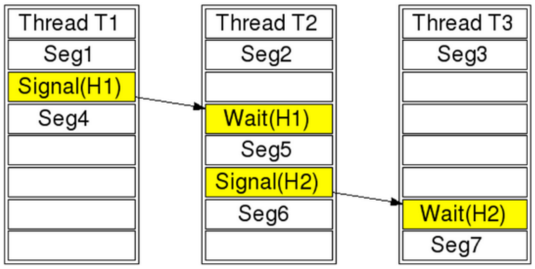


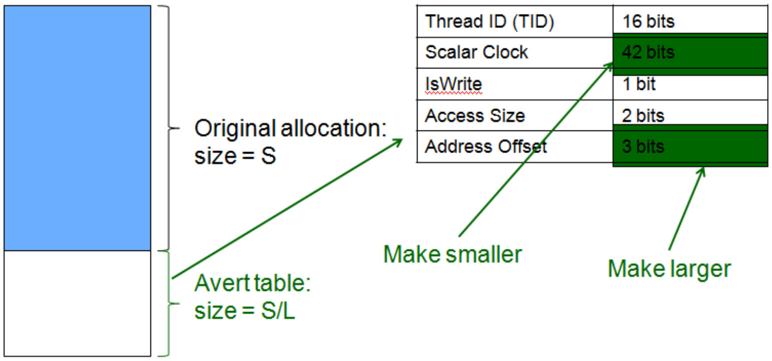
Figure 25: Illustration of the happens-before relationship

(from: https://code.google.com/p/data-race-test/wiki/ThreadSanitizerAlgorithm)

Figure 25 illustrates the happens-before relationship that is tracked by TSAN2 and similar algorithms. Each fork (e.g. thread creation), join, or signal/wait operation imposes an ordering between certain threads. For example, when a mutex in thread *A* is locked and then released, and later that same mutex is locked by thread *B*, we can order those threads. We know that anything after locking the mutex in thread *B* comes after anything that was before unlocking the mutex in thread *A*. Notice that the happens-before relationship holds only for an execution trace: the order of who locks the mutex first (thread *A* or *B*) may not be deterministic, and may depend on how the operation system schedules the threads.

Avert keeps track of the same information as TSAN2, but with some significant differences:

- Only global data memory (ELF sections .data and .bss) is fully shadowed. This allows us to keep memory overhead low and avoid the requirement of PIE compiled binaries.
- Heap is shadowed in a compressed format, using Twitcher Malloc. Figure 26 shows a sketch of our approach for heaps.
- The stack is not handled. We do not believe stack-based data races are a concern in most real-world programs.



Thread ID (TID)	16 bits
Scalar Clock	42 bits
IsWrite	1 bit
Access Size	2 bits
Address Offset	3 bits

Figure 26: Sketch of Avert's data race detection on heap memory.

Heap memory is not shadowed with 4x memory overhead, like TSAN2 does. Instead for every heap allocation of size *S* we reserve *S*/*L* extra bytes to serve as a compressed TSAN2 table. This means that multiple addresses with the heap allocation will share data race metadata. It also means that we need to change the meta data as shown in Figure 26; this will cause a slight loss in precision for the happens-before relationship.

Another difference between Avert and TSAN2 is that Avert groups memory addresses into atomic sets, and has one shadow memory location for the entire set. E.g., if you have two memory addresses A_1 and A_2 that are determined to be in an atomic set, Avert will detect a race where A_1 is accessed from one thread and A_2 is accessed from another.

Atomic sets are inferred statically. There are two conditions needed for memory addresses to be grouped together. First, they must be accessed close by in the machine code (within *T* instructions of each other, and within the same procedure). Second, for global memory the following can be in an atomic set:

- A global *g*
- Any dereferences of *g*, e.g. *(*g* + C)
- Any global boolean *h*, such that writes to *g* are control dependent on *h*

These will be heuristically inferred atomic sets, and may be incorrect. The boolean condition is meant to capture the idiom of using initialization flags, such as with the global singleton pattern.

Our goals with Avert were to reduce the memory and runtime overhead associated with state-of-the-art dynamic data race detection. We believe that only full shadowing the global data segment and shadowing the heap in a compressed way help meet this goal. To further reduce the runtime overhead, turning off race detection on heap allocations that are clearly not shared may help.

We started implementation of Avert, but halted development when we realized the implementation effort exceeded the remaining available contract effort.

3.7.5.2.2 **Perturbing Thread Schedules**

Race conditions and atomicity violations are rarely deterministic. To exploit these conditions attackers must either (a) be able to execute the racy code a large number of times or (b) increase the window of time between the operations that constitute the race.

Our approach is to inject a random delay prior to certain synchronization operations. The operations include thread creation, the first lock of each mutex (or decrement of a semaphore, etc.), and the joining of threads. In order to limit the overall impact on performance, we randomly choose a delay *d* in the range (0, MAX_DELAY). When then subtract that delay from the overall remaining delay, so MAX_DELAY = MAX_DELAY – *d*. This causes most of the changes in the schedule to happen early during process execution, and keeps the performance impact low.

Anecdotally, perturbing the thread schedule seemed to decrease the reliability of the atomicity violations present in our synthetic test suite. Additionally, the thought was that perturbing the schedule early in process execution might have a "ripple down" effect, i.e. it would make the schedule deeper in the process execution more non-deterministic.

3.8 C7: Memory-Safety Errors

3.8.1 Twitcher: Efficient Memory-Safety Enforcement

Twitcher is a sub-system in PEASOUP for automatically protecting against exploits of memory-corruption vulnerabilities. A software application may contain flawed logic, or *faults*. A carefully crafted malicious input may exploit the faults in an application to cause it to deviate from its intended behavior, possibly with dangerous consequences for the application's user. When this is possible, the faults are called *vulnerabilities*. Twitcher focuses on an important class of software vulnerabilities that lead to corruption of an application's in-memory data, also known as memory-corruption vulnerabilities. MITRE's Common Weakness Enumeration (CWE) provides a taxonomy of possible vulnerabilities [141]. A partial list of the vulnerabilities that Twitcher guards against includes:

- CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")
- CWE-121: Stack-based Buffer Overflow
- CWE-122: Heap-based Buffer Overflow
- CWE-124: Buffer Underwrite ("Buffer Underflow")
- CWE-126: Buffer Over-read
- CWE-127: Buffer Under-read
- CWE-129: Improper Validation of Array Index
- CWE-134: Uncontrolled Format String
- CWE-170: Improper Null Termination
- CWE-415: Double Free
- CWE-416: Use After Free
- CWE-457: Use of Uninitialized Variable
- CWE-590: Free of Memory not on the Heap
- CWE-665: Improper Initialization
- CWE-761: Free of Pointer not at Start of Buffer
- CWE-762: Mismatched Memory Management Routines
- CWE-805: Buffer Access with Incorrect Length Value
- CWE-806: Buffer Access Using Size of Source Buffer
- CWE-824: Access of Uninitialized Pointer
- CWE-908: Use of Uninitialized Resource

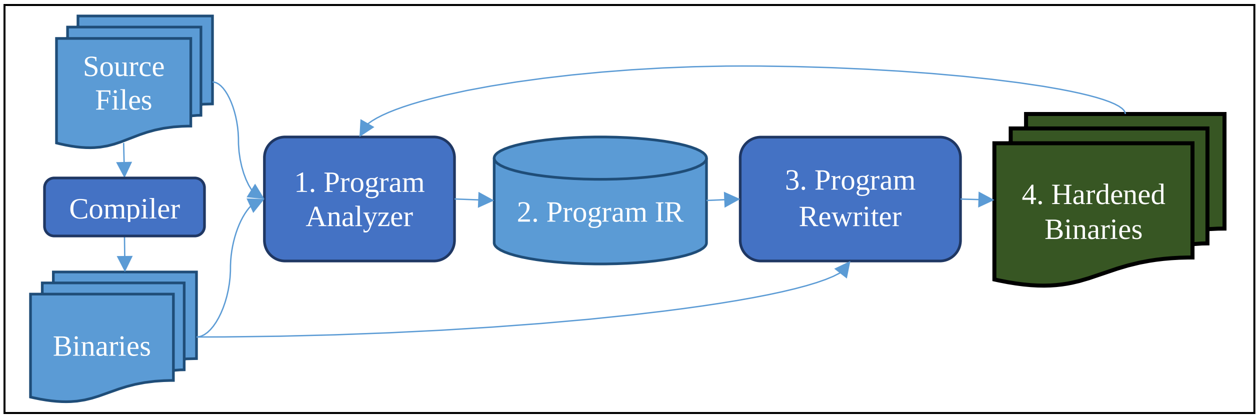


Figure 27 (Offline) Twitcher Preparation Stage

3.8.1.1 Idealized Platform (System Architecture)

Twitcher protects software that runs on general purpose electronic computing hardware, including personal computers, servers, and embedded devices such as smart phones or gaming consoles. Twitcher protects software that manually manages memory resources, typically by using a combination of a system library procedure, such as *malloc*, and a *runtime stack* that is typically maintained by updating one or more dedicated hardware registers.

3.8.1.2 Twitcher Architecture

Like PEASOUP, Twitcher operates in two stages: the *Preparation Stage* and the *Runtime-Monitoring Stage*. Figure 27 shows the flowchart for an offline implementation of the preparation stage, which is what is currently used by PEASOUP. During the preparation stage, Twitcher first uses program analysis (Figure 27, Box 1) to learn facts about the program that the user wants to protect, called the subject program. Many different types of analysis can be applied, including static and dynamic analysis and source code and machine-code analysis. Twitcher uses the program analyzer(s) to construct an intermediate representation (Figure 27, Box 2) that captures information such as the following:

1. *What instructions perform potentially dangerous memory operations?* A simple analysis to do this disassembles all possible instructions in the binary file and identifies any instructions that access memory. One implementation refines this analysis by assuming that instructions that access a fixed memory location or stack offset is/are safe.
2. *What instructions allocate and de-allocate memory regions?*
3. *What is the data layout of the program?*
4. *Which instructions directly address globally allocated or stack-allocated data?*
5. *What functions "recycle" heap-allocated buffers?*
6. *What functions are "wrappers" for the system's heap-management library?*
7. *What functions implement custom memory-management libraries on top of the system's heap-management library?*

In an example embodiment, all of this information is present. In alternate embodiments, some of this information may be missing or incorrect. In many such cases, Twitcher can still function; however, it may provide less protection, or require more extensive testing to ensure that the intended functionality of the subject program is not broken. PEASOUP primarily uses STARS (Section 3.3.4) and speculative transformation (Section 3.8.2) to discover the IR for items 1–4 listed above; IR for 5–7 is not currently recovered nor used in PEASOUP. Twitcher uses PEASOUP's rewriting facilities (Section 3.3.5) to insert Twitcher's defenses (which are described below).

The Runtime-Monitoring Stage of Twitcher's defenses occurs each time the protected application is used. The Twitcher defenses are implemented by the hardened libraries, hardened executable(s), and dedicated libraries. The dedicated libraries perform checks for memory-safety violations by interposing between the application and common system libraries, such as libc.so. As the hardened binaries run, they execute the protections inserted during the preparation stage. For example, the hardened binaries will use a modified data layout that includes *guard regions* that should not be accessed by the application code, maintain Twitcher metadata, including current locations of guard regions, and check for unsafe memory accesses, including accesses that should access the program's data but are accessing a guard region. When the hardened binaries call procedures in system libraries that manipulate memory, Twitcher will intercept the call in its hardened libraries, perform additional checking and updating of metadata, and call the underlying library procedure, if it is judged to be safe. Further detail is provided below.

Twitcher uses a combination of strategies to defend against memory corruption. These include:

- *Guard Regions*: Twitcher identifies regions of memory that the program has no legitimate reason to access; these are called *guard(ed) regions*. Furthermore, it transforms the layout of the program's data so that *guard regions* are interspersed with the program's legitimate data. The placement of guard regions is based on the IR that is recovered during the preparation stage: guard regions may be placed at the ends of heap blocks, between procedure activation records, between data objects on the stack, and/or between global data objects. Twitcher also replaces instructions or procedure calls that may perform unsafe memory accesses with code that checks that the locations to be accessed are not in guard regions before performing the accesses (See Section 3.8.1.2.1). Doing so helps ensure that the hardened program accesses data in a safe manner.
- *Clearing stale data*: Twitcher uses various techniques to recognize when a program is reusing a region of memory (see Section 3.8.1.2.3). Before reuse occurs, the old data is cleared and/or the region is converted to a class of guard region that indicates the region should be initialized before being read (see Sections 3.8.1.2.1 and 3.8.1.2.4). Certain example embodiments thus may supplement the garbage collector in a garbage-collected environment such as Java, e.g., by inferring and marking used memory segments itself. Doing so helps to ensure that data is scrubbed once it is no longer needed, e.g., reducing the likelihood that malicious programs will be able to access forgotten-about and/or leaked memory areas and the data stored therein. For example, this mechanism allows Twitcher to repair the infamous Heartbleed vulnerability in many instances.
- *Altering de-allocation patterns and actions*: When the program explicitly de-allocates a memory region by calling free(), Twitcher may delay the return of the memory to the pool of memory available for reuse. This is similar to the *free quarantine* mechanism used by *AddressSanitizer* [186]. As soon as a block is placed in quarantine, it is converted to a class of guard region that indicates it has been freed and should not be accessed by the subject program. If Twitcher detects an access to a guard region that is in the free quarantine, it indicates a use-after-free error. Twitcher is able to repair the error by avoiding re-allocation of the block (see Sections 3.8.1.2.1 and 3.8.1.2.4).

- *Taint inference and propagation*: Twitcher can use a combination of lightweight taint inference and taint propagation to recognize and repair certain classes of dangerous memory-usage errors such as use of unitialized memory and buffer overruns (see Section 3.8.1.2.3).
- *Repair strategies*: When Twitcher detects that a memory-corruption error is about to occur, it may use a variety of strategies to avoid the error. These include replacing the values that would be returned by an errant memory read and early termination of a computation (e.g., a loop, thread, or process). Twitcher will also report potential memory-corruption errors, which allows administrators to check for attacks and developers to repair faults. Twitcher's repair mechanisms are described in more detail in Section 3.8.1.2.4.

Some or all of these and/or other strategies are implemented by the modifications made to the subject program and in the Twitcher libraries. In the remainder of this section, details are provided regarding how Twitcher can implement the above strategies.

3.8.1.2.1 **Checking Potential Dangerous Memory Accesses**

As described above, Twitcher uses *guard regions* to check for potentially unsafe memory accesses. Twitcher modifies the subject program (1) to intersperse guard region with the program's data, (2) to update Twitcher's data about where the guard regions are located, and (3) to check that potentially unsafe instructions do not access a guard region in a disallowed manner (see 3.1.7 for a partial classification of allowed and disallowed accesses). Parts (1) and (2) involve modifications to the steps the program takes to allocate and deallocate memory. Part (3) involves modifications to instructions used to access memory.

In general, the modifications for part (3) reasonably can be expected to incur much higher runtime overhead than parts (1) and (2) in example implementations, as memory accesses may be expected to occur orders of magnitude more often than memory allocation and deallocation. However, Twitcher has multiple techniques for implementing (1)–(3). The exact techniques that are selected may depend on factors such as, for example, the performance characteristics of each subject platform, the context of individual instructions in the subject program to be instrumented, and/or the like. In order to make (3) effective, Twitcher in some respects follows the work of Hasabnis et al. in [152] by using a series of nested tests to check if the subject program should be allowed to access the values at a given address. The earlier tests are expected to be extremely cheap, but have a small false positive rate: it is possible the check will report that an address should not be accessed when it is safe to do so. Subsequent checks are more expensive, but have a lower false positive rate. The final check should have a vanishingly small probability of a false positive.

Guard Values

Twitcher fills guard regions with known *guard values* and the first check Twitcher performs to determine if an address *A* is safe is to check if the value stored at *A* is a guard value: if the value at *A* is not a guard value, *A* is assumed to not be in a guard region, and is assumed to be safe; if the value at *A* is a guard value, further checking is necessary. Twitcher performs the check for the guard value, performs subsequent checks, and selects guard values, in accordance with the procedures identified below. It will be appreciated that there are multiple strategies that can be used for various ones of these steps and, in certain example embodiments, multiple strategies may be used in any suitable combination, sub-combination, or combination of sub-combinations, e.g., to protect a single binary, e.g., as is made more clear below.

Suppose the subject program contains an instruction *instr* that accesses *n* bytes at address *A*. The following pseudo-code shows a technique to check for a guard value at *A*:

```
        Save registers and flags needed for comparison
        If (get_n_bytes(A) != n_byte_guard_value)
            goto SAFE;
        Perform additional checks
    SAFE:
        Restore registers and flags used in comparison
        instr              ;; the original instruction
```

It has the advantage of being simple and relatively cheap. Depending on the machine architecture, it may incur overhead for introducing additional memory accesses (to save and restore program state) and an additional conditional-branch instruction.

Checking Guard Values with Hardware Exceptions

Twitcher may opt to use a *branchless* implementation of some check for guard values at an address *A*. The idea is to read the value *v* stored at *A* and perform a calculation that will cause a hardware exception if *v* is a guard value. For example, to use a SIGSEGV fault for a check, before beginning execution of the subject program, Twitcher may initialize some static data as follows:

```
    int dummy = 0;
    int dummy_ptrs[256] = { & dummy, };
    dummy_ptrs[guard_value] = nullptr;
```

Then, to perform the desired check (“does A contain a guard-value byte?”), Twitcher could insert instrumentation that does the following:

```
    save contents of register r
    load byte at address A into r
    load dummy_ptrs[r] into r
    load *r into r  ;; faults iff r==guard_value
    restore the contents of r
```

In addition to avoiding the use of branches, this style of instrumentation may require saving and restoring less state. If Twitcher can identify a free register at the location where it wants to insert the check, it is possible no state will need to be saved or restored.

Twitcher may use a mixture of different checking strategies for different instructions in the subject program.

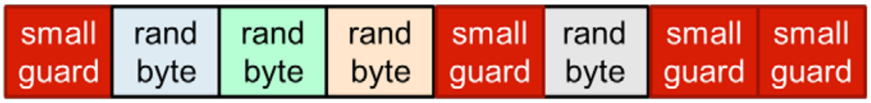


Figure 28 Layout of a Bipartite Guard on a Little-Endian Machine.

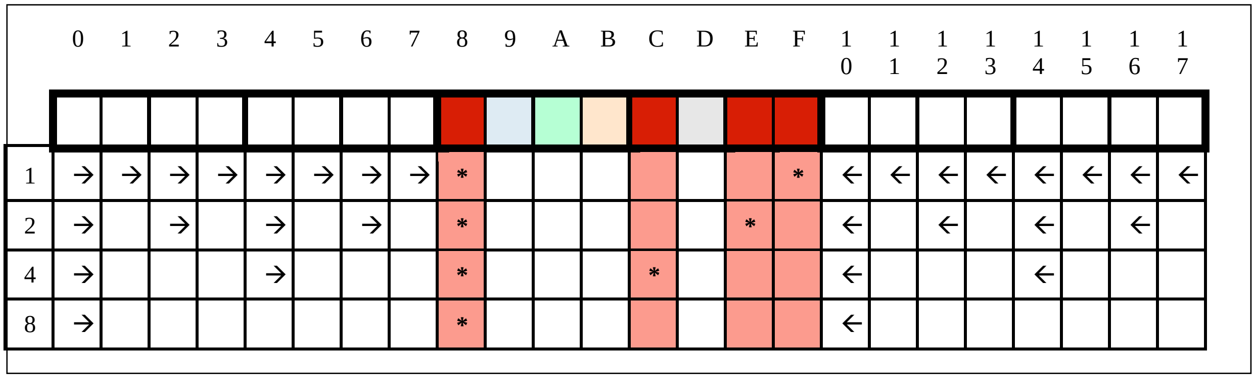
Mixing Guard-Value Checks

For each instruction with a potentially unsafe memory access in the subject program, Twitcher may use a customized instruction sequence to check if the instruction is safe. For some dangerous instructions, Twitcher may use a chain of compare-and-branch instructions; for others, it may use one of the hardware-exception mechanisms described in the previous section. An implementation of Twitcher that uses dynamic instrumentation to insert the checks may even change the checking instrumentation based on online profiling information.

Bipartite Guard Values

As described above, Twitcher typically checks for a guard value for its initial, quick-but-faulty check. Using the technique of *bipartite guard values*, the initial check is for a 1-byte *small-guard* value, and the subsequent more expensive check is for a full 8-byte *full-guard* value. If the memory access to be checked is for 8-bytes, then the small-guard check may be skipped, as the full-guard check should add no measurable overhead. The full (bipartite) guard is constructed at the beginning of program execution, as follows (assuming a little-endian machine):

```
Randomly select a small-guard value (from infrequent 1-byte values)
Set the bytes at offsets 0, 4, 6, and 7 to the small-guard value
```



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
									🟥	🟦	🟩	🟧	🟥	⬜	🟥	🟥								
1	→	→	→	→	→	→	→	→	*							*	←	←	←	←	←	←	←	←
2	→		→		→		→		*						*		←		←		←		←	
4	→				→				*				*				←				←			
8	→								*								←							

Figure 29 Strided Accesses Hit Small-Guard Values

```
Set the bytes at offsets 1-3 and 5 randomly
```

Figure 28 shows the layout of a bipartite guard on a little-endian machine. This layout is motivated by the observations that compilers usually try to align data on word boundaries, and that loops that overrun or underrun a buffer are usually accessing the buffer in a power-of-two stride (1-byte, 2-bytes, 4-bytes, or 8-bytes on an 8-byte machine). As shown in Figure 29, the layout of the bipartite guard, including the repetition of the small-guard value, ensures that aligned, strided buffer accesses will land on a small-guard value in the bipartite guard. The other bytes of the bipartite guard (at offsets 1, 2, 3, and 5) make it hard to guess the value of the full guard and vanishingly unlikely that the full guard value will arise by chance during the execution of the subject program.

One disadvantage of the bipartite guards is that they may be easier to circumvent: if an attacker can arrange a misaligned access, then the small-guard check will not detect the presence of the guard, and the attack may be able to corrupt the guard value. On the other hand, if the attacker can arrange misaligned accesses, she may already be able to side-step the guards.

Guard Maps

An alternative to using bipartite guards, or an approach that may be used with it, is to maintain *guard maps*. Using this approach, Twitcher maintains one or more maps of where in memory the guard regions are located. The maps are updated every time memory is allocated or deallocated. The guard value includes repeated copies of a randomly selected 1-byte value. The initial safety check always checks for n-copies of the (small-)guard value, where n is the number of bytes that are accessed in the instruction that is being checked. The more expensive check consults the maps for the presence of the untrusted address: if it is in a guard region, then the attempted access is unsafe.

Twitcher may use a *trie* or other data structure to implement a single guard map for the entire program. However, in more secure embodiments, Twitcher may use different types of maps, e.g., for different memory regions:

- For each thread stack, Twitcher may use a *direct map cache*, e.g., that contains a single bit for each byte in the stack. A direct-map cache is similar to a one-level trie. One advantage of using the direct-map cache is speed of update and access. Twitcher leverages the fact that every time an activation record for procedure *foo* is created on the stack, it will use exactly the same data layout, and hence, the guard regions will be exactly the same. This means that Twitcher can prepare a template that contains the appropriate bit pattern representing the locations of the guard regions in *foo*'s activation record and simply copy it to the end of the map for the currently running thread. Similarly, a template can be used for initializing the guard values in the activation record.
- For heap-allocated memory, Twitcher may leverage the internal data structures of the heap-manager to also keep track of the guard regions in the heap. Because Twitcher's implementation of the heap-manager already maintains the requisite information, no additional overhead or data structure is necessary (e.g., see [85]).
- Finally, for statically-allocated memory (such as, for example, the program's "global data"), Twitcher may simply use a hash table or similar data structure to store the locations of the guard regions, which do not change during the program's execution.

To implement the more expensive check, Twitcher may have to check each map.

Protecting Guard Maps

One danger in using guard maps is that the attacker may be able to force an errant-memory update that corrupts the map, thereby allowing more latitude in forcing other unsafe memory accesses. We have observed that with a straight-forward implementation of a guard map (e.g., using a direct-map cache), the majority of locations in a guard map may contain the value zero. We can leverage this observation by storing the bytes of the guard map after exclusive-or'ing them with the small-guard value, so that the majority of bytes in the guard map are the small-guard value. Furthermore, Twitcher marks the entire guard map as a "guard region." In this fashion, Twitcher uses its checks on potential errant-memory updates to protect the integrity of its own guard maps. Similar techniques may be used to guard other critical data.

Hybrid Guard implementations

Twitcher may use different guard implementations for different regions of memory. One attractive solution may be to protect stack memory with bipartite guards (thereby potentially obviating the need for guard-map updates when allocating/deallocating stack frames), and use homogenous guards to guard heap memory (where the guard map may be provided by the native heap manager). Using this configuration, the instrumentation to check if address A is safe could be implemented using the following pseudo-code:

```
    Save program registers and flags needed for check
    If (byte stored at A differs from the small-guard value)
        goto SAFE;
    If ((16-bytes stored at A differs from the full-guard value) AND
        (A is not in guard maps for heap and static data))
        goto SAFE;
    Initiate response for unsafe access
SAFE:
    Continue with execution of dangerous instruction
```

This particular configuration can be extended in such a way that we use both types of guards in the heap: when the program requests a heap block with N bytes, Twitcher increases N to allow space for (one) guard value and then rounds up to the next power of two; it actually reserves M > N bytes. Twitcher places a homogenous guard at the end of the M reserved bytes. Twitcher's heap manager provides a capability to look up an arbitrary address and determine if it is in an allocated block, and if so, the location and size of the block. Using this capability Twitcher implements a guard map for the homogenous guards placed at the end of blocks: if an address is at the end of the block, it is in the guard region. However,Twitcher's malloc implementation need not necessarily record the original requested size, N, anywhere. Consequently, overruns that exceed N but still do not hit the guard at the end of the M-byte block will not be detected. To detect such overruns, Twitcher may place a *bipartite* guard after the Nth byte of the allocated block and prior to the homogenous guard at the end of the block. As described above, this provides the full strength of the homogenous guard (which provides better protection against misaligned accesses) for the end of the allocated block, while still providing some protection for overruns that surpass the Nth byte in the middle of the buffer.

In some cases, M may be greater than or equal to 2*N. In these cases, Twitcher may place a homogenous guard at the (M/2)th byte, which is still easily located using the heap-management data structures. The presence or absence of a second, middle-of-buffer guard can be indicated with a single bit, which is much less than the amount of space needed to store N.

Guard Semantics

Up to this point, guard regions have been described generically and without explicitly defining the significance of an attempted access to a guard region. In some example implementations, any access to a guard region is disallowed. In Twitcher, there different classes of guard region, or different *guard classes.* Different guard classes are given different semantic interpretations, e.g., as to what types of memory accesses (read or write) should be allowed for guard of a given class. The guard classes used by Twitcher may include:

- Read/Write Guards: any attempted read or write access indicates an error or attempted exploit of the subject program. Read/write guards correspond to the guards of prior work [186], [152].
- Read Guards: any attempted read access indicates an error or attempted exploit, but write accesses are allowed and cause Twitcher to remove the guard. Read guards are useful for detecting when the program attempts to read memory it has not yet initialized.
- Page Guards: any attempted read or write access of any byte on the page containing the guard indicates an error or attempted exploit. Twitcher may place page guards on code pages and use them to detect attempts to scan code pages, e.g., for return-oriented programming (ROP) gadgets [43].
- Unallocated-Mem Guards: any attempted read or write access indicates an error or attempted exploit of the subject program. Twitcher places unallocated-mem guards on (heap) memory that the program was previously using but subsequently returned to the heap manager. An access of an unallocated-mem guard indicates a use-after-free error, which Twitcher will attempt to repair (see Section 3.8.1.2.4).
- Cloneable Read Guards: cloneable read guards disallow read accesses, except for the purpose of cloning the protected memory region. Cloneable read guards may be valuable for avoiding false positives in the presence of certain types of initialization patterns.

The implementation discussed above is modified as follows to support different guard classes:

- Small guard values are the same for all guard classes.
- There are different bipartite guards for each guard class, indicated by a few bits in one of the “random” bytes of the guard class.
- Guard maps are extended to indicate the class of each mapped guard.
- Guard checks are extended to consider the guard class. The initial, cheap check can be the same for all checked memory accesses. The secondary check will differ depending on the type of access:
    - read access: fails if any type of guard is found.
    - write access: fails if any class of guard except a read guard is found; read guards are removed but check passes.
    - bulk access (i.e., a library call that reads or writes to a range of memory): fails for page accesses that include a page guard.

Twitcher’s response when a check fails will also depend on the class of guard that caused the check to fail (see Section 3.8.1.2.4). For example, upon an attempted access of an unalloc-mem guard, Twitcher will attempt to restore the memory block, mark the block as allocated, and continue execution (see Section 3.8.1.2.4).

In addition, Twitcher places different guard classes in different situations:

- Twitcher (optionally) uses read/write guards to delineate program data, e.g., at the end of heap blocks.
- Twitcher (optionally) places read guards:
    - In some or all of newly allocated heap or stack memory.

    - In memory that could have been (over)written by a library call, but was not. For example, a call `read(fd,buf,M)` will read up to M bytes from fd into buf. If n<M bytes are actually read, then Twitcher may place read guards in the (n-M) bytes of buf after the portion that was filled by the call to read (see Section 3.8.1.2.3).
- Twitcher (optionally) places page guards on pages holding code or other sensitive metadata, such as guard maps or heap maps.
- Twitcher (optionally) places unalloc-mem guards on heap blocks before they are placed into a free quarantine or they are returned to the heap manager.

3.8.1.2.2 **Maintaining Guards**

As the subject program executes, the layout of its data in memory will change: memory regions are constantly allocated and released, or deallocated. Memory that has been deallocated may subsequently be reused to fulfill a new memory-allocation need. During the preparation stage, Twitcher learns the program's intended data layouts and modifies them to allow room for guard regions. During the monitoring stage, Twitcher helps ensure that the guards are properly maintained as memory is allocated and deallocated. As mentioned above, different mechanisms may be utilized for stack memory, heap memory, and static memory.

Maintaining Guards in Stack Memory

The subject program may have multiple threads, each with a *runtime stack* that is typically used to store temporary data, such as the arguments and variables for a function invocation. On a function call to a function *foo*, a new region is allocated on the "top" of calling thread's stack, called the *activation record* for foo. When foo's execution completes, its activation record is (implicitly) deallocated; the memory may be reused on subsequent function calls. At any time during execution, Twitcher helps ensure (a) that there are guard regions delineating the data objects on the "live" portions of the runtime stacks and (b) that it can reliably determine where these guard regions are. Ideally, any used memory on the stack (e.g., memory past the current stack top) would be tracked as belonging to one or more guard region(s).

One strategy is to update the guards and guard maps on every allocation and deallocation of an activation record. On a function call to a function *foo* (causing the creation of an activation record), Twitcher modifies the initialization of *foo*'s activation record to initialize the desired guard regions in the activation record with guard values. It also updates the guard maps with the locations of the new guard regions. When *foo*'s activation record is deallocated (either by a normal return or a non-local control transfer, such as an exception), the guard values are cleared, and the guard maps are updated to indicate those guard regions have been removed.

In some cases, the above strategy may have the lowest possible runtime overhead. In other cases, it may be inefficient, and it is overly conservative. In particular, the above strategy assumes that when an activation record is deallocated, none of the memory is protected (in a guard region). In fact, it may be desirable to protect all of that memory.

One option is to fill the entire deallocated region with guard values and update the guard map (if any) appropriately, although this is likely to incur significant additional overhead.

Another option is to simply leave the guard values and guard map alone during deallocation. This is extremely cheap, and provides partial protection for the deallocated region. In both of these cases, the extra guards are removed at any time before that region of the stack is re-used for a different allocation record. This can be done when new activation records are allocated, for instance.

Maintaining Guards in Heap Memory

Twitcher intercepts calls from the subject program to system libraries, such as the heap-management library (malloc, realloc, free, etc.). Twitcher may entirely replace the heap-management library with an implementation that provides greater security such as the DieHard and DieHarder libraries [85, 92]. On a call to allocate heap memory, Twitcher increases the size of the requested allocations to allow room for a guard value, and it places a guard at the end of the allocated block. It also optionally clears the other bytes in the heap block to ensure that no stale data is leaked. Other aspects of Twitcher's mechanism for maintaining guards in heap memory depend on the heap-management library that is ultimately used.

Some heap-management libraries (such as DieHard) are able to efficiently determine if an address resides in a heap block, and if so, what are the properties of that block (i.e., start and end addresses, size, is it currently allocated, etc.). Twitcher uses this capability to implement a guard map for the heap. If for some reason Twitcher's heap-management library cannot be used, then Twitcher will use an external data structure, such as a trie, to implement the guard map for the heap; using this approach, the trie is updated every time a call to allocate or free memory is intercepted. A third option is to use bipartite guards in the heap, and not bother with a guard map.

Guarding Deallocated Blocks

When the program requests that a block of heap memory be deallocated, or freed, it indicates that the program does not intend to access that block again. Existing references to the block at the time it is freed are considered to be stale. The heap-management library is free to use the block to satisfy future requests for memory. Use of a stale reference can lead to use-after-free vulnerabilities (e.g., as described in CWE 416), which cause the stale references to be aliased with new references when the block is re-allocated to fulfill a new allocation request. Twitcher uses many different techniques to reduce the likelihood of exploits of use-after-free vulnerabilities:

1. Twitcher may queue the program's requests for deallocation of blocks, temporarily prolonging the "lifetime" of the block allocation before it is reclaimed for reuse; an exploit cannot succeed until the block is reallocated.
2. Twitcher may randomize the order in which deallocation requests are processed from the free list. This makes it difficult to predict the circumstances under which a block will be reallocated, which is an operational principle behind many successful exploits.
3. At some point before Twitcher allows the block to be re-allocated (e.g., before it officially deallocates the block by marking it available for allocation), Twitcher may store a copy of some bytes of the block into the extra padding at the end of the block and/or overwrite some or all of the block with unalloc-mem guard values.
    a. One strategy is to only place an unalloc-mem guard at the beginning of the heap block, although some implementations for checking for guards may not always catch accesses past the first word of the block. One approach to compensate for this is based on the observation that frequently it is possible to infer the beginning of a block for a memory access that will access the middle of the block. For example, given a memory address (*base* + *offset*), often *base* will point to the

beginning of the block, while (*base* + *offset)* will be in the middle of the block. Given a potentially unsafe memory access to (*base* + *offset*), Twitcher may check *base* for an unalloc-mem guard.

   b. If the guard values are accessed before the block is reallocated, (i) it indicates a use after free and (ii) it will be detected. At that point, Twitcher may execute any one of several repair strategies (see Section 3.8.1.2.4).

4. Finally, Twitcher's heap-management implementation allocates from the set of free blocks at random. As with some of the above defenses, this makes it difficult to predict when the block will be reallocated, and it prolongs the time before the block is reallocated. Only when the block has finally been reallocated does a use-after-free exploit become possible.

Maintaining Guards in Static Memory

A program's static memory includes global variables and file-scope and function-scope static variables that have a fixed location potentially for the lifetime of the program. Each dynamic library used by the subject program may have its own static data segment, which has a fixed layout while the library is loaded. During the Preparation Stage, Twitcher should modify the layout of static data to insert guard regions; this is not yet implemented. During the Monitoring Stage, Twitcher could use a hash table (or other suitable data structure) to track the locations of guard regions in the program's static memory; again, this is not yet implemented. When the program's image or a dynamic library is loaded into memory, Twitcher initializes the guard regions in the static memory with guard values and it updates the guard map for static memory. When a dynamic library is unloaded from memory, the guard map for static memory is adjusted appropriately.

3.8.1.2.3 **Detecting Memory Reuse and Use of Uninitialized Memory**

Most programs use multiple mechanisms to recycle their memory and keep their overall memory footprint low: when the program is done using a memory block for one purpose, it is repurposed for a different use, e.g., by being "released" and then "reallocated." Many programs also contain defects such as "use of uninitialized memory" that can lead to sensitive data being exfiltrated from recycled memory regions. For example, here is one sequence of actions that might exfiltrate sensitive information to an attacker:

1. The program allocates a block of memory.
2. The program fills the block with sensitive information, e.g., a password.
3. The program releases the block (without clearing its contents).
4. The program reallocates the block for a new purpose.
5. The program reads and reports the contents of the block before overwriting some or all of the contents (i.e., the program reads contents that are "uninitialized" since the reallocation in step 4); as a result, the sensitive contents of the block from its last use are reported to the attacker.

Twitcher uses various heuristics to detect when memory is being reused and there may be a risk of leaking the old contents of the memory. Twitcher may use combinations of the following analyses and heuristics to determine when a buffer is being partially or completely reused:

- *Explicit allocation*: when Twitcher sees memory being (re)allocated via a call to malloc() or the creation of a new procedure frame, it may assume the new memory is being reused and should be considered "uninitialized."
- *Inferred recycling functions*: when a function might return uninitialized heap memory (obtained via a call to malloc) or some other heap block, then Twitcher may assume the function is implements *heap-block recycling*. A function that implements heap-block recycling arranges for reuse of heap blocks without returning them to the heap manager. Twitcher relies on its implementation of malloc to dynamically check if a value returned by a potential recycling function is the beginning of a heap block.
- *Fill gaps*: many library procedures will fill a variable amount of a buffer, up to some maximum number of bytes, *M.* At runtime, when the function actually fills just *n* bytes where *n* < *M*, the remaining (*M-n*) bytes are called the *fill gap*. Twitcher may assume that fill gaps should be considered to be uninitialized.
- *File-Descriptor Taint*: Twitcher hooks calls to read(), memcpy(), and other memory-copying functions. It may use space at the end of each heap block to record some *file-descriptor taint* about each heap block. When the application reads input into a buffer and overwrites pre-existing data with tainted data from a different file descriptor, Twitcher may conclude that the block is being reused, and the old data can be cleared. This may happen during a call to read() or to memcpy() or some later functions.

When Twitcher determines that memory is being reused — e.g., that the current contents are stale and the memory should be overwritten before it is reused — it may use a combination of the following strategies: (1) clear the memory; (2) place a read guard in the memory and modify unsafe read instructions to first check for the read guard.

3.8.1.2.4 **Repair Strategies**

Often, Twitcher is able to detect when an error has occurred. In these situations, Twitcher uses different strategies to attempt to repair the error and continue the program's execution.

Reallocation on Use-After-Free

As noted above in Section 3.8.1.2.2, Twitcher may sometimes detect an attempt to access a block that the application previously freed (indicated that it was done with the block). In these cases, Twitcher may attempt to restore the values in the block, if they've been saved, or clear the block, mark the block as allocated and then allow the program to continue execution. Marking the block as allocated has the effect of reviving the stale references and ensuring that no new references will be made that alias the stale references; the stale references can be safely used as if the requested deallocation never happened. The program execution will likely continue normally, and no use-after-free exploit will be possible.

Twitcher may also note the calling context that caused the use-after-free. If a particular use-after-free pattern is observed frequently, Twitcher may start tagging blocks according to where they were allocated, correlate the allocation tag with the use-after-free errors, and then take appropriate responses when a free is performed on a block with an allocation-tag correlated with frequent use-after-free errors, e.g., by increasing the delay for that free.

Automatically Growing Buffers

In some cases, Twitcher may detect an attempted overrun of a heap-allocated buffer. In these situations, there are several steps Twitcher may take to attempt to "grow" the buffer, essentially by performing a "realloc()" on the buffer. There are two possibilities, based on whether or not the neighboring heap block is allocated, or not. If the block is not allocated, Twitcher may mark the block as allocated, and allow the "overrun" to proceed into the neighboring block. Effectively, the heap block is increased in size.

If the neighboring block is already allocated, then Twitcher may take the following steps:

1. Allocate a larger block at some other location is memory.
2. Copy all of the data in the original heap block to the new heap block.
3. Overwrite all of the data in the original heap block with a homogeneous guard value. This guarantees that any attempted access to the original heap block will be detected.
4. Redirect all future attempts to access the original block to instead access the new block.

Replacing Over-Reads with Manufactured Data

When Twitcher detects an attempt to read past the end of a buffer, instead of performing the read (and potentially releasing sensitive information), it may replace the results of the read with manufactured data and allow the program to continue execution. For example, it may return common sentinel values, such as 0 and -1, or random data.

Early Loop Termination for Over-Writes

When Twitcher detects an attempt to write past the end of a buffer, instead of performing the write, it may terminate the loop that is attempting the write.

3.8.2 Stack-Layout Randomization Transformation (SLX)

In this section, we present an approach to stack-layout randomization that does not require access to the source program or other development artifacts. Randomization of the layout of local variables in the stack frame is implemented using dynamic binary translation. The goal of our approach to stack-layout randomization is to enhance the security of a software system using only the binary form of the software. The ability to randomize based on limited information is especially important, because having software of unknown quality and for which only the binary form is available is a common circumstance.

Many attacks on vulnerable software succeed because the perpetrators have knowledge of the layout of program elements in memory. Thus, one important form of artificial diversity is address space layout randomization (ASLR) in which the layout of elements in memory is randomized. Implementations that are currently deployed for binary programs [159] are coarse gained; they randomize details such as the base addresses of the stack, the heap, and the code. More aggressive fine-grained randomizations have been developed that randomize details such as the order of functions and the stack layout. These approaches rely, however, on the availability of the source program [70, 175, 176, 201].

The security benefits of randomizing the stack layout are many; exploits might be completely thwarted or severely limited. The benefits realized depend on the nature of the exploit, the vulnerability, and the randomization used.

There is a need, then, for a stack-layout randomization technique that can provide fine-grained randomization without using information from the program source. We sought to develop a technique that would meet the following goals:

- Operate on binary programs designed to execute on a common target platform (x86) using only data gathered from the binaries themselves.
- Require only relatively simple analysis of the binary program.
- Provide a fine-grained transformation; that is, randomize stack frames of functions so that the order of variables is changed.
- Apply to real programs and scale to large programs.
- Incur a small overhead during execution.

In this section we describe our technique, called Stack-Layout Randomization (SLR), and show that it achieves these goals.

SLR uses static analysis to develop a hypothesis about the layout of local variables in the stack frame for each function. It transforms each function to reorder the stack based on this hypothesis, then evaluates the hypothesis by executing the transformed program. In this sense, SLR is a speculative technique; static analysis of binary code rarely determines the stack layout perfectly, so static analyses must be verified empirically. If an analysis prescribes a transformation that changes the semantics of a function, different and less fine-grained randomizations are tested until a behavior- and semantics-preserving randomization is found.

The key contributions of this section are: (a) the use of a speculative and dynamically verified approach to the determination of the stack-frame layout, and (b) the use of dynamic binary translation to implement stack-frame randomization.

The way in which SLR develops the hypothesis about the stack layout is presented in Section 3.8.2.1. In Section 3.8.2.2 we describe the overall process by which the hypothesis is evaluated and modified if needs be. We describe the implementation of stack randomization in Section 3.8.2.3. Our approach to evaluation is described in Section 3.8.2.4.

3.8.2.1 Evaluating Stack Layout Hypotheses

Randomization of the stack frame layout for a function requires determination of:

- The current layout of the stack frame, i.e., the addresses of the function's local variables as set by the compiler.
- The instructions that manipulate the different variables in the stack frame.

In principle, if this information were available, the layout of the stack frame could be changed and the instructions that access the stack frame modified to reflect the new layout.

The new layout of the stack frame could be based on any security-relevant criteria. For example, memory objects could be placed in random order, padding introduced before, after or within the stack, canaries included, encryption applied, or scalars placed at lower addresses than arrays. Items on the stack could also be removed and promoted to the heap. In our current approach, we limit transformations to placement of memory objects in random order and the introduction of random length padding. This choice was made so as to gain information about the potential of simple stack-layout randomization.

Starting with a binary program, precise determination of the stack layout and the instructions that reference the stack is problematic. Modern compilers employ a wide range of techniques to minimize both the use of storage and program execution time. The result is binary programs with unpredictable structures. For example, when generating code for the x86 architecture, compilers frequently inline functions, fold constants, pack stack frames, unroll loops, and include hand-written assembly for common functions like memcpy.

Our approach to SLR is based on two assumptions:

- Knowledge of the size of variables is all that is needed to determine boundaries. Type information of stack variables is useful for SLR, because type information helps to determine the size of variables. This benefit from type information is especially important for determining fields in records (structs in C). However, our goal with this work was to assess the possibility of using simple heuristics for boundary determination, and so we do not determine nor use variable type information.
- The predominant mechanism by which instructions access stack variables is through scaled or direct addressing based on an offset indicating the variable starting location. Where indirect addressing is used, that use is for access to variables whose locations can be inferred from previous direct or scaled addressing based on an offset indicating the variable starting location.

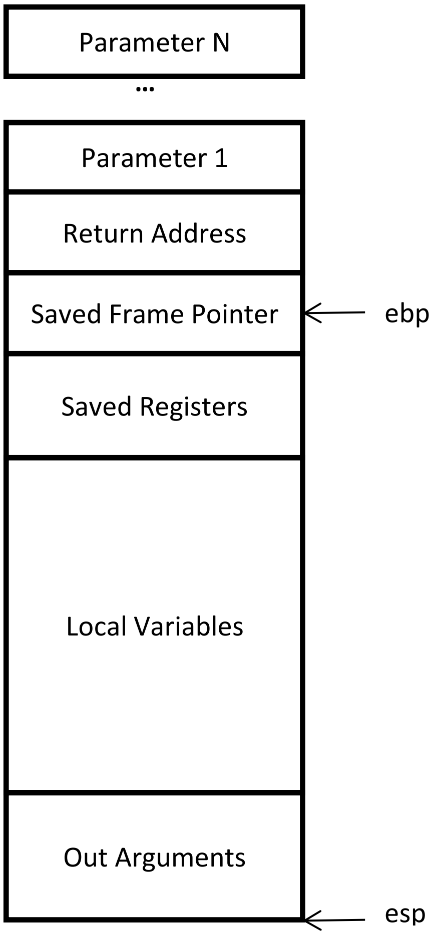


Figure 30. General Form of the Stack Layout

The second assumption does not necessarily hold, and so we use a speculative approach. The assumption is used to create an initial layout inference and an inference of which instructions access variables in this layout. These inferences are then evolved and refined.

Layout inferences are produced using (a) a set of simple heuristics, (b) assumptions about the manner in which the stack is allocated and deallocated, and (c) assumptions about the general stack frame layout. These assumptions hold for binaries produced by C/C++ compilers that use the cdecl x86 calling convention in which stack frames are in the form seen in Figure 30.

A key requirement for the heuristics is the use of an out arguments region for outgoing parameters for called functions. In principle, any calling convention where this general structure is maintained could be supported. Dynamically allocated stacks are out of scope for this work, as are compiler conventions where the out arguments region is not used and the stack expands and contracts before and after each function call.

To support the modification of instructions that access the stack, in the implementation of stack-layout randomization, all of the instructions in the binary program are inserted individually into a database. During static analysis, the binary program is processed by a static analysis system. We use a recursive descent disassembler (IDAPro) [9] and a linear scan disassembler (objdump). To ensure we have all instructions in the database, we added a disassembly validator module. The disassembly validator iterates over every instruction found by both IDA Pro and objdump, and verifies that both the fallthrough and (direct) target instructions are inserted into the instruction database. This information in the database allows the control-flow graph to be constructed.

Since exact instruction start locations in the executable segment are not known, some of the instructions in the instruction database may not represent instructions that were intended by the program's original assembly code. We make no attempt to determine which are the intended instructions and which are not. Instead, SLR modifies all instructions matching patterns of local variable access (see below). Any data address that is mis-identified as a code address will not be executed and will result in corrupted data if transformed by SLR. We rely on testing to catch those cases. We believe the probability of data being interpreted as a stack accessing instruction is low.

Finally, static analysis determines the functions and the out-arguments region size that IDA Pro detects. Once static analysis is complete, the next part of the approach is to find out if the subject function has a stack frame. The presence of a local-variable region in the stack is detected by scanning each instruction in the entry block of the function's control-flow graph for a stack allocation instruction:

```
    sub esp,<constant>
```

Scanning only the entry block avoids cases where a stack may be allocated differently depending on some condition. If the entry block does not contain this pattern, the function is determined to be non-transformable and is skipped, and the transformation process is restarted for the next function. Otherwise the stack size, the number of saved registers, and the out argument region size are recorded. The size of the out-argument region is determined during static analysis.

Once the existence of the local-variable region is established, each instruction of the function is analyzed and a set of local variable boundary inferences are determined based on instruction patterns accessing the stack. Memory access patterns accessing the stack can take one of three forms:

- Direct: A constant value added to `esp` or subtracted from `ebp` (e.g., `[esp+0x10]`).
- Scaled: A constant value added to an indexed (variable) offset from `esp` or `ebp` (e.g., `[esp+eax+0x10]`).
- Indirect: The stack is accessed by means other than through offsets to `esp` or `ebp` (e.g., `[ebx]`).

Inferring a boundary interface for every constant offset found in both direct and scaled memory accesses is a naïve approach but surprisingly high levels of accuracy have been reported [36].

The possibility of inaccuracy is the reason we describe SLR as speculative. SLR assumes these inferred boundaries are correct and determines their validity by testing. If testing reveals that the program's semantics have been changed, three additional heuristics are used: (1) variable boundaries are assumed for direct memory access offsets only, (2) variable boundaries are assumed for scaled memory accesses only, and (3) the entire stack frame is treated as one large variable as a “catch all” case.

In summary, the four inferences are:

- All Offset Inference (AOI): Any constant offsets of `ebp` and `esp` for direct and scaled accesses that access the local variables region of the stack are considered local variable boundaries.

- Scaled Offset Inference (SOI): Any constant offset used in scaled stack access instructions is considered a local variable boundary.
- Direct Offset Inference (DOI): Any constant offset used in direct stack access instructions is considered a local variable boundary.
- Entire Stack (ESI): The entire stack frame is considered one local variable.

A memory access relative to `esp` or `ebp` may access incoming parameters to the function, since the size of the frame is known, if a boundary exceeds the stack frame these offsets are ignored. Additionally, the out arguments region contains reusable space, and so any offset accessing this region is omitted from the layout inference.

For each function, all four inferences are applied to form four individual hypotheses. These four hypotheses are sorted by the number of variables detected and applied in order, highest to lowest. In practice, as might be expected, the order is almost always AOI, DOI, SOI, and ESI.

3.8.2.2 Stack-Layout Randomization Algorithm

The overall SLR process is shown in Figure 31. This process is applied to each function in the program independently. Thus, the whole process shown in Figure 31 is repeated a number of times equal to the number of functions in the program.

The four hypotheses of the stack layout for the subject function are created for each function in the binary program using the four inferences described above, and test data is created or acquired for the program. Recall that the layout hypotheses are limited to the boundary address between individual variables in the stack frame, and that no attempt is made to determine the types of the variables.

The dependence of this approach on testing raises the question of where test cases will come from. The approach has no specific requirements that constrain the test data. Existing test cases can be used or new tests developed using any test-case development technology. Novel test-coverage metrics are suggested by the SLR concept. When assessing the effects of a transformation by testing, measurement of the extent to which stack variables have been referenced or set would provide insight into the degree to which the randomization has been assessed. We do not presently capture this coverage information.

Using the hypothesized stack layout determined by AOI (AOI is the heuristic that detects the largest number of variables because it is the most aggressive), a new layout is created by randomizing the hypothesized layout for the subject function (randomization 1 in Figure 31). In this randomization, space for variables is contiguous, i.e., there is no padding between variables.

Once the randomization has been created, the necessary binary rewriting rules for dynamic binary translation are defined to transform the binary function from its original stack layout to the new, randomized layout, and then the program is tested.

If testing is successful, then the entire process is repeated for the subject function with the same hypothesis about the stack layout but a different randomization (randomization 2 in the Figure 31), again without padding. The program is then tested again. This second transformation is carried out to check the results of the first transformation. The claim is that, if two randomizations leave the program in a form that passes the available test cases, the chances of the randomizations having altered the program's semantics are small.

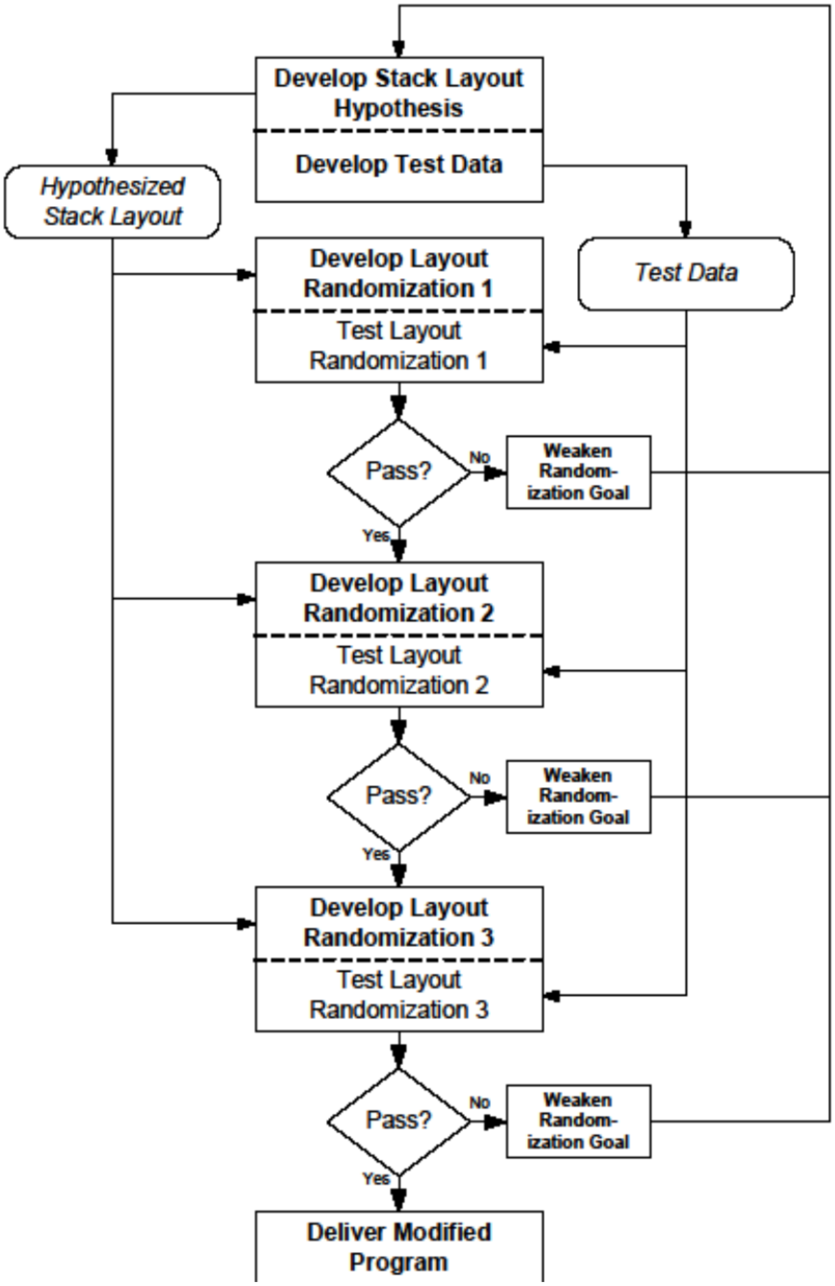

Develop Stack Layout Hypothesis
Develop Test Data
Hypothesized Stack Layout
Test Data
Develop Layout Randomization 1
Test Layout Randomization 1
Pass?
No
Yes
Weaken Random-ization Goal
Develop Layout Randomization 2
Test Layout Randomization 2
Pass?
No
Yes
Weaken Random-ization Goal
Develop Layout Randomization 3
Test Layout Randomization 3
Pass?
No
Yes
Weaken Random-ization Goal
Deliver Modified Program

Figure 31. Overall Approach to Stack Randomization

If testing with the second randomization is successful, a third transformation is applied to the subject function, again starting with the original binary program and the original stack-layout hypothesis. In this third transformation, random padding whose length lies between 4096 and 8192 is added between local variables, before the stack frame, and after the out arguments region (if one exists). This third randomization is then tested as the other two were. If testing is successful, this is the randomization of the stack that is used in the transformed program.

If testing fails for any of the three randomizations of the subject function, then the semantics of the program have been changed by the stack randomization, and so the hypothesized stack layout must have been wrong. In that case, the hypothesized stack layout that detected the next largest number of variables in the function is selected, and the three-phase test process is repeated. Again, any failing tests lead to abandoning of the hypothesized layout, and the process resumes with the next one.

In the worst case, all four hypothesized stack layouts from the four different inferences will affect the program's semantics and be rejected. In that case, the subject function is left unmodified. We present data on the application success with these four approaches to stack layout analysis below.

This speculative transformation and assessment process is repeated for each function in the program. Needless to say, for a large program with many functions, this process requires a lot of resources. Fortunately, the process can be conducted in parallel for all the functions in a program, and so the time taken for the analysis can be reduced by using additional equipment. Also, optimizations are possible in the transformation mechanism. For example, since transforms have a high success rate, processing more than one function at a time might be possible.

3.8.2.3 Randomization by Dynamic Binary Translation

SLR uses the dynamic rewriting technique described in Section 3.3.5. The translations in SLR that the PVM applies are the modifications to instructions that are needed to implement the randomization of the stack layout. These modifications are documented as a list of rewrite rules. The rewrite rules define the instructions that have to be modified, the associated modifications, and the fallthrough map, i.e., the address of the next instruction. The rewrite rules are written by the stack randomization system based on a randomization of a stack-layout hypothesis.

In SLR, the PVM loads the rewrite rules and the PVM's instruction-fetching mechanism checks and then reads the SLR rewrite rules as appropriate. After modifying an instruction, the PVM modifies the PC from the fallthrough map that SLR provides in the rewrite rules. (see Hiser et al [113] for further PVM details).

3.8.2.4 Experimental Evaluation

To evaluate SLR we conducted three assessment experiments:

- We measured the effectiveness of the transformation mechanism for a set of open-source programs for which accepted regression tests are available.
- We measured the performance overhead imposed by the transform using SPEC 2006 benchmarks [199].
- We applied SLR to a set of test programs designed to measure protection against buffer-overflow attacks.

The results of our evaluation are described in Section 4.2.

3.8.3 Phase 2 Heap Randomization

Memory management errors in the allocation, deallocation, and use of dynamically allocated memory leave applications vulnerable to attack. Heap exploits use knowledge of the heap layout and take advantage of memory management errors to launch arbitrary attacks such as denial of service or arbitrary code execution.

Memory management errors can be divided into several categories. Heap overflows and underflows occur when a buffer’s boundaries are breached. Dangling pointers is a situation that arises when a program frees an object that is still in use. A double free occurs when an object is de-allocated multiple times. Invalid frees occur when a program attempts to de-allocate objects that were not previously allocated.

Memory allocation functions such as `malloc` and `free` for C and `new` and `delete` for C++ are implemented in the runtime libraries. The implementation of these functions depends on the operating system. PEASOUP does not implement a full memory allocator. Rather, heap protection techniques are implemented in PEASOUP using the Strata software dynamic translator. The techniques implemented are similar to existing published heap protection techniques [85, 92, 93, 209]. These protection techniques are included to add depth to the defenses offered by PEASOUP.

3.8.3.1 Randomizing Heap Object Sizes

PEASOUP intercepts library calls which allocate dynamic memory such as `malloc,` `calloc,` and `realloc.` A randomized amount of additional padding above the size requested is added to the memory allocation. This padding technique gives probabilistic protection that a heap buffer overrun will not reach the allocated object's metadata or an adjacent heap-allocated object.

3.8.3.2 Randomizing Heap Memory De-allocation

PEASOUP also adds randomization to the time at which memory is de-allocated. This is discussed in further detail in the following section.

3.8.4 Phase 1–2 Heap-Usage Confinement

PEASOUP implements several known techniques for confining heap usage. Delaying frees prevents dangling pointer/use-after-free errors. Monitoring excessive allocations counteracts potential heap exhaustion attacks. Preventing frees of non-heap memory addresses potential memory corruption errors. Monitoring for and protecting against double free attempts prevents corruption of memory management data structures. These techniques are discussed in more detail in the following sections.

3.8.4.1 Delayed Frees

In PEASOUP, when a call to `free` is encountered to de-allocate memory, the memory is marked as free, but de-allocation of the memory chunk is delayed for a randomized time period. Actual freeing of the memory chunk occurs at PEASOUP's discretion. If the heap is deemed to be “overfull” (over a threshold value), then chunks that are marked as free, but have not yet been freed will be freed. This technique probabilistically addresses dangling pointer (“use after free”) errors by extending the effective lifetime of the freed object.

3.8.4.2 Excessive Allocation Sizes

The size parameter passed to heap allocation functions is monitored for excessive size requests. If the requested memory allocation size is over threshold, various policies might be enforced. For Phase 1, the enforcement policy printed a warning message describing the potential excessive allocation and effected a controlled exit of the application. Monitoring the requested allocation sizes addresses the scenario when a sign conversion and/or integer wraparound error occurs and that value is then passed to a memory allocation function.

3.8.4.3 Frees of Non-Heap Memory

All heap allocations are tracked during runtime. When a call to `free` is encountered for a particular address, the list of currently allocated heap memory is searched. If the address is not in the allocation list, PEASOUP produces a warning message describing the error encountered. Continued execution could be attempted, but for Phase 1, controlled exit was the policy used.

3.8.4.4 Double Frees

A double free error occurs when heap-allocated memory which has been previously freed is freed another time. PEASOUP addresses this error by maintaining a list of existing heap allocations. When `free` is called for a particular memory address, the allocation list is checked. If the address is not in the list, then an appropriate policy can be applied. For Phase 1, recording a warning message and affecting a controlled exit of the application was the policy selected.

3.8.5 The Twim Allocator: Phase 3 Heap Protection

In Phase 3, we changed PEASOUP's approach to protecting the heap. PEASOUP provides a heap manager, called the *Twim Allocator*, which is used in place of the standard heap allocator. Twim is closely based on the DieHard Allocator [85, 92]. For small and medium-sized requests, Twim uses the following strategy: each request is rounded to a power of two. A separate pool of block is maintained for each block size (i.e., each power of two up to the configurable cutoff, currently 64k). Within each pool, Twim allows reserves at least a constant factor greater than the maximum amount requested by the application from that pool. When fulfilling a request, Twim selects a free block from the pool *at random*.

For large requests, Twim uses a strategy that is closer to that used by glibc malloc. However, in Twim, all metadata, for all block sizes, is kept separate from the blocks returned to the application.

Like DieHard, Twim offers substantial protection against heap exploits: the random allocations complicate exploits based on the exact layout of the heap and make use-after-free exploits unlikely to succeed. The fact that metadata is not incline makes exploits based on corrupting the metadata significantly more difficult. Compared to the heap protections used in Phase 1 and 2, Twim's offers the following advantages:

1. The protections are easier to quantify (see [85]).
2. It provides an efficient mechanism to check if a given address is in a currently allocated block, and if so at what address. This capability is used by DieHard.
3. It is more robust against (unsafe) allocations from a signal handler, due to its extensive use of lock-free data algorithms and data structures.
4. It provides a garbage collection capability which can be used to survive in the face of extensive memory leaks.

Twim also imports some of the heap protections developed during Phase 1 and 2. In particular, it supports a *free quarantine* that is used to queue freed blocks and delay the time of their actual return to the heap [186].

3.9 C8: Null-Pointer Errors

PEASOUP focuses on defending against Null-Pointer Dereferences (NPDs) in server applications. Users can usually tolerate premature termination of a non-server application. PEASOUP's primary defense against NPDs is based on the observation that most server applications may tolerate the loss of a thread better than an uncaught SIGSEGV. Based on this observation, PEASOUP catches the SIGSEGV and performs a controlled-exit from the thread that caused the NPD. The controlled exit is performed by calling `exit(-1)`.

Some applications may already be prepared to handle a SIGSEGV. In order to limit altered functionality, PEASOUP only enables its defenses against NPDs (i) after a program indicates it is a server by listening for a socket connection and (ii) if the program does not install its own SIGSEGV handler.

3.10 General Defenses

Some PEASOUP defenses prevent many different types of exploits, regardless of the type of weakness being exploited. We describe some of those defenses in this section.

3.10.1 Instruction-Layout Location Randomization (ILR)

In this section we describe a novel technique, called Instruction Location Randomization (ILR) that conceptually randomizes the location of *every* instruction in a program. ILR can use the full address space of the process (e.g., 32-bits on 32-bit processors such as the x86). Information leakage attacks that discover information about the location of a code block (e.g., the randomized base address of a dynamically loaded module, the start of a function) are infeasible for two reasons: 1) the randomized code addresses are protected from leakage and 2) a leak provides no information about the location of other code blocks.

ILR changes a fundamental characteristic typically used by attackers---predictable code layout. For example, programs are arranged sequentially in memory starting at a base address, as shown in the left of Figure 32.[17]

In this example, the address used to return from function `foo` (7003) might be leaked if there is a vulnerability in the function. An attacker that learns this information can easily determine the location of all other instructions. Attackers routinely rely on the fundamental assumption of predictable code layout to craft attacks such as arc-injection and the various forms of return-oriented programming. In the example, an attacker might use the address of the `add` instruction to mount an ROP attack using `add eax, \#1;ret` as an ROP gadget.[18] For a detailed explanation of ROP gadgets and how they are combined to form an attack, please see Shacham's prior work [110].

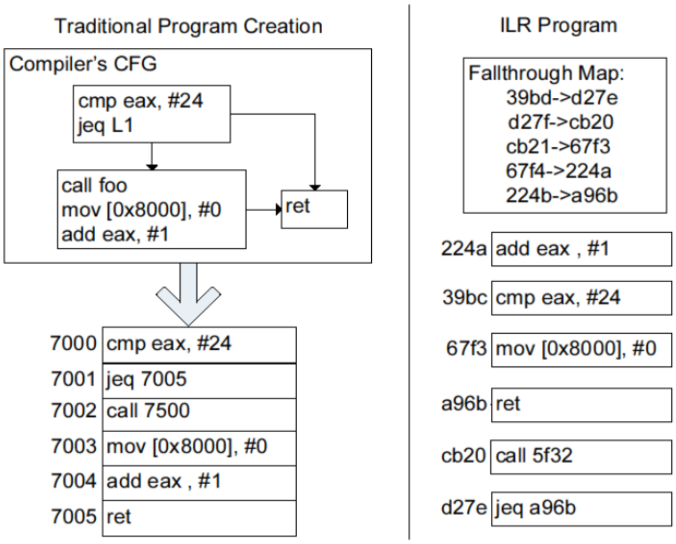
ILR adopts an execution model where each instruction has an explicitly specified successor. Thus, each instruction's successor is independent of its location. This model of execution allows instructions to be randomly scattered throughout the memory space. Hiding the explicit

[17] For simplicity, the figure and discussion assume all instructions are one byte. Our general approach, prototype implementation, and security discussion do not rely on this fact.

[18] ROP gadgets are short sequences of code, typically ending in a return instruction, that perform some small portion of the attack.

successor information prevents an attacker from predicting the location of an instruction based on the location of another instruction.

ILR's "non-sequential" execution model is provided through the use of a process-level virtual machine (PVM) based on highly efficient software dynamic translation technology [89, 138, 182]. The PVM handles executing the non-sequential, randomized code on the host machine.



```
cmp eax, #24
jeq L1

call foo
mov [0x8000], #0
add eax, #1

ret
```

```
7000 cmp eax, #24
7001 jeq 7005
7002 call 7500
7003 mov [0x8000], #0
7004 add eax , #1
7005 ret
```

```
Fallthrough Map:
  39bd->d27e
  d27f->cb20
  cb21->67f3
  67f4->224a
  224b->a96b
```

```
224a add eax , #1
39bc cmp eax, #24
67f3 mov [0x8000], #0
a96b ret
cb20 call 5f32
d27e jeq a96b
```

Figure 32. Traditional Program Versus an ILR Program

Several key contributions of the PEASOUP effort are presented in the remainder of this section. In particular, this section

- presents Instruction Location Randomization (ILR), a technique that provides high-entropy diversity for relocating instructions with low run-time performance overhead.
- demonstrates that ILR defeats arc-injection and ROP attacks on arbitrary binaries without need for compiler, linker, operating system or hypervisor support.
- provides a complete description of how ILR can achieve its goals despite inherent uncertainty about a program's structure, such as where code and data reside.

This section has been written to be self-contained. As such, it repeats some background information described elsewhere in the report.

3.10.1.1 Threat Model

ILR uses a very similar threat model to the threat model supposed by STONESOUP. Like STONESOUP, ILR assumes that the unprotected program is created and distributed to an end user (and possibly the attacker) in binary form. The program has been tested, but not guaranteed to be free from programmatic errors that might allow malicious exploit, such as memory errors. The program is assumed to be free from intentionally planted back doors, Trojans, etc. Furthermore, the program is to be protected and deployed in a setting where the other software on the system is believed to be operating correctly, and the system administrator is trusted. An attacker does not have direct access to the system or the protected program, but understands the protection methodology and may have access to tools for applying ILR protections. However, the attacker does have access to the unprotected version of the program, and can specify malicious input to the protected program.

ILR does not make any assumption about the type of vulnerabilities that the attacker may be able to exploit. In particular, ILR focuses on preventing attacks which rely on code being located predictably. This threat model includes a large range of possible attacks against a program. For example, many attacks against client and server software fit this model. Document viewers/editors (Adobe PDF viewer, Microsoft Word), e-mail clients (Microsoft Outlook, Mozilla Thunderbird), and web browsers (Mozilla Firefox, Microsoft Internet Explorer, Google Chrome) need to be protected from these types of threats anytime a user requests the program to examine data from an untrusted source.

3.10.1.2 ILR Implementation

ILR's goals are to achieve high randomization and low run-time overhead. An attacker, through knowledge of the instruction-set architecture and the executable format, can easily locate portions of code that may be useful in crafting an attack. For example, the attacker may identify the instruction sequence at locations 7004 and 7005 as being a gadget useful in crafting an ROP attack. This particular gadget adds one to register `eax`. By identifying a set of gadgets and exploiting a vulnerability, an attacker can cause a set of gadgets to be executed that effect the attack.

The right side of the figure shows the layout of the code when ILR is applied. The program instructions are randomly scattered through memory. With an address space of 32 bits, it is infeasible for an attacker to locate a set of gadgets that could be used to craft an attack.

To execute the randomized program, we employ a highly efficient PVM that fetches and executes the instructions in the proper order even though they are randomly scattered throughout memory. This process is accomplished via a specification that describes the execution successor of each instruction in the program. The PVM interprets the fallthrough map to fetch and execute instructions on the host hardware. The following subsections describe the process of automatically producing an ILR executable and its execution.

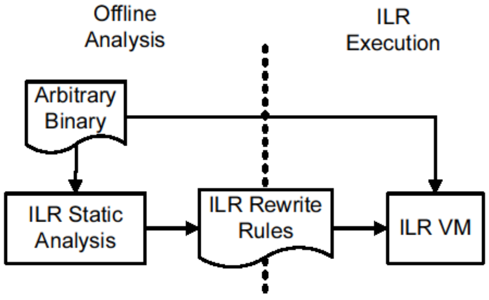


Figure 33. High-Level Overview of ILR

3.10.1.2.1 ILR Architecture

Figure 33 shows the high-level architecture of the ILR process. ILR has an offline analysis phase to relocate instructions in the binary and generate a set of rewriting rules that describe how and where the newly located instructions are to be executed, and how control should flow between them. The randomized program is executed on the native hardware by a PVM that uses the fallthrough map to guide execution.

The rewriting rules come in two forms, as described in Section 3.3.6.1. The first form, the instruction definition form, indicates that there is an instruction at a particular location. The first line of Figure 34 gives an example. In this example, address `0x39bc` has the instruction `cmp eax, #24`. Note that the rule indicates that if an instruction is fetched from address `0x39bc`, that it should be the `cmp` instruction. However, data fetches from address `0x39bc` are unaffected. This distinction allows ILR to relocate instructions even if instructions and data are overlapped.

An example of the second form of an ILR rewrite rule, the redirect form, is shown in the second line of Figure 34. This line specifies the fallthrough instruction for the `cmp` at location `0x39bc`. A normal processor would immediately fetch from the location `0x39bd` after fetching the `cmp` instruction. Instead, ILR execution checks for a redirection of the fallthrough. In this case, the fallthrough instruction is at `0xd27e`. The remaining lines show the full set of rewrite rules for the example in Figure 34.

```
39bc ** cmp eax, #24
39bd -> d27e
d27e ** jeq a96b
d27f -> cb20
cb20 ** call 5f32
cb21 -> 67f3
67f3 ** mov [0x8000], 0
67f4 -> a96b
224a ** add eax, #1
224b -> 67f3
a96b ** ret
```

Figure 34. Example ILR Rewrite Rules

The ILR architecture fetches, decodes and executes instructions in the traditional style, but checks for rewriting rules before fetching an instruction or calculating an instruction’s fallthrough address.

3.10.1.2.2 Offline Analysis

The static analysis phase creates an ILR program with random placement of every instruction in the program. For such randomization, the static analysis locates instructions, indirect branch targets, and identifies call sites for additional analysis. Figure 35 shows the organization of the static analysis used for ILR.

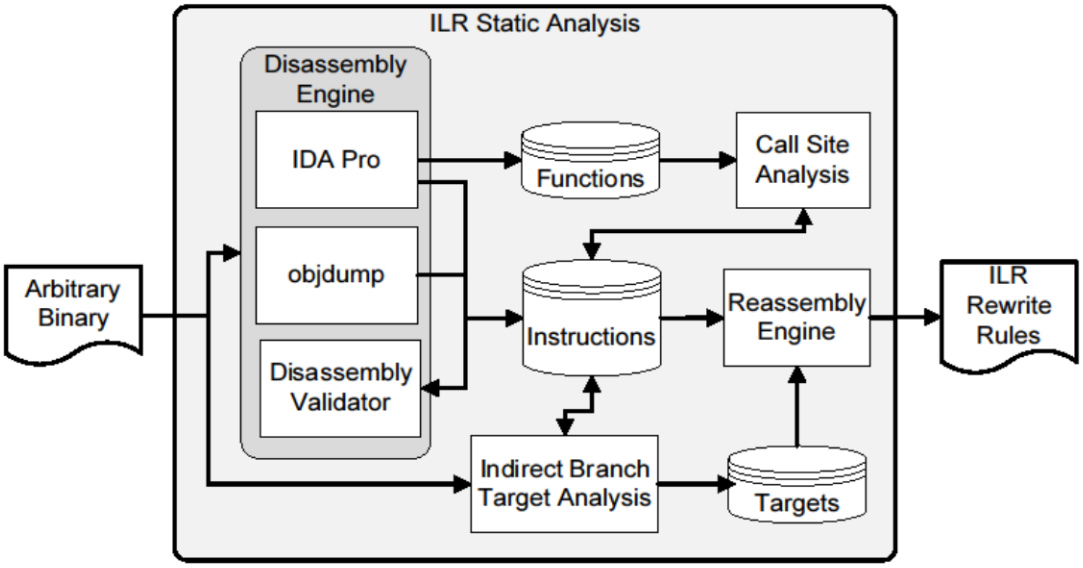


Figure 35. High-Level Overview of the STARS Analysis Engine Used in ILR

3.10.1.2.2.1*Disassembly Engine*

The goal of the ILR disassembly engine is to locate any byte that might be the start of an instruction. We use a recursive descent disassembler (IDA Pro) and a linear scan disassembler (`objdump`) [9]. To ensure that all instructions are identified, we added the disassembly validator module. The disassembly validator iterates over every instruction found by either IDA Pro and `objdump` and verifies that both the fallthrough and (direct) target instructions are inserted into the instruction database.

Since exact instruction start locations in the executable segment are not known, some of the instructions in the instruction database may not represent instructions that were intended by the program's original assembly code. We make no attempt to determine which are the intended instructions, and which are not. We simply choose to relocate all of them. Any data address that is mis-identified as a code address will not be executed, therefore the corresponding rewrite rules will simply never be accessed.

One last responsibility of the Disassembly Engine is to record the functions that IDA Pro detects. We record each function as a set of instructions.

3.10.1.2.2.2*Indirect Branch Target Analysis*

The goal of the indirect branch target analysis Phase 1s to detect any location in the program that might be the target of an Indirect Branch (IB). IBs create a distinct problem for ILR. The problem is that Indirect Branch Targets (IBTs) may be encoded in the instructions or data of a program, and it is challenging to determine which program bytes represent an IBT and which do not. Since we wish to randomize any arbitrary binary, our technique must tolerate imprecision in detecting which constants are an IBT in the program and which are not. Our solution is to perform a byte-by-byte scan of the program's data, and further scan the disassembled code to determine any pointer-sized constant which could feasibly be an indirect branch target.

We find that in most programs, this simple heuristic is sufficient (see Section 4.2 for details). However, when C++ programs use exception handling (`try-catch` blocks), the compiler uses location-relative addressing to encode IBTs for properly unwinding the stack, and invoking exception handlers. Our technique parses the portions of the ELF file that contain the tables used to drive the unwinding and exception throwing process, and records IBTs appropriately.

Rewriting the bytes in the program that encode an IBT might induce an error in the program if those bytes are used for something besides jumping to an instruction. To avoid breaking the program when the analysis is wrong, we choose to leave those program bytes unmodified. Unfortunately, not rewriting the IBTs encoded in the program means that the program might jump to the address of an original program (and hence unrandomized) instruction.

To accommodate indirect branches jumping to unrandomized addresses, each instruction that might be an IBT generates an additional ILR rule in the program. The additional rule uses the redirect form to map the unrandomized address to the new, randomized address. Thus, any indirect branch that targets an unrandomized address, correctly continues execution at the randomized address.

Unfortunately, attackers may know the unrandomized addresses in a program, and if they can inject a control transfer to one of these addresses, they might be able to successfully perform an attack.

The evaluation in Section 4.2 shows the number of IBTs detected in most programs is very limited, and restricting attacks to only these targets significantly reduces the attack surface.

3.10.1.2.2.3*Call Site Analysis*

Since unrandomized instructions may allow attacks, we wish to randomize the return address for function calls. The call site analysis phase analyzes the call instructions in a program to determine if the return address can be randomized. Typically, a `call` instruction stores a return address, and when execution of the function completes, a `ret` instruction jumps to the address that was stored. Most functions obey these semantics. Unfortunately, call instructions can be used for other purposes, such as obtaining the current program counter when position-independent code or data is found in a library. Such a call instruction is often called a *thunk*. Numerous other uses of a return addresses are possible.

The analysis proceeds as follows. If the call instruction is to a known location that starts a function, we analyze the function further. If the function can be analyzed as having only standard function exits (using the return instruction), having only entrances via the function's entry instruction, and having no direct accesses to the return value (such as with a `mov eax, [ebp+4]` instruction), then ILR declares that it is safe to rewrite the call instruction to store a randomized return address.

Our heuristic makes the assumption that indirect memory accesses should not access the return address. While not strictly true for all programs, we find that the heuristic generally holds for programs compiled from high-level languages. One exception to our heuristic is again the C++ exception handling routines that “walk the stack.” The routines use the return address to locate the appropriate unwinding, cleanup, and exception handling codes to invoke. Like with the IBT analysis, we adjust the call site analysis to take into account the exception handling tables, so that call sites with exception handling cannot push a randomized return address.

Once the analysis is complete, the ILR rules for calls need to be emitted. If the call site analysis determines that the call can randomize the return address, no additional rules are required, and the call instruction's location is randomized by simply emitting the standard rewrite rules. If, however, the non-randomized return address must be stored, we have two choices: 1) we could choose to pin the call instruction to its original location, so that the nonrandomized return address is stored, or 2) rewrite the call (using ILR rewrite rules) into a sequence of instructions that stores the unrandomized return address and transfers control appropriately. Since pinning instructions leads to a decrease in randomization, we choose the second option. Most machines can efficiently store the return address and perform the control transfer necessary to mimic a call instruction, typically using only 2-3 instructions. For example, on the IA32 instruction set architecture, a `call foo` instruction can be replaced with two instructions, `push <unrandomized address>; jmp foo`, resulting in only one extra instruction. This transformation is exactly what is performed by our call site analysis when we detect that a call instruction cannot push a randomized return address. Furthermore, the unrandomized return address is marked as a possible indirect branch target, since we are not sure how the return address in question will be used.

3.10.1.2.2.4***Reassembly Engine***

After completely analyzing the program's instructions, IBTs, and call sites, the reassembly engine gets invoked. The reassembly engine's purpose is to create the rewrite rules necessary to create the randomized program. For each instruction in the database, the engine emits a set of rewrite rules. First, it emits the rules necessary to relocate the instruction. Note that if the instruction has a direct branch target encoded in it (such as a `jmp L1`), that branch target is rewritten to the randomized address of the branch target. Then, the reassembly engine emits the rule to map the instruction’s fallthrough address to the randomized location for the fallthrough instruction.

As a post-processing step, each byte of the original executable text gets an additional rule. If the address of the program text is marked as a possible IBT, the reassembly engine adds a rule to redirect that address to the randomized address for that instruction, effectively pinning the instruction. Any other byte of the executable code segment gets a rule to map its address to a handler that prints an error message and exits in a controlled manner. Thus, any possible arc-injection or ROP attacks must jump to the start of a instruction, and not bytes located within an instruction.

3.10.1.2.3 **Running an ILR Program**

To apply the rewrite rules generated by the static analysis steps, ILR uses a specific ILR VM. We believe that a per-process virtual machine (PVM) is the best choice for the ILR VM since it can be easily deployed and has low performance and runtime overheads [135, 139, 182]. As described above, we use Strata to implement a PVM in PEASOUP.

3.10.1.3 Related Work

In this section, we focus on work specifically related to ILR.

3.10.1.3.1 **ROP Defenses**

The original authors of ROP have described ROP's salient feature as “Turing completeness without code injection” [200]. ROP invalidates the assumption that attack payloads are intrinsically external by nature as ROP re-uses code fragments already present in a target program. Defensive techniques such as various forms of instruction-set randomization that target code injection attacks directly are completely circumvented by arc-injection attacks in general [84, 112, 119], of which ROP, return-to-libc [144, 174], partial overwriting attacks of return addresses [82] are special cases. $W \oplus X$ is also circumvented as it implicitly assumes that external code will be executed from data pages [159].

Since the original seminal work on ROP [110], several defensive techniques have been proposed. Early defenses targeted what would emerge to be non-essential features of ROP attacks. For example, DROP [163] instruments binaries searching for short consecutive sequences of instructions ending in a return instruction. Li et al. avoid return instructions altogether when generating code [116]. ROPDefender [134] and TRUSS [177] look for mismatched calls and returns essentially using a shadow stack.

Checkoway et al. showed that the use of the return instruction is not a necessary condition in building ROP gadgets, thereby bypassing such ad-hoc defenses [200]. The balance against ad hoc defenses is further tilted by recent works that have automated the process of gadget discovery [10] [204] [173] and ROP exploit compilation and hardening [83].

TRUSS [177], ROPDefender [134], DROP [163], TaintCheck [115] use software dynamic translation frameworks for instrumenting code and implementing their respective defenses. TaintCheck uses dynamic taint analysis and provides a comprehensive approach to thwarting ROP attacks by detecting attempts at control-flow hijacking, though it suffers from high overhead (over 20X). Performance overhead for ROPDefender is approximately 2X overhead on the SPEC2006 benchmarks, while preliminary performances measurements for DROP range from 1.9X to 21X. While not directly comparable, ILR achieves average performance overhead of only 13–16%, which makes it practical for deployment.

3.10.1.3.2 **Defenses based on randomization**

In contrast to approaches that look for specific ROP patterns, ILR provides a comprehensive defense based on high-entropy diversification to thwart attacks. ILR provides 31 bits of entropy (out of a maximum of 32 for our experimental prototype) which makes derandomizing attacks impractical. ASLR on a 32-bit architecture only provides 16 bits of entropy and is susceptible to brute-force attacks [190]. Even on 64-bit architectures, there would be two potential problems.

The first is that ASLR is not applied universally throughout the address space. Even when using dynamically-linked libraries, it is common for the main program text to start at a known fixed location. Red Hat developed Position Independent Executable to remedy this situation [31]. However, PIE requires recompilation.

The second problem is that ASLR and other coarse-grained technology such as PIE do not randomize intra-library. Any information leaked as to the location of one function, or even one address, could be used to infer the complete layout of a library. Roglia et al. demonstrated a single-shot return-to-libc attack that used ROP gadgets to leak information about the base address of libc, and bootstrapped this information into all other libc functions [94]. Their proposed remedy of encrypting the Global Offset Table was specific to their attacks and leaves open the possibility of other leakage attacks.

Bhatkar, et al. use source-to-source transformation techniques to produce self-randomizing programs (SRP) to combat memory error exploits [175]. Unlike other compiler-based randomization techniques [114], SRP produces a single program image, which makes it more practical for deployment. SRP randomizes code at the granularity of individual functions and therefore retains a larger attack surface than the ILR approach of randomizing at the instruction level.

3.10.1.3.3 Control Flow Integrity

Control flow integrity (CFI) is designed to ensure the control flow of a program is not hijacked [24]. CFI relies on the Vulcan instrumentation system. The Vulcan system allows instruction discovery, static analysis, and binary rewriting.

Figure 36 shows an example program. In the figure, CFI enforces that the return instruction (in function `log`) can only jump to the instruction after a call to the `log` function. In this case, this policy allows an arc-injection attack if the `log` function is vulnerable. An attacker might be able to overwrite the return address to erroneously jump to L2, thereby granting additional access. Even the best static analysis cannot mitigate these threats using CFI.

Further, a partial overwrite attack might defeat ASLR in this example, since the distance between the two return sites is fixed. Since ILR randomizes this distance, ILR can defeat partial-overwrite attacks.

```
      call log
L1:   cmp [isRoot], #1
      jeq L3
      ...
      call log
L2:   call grantAccess
L3:   ...
log:  ...
      ret
```

Figure 36. Example Weakness with CFI

3.10.2 Secure In-process Monitoring (SIM): Phase 2 Protection of PEASOUP

This section discusses the techniques we used to protect PEASOUP, itself, from attack in Phases 1–2. These protections were superseded by the dual-process architecture described in the following section.

As shown in Figure 37, the Phase 2 execution manager is constituted of two major components, the Strata software binary translator (SDT) and the SIM sandbox. In this architecture, Strata is in charge of providing fine-grained confinement for the SOUP. However, as it is a user-level component, it is running in the same process address space as the SOUP. This means it is vulnerable to attacks once SOUP is compromised due to unhandled weakness class. That is, instead of try to compromise the SOUP (which is extremely difficult under Strata's confinement), an attacker could try to compromise Strata first. To overcome this limitation, we include an out layer sandbox, SIM to protect Strata from being attacked by SOUP.

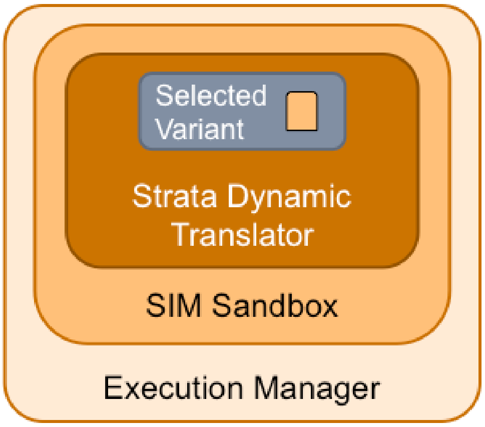


Figure 37. Phase 2 Execution Manager

SIM was original refer to our Secure In-VM Monitoring framework, which is designed to work with kernel-mode monitors. Because Strata is a user-mode SDT, to make them work together requires tremendous work, i.e., porting Strata to the kernel. Besides, the “untrusted kernel” assumption in the original Secure In-VM Monitoring work also makes it overkill for PEASOUP project, where the OS kernel is trusted. For this reason, we designed a new confinement technique called Secure In-process Monitoring.

3.10.2.1 Desgin

In this section, we present the design of our Secure In-process Monitoring technique.

3.10.2.1.1 **Threat Model**

In Phase 1, we assumed the analysis process and the protection provided by Strata is not perfect, therefore allowing an adversary to exploit vulnerabilities in the SOUP. The result of a successful exploit is not limited. And the adversary can launch complicated, multiple step attacks that may involve several vulnerabilities. Besides, we also assume the adversary knows of the existence of Strata, therefore can perform Strata-aware attacks.

However, we assume the whole environment besides the SOUP is trusted. The environment includes the virtual machine monitor (VMM), the operating system (OS), and the third party libraries and other processes that run inside the same virtual machine (VM). We assume these components cannot be exploited, nor be used to launch attacks targeting SOUP.

3.10.2.1.2 **Security Requirement**

The successfully confine attacks against Strata, we have following security requirements for our SIM technique:

- **Isolation**.
    - S1: Strata’s code and data must be isolated from SOUP
    - S2: Strata’s code cache can only be modified by Strata
- **Secure Invocation**
    - S3: Strata’s code should only be invoked from designated point
- **Integrity**
    - S4: The behavior of Strata should not be maliciously alterable

3.10.2.1.3 Bi-view based Confinement

Under our threat model for Phase 1, we satisfied the above security requirements through a bi-view based confinement technique. In this design, a single process address space will have two different memory access permission schemes, called views. The view corresponds to SOUP's context is called SOUP view, and the view corresponds to Strata's context is called SIM view. The differences between these two views are (Figure 38):

> **SOUP view**. Under this view, the original SOUP code will be mapped non-executable and the SOUP data will be mapped writable but not executable. The only executable code is the translated code in Strata's code cache and the entry/exit gate, but all kept un-writable. To protect Strata, its code and data will be mapped inaccessible.
>
> **SIM view**. Under this view, the code of Strata will be executable and the data will be writable, which allows Strata to perform the translation. And the gate will be both writable and executable. However, to avoid any execution of untrusted code, the original SOUP code will be kept non-executable.

View is switched via entry gates and exit gates. When translation is needed, the control flow will go through an entry gate and switch the context to Strata. And after the translation is done, the control flow will go through an exit gate so the context will be switched back to SOUP.

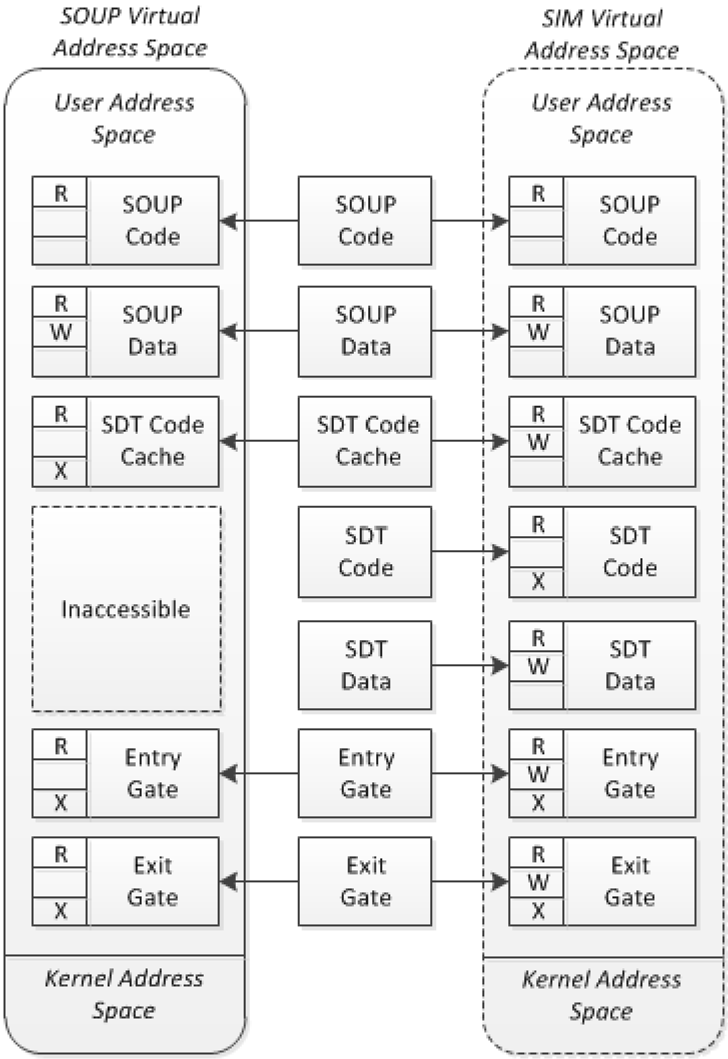


Figure 38. Bi-View Based Confinement

This confinement technique is built upon the hardware's memory protection support (i.e. page access permission). Since the only overhead is from the context switching, and context switching is rare due to Strata's optimization, as will show in the evaluation section, the performance overhead is very low.

3.10.2.1.4 Security Analysis

By using this bi-view based confinement technique, we can achieve following securing properties:

- **Isolation**
  - Strata's code and data are mapped inaccessible under SOUP view, this satisfies requirement **S1**;
  - Strata's code cache is mapped writable only under SIM view, this satisfies requirement **S2**;
  - SOUP cannot change access permissions of memory regions belong to Strata, this means the confinement cannot be break by SOUP.
- **Secure Invocation**
  - Strata's code is executable only after going through an entry gate; and
  - The destination of an entry gate is not modifiable for SOUP. By these two, our design satisfies requirement **S3**.
- **Integrity**
  - Strata is self-contained, combined with the isolation, this means Strata's behavior cannot be changed by SOUP;
  - While translation, Strata will mask-off signals, this means Strata's execution cannot be interrupted by SOUP. Together, this satisfies requirement **S4**.

3.10.2.1.5 System Components

SIM has three main components:

Memory Tracking Module. The memory tracking module tracks the virtual memory regions used by Strata and updates the view data structures accordingly. This information will be used to build the two different views.

Permission Change Checking Module. The permission change checking module has two jobs: first, it uses information collected by the tracking module to prevent the SOUP from changing the permission of the memory regions that belong to Strata; second, it prevents the SOUP from creating executable memory regions.

Entry and Exit Gates. Entry and exit gates are in charge of changing access permissions correctly during a context switch.

3.10.2.1.6 Integration with Strata

The interaction between Strata and SIM happens in two areas: memory usage tracking and context switch.

Memory Usage Tracking. The main difference between the SOUP view and SIM view relates to the access permissions of Strata's code, data and code cache. Therefore, to correctly change permissions during a context switch, SIM needs to know which memory region is used by Strata.

Moreover, since some regions allocated by Strata should be readable or writable under SOUP view, SIM also needs to know which regions should be given what access permissions under each view. Based on the fact that Strata uses *mmap/munmap* syscall to allocate/free memory from the system, SIM provides following functions to help Strata provide the required information:

```
sim_mmap(void *addr, size_t length, int prot_strata, int prot_soup, int
flags, int fd, off_t offset);

sim_munmap(void *addr, size_t length);
```

The parameters are generally the same as in *mmap/munmap* syscall, the only difference is *prot_soup*, which tells SIM what access permission should be granted to the allocated memory region under the SOUP View.

Context Switch. Strata has its own context switch mechanisms. Each has different purpose. Among them, the most important and most frequently used one is the trampoline. It is used when execution in the code cache has met a piece of code that has not been translated yet. A trampoline will save the current context and switch to Strata (in most cases, the translation entry). After the translation is done, the context will be restored and the execution will continue from the newly translated code. Some other cases like syscall monitoring and signal handling also requires context switch.

Once protected by SIM, these old context switch mechanisms will not work because the view is has not been changed yet. Therefore, another kind of interaction between SIM and Strata is context switch: when switching the context to Strata, a call to SIM (i.e. entry gate) must be made to also switch the view to SIM view; and before the context is switched back to SOUP, another call to SIM (i.e. exit gate) must be made to switch the current view back to SOUP view again.

3.10.2.2 Implementations

In Phase 2, we had two implementations of SIM: one is purely implemented at user-mode, and another one is implemented at kernel-mode. The user-mode implementation is easier to deploy because it is compatible with vanilla Linux kernel, while the kernel-mode implementation is more efficient, has more restricted access scheme for SOUP view and has more atomic context switch.

3.10.2.2.1 **User-Mode Implementation**

This section describes user-mode implementation of SIM.

View Data. The *View Data* is composed of *memory area descriptors*. Each memory area descriptor describes a memory area used by Strata, including the starting address, the length of the area, the permission under the SOUP View and the permission under the SIM View.

```
struct sim_vm_struct {
    void* addr;
    size_t length;
    int prot_strata;
    int prot_soup;
};
```

Based on the observation (from the source code) that Strata does not allocate/free memory frequently, we use an array to arrange the descriptors and use linear search to update the View

Data. If performance is detrimentally affected in the future, we will switch to binary search tree (e.g. RBTree) to store these descriptors.

We used three functions to allocate (*sim_vm_init*) the memory for the view data, to insert (*sim_vm_insert*) a new region, and to remove a region (*sim_vm_remove*). The view data is kept read-only except for updating (insert & remove) to prevent tampering.

Memory Tracking Module. In user-mode implementation, this module is implemented as a wrapper over the *mmap/munmap* syscall and is linked as part of Strata.

> **sim_mmap**: when Strata allocates memory through this interface, it first calls the real system service to allocate the memory; and if the allocation is successful, it then updates the view data according to the given permission and the result of the syscall.
>
> **sim_unmmap**: when Strata frees memory through this interface, it first calls the real system service to delete the memory mapping; if the call is successful, it then remove the corresponding record from the view data.

To make use of the above functions, we modified two functions of Strata:

> **strata_get_mem**: use *sim_mmap* to allocate memory, the permission for SIM view is read-write, and the permission for SOUP view is read-only;
>
> **strata_get_executable_mem**: use *sim_mmap* to allocate memory, the permission for SIM view is read-write-executable (the trampoline has not finished executing after the view is changed, therefore needs to be kept executable) and the permission for SOUP view is read-executable.

Important information related to Strata's memory usage is where the .strata section is mapped. This ELF section contains all the code and static data of Strata hence should be protected at runtime. In user-mode implementation, this information is retrieved at stratafy time. That is, we modified the stratafier to save this information in two global variables: `strata_section_entry` and `strata_section_size`. The limitations of this approach are: 1) if the section is relocated at runtime, then this information will be incorrect; and 2) it only works for stratafied program, if the program wants to statically link Strata, then this information is missing.

Permission Change Checking Module. To prevent SOUP from maliciously modify the access permission scheme (i.e. view), we intercept the two related syscalls, namely `mmap` and `mprotect`. In user-mode implementation, this module will be implemented using the syscall watching mechanism provided by Strata. In particular, it registers two callback functions:

> **intercept_mmap**: this function masks off the PROT_EXEC permission, so the SOUP cannot allocate any executable memory region;
>
> **intercept_mprotect**: this function does two things, 1) it checks the ownership of the target memory region, if it belongs the Strata, then the syscall is aborted and an error code is returned, this prevents the SOUP from changing access permission of Strata's code, data, code cache and the entry/exit gates; 2) it masks off the PROT_EXEC permission, so SOUP cannot make a memory region executable.

Entry/Exit Gates. In user-mode implementation, because SIM is linked as part of Strata, we relaxed the access permission of SOUP view, for the convenience of implementation. This scheme is called **read-only scheme**:

> Under this scheme, instead of being completely inaccessible, Strata's code and static data (i.e. the .strata section) will be mapped as read-only under SOUP view, and the code will remain executable. The executable permission is left for the entry/exit gates which are now part of the .strata section. And the readable permission is left for the entry/exit gates to read the necessary metadata to locate the view data. Although the permission is relaxed, since that ELF section is still un-writable under SOUP view, we think it is secure enough in most scenarios.

Another problem in user-mode implementation is signal. To satisfy the security requirements, the view switch must be atomic and the execution of Strata cannot be alterable by SOUP. Since in user-mode, the execution can be interrupted by signals, we need to mask off uncritical signals before switching the view to SIM view and keep them masked off until the view is switched back to SOUP view. As a result, the entry/exit gates in user-mode implementation work as:

When entering Strata context, the entry gate will:

1. Mask off signals;
2. Change the current view to the SIM view and;
3. Jump to the handler function.

When leaving Strata context, the exit gate will:

1. Change the current view to the SOUP View;
2. Turn on signals and;
3. Jump to the target address in the code cache.

For this purpose, we implemented four functions:

strata_maks_sigs: this function masks the unnecessary signals, it is extracted from *targ_mutex_lock*, and relies on *init_new_sigset* to initialize the mask;

strata_unmask_sigs: this function restores the signals before entering Strata, it is extracted from *targ_mutex_unlock*;

sim_unprotect: this function first turns on the write permission of the .strata section; it then iterates the view data and changes the access permission of each virtual memory region to *prot_strata*;

sim_protect: this function first iterates the view data and changes the access permission of each virtual memory region to *prot_soup*; and then turns off the write permission of the .strata section.

Since Strata already has its own context switch mechanisms, we extended these mechanisms with above functions to create entry/exit gates. For example, we would like to modify the *trampoline* to make it an entry gate and modify *targ_exec* to make it an exit gate. The context switching would be changed to act as follows.

For **translation trampolines**:

They all will use an entry gate similar to following:

```
LEA ESP, ESP-32
PUSHAD
PUSHFD
CALL sim_mask_sigs
CALL sim_unprotect
PUSH frag
PUSH target_addr
PUSH &targ_exec
JMP handler
```

This also means that the responsibility for signal masking and memory access permission will be moved from *strata_enter_builder* to the entry gate. The function *targ_exec* will be modified to be an exit gate and works as:

```
MOV [ESP+52], EAX
ADD ESP, 8
CALL sim_protect
CALL sim_unmask_sigs
POPFD
POPAD
LEA ESP, [ESP+32]
JMP [ESP-24]
```

For **known syscall watching and unknown syscall watching:**

We use a pair of entry/exit gate to wrap the calling to the callback function:

```
CALL sim_mask_sigs
CALL sim_unprotect
CALL callback
CALL sim_protect
CALL sim_unmask_sigs
```

For **signal handler**:

Since signals cannot be delivered during execution of the signal handler, we do not need to mask signal here, so we just simply added a call to *sim_unprotect* at the beginning of the function.

3.10.2.2.2 Kernel-Mode Implementation

This section describes kernel-mode implementation of SIM.

Limitations of user-mode SIM implementation. When SIM is implemented as part of Strata, the implementation strategy has following drawbacks:

- To change the view, SIM has to make multiple syscalls.
    - One mprotect call to make Strata section writable;
    - One sigprocmask call to mask signals;
    - One mprotect call to make code cache non-executable;
    - And several mprotect calls to change access permissions of other memory pools.

- The view switch is not fully atomic.
- The tight integration makes it harder to separate permissions (i.e. making Strata fully inaccessible while keeping the entry/exit gates executable).

Moving SIM to the kernel. To overcome these limitations, we can move SIM to the OS kernel:

- Only one syscall is required to change the view, saving lots of context switch.
- Syscall is atomic from the perspective of process.
- Turning off all access permission to Strata under SOUP's view is straightforward, as all the entry/exit gates now reside in the kernel.

View Data. The data structure of view data and its manipulation algorithm is the same as in user-mode implementation. The difference is, the related metadata (e.g. the pointer to the view data array, the current size etc.) is now stored inside the `task_struct`, the data structure Linux kernel used to keep process related data. The reason to store the metadata inside this data structure is, 1) this data structure can be easily located using the `get_current()` macro, and 2) no lock is required.

Memory Tracking Module. Memory tracking module has been completely moved into kernel, though the interaction interface for Strata remains the same (*sim_mmap* & *sim_munmap*). The difference is, in user-mode SIM, these two functions are wrappers over the libc *mmap* and *munmap*; but in kernel-mode SIM, they are implemented as two new syscalls:

```
#define __NR_sim_mmap           337
#define __NR_sim_munmap         338
```

The implementation of these two syscalls is similar to user-mode:

sys_sim_mmap: when this syscall is called, it first calls the handler for `SYS_mmap` (i.e. `sys_mmap_pgoff`) to allocate the memory; if the allocation is successful, it then updates the view data according to the given permission and the result of the allocation.

sim_unmmap: when this syscall is called, it first calls the handler for `SYS_munmap` (i.e. `sys_munmap`) to delete the memory mapping; if the call is successful, it then remove the corresponding record from the view data.

Another difference in kernel-mode implementation is, in user mode SIM, the information of the .strata section (start address and size) is gotten when stratafying the target binary; but in kernel mode SIM, this information is gotten when the executable is loaded into the process' address space. More specifically, when new process is created, the Linux kernel will call `load_elf_binary` to map the executable file. We therefore modified this function to let SIM parse the ELF header to located the .strata section (using the name "strata") and record where this section is mapped address at runtime. This solution is better than user-mode implementation because it can get the accurate mapping address.

Permission Change Checking Module. For performance optimization, the checking module is separated into two parts:

- The *mmap* checking, which prevents the SOUP from mapping any executable region is still done through Strata's syscall monitoring mechanism.
- And the *mprotect* checking, which prevents the SOUP from changing access permission of Strata's memory regions (code, data and code cache) is moved to kernel. More specifically, SIM intercepts the mprotect syscall and performs the checking before the original handler is called. The reason is because the required information (view data) now resides in kernel.

We decided to leave mmap checking at user-mode because syscall monitoring will impose performance overhead on all processes, but Strata's syscall monitoring will only affect the very process.

Entry/Exit Gates. In kernel-mode implementation, we applied the restricted access permission to SOUP view as in the original design. That is, unlike in user-mode implementation, Strata's code and data will be mapped completely inaccessible under SOUP view.

As a result of this restricted access mode, the entry and exit gates can no longer be part of Strata. Therefore, during initialization, SIM will allocate a special memory region, which is also guarded by the kernel component, to store the gates. Each gate will call one of the following syscalls to switch the view:

```
#define __NR_sim_enter          339
#define __NR_sim_exit           338
```

The implementation of these two syscall is similar to user-mode implementation:

sys_sim_enter: this function first turns on the access permission (read, write & execute) of the .strata section; it then iterates the view data and changes the access permission of each virtual memory region to *prot_strata*;

sys_sim_exit: this function first iterates the view data and changes the access permission of each virtual memory region to *prot_soup*; and then completely turns off the access permission of the .strata section.

Because Strata do not have existing support for emitting syscalls (i.e. int 0x80), so we still implemented two user level functions (`sim_enter` and `sim_exit`) to call above syscalls to switch the view. This also makes SIM's integration with Strata more unified between user-mode implementation and kernel-mode implementation. However, as mentioned above, the .strata section will be mapped inaccessible, so these two functions will be copied to the special gate memory region to construct entry/exit gates. Another difference from user-mode implementation is, since the view is switched through syscall and syscall is atomic to process, so we don't need to mask signals before changing the view.

As of user-mode implementation, we implemented several entry/exit gates for the integration of Strata's context switch mechanism:

For **translation trampolines**:

They all will use an entry gate similar to following:

```
LEA ESP, ESP-32
```

```
PUSHAD
PUSHFD
CALL sim_unprotect
PUSH frag
PUSH target_addr
PUSH sim_exec
JMP handler
```

Since *targ_exec* is no longer executable once the view is switched, we do not use it as the exit gate. Instead, we build the exit gate *sim_exec* for translation as:

```
PUSH EAX
MOV EAX, __NR_sim_exit
INT 0x80
POP EAX
MOV [ESP+52], EAX
ADD ESP, 8
POPFD
POPAD
LEA ESP, [ESP+32]
JMP [ESP-24]
```

For **known syscall watching and unknown syscall watching:**

We use a pair of entry/exit gate to wrap the calling to the callback function:

```
CALL sim_unprotect
CALL callback
CALL sim_protect
```

For **signal handler**:

As Strata's signal handlers (`intercept_signal` & `intercept_signal_act`) will be inaccessible when the signal is triggered under SOUP view, we replaced them with two entry gates, namely *sim_intercept_signal* and *sim_intercept_signal_act*. Each gate will first call *sim_enter* to switch the view, then jumps to Strata's corresponding handler.

3.10.2.2.3 **Protection Switch**

As part of the requirement (in Phase II), we implemented two options to turn-off the protection of SIM. One is compile time option: all SIM integration related code is surrounded by `SIM` macro (kernel-mode integration is surrounded by `SIM_KERNEL` macro). And the other option is runtime: by setting the environment variable `SIM_PROTECTION` as `1` (default) or `0`, user can turn on or turn off SIM's protection.

3.10.2.3 Correctness Validation

The evaluation includes two parts: the first part is the regression test set buildup, and the second part is the performance test and optimization.

3.10.2.3.1 Regression Test Buildup

The test set for SIM has two parts: the correctness tests which demonstrate the correctness of the implementation; and the functional tests which demonstrate the effectiveness of the protection, i.e. prevention of attacks against Strata.

Correctness Test. SIM shares the same correctness test set as Strata. The main purpose for these tests is to ensure SIM does not break the functionality of Strata and other protection mechanisms. Correctness test includes:

- SIM branch test lists: all test scripts under STRATA_SIM_BRANCH.
- Strata trunk test list: all test scripts after merged into STRATA_TRUNK.

In addition, SIM includes following correctness test for verifying the correctness of the newly added four syscalls in kernel-mode implementation. These tests are:

- Memory usage tracking
    - **sim_mmap**: make sure sim_mmap syscall can correctly allocate virtual memory regions with correct permissions;
    - **sim_munmap**: make sure sim_munmap syscall can correctly unmap given memory regions.
- Permission change checking
    - **mmap**: make sure SOUP cannot allocate executable memory regions
    - **mprotect**: make sure SOUP cannot modify memory regions protected by SIM
- Entry/Exit gates
    - **sim_exit**: make sure all protected memory regions are under SOUP view after sim_exit
    - **sim_enter**: make sure all protected memory regions are under SIM view after sim_enter

Functional Test. To demonstrate SIM's protection capability, we classify all potential attacks under our threat model into two major categories: primitive attacks and advanced attacks. Primitive attacks represent the simplest attacks an adversary may launch to compromise Strata, they include:

- Strata data overwritten: a compromised SOUP tries to modify Strata's data to disable its confinement;
- Strata code overwritten: a compromised SOUP tries to modify Strata's code to disable its confinement;
- Strata code cache overwritten: a compromised SOUP tries to modify Strata's code cache to disable its confinement;
- SOUP code overwritten: a compromised SOUP tries to modify the original SOUP's code to disguise itself;
- Strata code execution: a compromised SOUP tries to execute Strata's code;
- SOUP code execution: a compromised SOUP tries to execute the original SOUP's code without Strata's translation.

Advanced attacks include but are not limited to:

- ROP to Strata: use Strata's code as code gadgets to launch return-oriented programming (ROP) attack;
- ROP to SOUP: use original SOUP's code as gadgets to launch ROP attack;
- ROP to libc: when the SOUP is dynamically linked, use libc's code as gadgets to launch ROP attack;
- Two-step attack: use one attack to learn runtime information about our execution manager, then launch specified attack to bypass some protections.

We believe all advanced attacks can be break into one or more successful primitive attack, so in Phase 1, we will focus on demonstrating the successfulness of SIM's protection against primitive attacks. For this reason, we built following test cases:

- **strata_tracing**
  - Purpose: test if SIM can correctly protect Strata's global (static) data.
  - Attack: the test program will try to overwrite Strata's global variable strata_tracing to 1, thus making Strata output tracing information.
  - Expected behavior: with SIM_PROTECTION=0, the test program will output tracing information; but with SIM_PROTECTION=1, the attack will be prevented and test program will do a controlled exit
- **strata_pc_confine**
  - Purpose: test if SIM can correctly protect Strata's dynamic data.
  - Attack: the test program is an extended version of the dumbledore grading program, but before calling the vulnerable function, it will try to modify Strata's PC confinement metadata to mark the shellcode region as legitimate.
  - Expected result: with SIM_PROTECTION=0, the test program will give Jack an A even with STRATA_PC_CONFINE=1 (but without STRATA_PC_CONFINE_XOR); but with SIM_PROTECTION=1, the attack will be prevented and test program will do a controlled exit.
- **strata_build_main**
  - Purpose: test if SIM can correctly protect Strata's code.
  - Attack: the test program will try to overwrite strata_build_main function to print evil hello message and then exit
  - Expected behavior: with SIM_PROTECTION=0 the test program will print an evil hello message and exit; but with SIM_PROTECTION=1, the attack will be prevented and test program will do a controlled exit.
- **strata_code_cache**
  - Purpose: test if SIM can correctly protect Strata's code cache.
  - Attack: the test program will try to overwrite its translated code in Strata's code fragment and jump back to execute the un-translated code (original code)

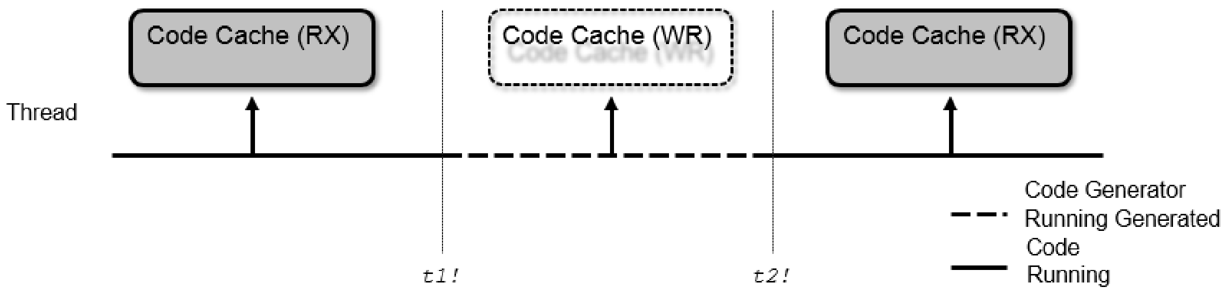


Figure 40. A permission switching based W⊕X enforcement. The code cache is kept as read-only when the generated code is executing. When the code generator is invoked (t1), the permission is changed to writable; and when the generator finishes its task (t2), the permission is changed back to read-only.

  - Expected behavior: with SIM_PROTECTION=0 the test program will tell it "jailbreaks" Strata's containment; but with SIM_PROTECTION=1, the attack will be prevented and test program will do a controlled exit.

Current Test Result. By the date of this report, the test results of the above tests are:

- For kernel-mode implementation
  - All correctness tests are passed;
  - All functional tests are passed.
- For user-mode implementation
  - All suitable correctness tests (except tests statically linked to Strata and tests for

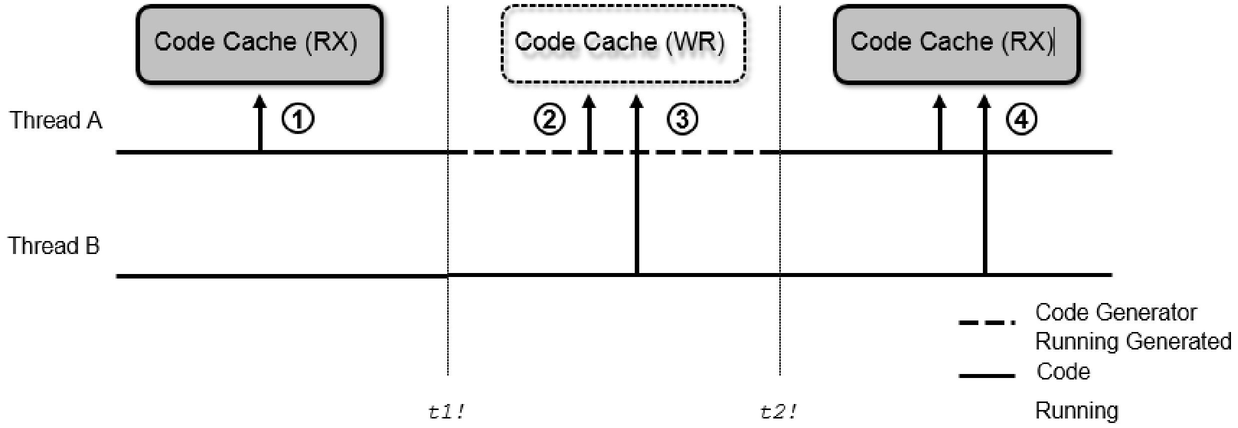
Figure 39. Race-condition-based attack using two threads. With switching based W⊕X enforcement, a single thread (A) can no longer attack the code cache (access 1). But the code cache can still be attacked using multiple threads. As when the code generator is serving one thread (access 2), the code cache will also become writable for other thread (access 3). The attack window refers to *t*2 - *t*1, as once the code generator finishes its task, the code cache becomes read-only again (access 4).

SIM kernel-mode implementation) are passed;
  - All functional tests are passed.

3.10.3 Secure Dynamic Code Generation (SDCG): Phase 3 Protection of PEASOUP

The material in this section was also published in NDSS 2015 [198]. We address the more general problem of protecting a software dynamic translator, such as Strata, which is at the heart of PEASOUP.

Exploits against software vulnerabilities remain one of the most severe threats to cyber security. To mitigate this threat, many techniques have been proposed, including data execution prevention (DEP) [27] and address space layout randomization (ASLR) [159], which have been widely deployed and very successful. DEP is a subset of the more general security policy W⊕X, which enforces that memory should either be writable but not executable (e.g., data segments), or be executable but read-only (e.g., code segments). This enforcement can completely mitigate traditional exploits that inject malicious shellcode into data segments. Consequently, attackers have to leverage more complicated exploit techniques, such as return-to-libc [190] and return-oriented-programming (ROP) [110]. Moreover, W⊕X memory has become the foundation of many other protection techniques, such as control flow integrity (CFI) [24], [226], [225].



However, the effectiveness of W⊕X can be undermined by another important compilation technique – dynamic code generation (DCG). With the ability to generate and execute native machine code at runtime, DCG is widely used in JIT compilers [34] and dynamic binary translators (DBT) [181], [164], [135] to improve performance, portability, and security. For example, JIT compilers for dynamic languages (e.g., JavaScript and ActionScript) can leverage platform information and runtime execution profile information to generate faster native code. And DBTs can leverage DCG to provide dynamic analysis capability [135], cross-platform or cross-architecture portability [182], [41], bug diagnostic [145], [165], and better security [39], [148], [53], [111], [107].

A fundamental challenge posed by DCG is that, the code cache, in which the dynamically generated code is stored, needs to be both writable (for code emitting, code patching and garbage collection) and executable. This violates the W⊕X policy and enables a new attack vector. We have observed a real world exploit that delivers shellcode into the writable code cache and successfully compromises the Chrome web browser [161].

Solving this problem seems trivial. A straightforward idea, which has been adopted in browsers like mobile Safari, is demonstrated in Figure 40. This technique keeps the code cache as read-only and executable (RX) when the generated code is executing; switches to writable but not executable (WR) when it needs to be modified (t1); and switches back to RX when the write operation finishes ($t2$). As a result, the code cache will remain read-only when the generated code is executing; and the attack demonstrated in [161] can thus be mitigated.

Unfortunately, in addition to performance overhead, this simple mechanism does not work well with multi-threaded programs. First, if the code generator uses a shared code cache for all threads (e.g., PIN [135]), then obviously the code cache cannot be switched to WR, because other concurrently running threads require the executable permission. Second, even if the code generator uses dedicated code cache for each thread (e.g., JS engines), the protection is still flawed and is subject to *race condition* attacks [146], as shown in Figure 39. More specifically, memory access permissions are applied to the whole process and are shared among all threads. When one thread, say thread A, turns on the writable permission for its code cache (e.g., for code emitting), the code cache also becomes writable to all other threads. Once the protection is gone, another concurrently running thread, thread B, can (maliciously) overwrite thread A’s code cache to launch attacks. This is similar to the classic time-to-check-time-to-use (TOCTOU) attacks [207], where the resource to be accessed is modified between the check and the use, by exploiting race conditions.

In this section, we demonstrate the feasibility of such racecondition-based code cache injection attacks, through a proofof-concept exploit against modern browsers that support the *Web Worker* [11] specification. Rather than relying on a permanently writable code cache [161], our attack leverages the race condition and can bypass the permission switching based W⨁X enforcement (Figure 40). In this attack, the malicious JS code utilizes web workers to create a multi-threaded environment. After forcing a worker thread into the compilation state, the main JS thread can exploit vulnerabilities of the browser to inject shellcode into the worker thread’s code cache.

To fundamentally prevent such attacks, we propose secure dynamic code generation (SDCG), a new architecture that 1) enables dynamic code generation to comply with the W⨁X policy; 2) eliminates the race condition; 3) can be easily adopted; and 4) introduces more acceptable performance overhead compared with alternative solutions. SDCG achieves these goals through a multi-process-based architecture. Specifically, instead of generating and modifying the code in the same process as the generated code, SDCG relocates the DCG functionality to another trusted process. The code cache is built upon memory shared between the original process and the trusted process. In the original process, the code cache is mapped as RX; and in the trusted process, the same memory is mapped as WR. By doing so, the code cache remains read-only under all circumstances in the untrusted process, eliminating the race condition that allows the code cache to be writable to untrusted thread(s). At the same time, the code generator in the trusted process can freely perform code generation, patching and garbage collection as usual. To enable transparent interaction between the code generator and the generated code, we only need to add a few wrappers that make the code generator invocable through remote procedure calls (RPC). Since only functions that modify code cache need to be handled, the effort for adding wrappers is small.

We have implemented SDCG for two types of popular code generators: JS engine and DBT. For JS engine, our implementation is based on V8 [5]. For DBT, our implementation is based on Strata [181]. Our implementation experience showed that porting code generators to SDCG only requires a small modification: besides the shareable part, which is about 500 lines of C code (LoC), we only added about 2,500 LoC for V8 and about 1,000 LoC for Strata. We evaluated the security of SDCG and the performance overhead of our two prototype implementations. The results showed that SDCG are secure under our threat model and the performance overhead introduced by our prototype implementations is small: around 6.90% (32-bit) and 5.65% (64-bit) for V8 benchmark suite; and around 1.64% for SPEC CINT 2006 (additional to Strata's own overhead).

In summary, we made the following contributions:

- Beyond the known exploit technique against permanently writable code cache [161], we demonstrated the feasibility of exploiting race conditions to maliciously modify the code cache that is protected by permission switching based W⊕X enforcement; and discussed the severity of such attacks.
- We proposed secure dynamic code generation (SDCG), a multi-process-based architecture that provides better security (mandatory, non-bypassible W⊕X enforcement), low performance overhead, and easy adoption.
- We implemented two prototypes of SDCG, one for V8 JS engine and one for Strata dynamic binary translator.
- We evaluated the performance overhead of our two prototype implementations.

3.10.3.1 Related Work

In this section, we discuss the techniques that could be used to protect the code cache from being maliciously modified and explain their disadvantages. We also discuss other forms of attacks against the JIT engines and their countermeasures.

Software-based Fault Isolation

Software-based fault isolation (SFI) [211] can be used to confine a program's ability to access memory resources. On 32-bit x86 platforms, SFI implementations usually leverage segment registers [87], [222] to confine memory accesses, for the benefit of low runtime overhead. On other platforms without segment support (e.g., x86-64, ARM), SFI implementations use either address masking [183] or access control list (ACL) [51], introducing higher runtime overhead.

Once memory accesses — especially write accesses — are confined SFI can prevent untrusted code from overwriting security sensitive data, such as the code cache. Our SDCG solution differs from SFI in several respects. First, SFI's overhead comes from the execution of the extra inline checks; but SDCG's overhead comes from the remote procedure call and cache synchronization on multi-core systems. Therefore, if the execution mostly stays within the code cache, SDCG will introduce less overhead than SFI. On the other hand, if the execution needs to be frequently switched between the code generator and the generated code, then SFI could be faster. Since most modern code generators try to make the execution stay as long as possible in the code cache, our approach is more suitable in this scenario.

Second, to reduce the overhead of address masking, many SFI solutions [183] use ILP32 (32-bit integer, long, pointer) primitive data types, limiting data access to 4GB space, even on a 64-bit platform. On the other hand, SDCG does not have this limitation.

It is worth noting that, some efforts have been made to apply SFI to JIT engines [29], [153]. Despite relatively higher overhead, the threat model of these approaches usually did not consider scenarios where the JIT compiler is only a component of a larger software, such as a web browser. Since most vulnerabilities of web browsers are found outside the JIT engines [15], to apply such techniques, one would have to apply SFI to other browser components as well. This could result in even higher performance overhead. From this perspective, we argue that our solution is more realistic in practice.

Memory Safety

The attacks on the code caches (at randomized locations) rely on the ability to write to memory area specified by attacker. Therefore, such attacks could be defeated by memory safety enforcement, which prevents all unexpected memory read and write. However, many programs are written in lowlevel languages like C/C++ that are prone to memory corruption bugs, leading to a majority of security vulnerabilities. Unfortunately, existing memory safety solutions [189], [80], [33], [158], [220], [142], [143] for C/C++ programs tend to have much higher performance overhead than SFI or other solutions, prohibiting their adoptions. For example, the combination of Softbound [143] and CETS [142] provides a strong spatial and temporal memory safety guarantee, but they were reported to have 116% average overhead on SPEC CPU 2000 benchmark. Compared with this direction of research, even though SDCG provides less security guarantees, it is still valuable because it fully blocks a powerful attack vector with minimal runtime overhead.

Control Flow Integrity

Control flow hijacking is a key step in many real world attacks. As DEP becomes ubiquitous, more and more attacks rely on return-to-libc [190] or ROP [110] to hijack control flow. Many solutions [24], [226], [225] are thus proposed to enforce control flow integrity (CFI) policy. With the CFI policy, the program's control flow cannot be hijacked to unexpected locations. It can protect the code cache in some way, e.g., attackers cannot overwrite the code cache by jumping to arbitrary address of the code generator.

However, attackers can still utilize arbitrary memory write vulnerabilities to overwrite the code cache without breaking CFI. Once code cache is overwritten, the injected code could be invoked through normal function invocations, without breaking the static CFI policy.

Moreover, when extending CFI to dynamically generated code, without proper write protection, the embedded enforcement checks can also be removed once attackers can overwrite the code. From this perspective, SDCG is complementary to CFI because it guarantees one basic assumption of CFI: code integrity protection.

Process Sandbox

Delegation-based sandbox architecture, a.k.a. the broker model [91], has been widely adopted by the industry and used in Google Chrome [20], Windows 8 [14], Adobe Reader [8], and etc. In this architecture, the sandboxed process drops most of its privileges and delegates all security sensitive operations to the broker process. The broker process in turn, checks whether the request complies with the security policy. SDCG is also based on the same architecture. Using this architecture, we 1) delegate all the operations that will modify the code cache (e.g., code installation, patching and deletion) to the translator process; and 2) make sure the W$\oplus$X policy is mandatory.

Attacks on JIT engines

Attacker have targeted the code cache for its writable and executable property. Currently, the most popular exploit technique is JIT spray [196], an extension to classic heap spray attacks [73]. Heap spray is used to bypass ASLR without guessing the address of injected shellcode. This technique became unreliable after DEP is deployed because the heap is no longer executable. To bypass this, attackers turned to JIT engines, The JIT spray attack abuses the JIT engine to emit chunks of predictable code, and then hijacks the control to the entry or middle of one of these code chunks. DEP or W⊕X is thus bypassed because these code chunks reside in the executable code cache. Most JIT engines have since deployed different mitigation techniques to make the layout of the code cache unpredictable, e.g., random NOP insertion, constant splitting, and etc. And researchers have also proposed more robust technique [215], [29] to prevent such attacks.

Rather than abusing the JIT engines to create expected code, attackers can also abuse the writable property of the code cache and directly overwrite the generated code [161]. In this paper, we first extend the attack [161] to show that, even with a permission switching based W⊕X enforcement, attackers can still leverage race conditions to bypass such enforcement. Then we propose a solution can fundamentally defeats all codecache injection based attacks.

3.10.3.2 Attacking the Code Cache

In this section, we describe in detail the code cache injection threat we are addressing in this paper. We begin this section with our assumptions and threat model. Next, we show how code cache can be attacked to bypass the state-of-the-art exploit mitigation techniques. Finally, we demonstrate how a

naive W⊕X enforcement can be bypassed by exploiting race conditions.

Assumptions and Threat Model

SDCG focuses on preventing remote attackers from leveraging the code cache as an attack vector to trigger arbitrary code execution. We focus on two classic attack scenarios discussed as follows. In both scenarios, we assume the code generator itself is trusted and does not have security vulnerabilities.

- *Foreign Attacks.* In this scenario, the code generator is a component of a program (e.g., a web browser). The program is benign, but components other than the code generator are assumed to be vulnerable when handling input or contents provided by attacker (e.g., a malicious web page). Attackers can then exploit the vulnerable components to attack the code cache.

- *Jail-break Attacks.* In this scenario, the code generator is used to sandbox or monitor an untrusted program, and attacks are launched within the code cache. This could happen under two circumstances. First, the sandboxed program itself is malicious. Second, the program is benign, but the dynamically generated code has vulnerabilities that can be exploited by attackers to jailbreak.

Without loss of generality, we assume that the following mitigation mechanisms for both general and JIT-based exploits have been deployed on the target system.

- *Address Space Layout Randomization.* We assume that the target system has deployed at least the base address randomization, and all the predictable memory mappings have been eliminated.

- *JIT Spray Mitigation.* For JIT engines, we assume that they have deployed a full-suite of JIT spray mitigation mechanisms, including but not limited to random NOP insertion, constant splitting and those proposed in [215], [29].

- *Guard Pages.* We assume the target system creates guard pages (i.e., pages without access permission) to wrap each pool of the code cache, like the Google V8 JS engine does. These guard pages can prevent buffer overflows, both overflows out of the code cache, and overflows into the code cache.

- *Page Permissions.* We assume that the underlying hardware has the support for mapping memory as nonexecutable (NX). And writable data memory like stack and normal heap are set to be non-executable. Furthermore, we assume that all the statically generated code has been set to non-writable to prevent overwriting. However, almost all JIT compilers map the code cache as both writable and executable.

The target system can further deploy the following advanced mitigation mechanisms for the purpose of sandboxing and monitoring:

- *Fine-grained Randomization.* The target system can enforce fine-grained randomization by permuting the order of functions [121] or basic blocks [213], randomizing location of each instruction [156], or even randomizing the instruction set [156], [84].

- *Control Flow Hijacking Mitigation.* The target system can deploy different kinds of control flow hijacking mitigation mechanisms, including (but not limited to): control flow integrity enforcement, either coarse-grained [225], [226] or fine-grained [24], [154]; return-oriented programming detection [75], [54]; and dynamic taint analysis based hijacking detection [148].

To allow overwriting of the code cache, we assume there is at least one vulnerability that allows attackers to write to attacker-specified address with attacker-provided contents. We believe this is a realistic assumption, because many types of vulnerabilities can be exploited to achieve this goal, such as format string [147], heap overflow [65], use-afterfree [16], integer overflow [17], and etc. For example, the attack described in [161] obtained this capability by exploiting an integer overflow vulnerability (CVE-2013-6632); and in [52], the author described how 5 use-after-free vulnerabilities (CVE-2013-0640, CVE-2013-0634, CVE-2013-3163, CVE-2013-1690, CVE-2013-1493) can be exploited to perform arbitrary memory write. It is worth noting that, in many attack scenarios, the ability to do arbitrary memory write can easily lead to arbitrary memory read and information disclosure abilities.

Overwriting the Code Cache

1) *Software Dynamic Translator:* For the ease of discussion, we use the term software dynamic translator (SDT) to represent software that leverages dynamic code generation to translate code in one format into another format. Before describing the attacks, we first give a brief introduction on SDT. A core task of all SDTs is to maintain a mapping between untranslated code and translated code. Whenever a SDT encounters a new execution unit (depending on the SDT, the execution unit could be a basic block, a function or a larger chunk of

code), it first checks whether the execution unit has already been translated. If so, it switches to execute the already translated code residing in the code cache; otherwise, it translates this new execution unit and installs the translated code into the code cache.

2) *Exploit Primitives:* In this section, we describe how code cache with full WRX permission can be overwritten. This is done in two steps. First, we need to bypass ASLR and find out where the code cache locates. Second, we need to be able to write to the identified location.

a) *Bypassing ASLR:* The effectiveness of ASLR or any randomization based mitigation mechanism relies on two assumptions: i) the entropy is large enough to stop brute-force attacks; and ii) the adversary cannot learn the random value (e.g., module base, instruction set).

Unfortunately, these two assumptions rarely hold in practice. First, on 32-bit platforms, user space programs only have 8 bits of entropy for heap memory, which is subject to brute-force guessing [190] and spray attacks [73]. Second, with widely available information disclosure vulnerabilities, attackers can easily find out the random value [187], [172]. In fact, researchers have demonstrated that even with a single restricted information disclosure vulnerability, it is possible to traverse a large portion of the memory content [197].

In the scenario of attacking code cache, this status quo implies that we can either launch JIT spray attack to prepare a large number of WRX pages on platforms with low entropy; or leverage an information disclosure vulnerability to pinpoint exactly where the code cache is. Note that, as one only needs to know the location of the code cache, most fine-grained randomizations that try to further randomize the content of the memory are ineffective for this attack. Since the content of code cache will be overwritten in the next step (described below), none of the JIT spray mitigation mechanisms can provide effective protection against this attack.

b) *Writing to Code Cache:* Once obtaining the location of the code cache, the next step is to inject shellcode to the code cache. In most cases, the code cache will not be adjacent to other writable heap memory (due to ASLR), and may also be surrounded by guard pages. For these reasons, we cannot directly exploit a buffer overflow vulnerability to overwrite the code cache. However, as our assumption section suggests, besides logic errors that directly allow one to write to anywhere in the memory, several kinds of memory corruption vulnerabilities can also provide the arbitrary memory write ability. In the following example, an integer overflow vulnerability is exploited to acquire this capability.

3) *An In-the-Wild Attack:* We have observed one disclosed attack [161] that leveraged the code cache to achieve reliable arbitrary code execution. This attack targeted the mobile Chrome browser. By exploiting an integer overflow vulnerability, the attack first gained reliable arbitrary memory read and write capabilities. Using these two capabilities, the attack subsequently bypassed ASLR and located the permanently writable and executable code cache. Finally, it injected shellcode into the code cache and turned control flow to the shellcode.

4) *Security Implication:* In practice, we have only observed this single attack that injects code into the code cache. We believe this is mainly due to the convenience of a popular ROP attack pattern, which works as: i) preparing a traditional shellcode in memory; ii) exploiting vulnerabilities to launch ROP attack; iii) using the ROP gadgets to turn on the execution permission of the memory where the traditional shellcode resides; and iv) jumping to the traditional shellcode to finish the intended malicious tasks. However, once advanced control flow hijacking prevention mechanisms such as fine-grained CFI are deployed, this attack pattern will be much more difficult to launch.

On the contrary, the code cache injection attack can easily bypass most of the existing exploit mitigation mechanisms. First, all control flow hijacking detection/prevention mechanisms such as CFI and ROP detection rely on the assumption that the code cannot be modified. When this assumption is broken, these mitigation mechanisms are no longer effective. Second, any inline reference monitor based security solution is not effective because the injected code is not monitored.

Exploiting A Race Condition

A *naive* defense against the code cache injection attack is to enforce W⊕X by manipulating page permissions (Figure 40). More specifically, when the code cache is about to be modified (e.g., for new code generation or runtime garbage collection), it turns on the write permission but turning off the execution permission ($t1$). And when the code cache is about to be executed, it turns off write permission but turn on execution permission ($t2$).

This solution prohibits the code cache to be both writable and executable at the same time. If the target program is single-threaded, this approach can prevent code cache injection attacks. Since the code cache is only writable when the SDT is executing and we assume that the SDT itself is trusted and not vulnerable, attackers cannot hijack or interrupt the SDT to overwrite the code cache. However, as illustrated in Figure 39, in more general multi-threaded programming environment, even if the SDT is trusted, the code cache can still be overwritten by other insecure threads (Thread B) when the the code cache is set to be writable for one thread (Thread A).

In this section, we use a concrete attack to demonstrate the feasibility of such attack, i.e., with the naive W⊕X enforcement, it is still possible to overwrite the code cache with the same exploit primitives described above.

1) Secure Page Permissions: Since the V8 JS engine does not have the expected page permission protection, i.e., the naive W⊕X enforcement, we implemented one in V8 for the demonstration of our attack.

Specifically, by default, when a memory region is allocated from the OS (e.g., via `mmap`) for the code cache, it is allocated as executable but not writable. We will turn on the write permission and turn off the execution permission of the code cache for:

- *New Code Installation.* Usually, the JavaScript program (e.g., a function) is first compiled into native code, and then copied into the code cache. To allow the copy operation, we need to turn on the write permission of the code cache.

- *Code Patching.* Existing code in the code cache is patched under certain circumstances. For instance, after a new code is copied into the code cache, its absolution address is thus determined; then instructions that require absolution address operands from this new code fragment is resolved and patched.

- *Runtime Inline Caching.* Inline caching is a special patching mechanism introduced to provide better performance for the JIT compiled programs written in dynamic typed languages. With runtime execution profile information, the JIT compiler caches/patches the result (e.g., the result of object property resolving) into the instructions in the code cache at runtime.

- *Runtime Garbage Collection.* The JavaScript engine needs to manage the target JavaScript program's memory, especially the garbage collection. The code cache needs to be modified for two main reasons. First, when unused code fragment needs to be removed from the code cache. Second, when a data object is moved to new addresses by the garbage collector, instructions referencing it have to be updated.

When these operations finish, or any code in the code cache needs to be invoked, we turn off the write permission of the code cache and turn on the execution permission.

To further reduce the attack surface, all the above policies are enforced in a fine-grained granularity. That is, 1) each permission change only covers memory pages that are accessed by the write or execution operations; and 2) the write permission is turned on only when an write operation is performed, and is turned off immediately after the write operation finishes. This fine-grained implementation provides a maximum protection for code caches.

2) Multi-threaded Programming in SDT: To launch the race-condition-based attack, we need two more programming primitives. First we need the ability to write multi-threaded programs. Note, some SDTs such as Adobe Flash Player also allows "multi-threaded" programming, but each "thread" is implemented as a standalone OS process. For these SDTs, since the code cache is only writable to the corresponding thread, our proposed exploit technique would not work. Second, since the attack window is generally small, we need the ability to coordinate threads before launching the attack.

- *Thread Primitives.* A majority of SDTs have multithreaded programming support. JavaScript (JS) used to be single-threaded and event-driven. With the new HTML5 specification, JS also supports multithreaded programming, through the specification of WebWorker [11]. There are two types of WebWorker: *dedicated* worker and *shared* worker. In V8, the dedicated worker is implemented as a thread within the same process; and shared worker is implemented as a thread in a separated process. Since we want to attack one JS thread's code cache with another JS thread, we leverage the dedicated worker. Note that although each worker thread has its own code cache, it is still possible to launch the attack, because memory access permissions are shared by all threads in the same process.

- *Synchronization Primitives.* To exploit the race condition, two attacker-controlled threads need to synchronize their operations so that the overwritten can happen within the exact time window when the code cache is writable. Since synchronization is an essential part of multithreaded programming, almost all SDTs support thread synchronization. In JS, thread synchronization uses the postMessage function.

3) A Proof-of-Concept Attack: Based on the vulnerability disclosed in the previous real-world exploit, we built a proofof-concept race-condition-based attack on the Chrome browser. Since the disclosed attack [161] already demonstrated how ASLR can be bypassed and how arbitrary memory write capability can be acquired, our attack focuses on how race conditions can be exploited to bypass naive W⊕X enforcement. The high level workflow of our attack is as follows:

i) *Creating a Worker.* The main JS thread creates a web worker, and thus a worker thread is created.

ii) *Initialize the Worker.* The worker thread initializes its environment, making sure the code cache is created. And then, it sends a message to the main thread through postMessage to inform that it is ready.

iii) *Locating the Worker's Code Cache.* Upon receiving the worker's message, the main JS thread locates the worker thread's code cache, e.g., by exploiting an information disclosure vulnerability. In Chrome V8 engine, attackers can locate the code cache like the previous disclosed exploit. But instead of following the pointers for the current thread, attackers should go through the thread list the JS engine maintains and follow pointers for the worker thread. Then, it informs the worker that it is ready too.

iv) *Making Code Cache Writable.* Upon receiving the main thread's message, the worker thread begins to execute another piece of code, forcing the SDT to update its code cache. In V8, the worker can execute a function that is big enough, forcing the SDT to create a new MemoryChunk as the code fragment and set it to be writable (for a short time).

v) *Monitor and Overwrite the Code Cache.* At the same time, the main thread keeps monitoring the status of the code cache, and tries to overwrite the code cache once its status is updated. In V8, the main thread can keep polling the head of MemoryChunk linked list to identify the creation of a new code fragment. Once a new code fragment is created, the main thread can then monitor its content. Once the first few bytes (e.g., the function prolog) are updated, the main thread can try to overwrite the code cache to inject shellcode. After overwriting, the main thread informs the worker it has finished.

vi) *Executing the Shellcode.* Upon receiving the main thread's new message, the worker calls the function whose content has already been overwritten. In this way, the injected shellcode gets executed.

It is worth noting that, the roles of the main thread and the worker thread cannot be swapped in practice. The reason is, worker threads do not have access to the document object model (DOM). Since lots of vulnerabilities are within the rendering engine instead of the JS engine, this means only the main thread who has the access to the DOM can exploit those vulnerabilities.

4) Reliability of Race Condition: One important question for any race-condition-based attack is its reliability. The first factor that can affect the reliability of our attack is synchronization, i.e., the synchronization primitive should be fast enough so that the two threads can carry out the attack within the relatively small attack window. To measure the speed of the synchronization between the worker and the main thread, we ran another simple experiment:

i) The main thread creates a worker thread;
ii) The worker thread gets a timestamp and sends it to the main thread;
iii) Upon receiving the message, the main thread sends an echo to the worker;
iv) Upon receiving the message, the worker thread sends back an echo;
v) The main thread and the worker repeatedly send echoes to each other in this way for 1,000 times.
vi) The main thread gets another timestamp and computes the time difference.

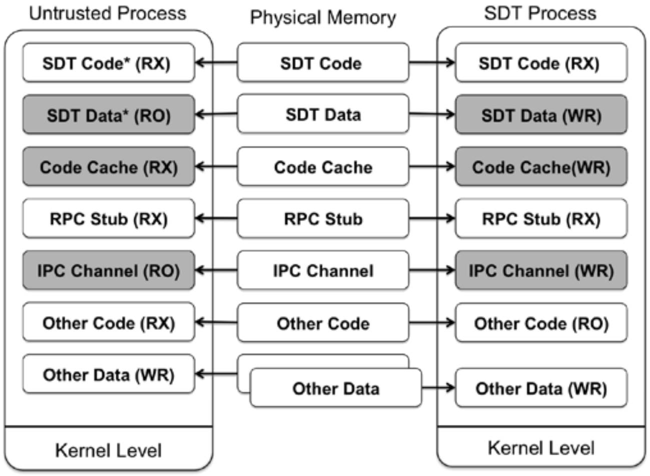


Figure 41. Overview of SDCG' multi-process-based architecture. The gray memory areas are shared memory, other are mapped as private (copy-on-write). Depending on the requirement, the SDT's code and data can be mapped differently.

The result shows that the average synchronization delay is around 23 μs. Comparing with this, the average attack window ($t2-t1$ in Figure 39) of our fine-grained naive W⊕X protection is about 43 μs. Thus in theory, the postMessage method is sufficiently fast to launch a race condition attack.

The second and more important factor that can affect the reliability of our attack is task scheduling. Specifically, if the thread under the SDT context (e.g., the worker thread) is descheduled by the OS while the attacking thread (e.g., the main thread) keeps executing, then the attacking window will be enlarged. The bad news (for defenders) is that, the only way to change the code cache's memory permission is through a system call, and a context switch is likely to happen during the system call. For example, the system call for changing memory access permission on Linux is mprotect. During the invocation of mprotect, since we are using fine-grained protection, the virtual memory area needs to be split or merged, it will trigger the thread to be de-scheduled. As a result, the main thread (with higher priority than the worker) can gain control to launch attacks.

Combining these two factors, our demo's result shows that the race-condition-based attack can succeed with a very high probability. We have tested the browser for 100 times, and the attack succeeded for 91 times.

3.10.3.3 System Design

In this section, we present the design of SDCG. We have two design goals: 1) SDCG should prevent all possible code injection attacks against the code cache under our adversary model; and 2) SDCG should only introduce trivial performance overhead. In addition, SDCG is designed to be integrated with the targeted SDT, and we assume that the source code of the SDT is available.

Overview and Challenges

Since the root cause of the attack is writable code cache (either permanently or temporarily), we can be prevent such attacks by two design choices: 1) ensuring that no one but the SDT can write to the code cache, e.g., through SFI or memory safety; and 2) ensuring that the memory occupied by the code cache is always mapped as RX. We chose the second choice for two major reasons. First, we expect that the performance overhead of applying SFI or memory safety to an entire, large, complex program (e.g., the web browser) would be very high. Second, implementing the first choice requires significant engineering effort.

Figure 41 shows the high level design of SDCG. The key idea is that, through shared memory, the same memory content will be mapped into two (or more) different processes, with different access permissions. In the untrusted process(es), the code cache will be mapped as RX; but in the SDT process, it will be mapped as WR. By doing so, SDCG prevents any untrusted code from modifying the code cache. At the same time, it allows the SDT to modify the code cache as usual. The only difference is, whenever the SDT needs to be invoked, e.g., to install a new code fragment, the request will be served through remote procedure call (RPC) instead of a normal function call.

To build and maintain this memory model, we need to solve following technical and engineering challenges.

i) *Memory Map Synchronization*. Since the memory regions occupied by the code cache are dynamically allocated and can grow and shrink freely, we need an effective way to dynamically synchronize memory mapping between the untrusted process(es) and the SDT process. More importantly, to make SDCG's protection mechanism work transparently, we have to make sure that the memory is mapped at exactly the same virtual address in all processes.

ii) *Remote Procedure Call*. After relocating the SDT to another process, we need to make it remotely invocable, by wrapping former local invocations with RPC stubs. Since RPC is expensive, we need to reduce the frequency of invocations, which also reduces the attack surface.

iii) *Permission Enforcement*. Since SDCG's protection is based on memory access permissions, we must make sure that untrusted code cannot tamper with our permission scheme. Specifically, no memory content can be mapped as both writable and executable, neither at the same time nor alternately.

Memory Map Synchronization

Synchronizing memory mapping between the untrusted process(es) and the SDT process is a bi-directional issue. On the one hand, when the SDT allocates a new code fragment in the SDT process, we should map the same memory region in the untrusted process(es) at exactly the same address; otherwise the translated code will not work correctly (e.g., creating incorrect branching target). On the other hand, the untrusted process may also allocate some resources that are critical to the SDT. For example, in the scenario of binary translation, when the untrusted process loads a dynamically linked module, we should also load the same module at the same address in the SDT process; otherwise the SDT will not be able to locate the correct code to be translated. Moreover, we want this synchronization to be as transparent to the SDT as possible, so we can keep the changes minimal.

When creating the shared memory, there are two possible strategies: on-demand and reservation-based. On-demand mapping creates the shared memory at the very moment a new memory region is required, e.g., when the SDT wants to add a new memory region to the code cache. However, as the process address space is shared by all modules of a

program, the expected address may not always be available in both the untrusted process and the SDT process. For this reason, we choose the reservation-based strategy. That is, when the process is initialized, we reserve (map) a large chunk of shared memory in both the untrusted process(es) and the SDT process. Later, any request for shared memory will be allocated from this shared memory pool. Note that, in modern operation systems, physical memory resources are not mapped until the reserved memory is really accessed, so our reservation-based strategy does not impose significant memory overhead.

Once the shared memory pool is created, synchronization can be done via inter-process communication (IPC). Specifically, when the SDT allocates a new memory region for the code cache, it informs the untrusted process(es) about the base address and the size of this new memory region. Having received this event, the untrusted process(es) maps a memory region with the same size at the same base address with the expected permission (RX). Similarly, whenever the untrusted process allocates memory that needs to be shared, a synchronization event is sent to the SDT process.

Remote Procedure Call

Writing RPC stubs for the SDT faces two problems: argument passing and performance. Argument passing can be problematic because of pointers. If a pointer points to a memory that is different between the untrusted process and the SDT process, then the SDT ends up using incorrect data and causes run-time errors. Vise versa, if the returned value from the SDT process contains pointers that point to data not copied back, the untrusted code ends up running incorrectly. One possible solution is that, instead of passing the pointer to the remote process, the stub serializes the object before passing it to the remote process. Unfortunately, not all arguments have builtin serialization functionality. In addition, when an argument is a large object, performing serialization and copy for every RPC invocation introduces high performance overhead. Thus, in general, stub generation is not easy without the support from the SDT or support from program analysis.

To avoid this problem, SDCG takes a more systematic approach. Specifically, based on the observation that a majority of data that the SDT depends on is either read-only, or resides in dynamically mapped memory, we extend the shared memory to also include the dynamic data the SDT depends on. According to the required security guarantee, the data should be mapped with different permissions. By default, SDCG maps the SDT's dynamic data as read-only in the untrusted process, to prevent tamper from the untrusted code. However, if noncontrol data attacks are not considered, the SDT's dynamic data can be mapped as WR in the untrusted process. After sharing the data, we only need to handle a few cases where writable data (e.g., pointers within global variables) is not shared/synchronized.

Since RPC invocations are much more expensive than normal function calls, we want to minimize the frequency of RPC invocation. To do so, we take a passive approach. That is, we do not convert an entry to the SDT to RPC unless it modifies the code cache. Again, we try to achieve this goal without involving heavy program analysis. Instead, we leverage the regression tests that are usually distributed along with the source code. More specifically, we begin with no entries being converted to RPC and gradually convert them until all the regression tests can pass.

While our approach can be improved with more automation and program analysis, we leave these as future work because our main goal here is to design and validate that our solution is effective against the new cod cache injection attacks.

Permission Enforcement

To enforce mandatory W⊕X, we leverage the delegationbased sandbox architecture [91]. Specifically, we intercept all system calls related to virtual memory management, and enforce the following policies in the SDT process:

(I) No memory can be mapped as both writable and executable.
(II) When mapping a memory region as executable, the base address and the size must come from the SDT process, and the memory is always mapped as RX.
(III) The permission of non-writable memory cannot be changed.

3.10.3.4 Implementation

We implemented two prototypes of SDCG, one for Google V8 JS engine [5], and the other for Strata DBT [181]. Both prototypes were implemented on Linux. We chose these two SDTs for the following reasons. First, JS engine is one of the most widely deployed SDT. At the same time, it is also one of the most popular stepping stone for launching attacks. Among all JS engines, we chose V8 because it is open sourced, highly ranked, and there is disclosed exploit [161]. Second, DBT have been widely used by security researchers to build various security solutions [39], [148], [53], [111], [107]. Among all the DBTs, we chose Strata because 1) it has been used to implement many promising security mechanisms, such as instruction set randomization [111], instruction layout randomization [107], etc.; and 2) its academic background allowed us to have access to its source code, which is required for implemented SDCG.

Shared Infrastructure

The memory synchronization mechanism and the system call filtering mechanism are specific to the target platform; but they can be shared among all SDTs.

1) Seccomp-Sandbox: Our delegation-based sandbox is built upon the seccomp-sandbox [19] from Google Chrome. Although Google Chrome has switched to a less complicated process sandbox based on seccomp-bpf [68], we found that the architecture of seccomp-sandbox serves our goal better. Specifically, since seccomp [67] only allows four system calls once enabled, and not all system calls can be fulfilled by the broker (e.g., mmap), the seccomp-sandbox introduced a trusted thread to perform system calls that cannot be delegated to the broker. To prevent attacks on the trusted thread, the trusted thread operates entirely on CPU registers and does not trust any memory that is writable to the untrusted code. When the trusted thread makes a system call, the system call parameters are first verified by the broker, and then passed through a shared memory that is mapped as read-only in the untrusted process. As a result, even if the other threads in the same process are compromised, they cannot affect the execution of the trusted thread. This provides us a perfect foundation to securely build our memory synchronization mechanism and system call filtering mechanism.

To enforce the mandatory W⊕X policy, we modified the sandbox so that, before entering the sandbox mode, SDCG enumerates all memory regions and converts any WRX region to RX.

For RPC invocation, we also reused seccomp-sandbox's domain socket based communication channel. However, we did not leverage the seccomp mode in our current implementation. There are several reasons. First, it is not compatible with the new seccomp-bpf-based sandbox used in Google Chrome. Second, it intercepts too many system calls that are not required by SDCG. More importantly, both Strata and seccompbpf provide enough capability for system call filtering.

2) Shared Memory Pool: During initialization, SDCG reserves a large amount of consecutive memory as a pool. This pool is mapped as shared (MAP_SHARED), not file backed (MAP_ANONYMOUS) and with no permission (PROT_NONE). After this, any mmap request from the SDT allocates memory from this pool (by changing the access permission), instead of going to the mmap system call. This guarantees any SDT allocated region can be mapped at exactly the same address in both the SDT process and the untrusted process(es).

After the sandbox is enabled, whenever the SDT calls mmap, SDCG generates a synchronized request to the untrusted process(es), and wait until the synchronization is done before returning to the SDT. In the untrusted process, the synchronization event is handled by the trusted thread. It reads synchronization request from the IPC channel and then changes the access permission of the given region to the given value. Since the parameters (base address, size and permission) are passed through the read-only IPC channel and the trusted thread does not use stack, it satisfies our security policy for mapping executable memory.

Memory mapping in the untrusted process(es) is forwarded to the SDT process by the system call interception mechanism of the sandbox. The request first goes through the system call filtering to make sure the security policy in enforced. SDCG then checks where the request comes from. If the request is from the SDT, or is a special resource the SDT depends on (e.g., mapping new modules needs to be synchronized for Strata), the request is fulfilled from the shared memory pool. If it is a legitimate request from the untrusted code, the request is fulfilled normally.

3) System Call Filtering: SDCG rejects the following types of system calls.

- mmap with writable (PROT_WRITE) and executable (PROT_EXEC) permission.
- mprotect or mremap with target region falls into a protected memory region.
- mprotect with executable (PROT_EXEC) permission.

SDCG maintains a list of protected memory regions. After the SDT process is forked, it enumerates the memory mapping list through /proc/self/maps, and any region that is executable is included in the list. During runtime, when a new executable region is created, it is added to the list; and when a region is unmapped, it is removed from the list. If necessary, the SDT's dynamic data can also be added to this list.

For Strata, this filtering is implemented by intercepting the related system calls (mmap, mremap and mprotect). For V8 (integrated with the Google Chrome browser), we rely on the seccomp-bpf filtering policies.

SDT Specific Handling

Next, we describe some implementation details that are specific to the target SDT.

1) Implementation for Strata: Besides the code cache, many Strata-based security mechanisms also involve some critical metadata (e.g., the key to decrypt randomized instruction set) that needs to be protected. Otherwise, attackers can compromise such data to disable or mislead critical functionalities of the security mechanisms. Thus, we extended the protection to Strata's code, data, and the binary to be translated. Fortunately, since Strata directly allocates memory from mmap and manages its own heap, this additional protection can be easily supported by SDCG. Specifically, SDCG ensures that all the memory regions allocated by Strata is mapped as either read-only or inaccessible. Note that, we do not need to protect Strata's static data, because once the SDT process is forked, the static data is copy-on-write protected, i.e., while the untrusted code could modify Strata's static data, the modification cannot affect the copy in the SDT process.

Writing RPC stubs for Strata also reflects the differences in attack model: since all dynamic data are mapped as readonly, any functionality that modified the data also needs to be handled in the SDT process.

Another special case for Strata is the handling of process creation, i.e., the clone system call. The seccomp-sandbox only handles the case for thread creation, which is sufficient for Google Chrome (and V8). But for Strata, we also need to handle process creation. The challenge for process creation is that, once a memory region is mapped as shared, the newly created child process will also inherit this memory regions as shared. Thus, once the untrusted code forks a new process, this process also shares the same memory pool, with its parent and the SDT process. If we want to enforce an 1 : 1 serving model, then we need to un-share the memory. Unfortunately, un-sharing memory under Linux is not easy: one needs to 1) map a temporary memory region, 2) copy the shared content to this temporary region, 3) unmap the original shared memory, 4) map a new shared memory region at exactly the same address, 5) copy the content back, and 6) unmap the temporary memory region. At the same time, the child process is very likely to either share the same binary as its parent, which means it can be served by the same SDT; or call execve immediately after the fork, which completely destroys the virtual address space it inherited from its parent. For these reasons, we implemented a N : 1 serving model for Strata, i.e., one SDT process serves multiple untrusted processes. And the clone system call can then be handled in the same way for both thread creation and process creation. The only difference is that, when a new memory region is allocated from the shared memory pool, all processes need to be synchronized.

2) *Implementation for V8:* Compared with Strata, the biggest challenge for porting V8 to SDCG is the dynamic data used by V8. Specifically, V8 has on two types of dynamic data: JS related data, and its own internal data. The first type of data is allocated from custom heaps that are managed by V8 itself. Similar to Strata's heap, these heaps directly allocate memory from mmap, thus SDCG can easily handle this type of data. The difficulty is from the second type of data, which is allocated from the standard C library (glibc on Linux). This makes it very challenging to track which memory region is used by the JS engine, and which is not. Clearly, we cannot make the standard C library to allocate all the memory from the shared memory pool. However, as mentioned earlier in the design section, we have to share data that is involved in the RPC so as to avoid serializing objects, especially C++ objects, which can be very complicated. To solve this problem, in our prototype implementation, we implemented a simple arenabased heap that is backed by the shared memory pool; and modified V8 to allocate certain objects from this heap. That is, only objects that are involved in the RPC need to be allocated from this heap, the rest can still be allocated from the standard C library.

Another problem is stack. Strata does not share the same stack as the translated program, so it never reads data from the program's stack. This is not true for V8. In fact, many objects used by V8 are allocated on the stack. Thus, during RPC handling, the STD process may dereference pointers pointing to the stack. Moreover, since the stack is assigned during thread creation, it is very difficult to enforce that the program always allocates stack from our shared memory pool. As a result, we ended up copying stack content between the two processes. Fortunately, only 3 RPCs require stack copy. Note that, because the content is copied to/from the same address, when creating the trusted SDT process, we must assign it a new stack, instead of relying on copy-on-write.

Writing RPC stubs for V8 is more flexible than Strata because the dynamic data is not protected. For this reason, we would prefer to convert functions that are invoked less frequently. To achieve this goal, we followed two general principles. First, between the entry of the JS engine and the point where the code cache is modified, many functions could be invoked. If we convert a function too high in the calling chain, and the function does not result in modification of the code cache under other context, we end up introducing unnecessary RPC overhead. For instance, the first time a regular expression is evaluated, it is compiled; but the next time, the compiled code can be retrieved from the cache. Thus, we want to convert functions that are post-dominated by operations that modify the code cache. On the other hand, if we convert a function that is too low in the calling chain, even though the invocation of this function always result in modification of the code cache, the function may be called from a loop, e.g., marking process during garbage collection. This also introduces unnecessary overhead. Thus, the second principle is that we want to convert functions that dominate as many modifications as possible. In our prototype implementation, since we did not use program analysis, these principles were followed empirically. In the end, we added a total of 20 RPC stubs.

Evaluation and discussion of the

3.10.4 Program-Counter Confinement

PEASOUP confines the subject program to only execute instructions that are intended to be executed by the programmers. In particular, PEASOUP uses program shepherding [122] to confine the program counter (PC) to only be within areas that are explicitly marked an code. The protection provided by program shepherding is redundant with SIM and ILR, but may sometimes provide for a cleaner exit.

3.10.5 Instruction Set Randomization (ISR)

PEASOUP integrates the technique for instruction-set randomization developed by the University of Virginia [112]. [3] demonstrated that software-dynamic translation is the key to efficient instruction-set randomization. Furthermore, the techniques in [3] were efficient enough to allow for strong cryptographic encoding (via AES) of the instructions.

We consider ILR to provide much greater protective strength than ISR. ISR is an effective defense against code-injection attacks. ILR is equally effective against code-injection attacks, but also prevents arc-injection attacks. For these reasons, we do not discuss ISR further in this report.

4.0 Results and Discussion

This section presents results and discussion of those results.

4.1 Phase 1 Independent Test and Evaluation Results

As we mentioned in Section 3.1, the independent test and evaluation for the PEASOUP project was conducted twice—a preliminary T&E run in December 2011 and the final T&E run in April 2012. In this section, we describe and discuss the T&E results.

4.1.1 Preliminary Test and Evaluation Results (December 2011)

The test case composition for the December T&E was as follows:

- 3 real-world test cases: bzip2, ngircd, and tinyproxy,
- 141 engineered benchmarks with memory-corruption vulnerabilities,
- 76 engineered benchmarks with number-handling problems

Below we present the results.

4.1.1.1 Real-world Test Cases

In the preliminary T&E run, PEASOUP successfully defended two of the real-world test cases and may have discovered an unknown bug in the third real-world test case. These results were obtained after disabling some overly protective number-handling checks. More specifically:

- PEASOUP appeared to **correct the behavior** of *bzip2* on the malicious test input. Normally, the malicious input exploited a number-handling error and a buffer overrun to crash the program. Under PEASOUP, bzip2 recognized that the malicious input was malformed, printed an error message, and exited. In addition, bzip2 appeared to correctly process all of the benign inputs.
- PEASOUP appeared to **correct the behavior** of *ngircd* on the malicious test input. As with bzip2, the malicious input to ngircd exploited a combination of a number-handling error and a buffer-overrun error. The malicious input allowed a login to ngircd with an invalid password. Under PEASOUP, ngircd refused to create a connection and appeared to continue operating when provided with the malicious input. As with bzip2, it still handled the benign inputs correctly.
- PEASOUP seemed to **discover an unknown double-free weakness** in *tinyproxy*. This test case only had a single input. PEASOUP reported a double free error and initiated a controlled exit, although this may not have been reported properly.

The table below summarizes the findings for T&E's real world test cases.

Test Case \ PEASOUP Config	Standard	Partial-C1
ngIRCd	PEASOUP changed behavior on good inputs, exploits defeated with continued execution.	Good inputs, unchanged. Exploit input defeated with continued execution.
Bzip2	PEASOUP changed behavior on good inputs, exploits defeated with continued execution.	Good inputs, unchanged. Exploit input defeated with continued execution.
Tinyproxy	PEASOUP reported a double-free error and caused a controlled exit (a change in behavior).	PEASOUP reported a double-free error and caused a controlled exit (a change in behavior).

4.1.1.2 Engineered Test Cases

According to the scores that were released by the Mitre team in March 2012, PEASOUP rendered unexploitable 65.5% percent of vulnerable memory-corruption test cases and 18.9% of vulnerable number-handling test cases. In some cases, PEASOUP defenses were overly conservative affecting the functionality of a program on good inputs. This happened for 13.2% of memory corruption test cases and for 28.9% of number-handling cases.

Mitre released the test cases along with the scores allowing us to conduct a thorough evaluation of PEASOUP in house. This resulted in many fixes and enhancements to the tool. Additionally, the analysis of test cases showed that many of the benchmarks were malformed and many of the test inputs for exposing good and bad program behaviors were mislabeled. This fact was observed by all other STONESOUP research teams as well. As the result, Phase 1 of the STONESOUP was extended by 3 month and a second T&E run was scheduled to obtain more definitive results.

4.1.2 Final Test and Evaluation Results (April, 2012)

The second and final test and evaluation run was conducted in April, 2012. In the interim between the T&E runs, the independent evaluation team significantly revised the body of engineered test cases and extended the set of good and bad inputs for them. PEASOUP tool chain was evaluated on the following corpus of tests:

- 2 real-world programs: bzip2 and ngircd,
- 30 engineered number-handling test cases,
- 211 engineered memory-corruption test cases.

Below, we present and discuss the evaluation results.

4.1.2.1 Real-World Test Cases

The evaluation results indicated that PEASOUP successfully rendered unexploitable the vulnerabilities in both of the real-world programs. However, it altered the functionality of one of them (ngircd) when applied to certain good inputs. The altered-functionality issue came as a surprise for us—we have been using ngircd for regression ever since the December T&E and were confident that we are not altering the behavior in any disallowed way. In fact, we successfully passed ngircd during the first T&E.

Our investigation of this issue indicated that the test system is showing PEASOUP as 'altering behavior' due to a race condition in the test script. The race condition is between ngircd proper and the coprocess that attempts to make a connection to ngircd to feed it predefined input. PEASOUP protections changed the timing characteristics of ngircd by incurring a 5-10 second startup delay, which caused the communication with the input-feeding coprocess to fail. This is the only way in which PEASOUP "altered behavior" of ngircd, and we believe it is allowed, at least implicitly, under the solicitation and the ROE. Note: we had to fix the same (or similar) race condition during the December T&E; however, we did not have time to complete a fix, during April T&E.

4.1.2.2 Engineered Test Cases

The following table shows the results of PEASOUP evaluation on the set of engineered test cases:

	Number Handling	Memory Corruption
Valid Test Cases	30	211
Successfully Processed	27 (90%)	211
Vulnerable Test Cases (test cases that are supplied with “bad” inputs that exploit the vulnerability)	27	210
Rendered Unexploitable	24 (88.9%)	132 (62.9%)
Altered Functionality	2 (7.4%)	11 (5.2%)

After the final test and evaluation run, we received all of the tests used by the independent evaluation team and the test system for automatic running and scoring of the test cases. We set it up locally and conducted an in-depth investigation of the cases where PEASOUP failed to meet our expectations. As the result we identified a number of malformed test cases and a number of invalid input/output pairs. We gave the detailed list and discussion of those test cases in Section 3.1.2. After accounting for malformed test cases and input, the overall results look as follows:

	Number Handling	Memory Corruption
Valid Test Cases	30	211
Successfully Processed	27 (90%)	211
Vulnerable Test Cases (test cases that are supplied with "bad" inputs that exploit the vulnerability)	27	210
Rendered Unexploitable	24 (88.9%)	140 (75.5%)
Altered Functionality	1 (3.7%)	5 (2.5%)

4.1.3 Post T&E Work

We continued working on improving our analysis after the April T&E. We observed that the use of a fixed stack address as the argument to certain libc functions could be a very good indicator of an object boundary. We incorporated this into our analysis and used it to insert padding and canaries in between stack-allocated data. As a result, we believe that we are rendering unexploitable an additional 14 of the memory-corruption tests. We confirmed these numbers by running these tests with T&E test system.

We believe that the improvement is even more dramatic and that we are also rendering unexploitable all or most of the 14 tests for CWE 127. There appears to be an issue in the part of test harness that supplies input for these tests that prevents them from being scored properly. This issue may have affected our original T&E results, although we suspect that is unlikely (the padding we inserted during T&E was too small to be effective on these tests).

In summary, we calculate PEASOUP rendered-unexploitable rate on the memory-corruption tests would be as follows:

- 83.8% if we include the 14 tests that were protected by better object-boundary identification heuristics (we have confirmed that those were rendered unexploitable by running T&E test apparatus).
- 87.1% if the 14 problematic tests for CWE 127 are marked invalid.
- 87.6% if we mark the 14 problematic tests for CWE 127 as passing (as we believe they are).

Overall, our infrastructure and technology seemed to behave well during T&E. Furthermore, Mitre brought some very useful technology to T&E and demonstrated superb flexibility in obtaining valid test results.

4.2 Phase 2 Independent Test and Evaluation

This section summarizes our evaluation of the Phase 2 Test and Evaluation Results.

4.2.1 Preserved Functionality

The primary reason PEASOUP was scored as altering functionality was because of tests that require malloc to return adjacent allocations. In total, this affects 386 tests.

We believe the altered functionality in the remaining tests will be due to a small number of PEASOUP bugs, maybe as few as one.

4.2.2 C1: x86 Binary Number Handling

	MITRE Score	GrammaTech Adjusted Score
Rendered Unexploitable	79.9%	83.0%
Preserved Functionality	87.6%	97.5%

The majority of failures in rendering unexploitable (RUE) for the number-handling tests were for CWE-196, unsigned to signed conversion error. It accounts for 75% of our failures to render unexploitable (in the GrammaTech adjusted scores). If these tests were handled, our adjusted RUE metric would be 95.7%.

All of the CWE-196 tests seem to be based on the same code pattern, and it is a pattern that PEASOUP cannot currently handle. We have plans to make the necessary extensions to PEASOUP needed to handle this pattern (primarily interprocedural numeric analysis), but we did not complete them in Phase 2.

The remainder of the RUE failures are for DOS_UNCONTROLLED_EXIT attacks against Cherokee tests. These might be handled by improving our management of process-crashing events. PEASOUP is already capable of preventing most these types of DoS attacks against Cherokee.

4.2.3 C7: x86 Binary Memory Corruption

	MITRE Score	GrammaTech Adjusted Score with CWE-126 !RUE	GrammaTech Adjusted Score with CWE-126 INVALID
Rendered Unexploitable	90.4%	89.0%	93.6%
Preserved Functionality	69.3%	96.7%	96.7%

All but 4 of the !RUE (Not Rendered UnExploitable) failures in the memory-corruption class were for the tests of CWE-126 and 127, buffer over- and under-read. The scoring for the CWE-126 tests is invalid, however, PEASOUP probably also failed to prevent the exploits.

The fundamental failure was in the analysis to delineate stack-allocated objects. This has been a major focus of our Phase 2 research, and we have made substantial progress: for unoptimized executables, for those portions of the stack that are mapped by debugging information, we are now able to get over 98% recall and precision on data boundaries. We may be doing equally well for optimized executables, but we need to improve our measurement capabilities.

Besides improving the data delineation analysis, we have other strategies to explore for improving our defenses against information leaks based on improved use of canary values. We are confident these tests can be addressed in Phase 3.

4.2.4 x86 Binary Injection

	MITRE Score	**GrammaTech Adjusted Score just bug fixes**	**GrammaTech Adjusted Score bug fixes + grammar extensions**
Rendered Unexploitable	76.8%	91.5%	95.5%
Preserved Functionality	84.4%	98.6%	98.4%

PEASOUP had several bugs in the parsing of strings in DLLs. These caused both altered functionality and failures to render unexploitable. The third column above shows the results of fixing these bugs.

The OS command injection attacks did identify some shortcomings with the approach taken by PEASOUP. These shortcoming were addressed by feature extensions that effectively improved the grammar that PEASOUP uses for parsing command injections. The fourth column shows the results of adding these extensions. We have planned further improvements in string matching that should eliminate the remaining failures to render unexploitable.

4.2.5 x86 Binary Null Pointer Errors

	MITRE Score	**GrammaTech Adjusted Score**
Rendered Unexploitable	90.8%	94.9%
Preserved Functionality	96.2%	96.2%

All of the failures to render unexploitable in the null-pointer-errors class were for DOS_UNCONTROLLED_EXIT attacks against Cherokee. These might be addressed by improving PEASOUP's policies for handling process-crashing events.

Investigating the Cherokee tests was challenging, for many reasons. Cherokee has some built in robustness, which PEASOUP further improves. In many cases, PEASOUP is able to entirely prevent DOS_UNCONTROLLED_EXIT attacks: the client's requests are still serviced, connections running concurrently with the attacked connection are not affected, and the server remains alive and servicing requests after the attack.

However, this did not always happen. Unfortunately, determining what happened with which tests was time consuming and error prone. In the end, MITRE did not have much time to review our suggested scoring changes for these tests.

4.3 Phase 3 Independent Test and Evaluation

Unfortunately, there were insufficient remaining funds for a close examination of the Phase3 Test and Evaluation results.

4.4 Data Delineation Analysis

This section outlines the results of the experimental evaluation of various aspect of Data Delineation Analysis (DDA) and its integration into the stack-layout transformation (SLX). DDA is also described in [98].

4.4.1 DDA Evaluation: 32-bit

Our paper submission to the International Conference on Software Engineering (ICSE), which is attached to this report, provides a detailed coverage of the experimental evaluation of DDA on 32-bit binaries. Here we present a high-level recap of the results.

There are two types of imprecisions that the DDA analysis yields. False positives correspond to inferred object boundaries that do not have counterparts in the ground truth. False negatives correspond to ground-truth object boundaries that are missed by the analysis. We measure both of these:

- *Precision:* is a measure of how many spurious boundaries the analysis infers (i.e., a measure of false positive rate).
- *Recall:* is a measure of how many true objects the analysis misses (i.e., the measure of false negative rate).

We use the following formulas to compute the two metrics. Let M denote the number of object boundaries that the analysis correctly identified, FP denote the number of false positives, and FN denote the number of false negatives. Also, let $GT = M + FN$ denote the number of ground-truth object boundaries.

$$Precision = \frac{GT}{GT + FP} \times 100\% \quad Recall = \frac{M}{GT} \times 100\%$$

The two metrics must be used together to assess the accuracy of the analysis: for an accurate analysis, both precision and recall should be close to 100%. In separation, the metrics could be easily misinterpreted: e.g., finding no boundaries at all yields 100% precision, but 0% recall; similarly, inferring that each byte in the activation record is a separate object, yields 100% recall, but low precision.

We have applied the DDA analysis to programs from Coreutils suite (about a 100 of small programs that share a common library), to the set of Phase 2 T&E base programs, and to a subset of LLVM utilities (large programs written in C++). We indicate the sizes of the benchmarks by stating the number of stack variables in the ground truth and compare the DDA results against IDA Pro. The table below shows the summary of the results that we obtained (a complete table is in the attached ICSE paper submission):

		Precision (%)			Recall (%)		
Benchmark	Vars	IDA	DDA	64-Bit	IDA	DDA	64-Bits
Coreutils	214	81.46	88.23	94.36	95.00	98.91	97.66
Coreutils (-O1)	76	59.03	74.45	85.37	77.23	85.86	84.29
Coreutils (-O2)	86	38.98	55.80	77.96	74.79	80.48	80.16
Ph. 2 T&E (base)	2193	79.30	89.84	93.45	98.68	99.41	98.32
LLVM utils	9504	90.16	93.28	96.44	99.76	99.80	98.46

The IDA column shows the precision and recall obtained by using internal IDA Pro variable identification. DDA column gives the results of DDA analysis as described in this report. 64-bit column shows the extension of DDA with a simple heuristic for identifying 64-bit integers in the 32-bit code: GCC implements 64-bit integers by using two 32-bit words that are manipulated separately, thus causing DDA to yield false positives. Our heuristic finds pairs of adjacent words in each stack frame that are always read or written together within a basic block and marks them as 64-bit words. As the results indicate, this simple heuristic proved to be very effective for detecting such 64-bit integers: the precision is significantly improved, while the decreases in recall are modest.

Overall, the evaluation showed that the DDA analysis significantly more precise than the mechanisms used by IDA Pro. Surprisingly, DDA also has better recall than IDA Pro does. Our investigation showed that it was caused by IDA Pro's failure to recognize variables in function `main(...)` for some of the benchmarks, possibly due to the presence of stack-alignment operations.

4.4.2 DDA Evaluation: 64-bit

In Phase 3, we extended DDA to support 64-bit binaries. We conducted an initial investigation on several benchmarks, though we did not have time to perform an in-depth analysis of the results. The table below shows the preliminary results we obtained:

		Precision (%)		Recall (%)	
Benchmark	Vars	IBI	DDA	IBI	DDA
Coreutils	-	90.77	94.67	96.97	96.93
Coreutils (-O1)	-	82.66	84.86	84.92	84.92
Coreutils (-O2)	-	60.84	68.00	75.19	75.17
OpenSSL	4728	75.78	82.13	85.43	84.20
Postgres	63048	92.47	95.83	97.13	97.08

We did not collect IDA Pro data for the 64-bit benchmarks, so we used the Initial Boundary Identification (see Section 3.3.5.1) results as the base line. Our IBI heuristic is similar in spirit to the IDA Pro internal mechanisms, though IDA Pro (on 32-bits) had a slightly better precision compared to IBI due to relying on information inferred by FLIRT (IDA Pro technique for recognizing standard library calls). Since DDA works by refining the boundaries identified by the IBI, the data for IBI represents the lowest possible precision and the highest possible recall that DDA, in its current incarnation, can achieve.

The results in the table show that for unoptimized code (OpenSSL and Postgres were built without optimizations), DDA eliminated from a third to half of IBI false positives without significantly affecting the recall. We investigated in more details the causes for DDA false positives in OpenSSL. It turned out that one of the main causes of false positives were the low-level encryption routines. Those routines use sets of local arrays that are accessed both in loops and with constant indices, which confuses the parameter offset analysis. Ultimately, more detailed investigation of analysis results on 64-bit binaries is necessary.

4.4.3 Investigation of False Positives

We performed an extensive study of the false positives yielded by our analysis. Below we briefly summarize the main source of false positives. While some of the false positive classes can be addressed with general infrastructure improvements, others require additional approached and techniques to be designed and integrated into the DDA.

Incomplete ground truth. Incomplete ground truth causes some of the identified valid object boundaries to be reported as false positives. The two main sources of imprecision in the ground truth are:

- Compiler-introduced variables—such as locations where intermediate computation results are stored—do not show up in DWARF information. Our analysis however identifies them as independent objects, which are the labeled as false positives during the evaluation of the analysis results. To avoid this imprecision, we chose to exclude from consideration the stack regions for which no DWARF information is given.
- Some variables are instances of the structures or aggregate types, but are only used locally and thus can be broken down by the stack layout transformation.

We need improved ground truth to account for these shortcomings of DWARF information. In a separate effort within the PEASOUP project, we worked on extracting the true object boundary information from the RTL (Register Transfer List) structures built internally by the GCC compiler. Our initial experiences with this approach where positive, but we did not have time to fully integrate the RTL-based ground truth extraction into out framework for evaluating DDA's precision. Instead, in all our experiments, we evaluated the precision of the analysis only in the areas of the stack that are covered by the DWARF information.

Indirect control transfers: The intermediate representation the Parameter Offset Analysis operates on does not have good information for indirect control transfers such as indirect function calls and even jump tables that are often used to implement C switch statements. As a result, the Parameter Offset analysis misses some of the function calls and subsequently some of the information about dereferenced offsets. For this issue, the improvements to the IR will automatically improve the precision of the object delineation analysis.

Local arrays that are both iterated and accessed with constant offsets. The false positives of this kind pose a serious challenge for our analysis because they requires sophisticated loop analyses to address. We may be able to use intra-procedural numeric analyses to conservatively identify portions of activation records that should be left contiguous. Our investigation however showed that such cases are fairly rare: we only so them in encryption routines, such as sha1sum and md5sum.

Passing and returning structures by value. There are several routines in the shared coreutil library that pass and return structures by value rather than by reference. This is often results in code that copies structures a word at a time. Such structure instances are not identified by our Parameter Offset analysis.

4.4.4 Investigation of False Negatives

The number of false negatives yielded by the DDA analysis is much smaller compared to the number of false positives. As a result, we spent less efforts on precisely characterizing their causes. The primary cause that we have observed is the incomplete intermediate program representation in the IRDB. Since our analysis does not see parts of the code it fails to collect the candidate object boundaries from that code. Note that this issue serves as both the source of false negatives (skipping code by the initial boundary identification (Section 3.3.5.1)) and the source of false positives (skipping code by the Parameter Offset Analysis (Section 3.3.5.2)). The root cause for this issue is a deficiency in IDA Pro that is used for disassembly by the STARS analysis which populates the IRDB.

Another source of false negatives that we have encountered is the lack of path sensitivity of the Parameter Offset Analysis. The following example illustrates the problem:

```
int foo(char * str) {
      if (strlen(str) > 16) {
            … str[15] …;
      }
      …
}
```

The parameter offset analysis will infer that the object passed to `foo` is at least 16 bytes long and will use that information at every call site of `foo`, even for objects that are potentially smaller. We have only seen a handful of false negatives that are due to this issue—not enough to justify extending the analysis to be path sensitive.

4.4.5 Evaluation DDA/SLX Integration

As we mentioned previously, unfortunately, we only had time to integrate together 32-bit versions of SLX and DDA. We have used the resulting tool to perform several experiments that tested the precision of DDA-inferred boundaries when used for program transformation. We picked nginx and apache, the web servers used as base programs in Phase 2 Test and Evaluation, as the benchmarks for our experiments. We used DDA-inferred boundaries to transform the stack layout for a large subset of the functions in the two benchmarks, and ran the transformed programs on the respective regression suites to check if the functionality has been altered. We used delta-debugging-like approach [224] to recursively narrow down the list of functions that are broken by the transformation.

For the unoptimized nginx, out of about 450 functions, only 8 were broken by the DDA-based SLX transformation. One of the functions was broken due to imprecision in the SLX transformation, which failed to handle properly a complex address computation. All other functions performed low-level string manipulations that combined standard string- and memory-manipulation operations with direct access to strings with constant offsets. The low-level string manipulation is definitely an area where DDA falls short and requires additional enhancements.

To investigate the effect of the optimization on the effectiveness of DDA-based SLX, we preformed the above experiment on nginx built with “-O2” option. In this experiment, the fraction of broken functions was higher: 22 out of 297. This result was predictable: the optimized code is harder to analyze and DDA consistently yields a higher false positive rate for optimized binaries (see Sections 4.4.1 and 4.4.2). However, we did not have an opportunity to investigate the exact causes for the breaks.

For the apache webserver, our evaluation showed that out of 1938 transformed functions 7 caused failures in the execution of the regression suite. On one hand, this shows that the precision of our analysis is high. On the other hand, we have poor understanding of the exact coverage provided by the test suite thus it is feasible that only a small portion of the code was exercised.

4.5 Checkpointing Test and Evaluation

To test the performance and stability of this primitive implementation of VM fork, we did the following test.

Hardware setup

- CPU: Intel Xeon X3450, 4 cores, 8 threads
- Memory: 16GB
- Disk: OCZ Revodrive X2 240G
- Swap: 32GB (on Revodrive)

Software setup

- Host OS: Debian 6.0.1 x64
- KVM-QEMU: 0.14.0
- KVM-KMOD: latest git checkout
- Guest OS: Ubuntu 10.04.2 LST server, clean install
- Guest CPU Number: 1
- Guest Memory Size: 512MB

Test Method

1. Start the base VM, run burnP6 stress test program;
2. Fork a child VM
3. After the child has loaded the snapshot, login through SSH and execution top command
4. Repeat until the VM is extremely slow or the forking is instable

Metrics

1. The time interval between creating the snapshot and the parent VM is resumed (performance)

2. The time interval of loading the snapshot (performance)
3. The ability to login into the child VM (stability)
4. The ability to find burnP6 in the child VM (stability)

4.5.1 Width-first Forking

Forking VMs from the same parent.

- Round 1: manually fork a VM, record the time and check if succeeded
    - VM successfully forked: 41
    - Fork time: 0.71s ± 0.08s
    - Load time[19]: 0.64s ± 0.05s
    - Memory used: 16371300k
    - Swap used: 0k

  The fork failed on the 42nd VM because the checkpoint file is full (8G).
- Round 2: increase the checkpoint file size to 32G, automatically forks a child every 30s for 63 times
    - VM successfully forked: 42
    - Fork time: 0.70s ± 0.07s
    - Load time: 0.61s ± 0.09s

  The fork failed on the 43rd VM because the default tmpfs size is 8G.
- Round 3: increase the tmpfs size to 32G, rest is the same as round 2.
    - VM successfully forked: 63
    - Fork time: 0.71s ± 0.11s
    - Load time: 0.62s ± 0.11s
    - Memory used: 16376008k
    - Swap used: 7468512k

  After the main memory was used up, the forking and loading time moderately increase by about 0.2s.

4.5.2 Depth-first Forking

Recursively fork child VM (i.e. parent fork a child, then the child fork another)

- Round 1: automatically forks up to 64 child VMs
    - VM successfully forked: 64
    - Fork time: 0.72s ± 0.14s
    - Load time: 0.64s ± 0.10s
    - Memory used: 16371228k
    - Swap used: 8487568k

  The max forking time is 1.04s and the max loading time is 0.87s.

[19] This is only the load time. Before loading the snapshot, we let the VM run for 2s, so the empirical start time for child VM is around 3s.

4.6 Ground-Truth IR Evaluation

The following table gives a snapshot of the current state of the (static) CodeSurfer/SWYX analysis of the 104 programs that constitute the coreutils suite. While these numbers were gathered using CodeSurfer/SWYX, we expect them to be similar to the measurements that we would obtain from STARS. We first compile each program with, –O2 and –gdwarf-2, under the DVT framework to produce an optimized executable with DWARF debugging information and ground truth IR. We then stripped this executable of its debugging information and analyzed it with CodeSurfer/SWYX, and compared its IR against ground truth.

	average		minimum			avg.	min.
	prec %	recall %	prec %	recall %		match %	match %
1. Proc. Entry (DVT)	100.0	97.5	100.0	76.4			
2. Proc. Entry (DWARF)	100.0	97.5	97.6	75.8	3. Proc. Ranges (DWARF)	58.7	42.3
4. Instructions (DVT)	100.0	100.0	100.0	100.0	5. Instr. sizes (DVT)	100.0	100.0
6. Data objects (DVT)	100.0	90.3	100.0	56.5	7. Data sizes (DVT)	87.5	67.6
8. Data objects (DWARF)	100.0	100.0	100.0	100.0	9. Data sizes (DWARF)	100.0	100.0
10. Stack Variables (DWARF)	100.0	93.0	100.0	87.5	11. Stack Variable Sizes (DWARF)	92.8	80.9
					12. DVT Instruction numeric/symbolic operands	99.6	98.0
					13. DVT instruction symbolic references	98.5	94.5
					14. DVT Data numeric/symbolic operands	43.6	22.2
					15. DVT Data symbolic references	3.9	1.7

For the items in the left column we show the average precision and recall over the 104 programs, and the minimum precision and recall among the 104 programs. Precision shows the proportion of CodeSurfer/SWYX IR that matched ground truth, while recall shows the proportion of ground truth that was discovered by CodeSurfer/SWYX. It is important to mention that the domain of the precision computation only includes the regions of the program covered by ground truth. This is because we only have ground truth information for a subset of the final executable (generally ground truth is lacking for statically-linked libraries), and it is not possible to classify any IR outside of the ground truth region as good or bad. In previous versions of the ground truth tool, we were only reporting on precision.

Items in the right column include only a single match percentage, showing both the average over the 104 programs and the minimum (worst match percentage among the 104 programs).

Items 1 and 2 compare procedure entry points, with ground truth from both DVT and DWARF. The precision and recall are all above 94% with the exception of three outliers with recall at around 75%. These outliers are for the programs dir, ls, and vdir (essentially the same program with different names and slightly different behavior), and are due to 52 sort functions that are stored in an array of function pointers and are only ever called indirectly. The CodeSurfer/SWYX IR recovery currently fails to recognize these 52 function entry points, due to a faulty heuristic.

Item 3 compares the procedure boundaries (address ranges). Currently the match rate is not very good. This is due mostly to the granularity of the CodeSurfer/SWYX IR, which excludes padding bytes from its computed ranges. We believe that with a minor tweak to account for these padding bytes we will achieve a much higher rate.

Items 4 and 5 compare instruction starting points and sizes respectively. Encouragingly our precision, recall, and match rate for identifying code is 100% within the ground truth region.

Items 6-9 compare (static) data objects and sizes, with ground truth from both DVT and DWARF. The numbers for DWARF are 100% across the board, but the data object ground truth available from DWARF is a much smaller subset of the static data; in particular, it excludes string literals. The recall and match rate with DVT is worse, though reasonably good. We believe many of the mismatches are still related to string literals.

Items 10 and 11 compare stack variables and their sizes, with a reasonable percentage and room for improvement. It is worth noting for stack variable ground truth two issues which we are ignoring for the moment. First, we assume a unique stack layout for a given procedure; but with a C function includes subscopes, this assumption does not hold. Among the 17499 stack variables in 9205 procedures in the coreutils suite covered in the DWARF ground truth region, there were 893 “conflicting” stack variables occurring in 367 procedures. Of the 893 conflicting stack variables, 277 had different sizes. In these cases we take the larger of the conflicting variables as ground truth.

The second noteworthy issue is that coreutils makes heavy use of inlined functions, and the stack variable layout of such functions – which now occur within the context of the procedure into which the function is inlined – is a bit trickier to define, and are currently excluded from the ground-truth comparison. We do not currently see a way to deal with this issue and may have to live with some noise in the ground truth measurements.

Items 12–15 compare symbolic references, distinguishing between references from code and from data. For code, we consider only immediate and memory-reference operand, while for data we consider data object up to a word size. We measure our match rate by two methods. First, we consider every potentially symbolic operand, and count each such operand as a match if both ground truth and SWYX agree on whether the operand is symbolic, and if so what offset value that symbolic operand has. These are presented in items 12 and 14 in the table. The numbers are good for references in code, and not very good for references in data (at just under 50%). However, we soon realized that since most operands are non-symbolic (which is easier for IR recovery to get right), this method of counting actually skewed the numbers in favor of the IR recovery.

We therefore measured our match rate by a second method, which is to only consider operands for which either ground truth or SWYX determined that it is symbolic. Across the 104 coreutils program, this covers 26% of the code operands and 51% of the data objects. With this method, an operand is counted as a failure if one side determines it to be symbolic while the other doesn’t, or if both sides determine it to be symbolic but with different offsets. Items 13 and 15 give the match percentages when considering only this subset of operands. For code the match is still quite good, but for data the match rate is very poor. An initial investigation shows some of these to be jump tables that failed to be recognized as such by SWYX.

4.6.1 Improved Object-Boundary Recovery

Precise object-boundary recovery is important for effective stack-layout randomization. In the extension of Phase 1, we started investigating novel techniques for improving the precision of object-boundary identification. We judge the effectiveness of object boundary identification by both precision and recall. Precision and recall are affected by the false positives (stack offsets that are incorrectly marked as separating two data objects) and false negatives (stack offsets that are on the boundary of a data object but are not identified as such). For our purposes, we define precision and recall as follows:

- Precision measures the percentage of hypothesized object boundaries that are actually object boundaries. PEASOUP requires perfect precision to ensure that SLR does not break correct programs.
- Recall measures the percentage of object boundaries that are correctly identified. A higher recall rate means that PEASOUP will be more effective at providing protection. (If an object boundary is missed, than SLR cannot protect overruns or underruns that cross the unidentified boundary.)

PEASOUP mostly relies on the analyses in IDA Pro to generate candidate object boundaries and then uses BED and TSET to improve precision by discarding boundaries that are crossed during “normal” execution. Unfortunately, there are cases where we incorrectly removed real object boundaries, causing recall to suffer while maintaining precision.

The primary heuristic used in IDA Pro to determine object boundaries is to look for instruction operands that refer to a constant offset from the stack pointer. Each such offset is assumed to be the boundary of an object. Unfortunately, this heuristic can frequently be wrong, which is why BED and TSET are important for achieving high precision. Last month, we observed that there are variations of this heuristic that are likely to be much more precise such as:

- Assume that constant stack pointer offsets that are passed as function parameters correspond to stack object boundaries.
- Assume that constant stack pointer offsets that are passed to specific functions (such as memset) as parameters correspond to stack object boundaries.

Many other variations are possible. These refinements are more effective than the original heuristic because they eliminate some of the common causes of false positives incurred by the first heuristic, such as an instruction that wrings a prophylactic sentinel at the end of an array. In Phase 1 extension, we implemented the second refinement for the memset function. The memset function is often used to initialize entire objects, and therefore its first argument is very likely to be the beginning of an object. In Phase 2, we intend to explore many different variations of the above heuristics and use a Bayesian learner to determine which heuristics are most effective.

4.7 C5: Command Injection

To evaluate the security and performance of S^3, we applied our system to a variety of engineered and real-world benchmarks. The following sections describe the Experimental Setup, Benchmarks, Performance Evaluation and Security Evaluation in more detail.

4.7.1 Experimental Setup

For our evaluation, we used a 32-bit VirtualBox virtual machine running Ubuntu 12.04 with 4 GB of RAM and a 2GHz Xeon E5-2620 processor.

4.7.2 Benchmarks

To evaluate the performance and security of S^3, we have collected a variety of benchmarks. For real-world benchmarks with CVE reports, we used the SpamAssassin Milter Plugin [99], an email filter interface for detecting spam, and cbrPager [66] version 0.9.16, a program to decompress and view high-resolution images. We configured SpamAssassin Milter version 0.3.1 with SpamAssassin version 3.3.2 and Postfix version 2.9.6. Both of these programs have real-world OS command injection vulnerabilities.

We also used a set of vulnerable programs independently developed by MITRE Corporation from real-world, opensource software. Each program was seeded with a command injection vulnerability. This process was repeated to create many variants with the vulnerability at many locations. Each variant has inputs that represent normal program input, as well as exploit inputs.

Finally, we used a set of small exploitable programs, most less than 100 lines, that were developed by Raytheon. Like the MITRE programs, normal and exploit inputs are provided.

Lastly, to help evaluate performance, we developed a series of microbenchmarks. These benchmarks create an OS command from command line input, and use a tight loop to execute that command as frequently as possible, doing no other work. There are two dimensions of variation in the micro benchmarks: 1) the command to be executed and 2) the primitive used to invoke the command. There are two possible commands to be executed. The command to be executed in one case is echo hello, and in the second case is bzip2 dickens.txt [188], [81]. The two cases represent a fast command and a somewhat more reasonable workload that compresses a 775 KB file. Each microbenchmark uses one of the following primitives to invoke the command: execv, popen, or system.

4.7.3 Security Evaluation

We used a combination of programs with reported realworld vulnerabilities, synthetic test programs, and real-world programs seeded with vulnerabilities by an independent testing team to evaluate the strength of the S^3 approach.

1) Real-World Attacks: We evaluated S^3 against two reported command-injection vulnerabilities that we were able to reproduce in open-source binaries.

The first attack, based on CVE-2008-2575, is a command injection in cbrPager [4]. To extract images, cbrPager invokes the system library call to execute unzip or unrar on the archive, without sanitizing the filename. By crafting an input such as ";rm -rf *;".cbr and providing it where a filename is expected, cbrPager is tricked into executing a malicious command when it attempts to open the putative file. S^3 is able to detect attempts to open a malicious filename and return an error from the system library call. These actions result in the program displaying a message that the file cannot be opened, and exiting harmlessly.

Table 3. Performance overhead in milliseconds. Asterisks indicate the differences are not statistically siginifacnt from the 50 trial runs performed.

Benchmark Type	Benchmark	Native (ms ±95%CI)	S$_3$ (ms ±95%CI)	Absolute Diff. (ms)	% difference
MITRE Seeded	C-C078-NGIN-04-DT03-02	10.6±1.5	10.8±1.8	0.2*	2.1%*
MITRE Seeded	C-C078-CHER-04-DF09-02	11.7±3.3	11.5±1.4	−0.2*	−1.5%*
Real world	spamass-milter 0.3.1	87±70	82±43	−4.4*	−5%*
Real world	cbrPager 0.9.16	121.3±11.6	121.2±9.0	−0.1*	−0.1%*
Micro	echo (system)	1,126±18	1,222±5	96	8.4%
Micro	echo (popen)	1,236±12	1,240±6	4	0.32%
Micro	echo (execv)	1,243±4	1,514±11	271	22%
Micro	bzip2 (system)	1,377±13	1,380±14	2.8	0.2%
Micro	bzip2 (popen)	1,379±12	1,381±12	3.0	0.2%
Micro	bzip2 (execv)	1,366±14	1,373±13	7.2	0.5%

The second attack, based on CVE-2010-1132, is a remote exploit in the SpamAssassin Milter Plugin [7] (spamassmilter), which integrates the SpamAssassin spam filter with either sendmail or Postfix. The vulnerability occurs when the milter is invoked with the -x “expand” option, to pass the email address through alias and virtusertable expansion to allow emails to be redirected to other accounts. In this case, the popen function is invoked on sendmail with the email address provided from SMTP as an argument, without properly sanitizing the email address, which can contain a pipe character. With an SMTP command such as RCPT TO:<username+:"|rm /var/spool/mail">, arbitrary commands can be executed; with careful crafting, these may be sufficient to open a remote shell. Our technology was able to harmlessly block any command injections. The signatures extracted from spamassmilter do not include the vertical bar (pipe) character, foiling any attempt to exploit this weakness. Moreover, the Milter plugin properly error-checks the popen function call, so it continues to function without loss of service in the face of an attempted exploit.

1) *Synthetic Attacks:* We evaluated S^3 against engineered test suites developed by Raytheon and independently by MITRE. The Raytheon engineered suite consists of 18 microtests demonstrating command injections with 22 good inputs and 35 bad inputs, using 9 different function calls ([f]exec[l,le,lp,v,ve,vp], system and popen) and a variety of input-processing techniques. S^3 mitigates all of the bad inputs while breaking none of the good inputs in this test suite.

The MITRE test suite includes 477 OS command injection (based on CWE-78) and 516 OS argument injection (based on CWE-88) test cases [141]. These test cases are based on inserting vulnerabilities into seven base programs: Cherokee, grep, nginx, tcpdump, wget, w3c (from libwww), and zsh. Each test case involves inserting a vulnerable call to popen at various locations in the base program. For the CWE-78 test cases, this call invokes nslookup with an unsanitized argument specified from an environment variable or untrusted file. For the CWE88, the program builds the command using the format string “find / -iname %s.” Semicolon characters are properly sanitized when constructing the command, but the user can still include input that has a -exec argument that is ultimately passed to find. Consequently, they could use an input such as “* -exec rm {} \;” to remove files or execute other commands. For each test case,

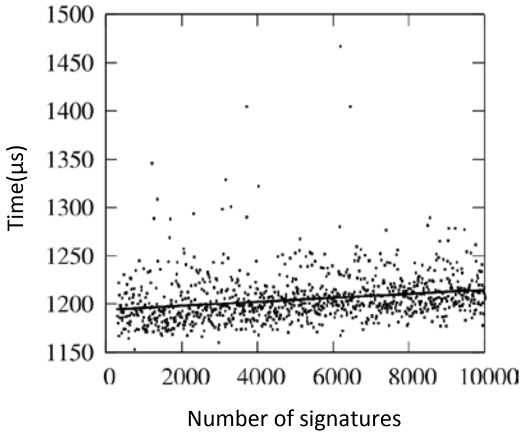


Figure 42. Average time to invoke system versus number of signatures.

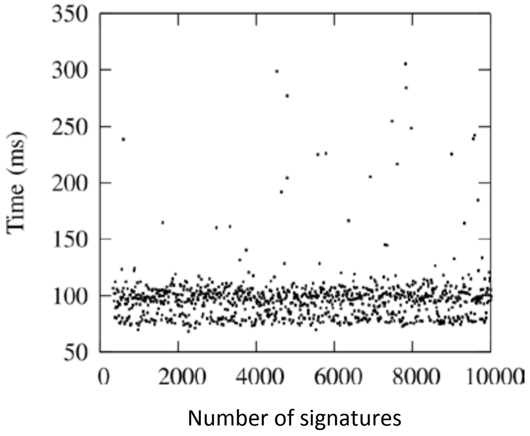


Figure 43. Average time for email transaction versus number of milter signatures.

ten good inputs and two bad inputs are provided. In each case, S^3 was able to intercept the bad inputs without altering behavior on the good inputs.

4.7.4 Performance Evaluation

Table I shows the performance overhead of S^3. The columns show the type of benchmark, benchmark name, performance timing without and with S^3 and finally an absolute and percentage difference, indicating the slowdown S^3 introduces. A 95% confidence interval is shown where appropriate.

We selected two of the benchmarks from the MITRE suite where the seeded vulnerability was in the main loop of a server; most vulnerabilities were injected into startup or shutdown code, and there was no significant performance difference. The seeded vulnerability was set to execute only once, but for timing purposes we modified the code slightly so that it executed on every request to the server. The benchmarks are based on Cherokee (C-C078-CHER-04-DF09-02) and nginx (C-C078-NGIN-04-DT03-02), two production-quality web servers. We performed 50 timings, each consisting of downloading a small html file (574 bytes). Even with the seeded vulnerability

Table 4. Analysis time in seconds

Benchmark	Size (KiB)	Total (s)	Extraction (s)
spamass-milter 0.3.1	142	17.3	0.60
cbrPager 0.9.16	198	25.8	0.69
C-C078-NGIN-04-DT03-02	708	238.6	8.3
C-C078-CHER-04-DF09-02	548	224.2	4.8

in the main loop and the small download size, no statistically significant difference in timing was observed with S^3.

For SpamAssassin Milter, we wrote a simple client that uses gettimeofday to measure the time spent in processing an email transaction. We also modified cbrPager to measure the time to render the first page of a 49 MB input file. Like the MITRE benchmarks, these benchmarks show no statistically significant overhead.

Unfortunately, the server applications have relatively high variance due to network latencies, disk caching, etc. To deal with this issue and benchmark worst-case overhead, we use the microbenchmarks described in Section V-C2. For these benchmarks, we perform 50 timings, where each timing invokes 10 or 1,000 OS commands for the bzip2 or echo microbenchmark, respectively. The microbenchmarks that invoke bzip2 to compress a file show that S^3 causes practically no overhead, only 0.2%. The true worst-case performance overhead is when the program does nothing but issue OS commands, and each OS command invocation completes extremely quickly. This case is represented by the microbenchmark that issues the echo hello command. These benchmarks show that the absolute worst case overhead might be as high as 22%. However, in practice the actual work performed by the program and by executing the OS command clearly dominates the overall runtime. Only our worst-case microbenchmarks demonstrate that S^3 generates any measurable overhead.

To verify the move-to-front heuristic was working properly, we measured the overhead of the echo microbenchmark that uses the system function to invoke OS commands as we vary the number of DNA fragments. We automatically added randomly generated strings to the program's DNA fragments. Figure 42 shows the average time in microseconds for the microbenchmark to execute the system call 100 times, using from 300 to 10,000 signatures (timing starts after steady state has been reached). There is a very slight positive correlation as shown by the line of best fit $y = 0.002x + 1194$. Our investigation indicates that the command processing and matching time after initialization was fixed across the differing number of signatures, but that as there are more signatures in the process's address space, the fork system call (used to implement system) takes longer. We suspect this is a result of taking slightly longer to copy additional page table entries for the new process.

In practice, this additional overhead is negligible since most programs have few signatures. For example, SpamAssassin Milter has 316 signatures and nginx has 2,017. Figure 43 shows the average time in milliseconds to process an email transaction over 50 trials applying S^3 with from 320 to 10,000 signatures to SpamAssassin Milter, which shows no trend. We would not have expected to see any correlation, given an expected increase of only 20 microseconds based on our microbenchmark and the higher time variance of the email benchmark.

Based on these microbenchmark and real-world benchmark performance results, we believe that in practice the S^3 system would introduce no measurable overhead, and is the fastest OS command injection detector to date.

4.7.5 Analysis Time

We measured the time for offline analysis (i.e., DNA Fragment Extraction) of the real-world benchmarks. The results are shown in Table II. This table shows the size of the analyzed executables and libraries, the entire static analysis time, and the portion of that time spent in fragment extraction and processing. This analysis needs to be performed only once. Our results show the analysis taking up to four minutes for nginx. The time is dominated by the disassembly and IR recovery steps that can be shared by other binary analyses and protections. The actual time devoted to string extraction and post-processing amounts to about 2% of the analysis, completing in between 0.5 to 8 seconds on our benchmarks.

4.7.6 Security Discussion

4.7.6.1 Spurious Attack Detection (False Positives)

The current S^3 prototype makes the assumption that command words originate from static strings in the binary or dependent libraries. If this assumption is violated, S^3 may incorrectly flag a benign command as an attack, thereby breaking programs. As observed by Halfond et al., these situations can often be detected easily during a pre-deployment testing phase [101]. Since the S^3 architecture cleanly separates the specification of fragments from their use at run-time, additional fragments can be incorporated simply by appending to the DNA fragment list. This step could be done manually or automatically through further tool support.

Another possibility is that the strings in the program that form commands cannot be detected via static analysis of the binary program. Such a situation might occur if the program is encrypted or highly obfuscated. Such situations are expected to be uncommon. However, we do not have experimental evaluation of the prevalence of obfuscated binaries and our evaluation suite may not be representative in this respect.

4.7.6.2 Missed Attack Detection (False Negatives)

The extraction heuristic introduced in Section IV-A may lead to spurious fragments, i.e., string fragments that do not actually exist in the binary’s source code. Spurious fragments may also be introduced as a result of separating of fragments containing format string specifiers into sub-fragments (Section IV-A2). For example, the fragment "echo ’%s’" would be sub-divided into the fragments "echo ’" and "’". These spurious fragments may inadvertently match special shell operators or command words that might be useful in an attack, e.g., ’, ‘, <, >, $(, |, ;, &.

Another issue is that a binary may truly contain dangerous string fragments. Consider the following code snippet used for filtering out dangerous characters:

```
    if (strstr(s, "’") || strstr(s, "‘") || strstr(s, ">")) ...
```

Or code that uses concatenation to assemble strings:

```
    string s = "echo ’"; s += name; s += "’";
```

Such strings may thwart the ability of S^3 to detect all attacks. The current S^3 system handles this situation by using the policies described in Section IV-E. These policies enforce the constraints that critical command names, along with shell characters that start sub-command, must come from a single signature. These policies also handle the case where there may be fragments for each character in the alphabet, which could be reassembled to form any command name. Such a situation might be common for some programs that utilize many different command line options: programs often contain each command line option as a single character string! Despite this, S^3 has been able to detect all command injections in practice using the policies described previously.

1) Future Work: Reducing the Attack Surface: We plan on investigating simple data flow analyses methods to prune the fragments to just the set which might reach an OS command site. We would omit a fragment if we could prove that it never flowed into a critical command, as is the case with the strstr example above. Furthermore, we plan on refining our fragment matching algorithm to allow for regular expressions. Instead of breaking up fragments when they contain a format string specifier, we would instead substitute the corresponding regular expression pattern specifier, e.g., [0-9]+ for %d.

4.7.6.3 Subtle Injections

Some “command injections” are particularly difficult to detect using tainting information. Consider this program snippet:

```
    system("make");
```

The parameter to the command execution library is constant. However, if the program has root privileges and the user has the ability to control the executable search path, they can modify the make command to do as they wish, and avoid the programmer’s intended security policies. Likewise, consider this program snippet:

```
    sprintf(buf,"cat %s", argv[60]); system(buf);
```

The user is presumably allowed to specify a file to be displayed by the program. However, if the user specifies a file with a relative or absolute pathname, a security policy may be violated. Perhaps even trickier is if the user specifies no file at all, and the command becomes simply cat with no parameter. Terminal input is then used instead of a file on the file system. Again, no command is “injected” into the program.

Lastly, if the program specifies that an external intepreter should be used to interpret commands, detection may be challenging. Consider this program snippet:

```
    sprintf(buf,"echo %s | bc", argv[60]); system(buf);
```

The program snippet indicates that the user should be able to specify input to the bc program. However, bc accepts a large variety of commands, which may have effects that were not anticipated by the programmer. Many programs in a standard Linux install have this characteristic, e.g. bash -c, zsh, find -exec, psql -c, printf, etc., all have flags that allow them to interpret arbitrary commands. Furthermore, there are many non-standard interpreters. There is no *a priori* way to establish whether a program is an interpreter or not, which language it might accept, and which parts of the language are intended by the programmer.

While each of these cases are hard to detect with S^3, they also represent the most challenging command injections to handle automatically. They represent the fundamental problem that programmer intent is not typically available. Without clear, correct, and formally represented programmer intent, no tool can detect all OS command injections. Even expensive taint propagation systems, which are considered largely effective, would be ineffective against the attacks shown.

4.8 C6: Concurrency-Error Defenses

This section presents the results of our evaluation of the defenses against concurrency vulnerabilities discussed in Section 3.7

4.8.1 TOCTOU Defenses

In all real-world programs we tested, injecting our TOCTOU tool produced no noticeable runtime overhead. Presumably this is because we are hooking file system operations which are typically infrequent and already slow, since they are accessing persistent storage.

We were unable to obtain a real-world program with a file system TOCTOU vulnerability and a corresponding exploit. As such, we evaluated our mitigations against synthetic tests that exhibit TOCTOU vulnerabilities for file operations that we support.

4.8.2 Deadlock Defenses

We evaluated the effectiveness of our deadlock tool against many synthetic tests, as well as against a real-world deadlock from the sqlite database library. It effectively detected and recovered from all deadlocks, and was able to avoid them on the second run of the program.

Performance was an important consideration for our approach. Our use of a completely dynamic technique meant that we had to run the deadlock detection algorithm at every lock() operation, which can be quite expensive. Initially performance was quite bad: approximately 800% on some applications. We then modified our approach to attempt a few non-blocking calls to the lock() operation before performing deadlock detection. If any non-blocking lock() call succeeds then we are clearly not in a deadlock, and the detection step can be skipped. This drastically reduced overhead, as you can see below.

We evaluated performance on parallel bzip2 (Pbzip2) and Apache. We ran PBZip2 on a 20MB compressed file, and varied the number of threads. We ran each test 10 times. Table 5 shows the results, where “Protected” means it was run using the deadlock tool. Overhead ranges from <1% to about 13% (mostly due to particularly large outlier in one of the runs).

Table 5: PBZip2 experiments with a 20MB bz2 file. Times are in seconds, and are averaged over 10 runs.

Threads	**2**	**4**	**6**	**8**
Compress (Unprotected)	13.48	7.29	6.32	5.73
Compress (Protected)	13.54	7.50	6.54	5.98
Compress Overhead	*0.4%*	*2.9%*	*3.5%*	*4.3%*
Decompress (Unprotected)	3.01	1.78	1.50	1.30
Decompress (Protected)	3.07	2.02	1.56	1.42
Decompress Overhead	*2.0%*	*13.5%*	*4.0%*	*9.1%*

For testing on Apache, we used the ab (Apache Bench) utility and the default home page. For each test we did 10 runs of 10000 requests each. These are short, frequent requests so it exercises the deadlock code very heavily. From Table 6 you can see that the tests with fewer threads have lower overhead, probably because of less lock contention.

Table 6: Apache tests, for a variety of simultaneous connections (threads). Times are in seconds, and are averaged over 10 runs.

Connections	**20**	**40**	**60**	**80**	**100**
Average Time (Unprotected)	0.6911	0.6643	0.633	0.6024	0.6479
Average Time (Protected)	0.9568	0.927	0.9182	0.876	0.9367
Increase	38.4%	39.5%	45.1%	45.4%	44.6%

Our profiling indicates that the primary remaining cause of performance overhead is our collection of stack traces at synchronization operations. This is upwards of 90% of the time the process spends inside the deadlock tool. To improve upon this we would look into extending an approach like Probabilistic Calling Context [44] to do only the last *K* functions on the stack. Unwinding the stack on the fly (which is what we do now) is simply an expensive operation.

4.8.3 Signal-Handler Defenses

We evaluated our signal handling against many synthetic tests. Errors related to CWEs 479 and 831 were handled very nicely by this approach. It further has the advantage of protecting our other mitigations (namely, Twitcher malloc and our deadlock detection) from signal handler interruption.

Our testing indicates that buffering signals has essentially no overhead for most programs. In some of our micro tests (which process significant numbers of signals) we actually noticed that buffering improved execution time. The deadlock performance numbers in the previous section actually include signal buffering enabled as well.

4.8.4 Atomicity-Violation Defenses

The preliminary evidence indicates that perturbing the schedule early in process execution does not necessarily change the schedule later in process execution. Below we describe a simple experiment where we attempted to roughly measure the overall impact of early, random delays on the thread schedule.

In [72] they describe a Deterministic Multi-Threading (DMT) approach where part of their evaluation is based on memory order determinism. The referenced DMT approach perturbs the schedule only at synchronization API calls (e.g., pthread_mutex_lock()) and shows that this correlates significantly with changes to the order in which memory accesses occur between threads. Other DMT approaches [71] have also used synchronization API-based determinism as a proxy for actual schedule determinism. We use this connection between synchronization API ordering and schedule determinism to measure the impact of our random delays.

We performed the experiment using Parallel BZip2 (pbzip2) with 8 threads. We traced all pthreads synchronization API calls, along with their argument (e.g., which mutex) and the thread that performed the operation. Thread IDs were normalized in a first-come, first-served fashion.

ASLR was disabled to get determinism run-to-run in the API arguments. We then used diffutils as an approximation for computing the edit distance between these traces (since they were line-based traces). We tested delays of none, 0.5 seconds, 1 second, 2 seconds, and 4 seconds. We did 30 runs for each delay value.

Figure 44 shows the average of taking the pair-wise trace differences between the runs, at each delay level. As you can see, these results seem to indicate that determinism actually increased with any introduced delay.

Delay	**None**	**0.5 s**	**1s**	**2s**	**4s**
Average Pair-Wise Trace Difference	4022.12	3800.74	3887.67	3789.73	3773.63

Figure 44: Average pthreads trace difference between 30 runs of pbzip2 with 8 threads.

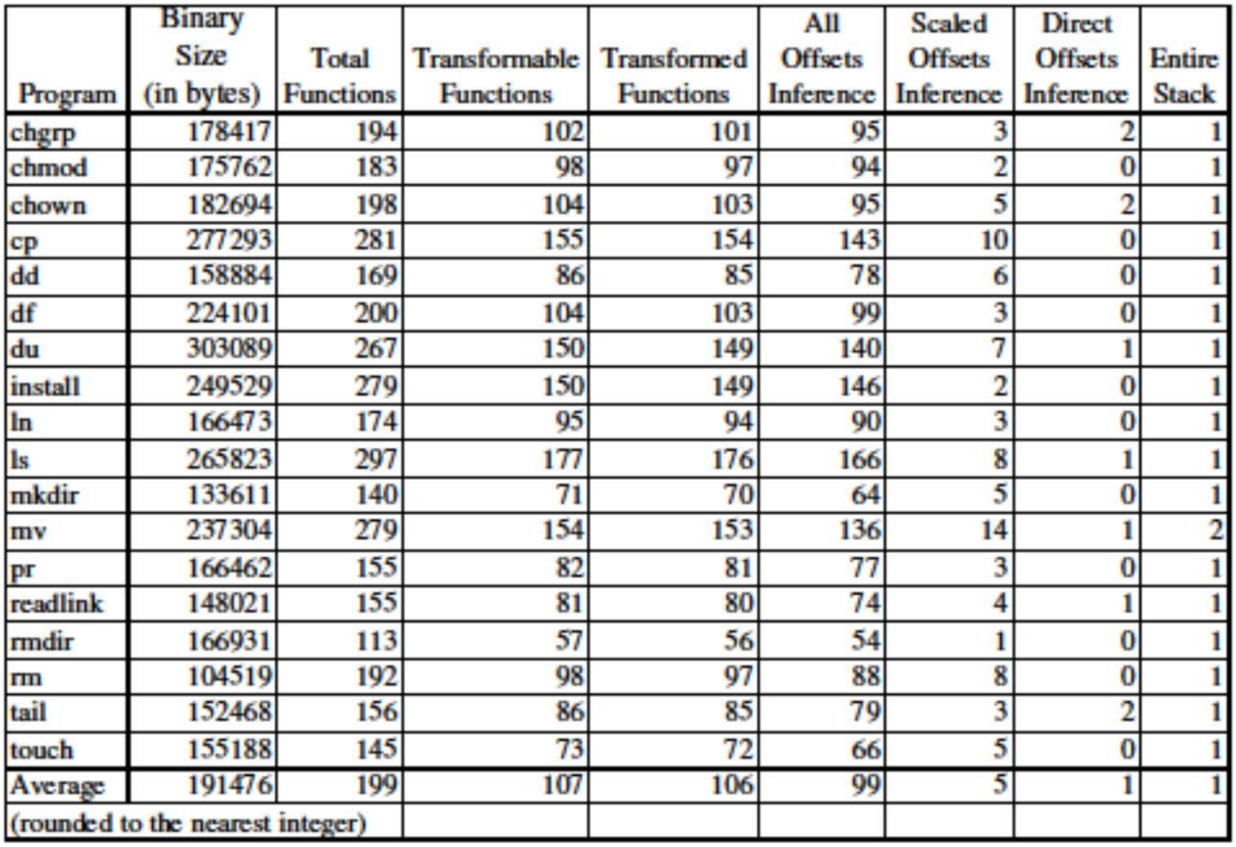
One possibility is that using a trace of pthreads operations as a proxy for determinism is not valid, or simply not a fine enough granularity measurement. The real proof of whether this technique is effective would be to test against actual, exploitable atomicity violations. On real, deployed systems, the non-determinism inherent in the system may outweigh any injected delays and be sufficient to prevent the repeatable exploitation of races.

4.9 C7: Stack-Layout Randomization Evaluation

4.9.1 Transformation Metrics

To evaluate the effectiveness of SLR on real programs, we transformed 18 Unix core utilities version 7.4. Stack layout randomization depends on regression tests to validate layout inferences, and so we selected eighteen based on the availability of regression tests provided by the vendor.

We compiled all 18 utilities using gcc version 4.4.3 with O3 optimizations and dynamic linking on Linux kernel 2.6.32-35-generic as part of Ubuntu 10.04.03 LTS.



Program	Binary Size (in bytes)	Total Functions	Transformable Functions	Transformed Functions	All Offsets Inference	Scaled Offsets Inference	Direct Offsets Inference	Entire Stack
chgrp	178417	194	102	101	95	3	2	1
chmod	175762	183	98	97	94	2	0	1
chown	182694	198	104	103	95	5	2	1
cp	277293	281	155	154	143	10	0	1
dd	158884	169	86	85	78	6	0	1
df	224101	200	104	103	99	3	0	1
du	303089	267	150	149	140	7	1	1
install	249529	279	150	149	146	2	0	1
ln	166473	174	95	94	90	3	0	1
ls	265823	297	177	176	166	8	1	1
mkdir	133611	140	71	70	64	5	0	1
mv	237304	279	154	153	136	14	1	2
pr	166462	155	82	81	77	3	0	1
readlink	148021	155	81	80	74	4	1	1
rmdir	166931	113	57	56	54	1	0	1
rm	104519	192	98	97	88	8	0	1
tail	152468	156	86	85	79	3	2	1
touch	155188	145	73	72	66	5	0	1
Average	191476	199	107	106	99	5	1	1
(rounded to the nearest integer)								

Figure 45. Statistics for the Binary Programs Used for Assessment

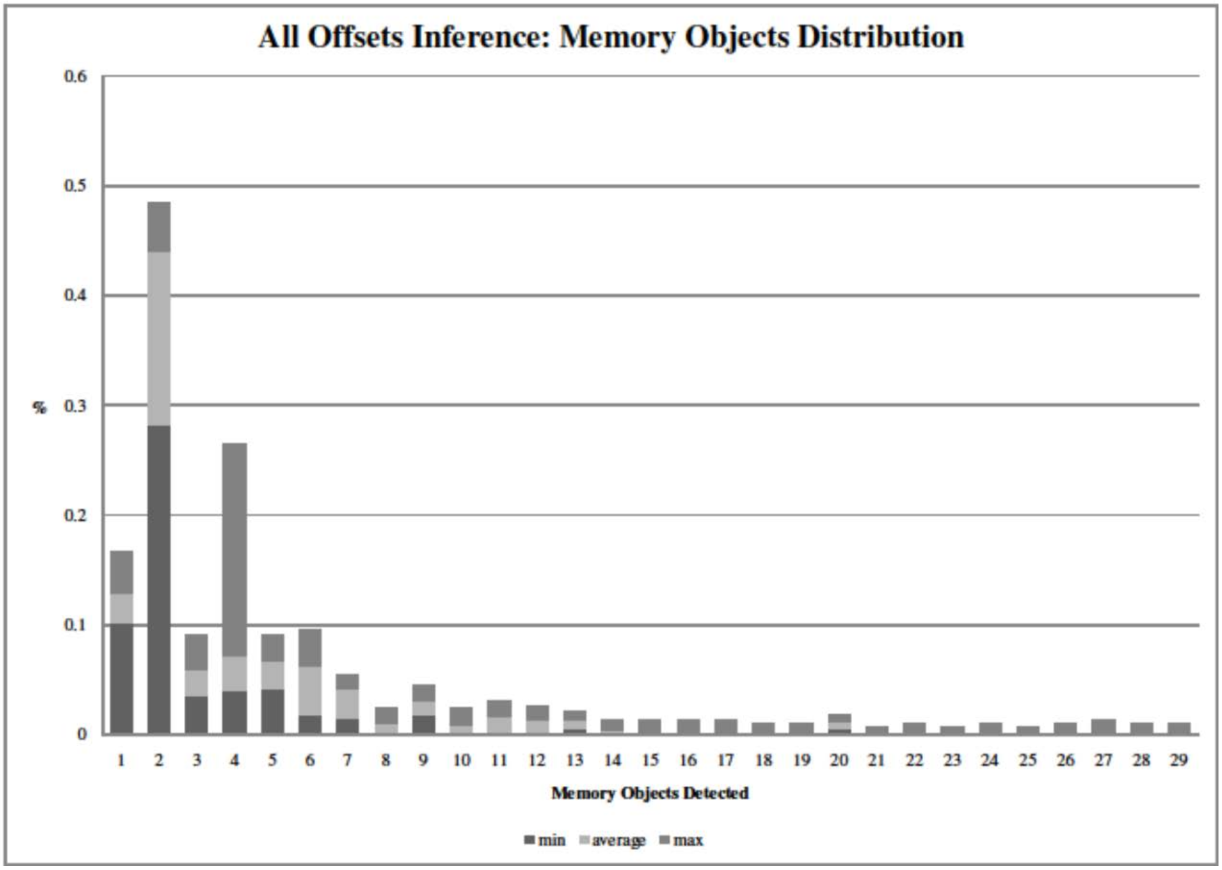


Figure 46. Performance of the All Offsets Inference Heuristic

Only those functions that were not dynamically linked were transformed, i.e., only those functions whose definitions can be found statically in the binary. Libraries should be transformed and evaluated separately as separate regression tests are needed to properly test all library functions. Additionally, libraries should only need to be transformed once and reused by any number of binaries.

Figure 45 shows the statistics of the binaries used and the way in which functions were transformed. For each program, all but one of the transformable functions were transformed using one of the layout inference heuristics. The exception was present in all programs and removed from transformation through regression testing.

Also present in every program was a function that was always reverted to using ESI, the most conservative layout heuristic. The vast majority of transformations, 93.6%, were based on layouts produced by our most aggressive heuristic, AOI. Thus 6.4% of all functions initially transformed with AOI resulted in semantic changes detected by regression testing that triggered a transformation rollback.

Figure 46, Figure 47, and Figure 48 show the distribution of the number of memory objects detected on the stack by each layout heuristic used for each function randomization, where a memory object may be a local variable or the out arguments region. In Figure 46, the distribution for AOI shows that 56.9% of all AOI inferences used found only one or two stack memory objects per function. Since AOI is the first inference used by default, this result suggests most functions use few or no local variables, perhaps leaving the stack with only an out-arguments region.

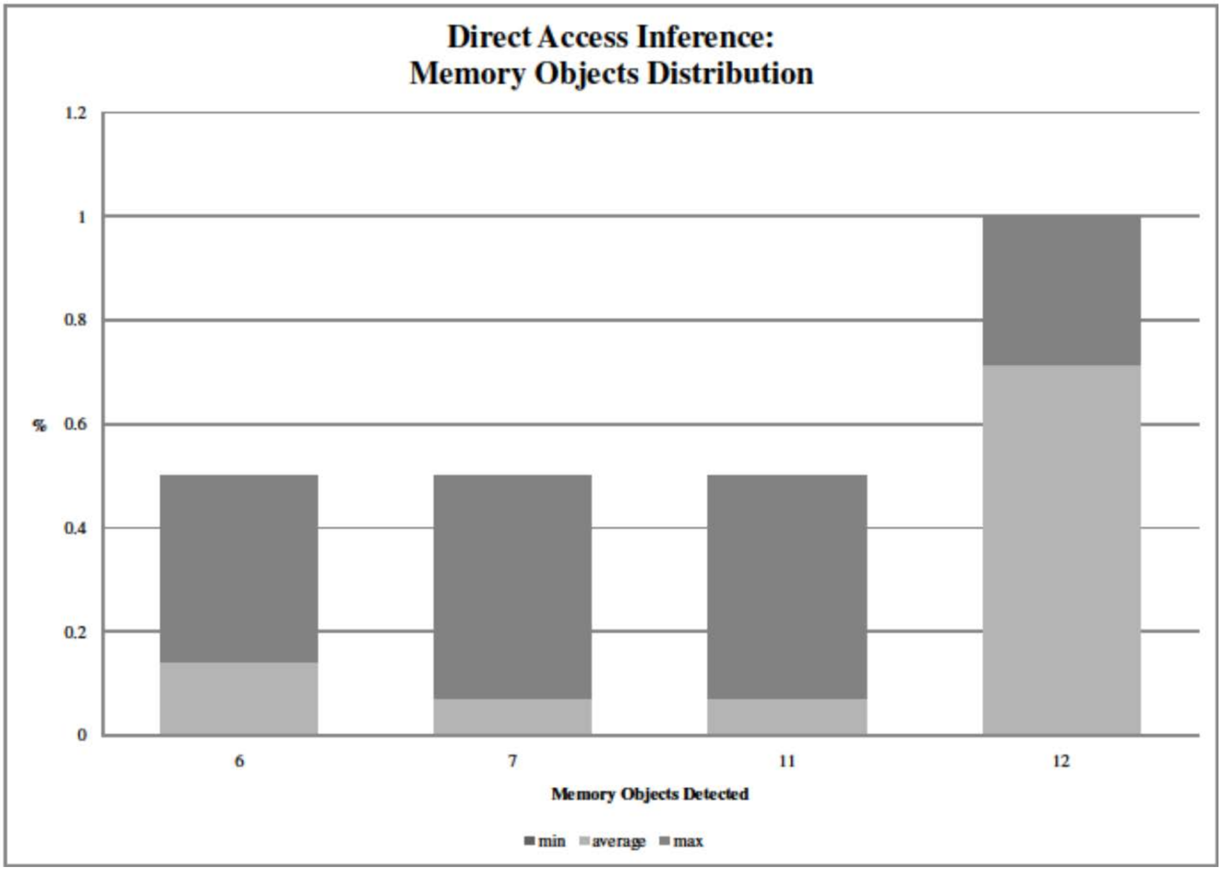


Figure 47. Performance of the Direct Access Inference Heuristic

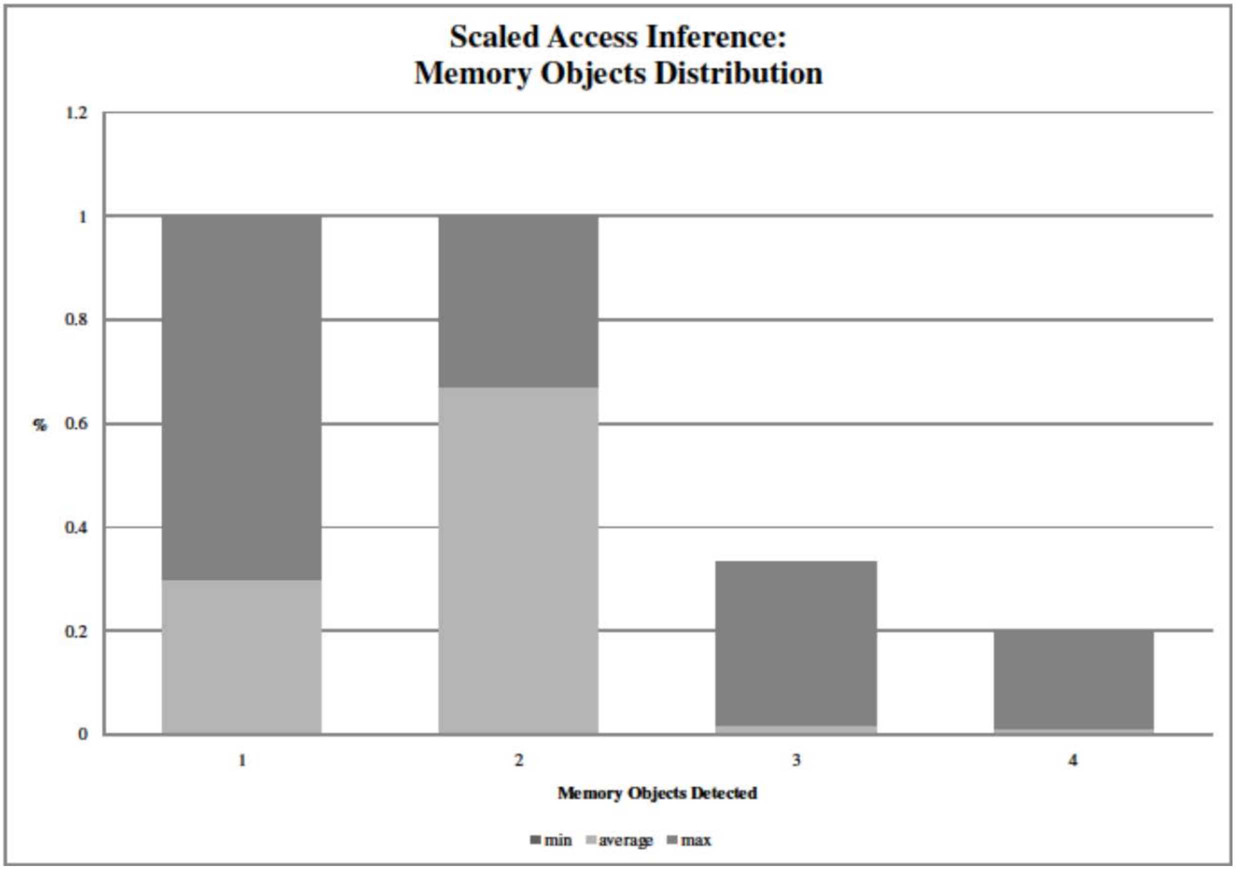


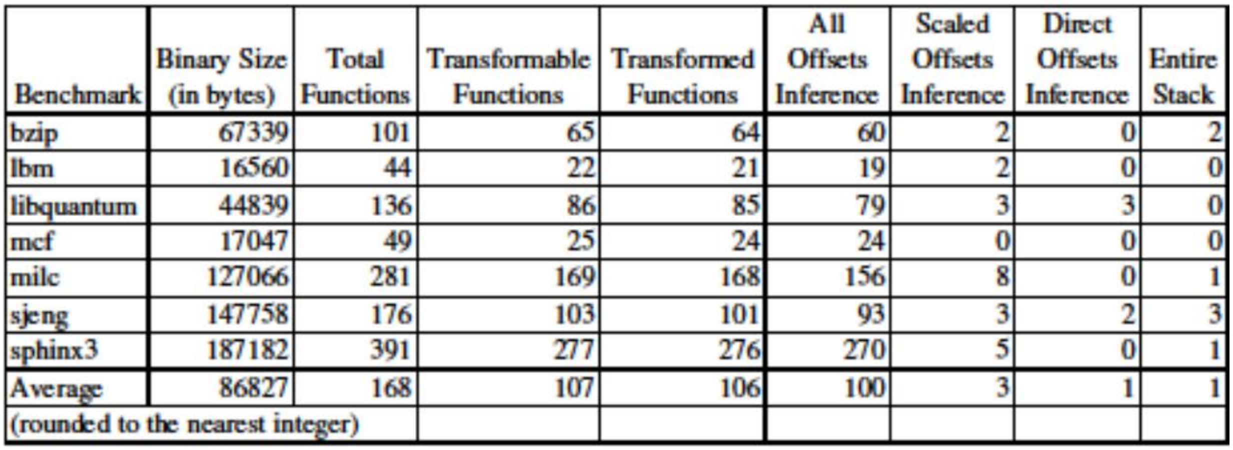
Figure 48. Performance of the Scaled Access Inference Heuristic

The distributions for both DOI and SOI, Figure 47 and Figure 48, show that DOI is generally more aggressive than SOI. Bhatkar et al [42] report dynamic analysis performed on a suite of eight programs for which they find an average of 87% of the stack accesses made are to non-buffer stack variables. If non-buffer stack variables are most prevalent, then prevalence of direct accesses is also expected since this is the typical access mechanism for non-buffer variables. DOI would also be expected to infer the presence of larger numbers of variables than SOI because direct references are more common.

If a layout inference is only able to find one stack memory object, then the layout is reducible to the most conservative layout inference, ESI. In many cases SOI is either reducible to ESI, see Figure 48, or produces a very course grained layout consisting of only two memory objects usually indicating the separation between the out arguments region and the remaining stack. This might explain why few functions (at most two unique functions) are transformed with ESI. Course-grained inferences are less likely to break expected behavior, and by default if ESI and SOI produce the same number of memory objects, the scaled-offset inference is attempted first.

4.9.2 Performance Metrics

To evaluate run-time overhead, SLR was applied to a set of seven SPEC CPU2006 C benchmarks. Execution-time overhead (wall clock time) was measured on a system with a dedicated 4-core, AMD Phenom II B55 processor, running at 3.2 GHz. The machine has a 512KB L1 cache, a 2MB L2 cache, a 6MB L3 cache, and 4GB of main memory. Performance numbers were generated by running the benchmark 3 times. For these measurements, the test programs were compiled using gcc version 4.4.3 with O2 optimizations and dynamic linking on Linux kernel 2.6.32-34-generic as part of Ubuntu 10.04.03 LTS. The regression tests used for each benchmark consisted of a subset of the SPEC train and test input suites for each program.



Benchmark	Binary Size (in bytes)	Total Functions	Transformable Functions	Transformed Functions	All Offsets Inference	Scaled Offsets Inference	Direct Offsets Inference	Entire Stack
bzip	67339	101	65	64	60	2	0	2
lbm	16560	44	22	21	19	2	0	0
libquantum	44839	136	86	85	79	3	3	0
mcf	17047	49	25	24	24	0	0	0
milc	127066	281	169	168	156	8	0	1
sjeng	147758	176	103	101	93	3	2	3
sphinx3	187182	391	277	276	270	5	0	1
Average	86827	168	107	106	100	3	1	1
(rounded to the nearest integer)								

Figure 49. Statistics of the SPEC2006 Benchmarks Used for Timing Assessment

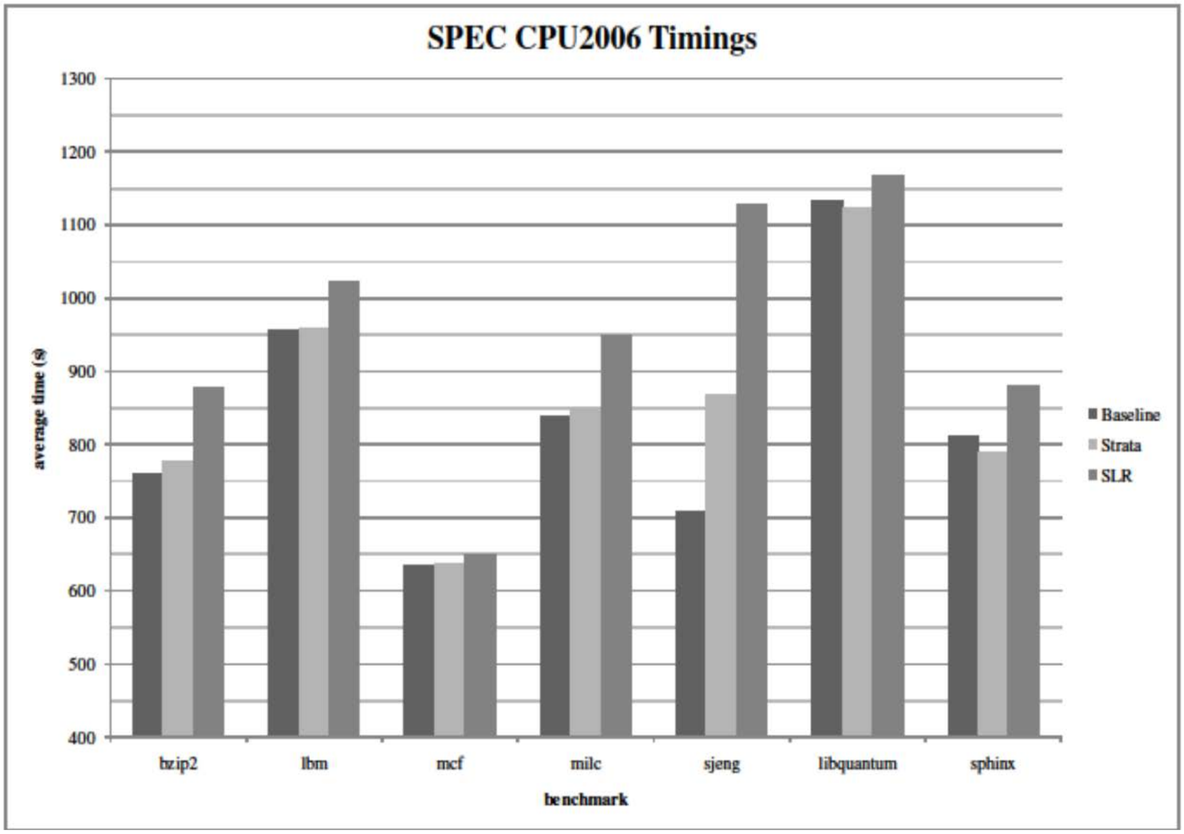
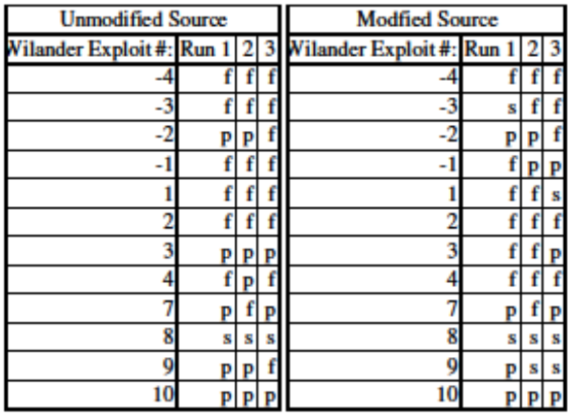


Figure 50. Timing Assessment of SLR on the SPEC 2006 Benchmarks



Unmodified Source				Modfied Source			
Wilander Exploit #:	Run 1	2	3	Wilander Exploit #:	Run 1	2	3
-4	f	f	f	-4	f	f	f
-3	f	f	f	-3	s	f	f
-2	p	p	f	-2	p	p	f
-1	f	f	f	-1	f	p	p
1	f	f	f	1	f	f	s
2	f	f	f	2	f	f	f
3	p	p	p	3	f	f	p
4	f	p	f	4	f	f	f
7	p	f	p	7	p	f	p
8	s	s	s	8	s	s	s
9	p	p	f	9	p	s	s
10	p	p	p	10	p	p	p

Legend: “f” – segfault; “p” – exploit prevented; “s” exploit succeeded

Figure 51. Results of Running the Wilander Buffer-Overflow Exploit Tests

Figure 49 shows the statistics for each program and how functions were transformed. This data is consistent with the results from the coreutils programs (see Figure 45). However, the regression tests used for SLR were not extensive as the goal of this experiment was to provide execution-time performance results. We provide these statistics to show that actual transformations were made to these benchmarks, and most transformations used our most aggressive layout heuristics. However, without extensive regression testing many of these transformations might actually be based on incorrect layout assumptions. The only functionality of any consequence for these benchmarks is that which is executed for performance testing, so as long as this expected output did not break and SPEC did not report any errors or output discrepancies, as compared with the baseline data, we considered the transformation successful.

Figure 50 and Figure 52 show the runtimes and run-time overheads of seven programs with and without SLR applied. The execution-time overhead of the PVM, the software dynamic translator, averaged 3.4% over the execution times of the compiler-produced binary (native) code. The average execution-time overhead of the PVM running the SLR binary rewriting rules over the native run was 15.6% and ranged between 2.4% and 59.5%. SLR, then, only incurs a 9% overhead over the PVM alone, i.e., dynamic binary translation but with no transformations being applied.

The current implementation of SLR makes no effort to keep stack variable alignment, and this can cause cache performance losses that can greatly increase the overhead in certain programs. The worst overhead we found was 59.5% for sjeng. Retransforming this program with stack alignment, we were able to improve the overhead to 36.8%. The results of security analysis (see section C) suggest misalignment of the stack may have security benefits; we therefore leave it to the user to decide if the performance penalty of misalignment is worth the security benefits. The remaining programs had an overhead of less than 16%, with many falling under 10%. If we consider the overhead for unaligned sjeng as intolerable and accept the overhead for all other programs, then replacing the overhead results for sjeng unaligned with the alignment enforced randomization, the average overhead from SLR is 12.4%.

The more functions that are transformed by SLR, the more instruction rewriting rules must be generated. Reading and processing this file is part of the transformation's overhead. For the SPEC benchmarks we analyzed, less than 2 MB of rewriting rules were generated. However, for very large programs, it is conceivable that SLR could require large instruction-rewriting rule files. The rewriting rules are currently stored in ASCII plaintext, and we believe that a binary encoding of the rewrite rules and an efficient storage technique, such as gzip, could easily reduce the processing time and reduce the memory footprint.

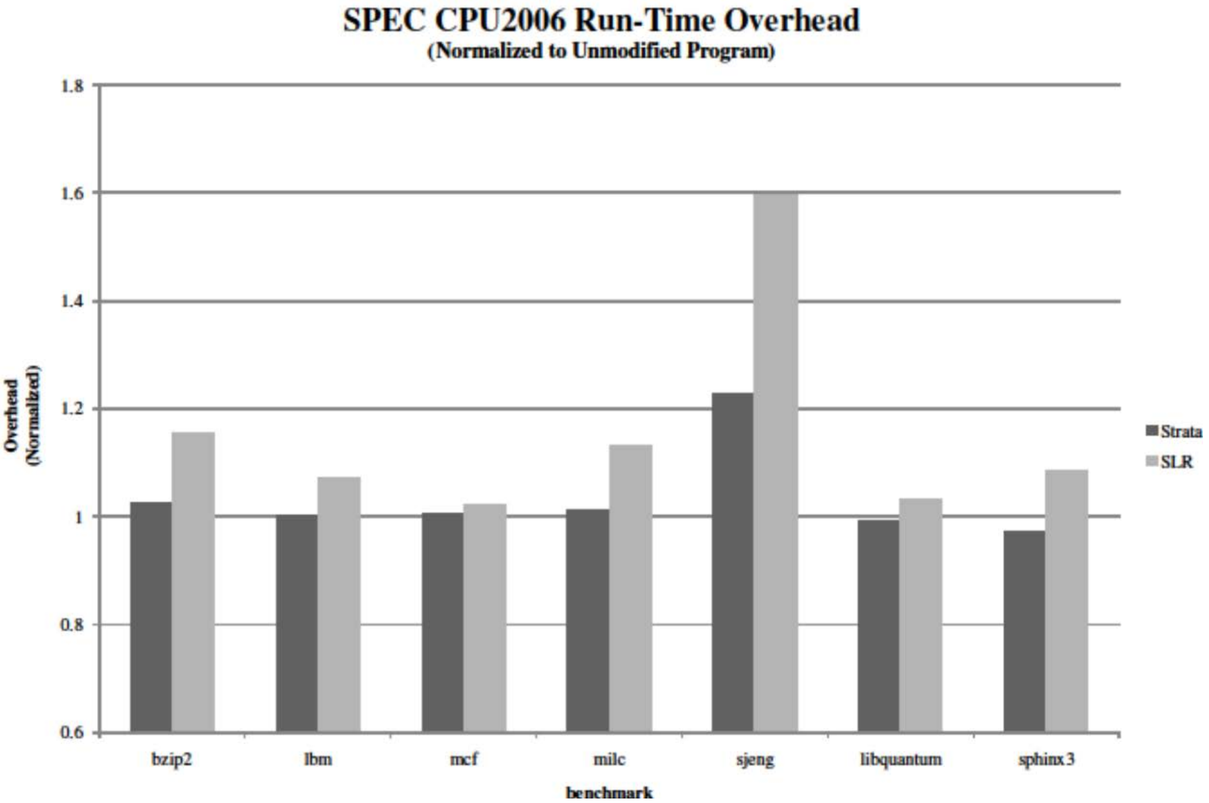


Figure 52. SPEC CPU2006 Run-Time Overhead

4.9.3 Security Discussion

We applied SLR to the Wilander buffer overflow test suite [217], specifically to the stack-based buffer overflow subset consisting of twelve exploits. The Wilander suite provides buffer overflows that are able to determine the appropriate amount by which to overflow the vulnerable buffer at run-time. This functionality makes these overflows some of the most difficult cases to thwart.

For SLR, the Wilander suite is problematic because all valid inputs, except for no input, trigger a buffer overflow that results in execution of arbitrary code. Developing a set of regression tests for this suite is problematic, because the expected behavior (in some sense the "correct" behavior) is a buffer overflow that opens a shell. But this is precisely what SLR intends to defeat.

In order to apply SLR, we compared the binary with the original source code manually to determine which of the four layout heuristics could be used for each function.

The stack-based subset of the Wilander suite contains six functions each containing two exploits for a total of twelve stack-based exploits. From the manual analysis of the binary, we determined that five functions could be transformed with AOI, which correctly identified all variable boundaries.

For the sixth function, a buffer is randomly accessed using constant indices in the source code. As a result, the binary contains constant offsets that refer to part of the buffer, which AOI incorrectly identifies as variable boundaries. The most aggressive layout inference we were able to use that was not affected by the random access of array elements was SOI. The inference produced by SOI was very course grained, only identifying a boundary between the out-arguments region and the local-variables region.

The Wilander test suite was randomized three times using SLR, and all stack-based exploits were attempted for all three randomized versions. Any exploit which failed to achieve the goal of opening a shell was considered thwarted. In all three randomizations, SLR thwarted all but one of the twelve exploits; see the left side of Figure 51. The only exploit that managed to succeed was implemented in the function which was transformed using SOI. Analysis of the exploit indicated that the exploit could be thwarted by layout randomization alone, because the overflow targets another variable on the stack. However, SOI was not sufficiently fine-grained to find the boundaries necessary for successful randomization.

For the other eleven exploits, the run-time behavior was either a segmentation fault or a test suite exploit failure report. Since the layout inferences for each function were validated manually to identify variables correctly, a segmentation fault was considered evidence that SLR thwarted the attack.

The exploits that produced segmentation faults and those that produced exploit failure reports from the test suite were not consistent across the three randomizations. Some exploits require a certain layout of variables to succeed. If this layout is not found, such as if the target data is below the buffer, the attack is considered not possible and the exploit is aborted after printing a message indicating the attack was not possible. Since the layout of variables is random, in some variations the layout was susceptible to an attack while in others the attack was thwarted.

The cause of the segmentation fault for exploits not resulting in an exploit failure report was the random padding added by SLR. The padding caused the exploit to overflow a buffer not intended to be overflowed. The Wilander overflow exploits are very robust. At run-time, they calculate the distance from the buffer being overflowed to the target data. After calculating this distance, a block of characters equal in length to the distance are placed in a temporary buffer of static size. However, SLR adds a random amount of padding between variables, thus increasing the distance. The minimum padding amount between any two variables exceeds the statically allocated space for the temporary buffer, and, as a result, an overflow occurs triggering a segmentation fault. In most situations this overflow occurs in the glibc function memset.

While these results show both padding and randomization were effective in thwarting all but one exploit, had SLR been applied to memset, the added padding might have obfuscated the overflow that generated the segmentation fault, thus allowing the exploit to continue and possibly succeed. We altered the Wilander test suite so that the temporary buffer had as much space as necessary to prevent the segmentation fault overflow, and reran the experiment. The results are shown on the right side of Figure 51.

Because the test suite calculates distances necessary for overflow at runtime, and the previous cause of exploitation failure had been removed, the expected result was attack success for randomizations where the target data had not been moved so as to prevent the attack. Instead, SLR was successful in thwarting most attacks. All exploits rely on the assumption that local variables are placed on boundaries with addresses that are divisible by four. Stack-layout randomization adds a random padding between variables, and therefore four-byte alignment is not guaranteed. The buffer is still overflowed, but target data was not corrupted in the necessary way to execute the payload.

The Wilander suite was engineered to provide sophisticated exploits that are highly likely to succeed in overflowing a buffer. Nevertheless, SLR was able to thwart attacks, even when the code was altered to improve the probability of attack. Unexpectedly, the fact that the padding added between variables left boundaries on addresses that were not divisible by four defeated some attacks. For more common and likely less well engineered attacks, we would expect SLR to be equally successful.

4.10 C7: Twitcher Evaluation

Unfortunately, we did not have enough funds for a comprehensive review of Twitcher's security. We did demonstrate Twitcher's ability to prevent the Heartbleed attack against many clients of OpenSSL, including the popular web server NGINX [13]. We also were able to gather some performance numbers based on inserting and checking guards by hooking libc functions. Twitcher's raw performance are given in the following table:

Benchmarks	Base (s)	+ Twim (s)	+ RdWr Guards (s)	+Read Guards (s)	+Free Guards (s)
400.perlbench	523	580	612	632	696
401.bzip2	733	695	709	703	720
403.gcc	464	478	477	482	564
429.mcf	832	871	814	817	836
445.gobmk	649	640	635	648	643
456.hmmer	605	594	601	615	615
458.sjeng	747	743	759	744	747
462.libquantum	613	632	613	629	628
464.h264ref	880	912	1109	1111	1100
471.omnetpp	524	617	659	662	709
473.astar	697	645	660	668	675
483.xalancbmk	390	462	478	497	530
Average	**638**	**656**	**677**	**684**	**705**

The first column shows the name of the benchmark and the second shows the time to run the benchmark natively on a 64-bit, 2-core VM running on an Intel Core i7-4510U CPU at 2.6 GHz host. Columns 3–7 show the performance as additional features are cumulatively enabled, including the Twim memory manager, read/write guards, read guards, and free guards and a free quarantine. The following table summarizes the performance for each additional feature (cumulatively) as a percentage of the base performance:

Benchmark	+Twim	+RdWr	+Read	+Free
400.perlbench	111%	117%	121%	133%
401.bzip2	95%	97%	96%	98%
403.gcc	103%	103%	104%	121%
429.mcf	105%	98%	98%	100%
445.gobmk	99%	98%	100%	99%
456.hmmer	98%	99%	102%	102%
458.sjeng	100%	102%	100%	100%
462.libquantum	103%	100%	103%	102%
464.h264ref	104%	126%	126%	125%
471.omnetpp	118%	126%	126%	135%
473.astar	93%	95%	96%	97%
483.xalancbmk	119%	123%	128%	136%
Geometric Mean	**103%**	**106%**	**108%**	**111%**

The overall impact on performance is very low.

4.11 Instruction-Location Randomization

We implemented ILR as one of the defenses provided by PEASOUP. We demonstrated the effectiveness of ILR using a set of vulnerable programs (including a binary distributed by Adobe to read PDF files) and ASLR- and $W\oplus X$-defeating exploits. An important consideration of any mitigation technique is the run-time overhead. Many proposed mitigation techniques incur high overheads — as much as 90% to 2000% [134, 163]. Using a large industry standard CPU performance benchmark suite [199], we compared the run time of ILR-protected executables to that of native executables. The average run-time overhead of ILR was 13% with over half of all programs having effectively no overhead (less than 3%) indicating that ILR is a realistic and cost-effective mitigation technique.

4.11.1 Experimental Setup

We evaluated the effectiveness and performance of the ILR prototype using the SPEC CPU2006 benchmark suite [199]. These benchmarks are state-of-the-art, industry-standardized benchmarks designed to stress a system. The benchmarks are processor, memory and compiler stressing. The benchmarks are provided as source, and we compiled them with gcc, g++, or gfortran (as dictated by the program's source code) version 4.4.3 before applying our ILR technique. The benchmarks are compiled at optimization level -O2, and use static linking. We used static linking to thoroughly demonstrate the effectiveness of our system at randomizing large bodies of code, and to fully stress the system using all the odd, compiler-specific, language-specific, hand-coded, or otherwise abnormal code that is often found in libraries. Furthermore, having all the code packaged into one executable increases the attack surface making it easier to locate an ROP gadget. Thus, we believe our evaluation is a worst-case analysis for these benchmarks.

We run our experiments on a system with a 4-core, AMD Phenom II B55 processor, running at 3.2 GHz. The machine has 512KB of L1 cache, 2MB of L2 cache, 6MB of L3 cache, and 4GB of main memory.

Performance numbers are generated by running the benchmark 3 times. Unless otherwise noted, the performance of a protected binary is reported by normalizing its run time to the run time of the corresponding original binary produced by the compiler.

4.11.2 Security-Related Experiments

To verify that our technique stops attacks that are successful against ASLR and W$\xor$X protected systems, we performed a number of tests on vulnerable programs. For each test, ASLR and $W \oplus X$ were enabled.

In the first test, we used a small program (44 lines of code) that had a simple stack-based buffer overflow. The program assigns grades to students based on the program's input, the student's name. A malicious input can cause a buffer overflow enabling an attack.

We created a simple arc-injection attack which causes the program to print out a grade of B when the student should receive a D. It was trivial to perform the arc-injection. ASLR was ineffective because no randomized addresses were used—only the unrandomized addresses in the main program. Similarly, $W \oplus X$ was ineffective because the attack only relied on instructions that were already part of the program. We also used a tool called ROPgadget [10] to craft an ROP attack that causes the program to start a shell program which prints that the student's grade is an A.

Again, ASLR and $W \oplus X$ were ineffective. ILR, however, thwarted the attack.

We next verified our technique against a vulnerability in a realistic, dynamically linked program: Ubuntu's PDF viewer, `xpdf`. We seeded a vulnerability in the input processing routines. An appropriately long input can trigger a stack overflow. In this case, we were able to use ROPgadget to craft an attack to create a shell. ILR was again able to prevent the attack.

Lastly, we used a version (9.3.0) of Adobe's PDF viewer, `acroread`, that we downloaded from Adobe's website in binary form. The program has a well-documented vulnerability when parsing image files (see CVE-2006-3459) that allows arc-injection and ROP attacks [6]. Again, we used ROPgadget to craft an ROP attack payload for this vulnerability to start a shell program.

Because exploiting the vulnerability is more complicated, it took some additional effort to adapt the attack. Using information from Security Focus’s website, we were able to create a malicious PDF file that effected the ROP attack [6]. ILR successfully processes and randomizes the 24MB executable, and thwarts the attack.

Section 4.11.4 discusses ILR’s effect on the use of such tools as ROPgadget, and Section 4.11.6 describes how randomized addresses needed for the attack are protected from exfiltration by the ILR VM. Consequently, we believe attacks using programs such as ROPgadet are not possible when ILR is employed.

4.11.3 Effectiveness of ILR Components

We continue our evaluation of ILR by examining the effectiveness of the individual components that make up the offline ILR transformation.

4.11.3.1 Disassembly Engine

The goal of the Disassembly Engine is to locate any instruction which might be executed, so that the instruction can be relocated later. For our benchmarks, we found that the disassembly engine successfully located 100% of the executed instructions for all benchmarks. The Disassembly

Engine has met its first goal. We omit further discussion on disassembly as such techniques are well studied [25, 123, 180].

A secondary goal of the Disassembly Engine is to introduce few conflicting facts about instruction locations into the instruction database. We measured the fraction of bytes in the executable segments that belonged to more than one instruction. On average, only 0.005% of bytes were represented as part of more than one instruction with the worst-case having only 0.012% of bytes in conflict. Thus, we believe that the ILR disassembly engine has met its second goal.

4.11.3.2 Call Site Analysis

Figure 53 shows the percentage of call sites that were marked as safe to randomize their return addresses. The first bar shows that our technique works well for some benchmarks. `403.gcc` for example, has 91% of the return addresses randomized while `416.gamess` reaches 97%. Other benchmarks do not perform as well; `447.dealII` and `483.xalancbmk` only manage to identify 5% and 3% of return addresses as randomizable. The C++ benchmarks (`447.dealII`, `450.soplex`, `453.povray`, `470.lbm`, and `471.omnetpp`) do especially poorly. Only 10% of calls can use a randomized return address.

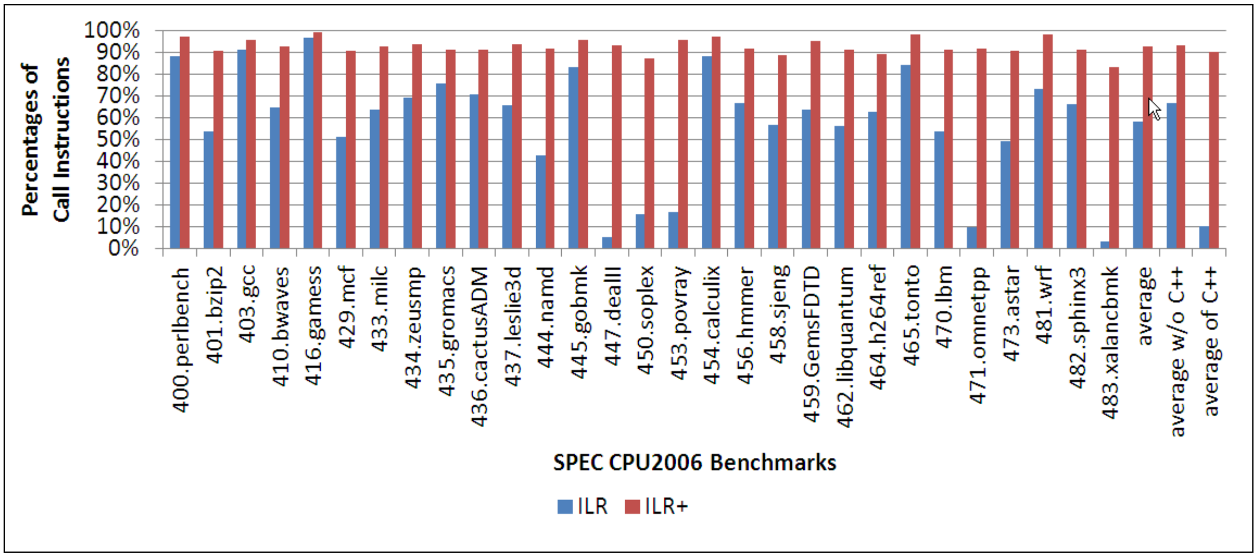


Figure 53. Percent of Call Instructions Given Randomized Return Address

To understand why the call site analysis phase was less effective on some benchmarks, we examined the reasons that the call site analysis indicated that a randomized return address could not be used.

Figure 54 shows the results as a fraction of all call instructions. We find that indirect calls (which cannot use a randomized return address because our analysis does not attempt to determine possible targets) result in a small fraction of unrandomized return addresses, resulting in 5% of calls on average. Possible non-standard uses of the return address, such as thunks, result in only 7.6% of return addresses. Interestingly, we find that direct call instructions to targets that we were not able to include in our disassembly result in 1.2% of the total call instructions. Closer inspection indicates that the compiler is actually emitting a `call 0x0` instruction in many library functions. If this type of call instruction were to ever execute, it would cause a fault in the program, but the call is (dynamically) unreachable code. The compiler cannot detect this fact, and so cannot eliminate the call. A minor improvement would randomize the return address for this type of call, knowing that the return address cannot be used if the call instruction causes a fault. Together, these causes represent only 21% of all unrandomized call instructions.

The top bar in the figure shows the real cause of the poor performance, especially in C++ programs. More than 32% of call instructions are marked as not being able to randomize the return address because of the exception handling tables used in the ELF file. In the C++ programs, this number jumps up to an average of 79%! In C++ programs, the compiler typically cannot calculate when a function, **f**, makes a call, whether the called function will throw an exception and need to clean up **f**'s stack. Consequently, the C++ compiler emits cleanup code into f, and adds to the `.eh_frame` and `.gcc_except_table` ELF sections to drive the exception handling routines. Since most functions with a call site fit this form, most call instructions cannot have a randomized return instruction.

It is interesting that even the C and Fortran benchmarks use the exception handling table. The C/Fortran benchmarks' application code does not seem to directly add to these tables. Instead, the table entries come from library routines that are compiled to work with C++ source.

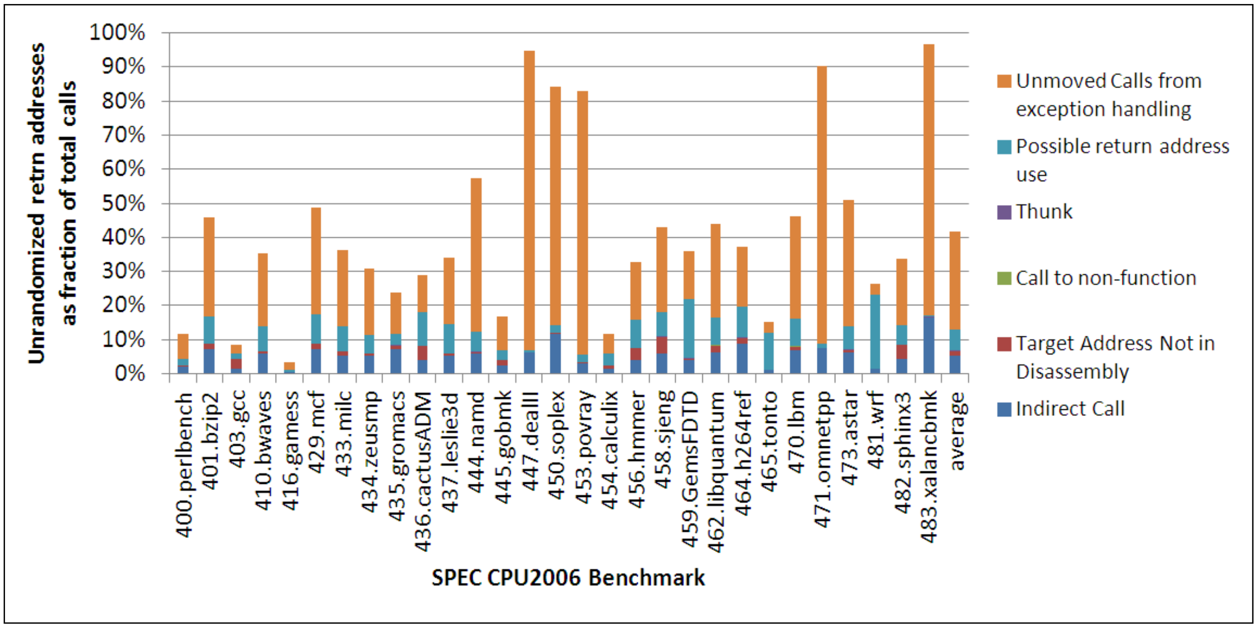


Figure 54. Breakdown of Call Instructions with Original Return Address

We believe that modifying the PEASOUP toolchain to edit the exception handling tables to reflect the randomization would be feasible. The tables are in a fixed, known format and can easily be rewritten with randomized addresses. Other solutions are possible as well. For example, detecting if C++ exception handling is actually used in the program or a portion of the program would allow return address randomization to be selectively applied. While fully exploring this idea is beyond the scope of this paper, we were able to modify our ILR toolchain to ignore the exception handling tables when calculating safe calls. We term the ILR toolchain with this modifications ILR+. ILR+ represents a very close approximation to a system that could easily be achieved by rewriting the exception handling tables in a binary.

With ILR+, the call site analysis performs well across all benchmarks. As Figure 53 shows, 93% of all calls are marked as using a randomized return address.

4.11.3.3 Indirect Branch Target Analysis

We continue our evaluation of ILR by measuring the effectiveness of the analysis of indirect branch targets (including return addresses). Figure 55 shows the fraction of instructions that were detected as possible indirect branch targets. On average, only 2.2% and 0.60% of the instructions are marked as indirect branch targets for ILR and ILR+, respectively. Consequently, we believe our simple scheme for detecting possible IBTs is not too aggressively marking instructions as possible indirect branch targets.

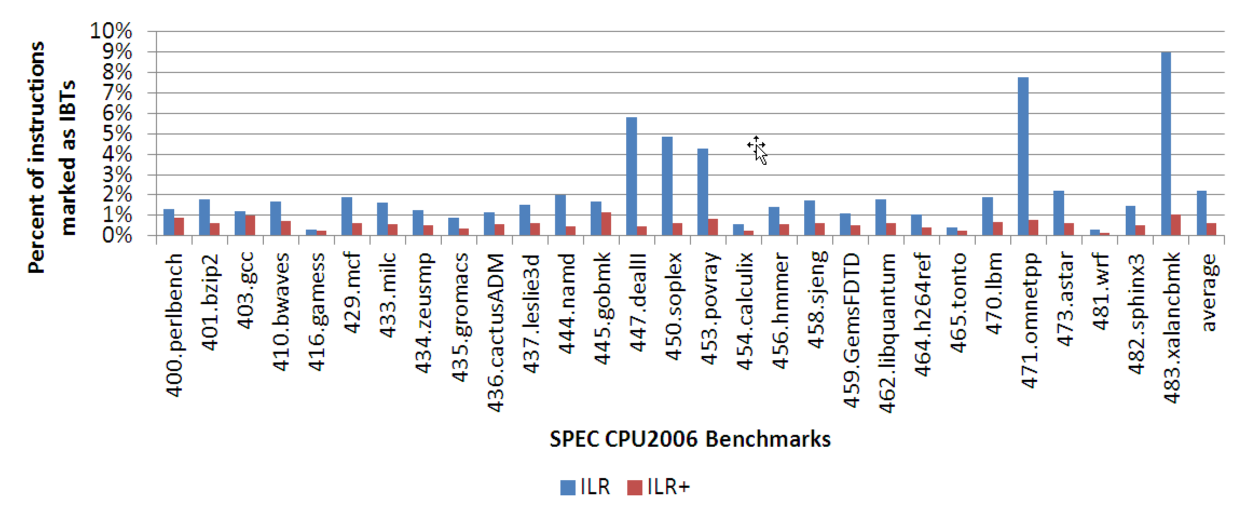


Figure 55. Percent of Instructions Marked as Possible Indirect Branch Targets

4.11.3.4 Moved Instructions

Since we emit rewrite rules for every byte of the executable segment, technically all instructions are moved. However, IBTs get a rule that maps the unrandomized address to the relocated instruction. Despite technically being moved, we consider this an unmoved (or pinned) instruction because if an attacker were to inject an arc or locate an ROP gadget at the unrandomized address, they could still exploit that information in the randomized program.

Figure 56 shows the percentage of instructions we can move for our benchmarks. The first bar in the figure shows effectiveness of ILR without the call site analysis; approximately

95.0% of instructions were successfully and safely located at a randomized address. The second bar shows call site analysis for standard ILR; 97.4% of instructions are moved. The last bar shows the results for ILR+, almost all instructions (99.1%) are assigned to a randomized location in memory. This randomization represents a two order of magnitude reduction in the attack surface for arc-injection and ROP attacks.

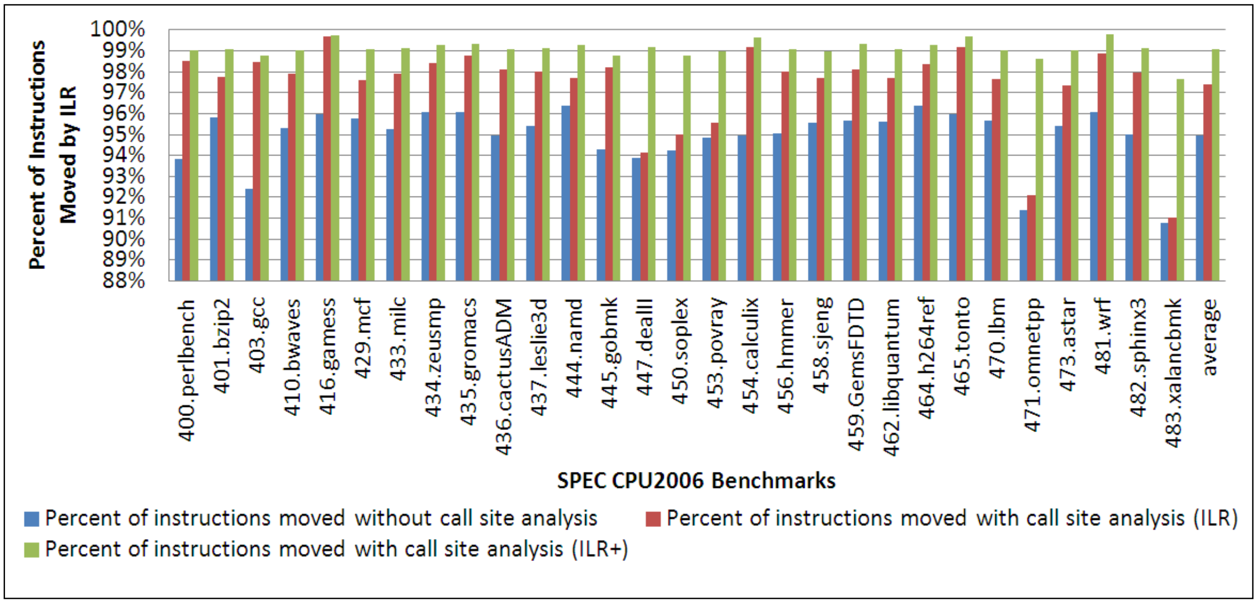


Figure 56. Percent of Instructions that were Moved Using ILR

4.11.4 ILR Security

To assess the security of ILR, we first note that up to 99.7% of the instructions can be randomized. Furthermore, all of the executable bytes of a program that do not make up a compiler-intended instruction sequence are marked as invalid for execution. These features of ILR reduce the attack surface for arc-injection by over two orders of magnitude. We believe it would be very difficult for an attacker to inject even one control-flow arc that achieves a meaningful result.

However, it has recently been shown that even small programs (with at least 20KB of program text) contain enough executable bytes to successfully produce an ROP attack [83]. The basic ILR algorithm reduces the unrandomized program text to less than 20KB for 26 of the 29 SPEC2006 benchmarks, while ILR+ reduces the attack surface to below 20KB for 28 of 29 benchmarks. On average, ILR+ reduces the attack surface to just 3KB! Thus, even state-of-the-art gadget compilers likely cannot detect enough gadgets to mount an ROP attack in a ILR+-protected program.

To more directly validate that ILR successfully randomizes enough gadget locations to make ROP attacks infeasible, we further examine the SPEC benchmarks. While we know of no vulnerabilities in these benchmarks, they, like all large pieces of software, may in fact have an error that might allow an ROP attack. We study the feasibility of such an attack on these large applications if an appropriate vulnerability were to be found or seeded.

To search for gadgets in these benchmarks, we use a tool available online, ROPgadget [10]. The tool contains a database of gadget patterns and can scan binary programs to identify specific gadgets within an executable. For example, one of the gadget patterns is `mov e?x, e?x;ret`, which identifies gadgets that move one register to another. We experiment with two versions of the tool, version 2.3 and 3.1. Version 2.3's database contains 60 gadget patterns, while version 3.1 has significantly more: 185 gadget patterns. Version 3.1 also contains a simple gadget compiler that matches gadgets with an attack template to form a complete attack payload. While these payloads do not automatically exploit a vulnerability in a program, they represent a significant portion of the attack. Converting this attack payload into an actual attack is dependent on the exact vulnerability, and is not automated. However, if ROPgadget cannot assemble the attack payload from the attack template, this failure indicates that the templated ROP attack could not proceed, even with a suitable vulnerability. ROPgadget 3.1 comes with two, simple attack templates.

For the experiment, we modified both versions of ROPgadget to ignore randomized addresses, so that the tool can only locate gadgets at the unrandomized code addresses. This modification mimics an attacker's abilities via a remote attack. Figure 57 shows how ILR affects an attacker's ability to mount an ROP attack. The first bar shows the percentage of unique gadgets that have been moved by ILR. We count unique gadgets because typically an attacker could re-use a gadget if needed, and any particular instance of a gadget is likely enough to mount an attack which used that gadget. Over 94% are moved on average, with `483.xalancbmk` being the worst performing at only 87%. The second bar shows the results for ROPgadget version 3.1. Even more of the gadgets appear to be hidden; over 90% in all cases, and 96% on average. What the figure does not show, however, is that version 3.1 located slightly more gadgets in the ILR-protected version, but found many more gadgets in the unprotected version, thus the overall ratio has improved, indicating that ILR is effective at hiding most gadgets in a program, even in the face of a better gadget identification framework. This result is quantified in the last bar of the figure where we count not unique gadgets, but all gadgets (including duplicates). On average, 99.96% of the total gadgets have had their location randomized.

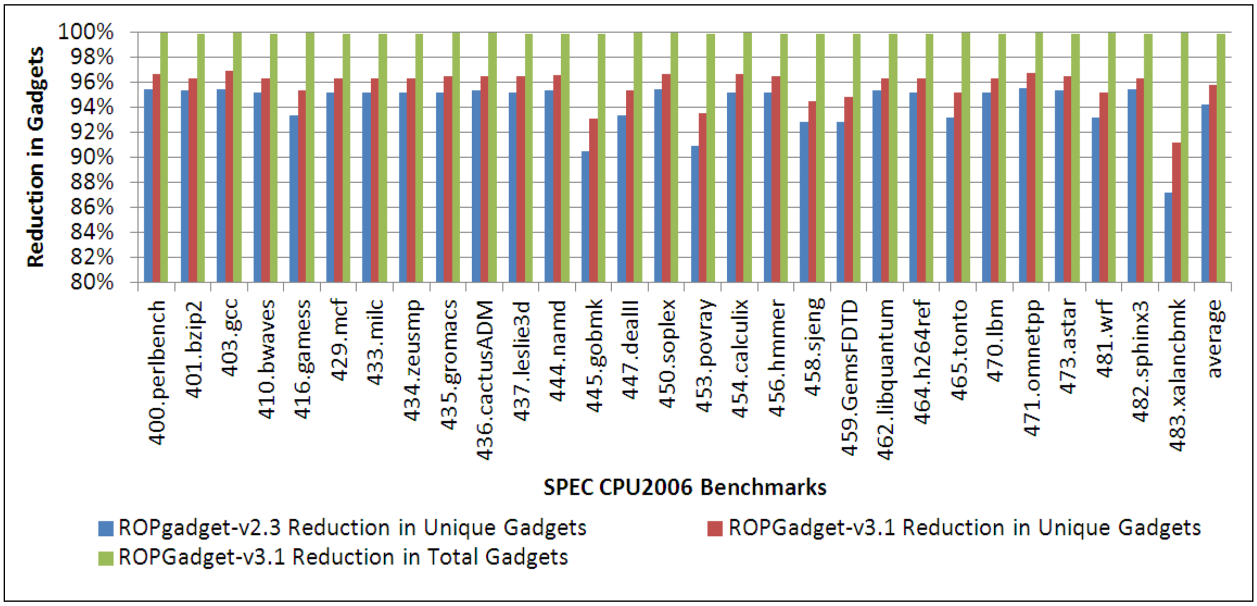


Figure 57. Reduction of Number of Gadgets Found After ILR

On average, only 2.48 gadgets remain in the protected program. The worst performing benchmark (`483.xalancbmk`) has 6 unique gadgets, versus 67 for the unprotected program. Six gadgets is not enough to mount an attack in most cases. Even the two very simple attack templates included with ROPgadget version 3.1 require 8 and 9 gadgets. We note that on an unprotected application, the gadget compiler can successfully generate an attack payload for every program. In fact, both attacks are automatically detected as possible on 9 of the benchmarks. On the protected program, no attack payloads are ever successfully generated.

With ILR+ (full results not shown) the probability of mounting an attack is further reduced. Most ILR+ protected applications have only one gadget (21 of 29 benchmarks). In every case, this lone gadget is an `int 0x80` sequence. Used alone, this gadget cannot successfully mount an attack. On average, only 1.5 gadgets remain available to mount an attack with ILR+.

4.11.5 Performance Metrics

4.11.5.1 Run-time Overhead

Figure 58 shows the performance overhead of the base VM (Strata), as well as the overhead of ILR and ILR+. We see that Strata adds much of the overhead for the applications, and applying the randomization costs little additional overhead. On average, Strata adds only 8% overhead, with an additional 8% used for ILR. This extra overhead occurs in the short-running, but large code size benchmarks, for example, `400.perlbench`, `403.gcc`, and `416.gamess`). The overhead added is mostly due to the startup overhead of reading the rewrite rules. In `481.wrf` benchmark, for example, we note that reading the rewrite rules takes about 45 seconds, and that the 7% overhead difference between basic virtualization and ILR also corresponds to about 46 seconds. We believe that this startup overhead could be greatly reduced by a better rewrite rule format than ASCII.

ILR+ actually reduces the overhead (by 3% to only 13%) compared to ILR. This reduction is due to more call sites being randomized. As described above, storing an unrandomized return address takes one extra instruction. With more return addresses randomized, the instruction count is reduced. Since ILR+ has the largest effects on the C++ benchmarks, we see this difference most in the C++ that are ILR+ compatible (`447.dealII`, `450.soplex`, and `483.xalancbmk`).

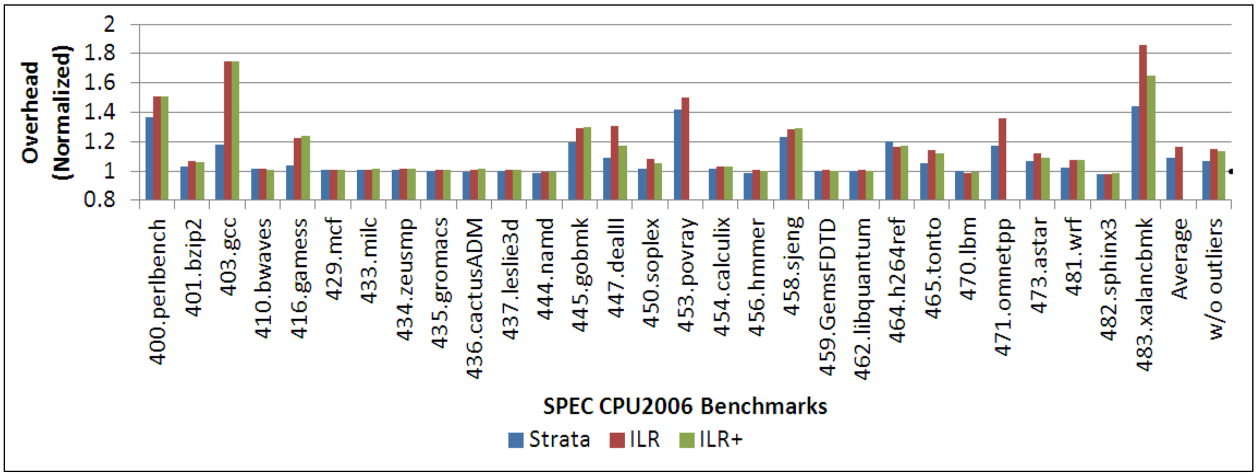


Figure 58. Performance Overhead of ILR and ILR+

Taken together, we believe there is strong evidence that ILR can be implemented efficiently, perhaps as low as the basic virtualization overhead of only 8%. Even our prototype implementation, which has overheads of 13%-16% overhead on average could be used to protect many applications.

4.11.5.2 Space Overhead

Our prototype implementation has memory overhead from two sources. The first is from the PVM we used to implement the ILR VM. Such overheads are well studied, and not particularly significant for modern systems [30, 108].

The second source of overhead is the handling of the ILR rewrite rules. In our prototype implementation, we made the design choice to use ASCII for the ILR rewrite rules. Our choice makes sense for an evaluation prototype: we favored human readability and ease of debugging over raw performance or storage efficiency. Consequently, we note that the on-disk size of the rewrite rules can be quite large. For example, the largest benchmark, `481.wrf`, has 264MB of rewrite rules. The in-memory size is even worse, 345MB. This overhead is largely due to our hashtable implementation that stores each byte of an instruction in a separate hash bucket, which allocates many words of data for each byte stored in an ILR rewrite rule. However, `481.wrf` is clearly a worst-case for our benchmarks. The average size of the rewrite rules (104MB) is less than half that for `481.wrf`.

While our prototype implementation is currently inefficient, we do not believe the size of the rewrite rules is an inherent limitation of ILR. Many techniques exist for minimizing this overhead. For example, we used the gzip compression utility to compress the rewrite rules, and obtained an average size of 14MB. We believe that a binary encoding of the rewrite rules and an efficient memory storage technique could easily reduce the memory used well smaller than 14MB. On today's systems with multiple gigabytes of main memory, such space overhead should be easily tolerated.

4.11.5.3 Analysis Time

We measured the analysis time of the ILR technique. We were able to process the SPEC benchmarks in an average of 23 minutes each. Only the last step of the process creates any randomization, so most of that processing time can be re-used if one wanted to re-randomize. The randomization step itself took only 36 seconds, indicating that re-randomization once analysis is complete could proceed very quickly.

4.11.6 Security Discussion

We expect the other protection mechanisms in PEASOUP, such as SIM, to protect the ILR runtime implementation. This section discusses other issues related to the security of ILR.

4.11.6.1 Entropy Exhausting Attacks

The entropy of the ILR technique can be quite high. Since the ILR technique separates data and instruction memory, randomized instructions can be located anywhere in memory, even at the same addresses as program data, VM code or data. Many operating systems reserve some pages of memory specifically for code to interface with the operating system, so those pages could not be used for randomized addresses. Further, any unrandomized instructions restrict the entropy of the remaining instructions. Since there are very few unmoved instructions, and almost all other addresses are available for randomization, we believe it would be easy to produce a system that has at least 31-bits of entropy on a 32-bit address system and at least 63-bits of entropy on a 64-bit system. Thus, randomly attempting to guess gadget addresses is completely infeasible and ILR can evade attacks which attempt to reduce the entropy of a system.

4.11.6.2 Information Leakage Attacks

A more likely attack scenario is that an attacker is able to leak information about randomized addresses. Fortunately, the memory-page protection techniques described previously prevent leaking of information about most randomized addresses. The only randomized addresses that might be leaked are those that potentially end up in the application’s visible data. For ILR, that is the randomized return addresses that might be stored in the application's stack. For a complete ILR+ implementation, it also includes any randomized addresses that are written into the application's exception handling tables.

In theory, all of these addresses might be leaked to an attacker. However, revisiting Figure 56, we see that on average only 5% of addresses in the total program could be known by the user. In practice, only a few randomized return addresses are available in the application at any instance, and most return addresses could not actually be leaked. If it were possible for the entire exception handling table to be leaked, the number of available addresses would likely be very close to the ILR results, and no ROP attacks are available against ILR in our benchmarks.

Furthermore, since our ILR technique is designed to be applied to arbitrary executables, re-randomization could occur regularly with little overhead. Regular re-randomization of high-entropy systems has been shown to be effective in the context of information leakage [28].

Thus, information leakage is not a problem for ILR.

4.11.6.3 False Detections

A false detection occurs when the program performs an operation that is detected as illegal, when there is no attack underway. On our benchmark suite, we found that there were no false detections with ILR. Since our implementation of ILR+ is incomplete, we did observe two false detections. Both `453.povray` and `471.omnetpp` resulted in incorrect output (from faulting the program) when attempting to throw an exception. A complete implementation of ILR+ would not demonstrate this problem. We believe this indicates that false detections would be rare in real programs.

Nonetheless, we discuss some possible mechanisms by which false detections might occur.

False detections might occur if a program were to calculate an indirect branch target, instead of simply storing the target in data memory as is most common. We found one example of this type of code in `gcc`'s library for doing arbitrary precision arithmetic. The example, shown in Figure 59 and originally written in assembly, is used to dispatch into a switch-style table of code blocks. Each block in the table is 9 bytes long. The assembly multiplies register `eax` by 9 (`eax+eax*8`), then adds the the base of the first code block before finally jumping to that address. A similar construct might be generated by a compiler, but we know of no compilers which generate this type of code for a switch statement.

Other constructs exist that might hide code addresses. For example, a function pointer might be calculated for some reason, such as for program obfuscation techniques.

```
lea  eax, [eax+eax*8+0x80b4545]
jmp  eax
```

Figure 59. Example of a Calculated Branch Target

A more common compiler construct that might calculate an indirect branch target is position independent code (PIC). In PIC mode, the compiler will often generate a code address by emitting a sequence of instructions that adds the current PC and a constant offset, knowing that the desired code address is a fixed distance from the current PC. PIC code is not standard due its performance overhead.

We believe that in most of these cases that a more advanced indirect branch analysis would solve the problem. For example, the code in Figure 59 is prefixed by code to verify that register `eax` is in proper bounds. A simple range analysis on the values that can reach the `jmp` instruction would reveal the possible indirect branch targets.

Furthermore, our experience indicates that the ILR technique can easily print the address of an indirect branch target if a false detection is encountered. A profile-based or feedback-based mechanism that incorporates newly discovered IBTs would be easy to implement to reduce false detections over time if the IBT can be detected as derived only from safe sources.

4.11.6.4 Shared Libraries

Modern computer systems are built using libraries that are loaded on demand, and possibly shared among many processes. Linux uses the `.so` (Shared Object) format, while Windows uses the `.dll` (Dynamically Linked Library) model. Our system is capable of processing and randomizing a program that uses dynamic linking. Generally, analysis of these types of programs is easier for our system. Since the code is divided into libraries, we know that if a library contains a constant, that the constant can only be an IBT in the library being considered, not to other libraries.

Thus, this separation dramatically reduces the number of potential IBTs for a library. Furthermore, externally visible functions and symbols need to be referenced by a handle that is given in the library's headers. Extracting these types of indirect branch targets is trivial.

While our prototype can process and effectively randomize programs that require shared libraries, it does not actually randomize the libraries. Both Linux and Windows support some form of ASLR which provides coarse-granularity randomization of shared libraries. We believe our technique could easily be extended to include full randomization of shared libraries, but it is not clear that doing so would always be the best solution. When feasible, it seems better to provide randomization within the library itself. On Linux, this randomization could be accomplished by using a randomizing compiler to generate a per-system version of the libraries. When library source code is not available, such as on Windows-based systems, ILR-based randomization would be important. To achieve this, ILR-rewrite rules for a library would have to be loaded and symbolic addresses resolved whenever a new library entered the system. Such a mechanism could be easily included in a dynamic loader, or by having the ILR VM watch for library loading events.

4.11.6.5 Self-modifying Code

Our ILR implementation does not currently support self-modifying, dynamically generated, or just-in-time compiled (JITted) code because our underlying VM does not support such constructs. However, the ILR mechanism itself should operate properly with dynamically generated and JITted code, which is significantly more common than self-modifying code. ILR would not randomize the generated code, but we believe that to be an easy task for the JITter. A security-minded JITter would perform this simple operation.

4.11.7 Conclusions

This summarizes the evaluation of instruction location randomization (ILR), a high-entropy technique for relocating instructions within an arbitrary binary. ILR is shown to effectively hide 99.96% of ROP gadgets from an attacker, a 3.5 order of magnitude reduction in attack surface.

We have described the general technique, as well as evaluates two versions of an ILR prototype. It further discusses the security implications of ILR. We find that ILR can be applied to a wide range of binary programs compiled from C, Fortran, and C++. Performance overhead is shown to be as low as 13% across the 29 SPEC CPU2006 industry-standard benchmarks [199].

ILR surpasses state-of-the-art techniques for defeating attacks in a variety of ways. In particular, the technique:

- can be easily and efficiently applied to binary programs,
- provides up to 31 bits of entropy for instruction locations on 32-bit systems,
- can regularly re-randomize a program to thwart entropy-exhausting or information-leakage attacks,
- provides low execution overhead,
- randomizes statically and dynamically linked programs, and
- demonstrated that it defeats attacks against large, real-world programs including the Linux PDF viewer, `xpdf`, and Adobe's closed-source PDF viewer, `acroread`.

Taken together, these results demonstrate that ILR can be used in a wide variety of real-world situations to provide strong protection against attacks and demonstrate the flexibility and power of the PEASOUP infrastructure.

4.12 SIM

This section describes the results of evaluation of early prototypes of SIM, which has been superseded by SDCG (Section 3.10.3).

4.12.1 Performance Test

To test the performance overhead imposed by SIM's protection, we conducted following performance evaluations. All evaluations are done using kernel-mode implementation and the test suite is SPEC CPU INT 2006.

Test Machine Configuration

- Host CPU: Intel Core i7 930
- Host memory: 6GB
- Host OS: Ubuntu 11.10 32bit
- Guest VM: KVM, 1 VCPU, 1.8G memory
- Guest OS: Ubuntu LTS 10.04 32bit

Test Setup

- **Baseline**: this test set the baseline for the rest test. It is done using a vanilla Ubuntu kernel, and without stratafying the test programs.
- **SIM kernel built without Stratafy**: this test demonstrates how much overhead is introduced by SIM's kernel modification
- **SIM kernel built with SIM turned off**: This test uses stratafied test programs but sets environment variable SIM_PROTECTION=0 to turn off SIM's protection. It sets another baseline for the next test. For optimization, we used STRATA_SIEVE=1

STRATA_RC=1 STRATA_PARTIAL_INLINING=0 and none optional protection mechanisms.

- **SIM kernel built with SIM turned on**: This test shows the performance penalty introduced by SIM's protection. For optimization, we used the same optimization option (STRATA_SIEVE=1 STRATA_RC=1 STRATA_PARTIAL_INLINING=0) and used no other protection mechanisms.

Test Result

Every test is done after a reboot and the iteration is set to 5. The result is shown in Table 7, the corresponding result is shown in Table 8.

Table 7 SIM Performance Test Results

		Baseline		**w/o SIM**		**SIM w/o RC**		**SIM w/ RC**	
BENCH	BASE REF	TIME	RATIO	TIME	RATIO	TIME	RATIO	TIME	RATIO
perlbench	9770	424	23.0	646	15.1	728	13.5	670	14.6
bzip2	9650	716	13.5	753	12.8	741	12.8	756	12.8
gcc	8050	392	20.6	501	16.1	622	12.7	570	14.1
mcf	9120	504	18.1	510	17.9	480	17.8	504	18.1
gobmk	10490	533	19.7	676	15.5	817	12.9	680	15.4
hmmer	9330	993	9.4	994	9.4	1030	9.3	991	9.4
sjeng	12100	638	19.0	850	14.2	969	12.3	860	14.1
libquantum	20720	672	30.8	674	30.8	647	31.2	670	30.9
h264ref	22130	878	25.2	1061	20.9	1157	19.2	1097	20.2
omnetpp	6250	404	15.5	494	12.7	520	11.7	506	12.4
astar	7020	619	11.3	665	10.6	643	10.5	668	10.5
xalancbmk	6900	313	22.0	564	12.2	655	10.3	555	12.4

Table 8 SIM Performance Overhead

	Baseline	w/o SIM	w/o RC	w/ RC	to w/o SIM	to w/o RC
perlbench	0.00%	52.36%	71.70%	58.02%	5.66%	-13.68%
bzip2	0.00%	5.17%	3.49%	5.59%	0.42%	2.09%
gcc	0.00%	27.81%	58.67%	45.41%	17.60%	-13.27%
mcf	0.00%	1.19%	-4.76%	0.00%	-1.19%	4.76%
gobmk	0.00%	26.83%	53.28%	27.58%	0.75%	-25.70%
hmmer	0.00%	0.10%	3.73%	-0.20%	-0.30%	-3.93%
sjeng	0.00%	33.23%	51.88%	34.80%	1.57%	-17.08%
libquantum	0.00%	0.30%	-3.72%	-0.30%	-0.60%	3.42%
h264ref	0.00%	20.84%	31.78%	24.94%	4.10%	-6.83%
omnetpp	0.00%	22.28%	28.71%	25.25%	2.97%	-3.47%
astar	0.00%	7.43%	3.88%	7.92%	0.48%	4.04%
xalancbmk	0.00%	80.19%	109.27%	77.32%	-2.88%	-31.95%
AVG	0.00%	23.14%	33.99%	25.53%	2.38%	-8.47%

As show in Table 8, the SIM kernel introduces an average overhead of 1.36%, which is a tolerable runtime deviation. For Stratafied binaries, with SIM off and many optimization options on, the average overhead is 23.14%. Note, this is considerably higher than the overhead we have observed for Strata in other situations, raising some questions about an error in this experimental setup. With SIM on, the average overhead is 25.53%. This means, the average overhead of imposed by SIM's protection is only 2.38%.

4.12.2 Compatibility with other Protection and Optimization Techniques

We have also tested the compatibility of SIM and other PEASOUP protection and optimization techniques. More specifically, we tested its compatibility with following protection techniques:

- Heap Randomization;
- Stack Randomization;
- PC confinement;
- Instruction Set Randomization

SIM works harmoniously with these mechanisms. We also tested SIM's compatibility with Strata's performance optimization techniques:

- Sieve optimization (STRATA_SIEVE)
- Return cache optimization (STRATA_RC)

SIM works fine with sieve optimization. But the original protection scheme breaks return cache optimization. Because return cache need to be writable under SOUP view but as it is part of Strata data, it will be mapped as inaccessible. To solve this incompatibility, we created a special memory region for return cache that will be kept writable under SOUP view.

4.12.3 Conclusion

In Phase 1, as part of our defense-in-depth strategy, we design and implemented a new technique called Secure In-process Monitoring, which protects the in-process monitor, Strata software dynamic translator. Our evaluation results showed, this protection is effective and efficient. Together with Strata's fine-grained confinement, they build up a robust sandbox environment for SOUP that can resist most kind of exploits.

4.13 Secure Dynamic Code Generation

In this section, we describe the evaluation of the effectiveness and performance overhead of our two prototype implementations of SDCG.

4.13.1 Security Analysis

In this section, we analyze the security of SDCG under our threat model. First, we show that our design can enforce permanent W⊕X policy. The first system call filtering policy ensures that attackers cannot map memory that is both writable and executable. The second policy ensures that attackers cannot switch memory from non-executable to executable. The combination of these two policies guarantees that no memory content can be mapped as both writable and executable, neither at the same time nor alternately. Next, the last policy ensures that if there is critical data that the SDT depends on, it cannot be modified by attackers. Finally, since the SDT is trusted and its data is protected, the second policy can further ensure that only SDT-verified content (e.g., code generated by the SDT) can be executable. As a result, SDCG can prevent any code injection attack.

4.13.2 Performance

4.13.2.1 Experimental Setup

For our port of the Strata DBT, we measured the performance overhead using SPEC CINT 2006 [199]. Our port of the V8 JS engine was based on revision 16619. And the benchmark we used to measure the performance overhead is the V8 Benchmark distributed with the source code (version 7) [22]. All experiments were run on a workstation with one Intel Core i7-3930K CPU (6-core, 12-thread) and 32GB memory. The operating system is the 64-bit Ubuntu 13.04 with kernel 3.8.035-generic.

Table 9. RPC Overhead During the Execution of the V8 Benchmark.

	Avg Call Latency	Avg Return Latency	# of Invocations	Stack Copy (%)	No Stack Copy (%)
Richards	4.70 µs	4.54 µs	1525	362 (23.74%)	1163 (76.26%)
DeltaBlue	4.28 µs	4.46 µs	2812	496 (17.64%)	2316 (82.36%)
Crypto	3.99 µs	4.28 µs	4596	609 (13.25%)	3987 (86.75%)
RayTrace	3.98 µs	4.00 µs	3534	715 (20.23%)	2819 (79.77%)
EarlyBoyer	3.87 µs	4.28 µs	5268	489 (9.28%)	4779 (90.72%)
RegExp	3.82 µs	5.06 µs	6000	193 (3.22%)	5807 (96.78%)
Splay	4.63 µs	5.04 µs	5337	1187 (22.24%)	5150 (77.76%)
NavierStokes	4.67 µs	4.82 µs	1635	251 (15.35%)	1384 (84.65%)

Table 10. Cache Coherency Overhead Under Different Scheduling Strategies.

	Schedule 1	Schedule 2	Schedule 3	Schedule 4	Schedule 5	Schedule 6
Richards	4.70 µs	13.76 µs	4.47 µs	14.25 µs	12.85 µs	13.37 µs
DeltaBlue	4.28 µs	13.29 µs	4.31 µs	13.85 µs	14.09 µs	15.84 µs
Crypto	3.99 µs	10.91 µs	3.98 µs	14.07 µs	12.47 µs	13.48 µs
RayTrace	3.98 µs	14.99 µs	4.05 µs	14.76 µs	13.15 µs	12.35 µs
EarlyBoyer	3.87 µs	13.70 µs	3.87 µs	14.27 µs	13.42 µs	13.47 µs
RegExp	3.82 µs	14.64 µs	3.85 µs	14.48 µs	13.55 µs	12.32 µs
Splay	4.63 µs	12.92 µs	4.49 µs	13.22 µs	13.36 µs	15.11 µs
NavierStokes	4.67 µs	12.06 µs	4.47 µs	13.02 µs	14.80 µs	12.65 µs

4.13.2.2 Effectiveness

In Section 4.13.1, we provided a security analysis of our system design, which shows that if implemented correctly, SDCG can prevent all code cache injection attacks. In this section, we evaluated our SDCG-ported V8 prototype to see if it can truly prevent the attack we demonstrated in Section 3.10.3.2.

The experiment was done using the same proof-of-concept code as described in Section 3.10.3.2. As the attack relies on race condition, we executed it for 100 times. For the version that is protected by naive W⊕X enforcement, the attack was able to inject shellcode into the code cache for 91 times. But for SDCG-ported version, all 100 tries failed.

4.13.2.3 Micro Benchmark

The overhead introduced by SDCG comes from two major sources: RPC invocation and cache coherency.

1) *RPC Overhead:* To measure the overhead for each RPC invocation, we inserted a new field in the request header to indicate when this request is sent. Upon receiving the request, the handler then calculates the time elapsed between this and the current time. Similarly, we also calculated the time elapsed between the sent and the receiving of return values. To eliminate the impact from cache synchronization, we pinned all threads (in both the untrusted process and the SDT process) to one single core.

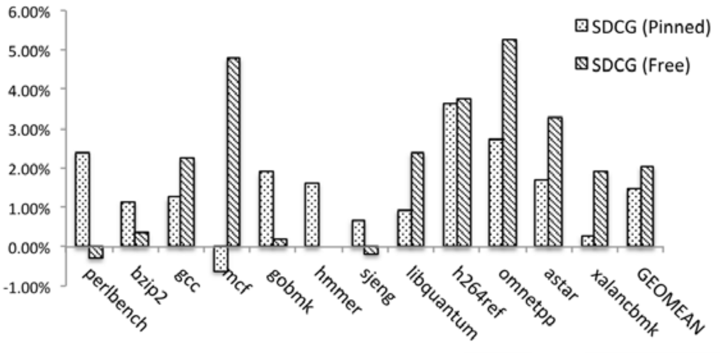


Figure 60. SPEC CINT 2006 Slowdown. The baseline is the vanilla Strata.

Table 11. SPEC CINT 2006 Results. Since the standard deviation is quite small (less than 1%), we omitted this information.

	NATIVE	STRATA	SDCG (PINNED)	SDCG (FREE)
PERLBENCH	364	559	574	558
BZIP2	580	600	613	602
GCC	310	403	420	410
MCF	438	450	479	471
GOBMK	483	610	623	611
HMMER	797	777	790	777
SJENG	576	768	784	767
LIBQUANTUM	460	463	511	474
H264REF	691	945	980	971
OMNETPP	343	410	450	428
ASTAR	514	546	587	563
XALANCBMK	262	499	515	504
GEOMEAN	**461**	**566**	**592**	**576**

Besides the overhead for each RPC invocation, another important fact that affects the overall overhead is the frequency of RPC invocation, i.e., the more frequent an RPC is invoked, the worse the performance is. So we also collected this number during the evaluation.

Table 9 shows the result from the V8 benchmark, using the 64bit release build. The average latency for call request is around 3-4 μs and the average latency for RPC return is around 4-5 μs. So the average latency for an RPC invocation through SDCG's communication channel is around 8-9 μs. The number of RPC invocations is between 1,525 and 6,000. Since the input is fixed, this number is stable, with a small fluctuation caused by garbage collection. Comparing with the overall overhead presented in the next section, it meets the expectation that the larger the number of invocations is, the higher the overhead is. Among all RPC invocations, less than 24% require stack copy.

2) Cache Coherency Overhead: SDCG involves at least three concurrently running threads: the main thread in the untrusted process, the trusted thread in the untrusted process, and the main

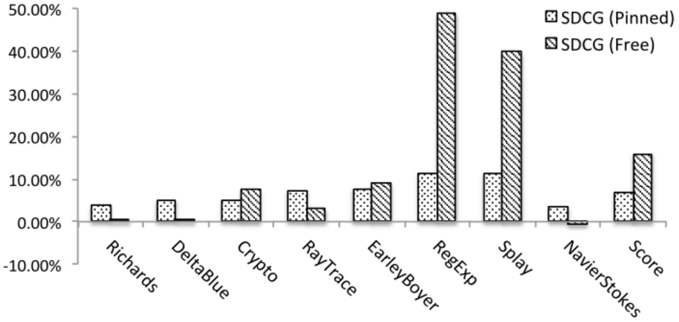


Figure 61. V8 Benchmark Slowdown (IA32)

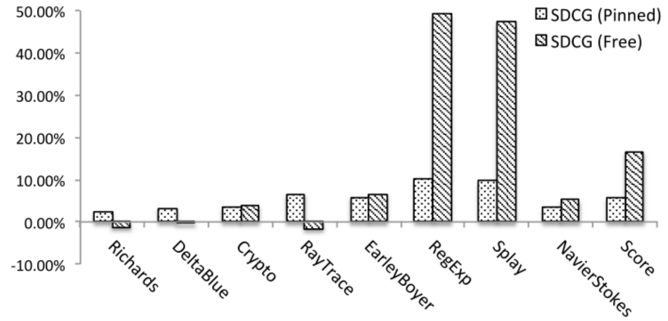


Figure 62. V8 Benchmark Slowndown (x64).

thread in the SDT process. This number can be more if the SDT to be protected already uses multiple threads. On a platform with multiple cores, all these threads can be scheduled to different cores. Since SDCG heavily depends on shared memory, the OS scheduling for these threads can also affect the performance, i.e., cache synchronization between threads executing on different cores introduces additional overhead.

In this section, we report this overhead at the RPC invocation level. In next section, we will present its impact on the overall performance. The evaluation also used V8 benchmark. To reduce the possible combination of scheduling, we disabled all other threads in V8, leaving only the aforementioned three threads. The Intel Core i7-3930K CPU on our test bed has 6 cores. Each core has a dedicated 32KB L1 data cache and 256KB integrated L2 cache. A 12MB L3 cache is shared among all cores. When Hyperthreading is enabled, each core can execute two concurrent threads.

Given the above configuration, we have tested the following scheduling:

1) All threads on single CPU thread (affinity mask = {0});
2) All threads on single core (affinity mask = {0,1});
3) Two main threads that frequently access shared memory on single CPU thread, trusted thread freely scheduled (affinity mask = {0},{*});
4) Two main threads on single core, trusted thread freely scheduled (affinity mask = {0,1},{*});
5) All three threads on different core (affinity mask = {0},{2},{4}); and
6) All three threads freely scheduled (affinity mask = {*},{*},{*}).

Table 12. V8 Benchmark Results (IA32). The score is the geometric mean over 10 executions of the benchmark suite. Number in the parentheses is the standard deviation.

	BASELINE	SDCG (PINNED)	SDCG (FREE)
RICHARDS	24913 (2.76%)	23990 (0.28%)	24803 (1.72%)
DELTABLUE	25657 (3.31%)	24373 (0.43%)	25543 (3.86%)
CRYPTO	20546 (1.61%)	19509 (1.27%)	19021 (1.95%)
RAYTRACE	45399 (0.38%)	42162 (0.75%)	43995 (6.46%)
EARLYBOYER	37711 (0.61%)	34805 (0.27%)	34284 (0.82%)
REGEXP	4802 (0.34%)	4251 (1.04%)	2451 (3.82%)
SPLAY	15391 (4.47%)	13643 (0.71%)	9259 (8.18%)
NAVIERSTOKES	23377 (4.15%)	22586 (0.42%)	23518 (1.26%)
SCORE	**21071 (0.72%)**	**19616 (0.35%)**	**17715 (1.86%)**

Table 13. V8 Benchmark Slowdown (x64). The score is the geometric mean over 10 executions of the benchmark suite. Number in the parentheses is the standard deviation..

	BASELINE	SDCG (PINNED)	SDCG (FREE)
RICHARDS	25178 (3.39%)	24587 (2.31%)	25500 (3.24%)
DELTABLUE	24324 (3.65%)	23542 (0.38%)	24385 (2.54%)
CRYPTO	21313 (3.16%)	20551 (0.26%)	20483 (2.57%)
RAYTRACE	35298 (5.97%)	32972 (1.03%)	35878 (1.66%)
EARLYBOYER	32264 (4.42%)	30382 (0.61%)	30135 (1.04%)
REGEXP	4853 (3.59%)	4366 (0.82%)	2456 (7.72%)
SPLAY	13957 (6.02%)	12601 (2.92%)	7332 (9.85%)
NAVIERSTOKES	22646 (2.48%)	21844 (0.30%)	21468 (3.45%)
SCORE	**19712 (3.57%)**	**18599 (0.62%)**	**16435 (1.03%)**

Table 10 shows the result, using 64bit release build. All the numbers are for RPC invocation, the return latency is omitted. Based on the result, it is clear that the scheduling has a great impact on the RPC latency. If the two main threads are not scheduled on the same CPU thread, the average latency can exacerbate to 3x-4x slower. On the other hand, the scheduling for trusted thread has little impact on the RPC latency. This is expected, because the trusted thread is only waken up for memory synchronization.

4.13.2.4 Macro Benchmark

In this section, we report the overall overhead SDCG introduces. Since OS scheduling can have a large impact on performance, for each benchmark suite, we evaluated two CPU schedules. The first one (*Pinned*) pins both main threads from the untrusted process and the SDT process to one single core; and the second one (*Free*) allows the OS to freely schedule all threads.

1) *SPEC CINT 2006:* Both the vanilla Strata and the SDCG-ported Strata are built as 32-bit. The SPEC CINT 2006 benchmark suite is also compiled as 32-bit. Since all benchmarks from the suite are single-threaded, the results of different scheduling strategies only reflect the overhead caused by SDCG.

Table 11 shows the evaluation result. The first column is the result of running natively. The second column is the result for Strata, but without SDCG. We use this as the baseline for calculating the slowdown introduced by SDCG. The third column is the result for SDCG with pinned schedule, and the last column is the result for SDCG with free schedule. Since the standard deviation is quite low (less than 1%), we omitted this information.

The corresponding slowdown is shown in Figure 60. For all benchmarks, the slowdown introduced by SDCG is less than 6%. And the overall (geometric mean) slowdown is 1.46% forpinned schedule, and 2.05% for free schedule.

Since SPEC CINT is a computation-oriented benchmark suite and Strata does a good job reducing the number of translator invocations, we did not observe a big difference

between pinned schedule and free schedule.

2) JavaScript Benchmarks: Our port of V8 JS engine was based on revision 16619. For better comparison with SFI-based solution [29], we performed the evaluation on both IA32 and x64 release builds. The arena-based heap we implemented was only enabled for SDCG-ported V8. And to reduce the possible combination of scheduling, we also disabled all other threads in V8.

Table 12 shows the results for IA32 build, and Table 13 shows the results for x64 build. The first column is the baseline result; the second column is the result of SDCG-ported V8 with

pinned schedule; and the last column is the result of SDCG-ported V8 with free schedule. All results are the geometric mean over 10 executions of the benchmark. The number in the parentheses is the standard deviation in percentage. As we can see, the fluctuation is small; with the baseline and free schedule slightly higher than pinned schedule.

The corresponding slowdown is shown in Figure 61 (for IA32 build) and Figure 62 (for x64 build). Overall, we did not observe a big different between IA32 build and x64 build. For four benchmarks (Richards, DeltaBlue Crypto, NavierStokes), the slowdown introduced by SDCG is less than 5%, which is negligible because they are similar to the standard deviation.

The rest of our benchmarks (RayTrace, EarlyBoyer, RegExp, Splay) have higher overhead, but with pinned schedule, the slowdown is within 11%, which is much smaller than previous SFI-based solution [29] (79% on IA32).

There are the two major overhead sources. For RPC overhead, we can see a clear trend that the more RPC invocation is (Table 9), the larger the slowdown is. However, the impact of cache coherency overhead caused by different scheduling strategies is not that consistent. For some benchmarks (Richards, DeltaBlu, RayTrace), the free scheduling is faster than the pinned scheduling. For some benchmarks (Crypto, EarlyBoyer), the overhead is almost the same. But for two benchmarks (RegExp and Splay), the overhead under free scheduling is much higher than the pinned scheduling. We believe this is because these two benchmarks depend more heavily on data (memory) access. Note that, unlike Strata, for SDCG-ported V8, we not only shared the code cache, but also shared the heaps used to store JS objects, for the ease of RPC implementation. Besides RPC frequency, this is another reason why we observed a higher overhead compared with SDCGported Strata.

4.13.3 Discussion

In this section, we discuss the limitations of our work on SDCG and potential future work.

4.13.3.1 Reliability of Race Conditions

Although we only showed the feasibility of the attack in one scenario, in practice, the dynamic translator can be invoked under different situations, each of which has its own race condition window. Some operations can be very quick (e.g., patching), others may take a longer time to finish. By carefully controlling how the translator is invoked, we can enlarge the race condition window and make such attack more reliable.

In addition, OS scheduling can also affect the size of the attack window. For example, as we have discussed in Section 3.10.3.2, the invocation of mprotect is very likely to cause the thread to be swapped out of the CPU, which will enlarge the attack window.

4.13.3.2 RPC Stub Generation

To port a dynamic translator to SDCG, our current solution is to manually rewrite the source code. Even though the modification is relatively small compared to the translator's code size, the process still requires the developer to have a good understanding of the internals of the translator. This process can be improved or even automated through program analysis. Firstly, our current RPC stub creation process is not sound. That is, we relied on the test input. Thus, if a function is not invoked during testing, or the given parameter does not trigger the function to modify the code cache, then we miss this function. Second, to reduce performance overhead and the attack surface, we want to create stubs only for functions that 1) are post-dominated by operations that modifies the code cache; and 2) dominate as many modification operations as possible. Currently, this is done empirically. Though program analysis, we could systematically and more precisely identify these "key" functions. Finally, for the ease of development, our prototype implementation uses shared memory to avoid deep copy of objects when performing RPC. While this strategy is convenient, it may introduce additional cache coherency overhead. With the help of program analysis, we could replace this strategy with object serialization, but only for data that is accessed during RPC.

4.13.3.3 Performance Tuning

In our current prototype implementations, the SDTs were not aware of our modification to their architectures. Since their optimization strategy may not be ideal for SDCG, it is possible to further reduce the overhead by making the SDT be aware of our modification. For example, one major source of SDCG's runtime overhead is RPC invocation, and the overhead can be reduced if we reduce the frequency of code cache modification. This can be accomplished in several ways. For instance, we can increase the threshold to trigger code optimization, use more aggressive speculative translation, and separate the garbage collection, etc.

Second, in our implementations, we used the domain socked-based IPC channel from seccomp-sandbox. This means, for each RPC invocation, we need to enter the kernel twice; and both the request/return data need to be copied to/from the kernel. While this approach is more secure (in the sense that a sent request cannot be maliciously modified), if the request is always untrusted, then using a faster communication channel (e.g., ring buffer) could further reduce the overhead.

Third, we also used the same service model as seccompsandbox in our prototypes. That is, RPC requests are served by a single thread in the SDT process. This strategy is sufficient for SDTs where different threads share the same code cache (e.g., Strata), because modifications need to be serialized anyway to prevent data race. However, this service model can become a bottleneck when the SDT uses different code cache for different thread (e.g., JS engines). For such SDTs, we need to create dedicated service threads in the SDT process to serve different threads in the untrusted process.

In addition, our current prototype implementations of SDCG are also not hardware-aware. Different processor can have different shared cache architecture and cache management capability, which in turn affects cache synchronization between different threads. Specifically, on a multi-processor system, two cores may or may not share the same cache. As we have demonstrated, if the translator thread and the execution thread are scheduled to two cores with different cache, then the performance is much worse than when they are scheduled to cores with the same cache. To further reduce the overhead, we can assign the processor affinity according to the hardware features.

4.14 Publications

The STONESOUP program focused on advancing fundamental research in software security. As such, one of the metrics of success is the rate of publication of results in peer-reviewed journals and conferences. The PEASOUP project has resulted in multiple publications in respected venues. The following publications can be fully or partially attributed to PEASOUP:

- Gopan, D., Melski, D., Nguyen, D., Naydich, D., and Driscoll, E., *Data-Delineation in Software Binaries and its Application to Buffer-Overrun Discovery.* In *ICSE 2015*. 2015. Firenze, Italy.
- Song, C., Zhang, C., Wang, T., Lee, W., and Melski, D., *Exploiting and protecting dynamic code generation.* In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*. 2015.
- Ghosh, S., Hiser, J. D., and Davidson, J. W., *What's the PointiSA?* In *2Nd ACM Workshop on Information Hiding and Multimedia Security*. 2014. Salzburg, Austria: ACM. pp. 23-34.
- Hiser, J. D., Nguyen-Tuong, A., Co, M., Rodes, B., Hall, M., Coleman, C., Knight, J. C., and Davidson, J. W., *A Framework for Creating Binary Rewriting Tools.* In *Proceedings of the 2014 Tenth European Dependable Computing Conference*. 2014. Washington, DC. pp. 142-145.
- Ghosh, S., Hiser, J., and Davidson, J. W., *Software Protection for Dynamically-generated Code.* In *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. 2013. Rome, Italy: ACM. p. 1:1-1:12.
- Ghosh, S., Hiser, J., and Davidson, J. W., *Replacement Attacks Against VM-protected Applications.* In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 2012. London, England, UK: ACM. pp. 203-214.
- Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J., *ILR: Where'd My Gadgets Go?* In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 2012. Washington, DC: IEEE Computer Society. pp. 571-585.
- Jang, Y., Chung, S., Payne, B. D., and Lee, W., *Gyrus: A Framework for User-Intent Monitoring of Text-Based Networked Applications.* In *21st Annual Network and Distributed System Security Symposium (NDSS)*. 2014. San Diego, CA.
- Lee, B., Lu, L., Wang, T., Kim, T., and Lee, W., *From Zygote to Morula: Fortifying Weakened ASLR on Android.* In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. 2014. Washington, DC: IEEE Computer Society. pp. 424-439.

- Nguyen-Tuong, A., Hiser, J. D., Co, M., Davidson, J. W., Knight, J. C., Kennedy, N., Melski, D., Ella, W., and Hyde, D., *To B or not to B: Blessing OS Commands with Software DNA Shotgun Sequencing.* In *Proceedings of the 2014 European Dependable Computing Conference (EDCC '14)*. 2014. Washington, DC: IEEE Computer Society. pp. 238-249.
- Rodes, B. and Knight, J., *Speculative Software Modification and its Use in Securing SOUP.* In *To Appear in Proceedings of the 2014 European Dependable Computing Conference (EDCC '14)*. 2014.
- Rodes, B. and Knight, J. C. *Reasoning about software security enhancements using security cases.* In *1st International Workshop on Argument for Agreement and Assurance*. 2013. Kanagawa, Japan.
- Rodes, B., Knight, J. C., and Wasson, K. S., *A Security Metric Based on Security Arguments.* In *Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics*. 2014. Hyderabad, India: ACM. pp. 66-72.
- Rodes, B., Nguyen-Tuong, A., Hiser, J., Knight, J., Co, M., and Davidson, J., *Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation.* In Runtime Verification 2013. Springer Berlin Heidelberg. pp. 308-313.
- Wang, T., Jang, Y., Chen, Y., Chung, S., Lau, B., and Lee, W., *On the Feasibility of Large-scale Infections of iOS Devices.* In *Proceedings of the 23rd USENIX conference on Security Symposium*. Berkeley, CA, USA: USENIX Association. pp. 79-93.
- Wang, T., Lu, K., Lu, L., Chung, S., and Lee, W., *Jekyll on iOS: When Benign Apps Become Evil.* In Proceedings of the 22Nd USENIX Conference on Security. 2013. Berkeley, CA: USENIX Association. pp. 559-572.
- Wang, T., Song, C., and Lee, W., *Diagnosis and Emergency Patch Generation for Integer Overflow Exploits.* In *11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2014)*. 2014. Egham, UK: Springer. pp. 255-275.
- Co, M., Davidson, J. W., Hiser, J. D., Knight, J. C., Nguyen-Tuong, A., Cok, D., Gopan, D., Melski, D., Lee, W., Song, C., Bracewell, T., Hyde, D., Mastropietro, B., *PEASOUP: Preventing Exploits Against Software Of Uncertain Provenance (Position Paper)*, 7th International Workshop on Software Engineering for Secure Systems, Waikiki, Hawaii, May, 2011.
- Jose A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. *Enabling Dynamic Binary Translation in Embedded Systems with Scratchpad Memory.* ACM Transactions on Embedded Computing Systems (TECS), 11(4), 89.
- Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. *Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection.* In Proceedings of The 2011 IEEE Symposium on Security and Privacy. Oakland, CA, May 2011.
- Shuaifu Dai, Tao Wei, Chao Zhang, Tielei Wang, Yu Ding, Zhenkai Liang, and Wei Zou. 2012. *A Framework to Eliminate Backdoors from Response Computable Authentication*. In Proceedings of The 2012 IEEE Symposium on Security and Privacy. San Francisco, CA, May 2012. (to appear)
- J. D. Hiser, D. W. Williams, W. Hu, J. W. Davidson, J. Mars and B. R. Childers. *Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems*. ACM Transactions on Architecture and Code Optimization, 8(2), July 2011, Article No. 9.

- T. Dey, W. Wang, J. W. Davidson, M. Soffa. *Characterizing Multi-threaded Applications based on Shared- Resource Contention*. Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software, Austin, TX, April 2011, pp. 76–86.
- R. Rajkumar, A. Wang, J. D. Hiser, A. Nguyen-Tuong, J. W. Davidson, and J. C. Knight. *Component-Oriented Monitoring of Binaries*. Proceedings of the 44th Hawaii International Conference on System Sciences, Kauai, HI, January 2011, pp. 1–10.
- David Evans, Anh Nguyen-Tuong, and John Knight. *Chapter in Moving Target Defense: An Asymmetric Approach to Cyber Security*, edited by Sushil Jajodia. Springer. 2011.
- John Knight. Diversity. *Festschrift in Honor of Brian Randell*. Lecture Notes in Computer Science 6875. Springer. 2011.

5.0 CONCLUSIONS

We consider the PEASOUP project to have been a great success, with many important results. The STONESOUP program set very high goals, both in performing fundamental research to advance the state of knowledge and in engineering a robust prototype for strenuous testing of hypothesis. On the latter objective, PEASOUP was not a complete success. In the final phase of the project, we were called upon to upgrade PEASOUP from a 32-bit to a 64-bit technology, process large binaries (4 times the size originally targeted in the solicitation), add defenses for two very subtle, difficult warning classes, and increase the level of defense provided by the four warning classes we handled in the previous phases. Trying to meet all of these goals proved to be too much, and PEASOUP's results on the Phase 3 T&E were apparently poor. However, we believe that the Phase 3 T&E performance was due to an engineering failure, which does not undermine the numerous scientific results that we obtained during the course of the project.

The PEASOUP effort resulted in advances in:

- Automated binary analysis.
- Techniques for building binary-hardening tools.
- Techniques for automatically protecting software binaries without the benefit of a specification.

In the remainder of this section, we discuss each of these points in greater detail. We close with some notes on the current status of the prototype and potential future research.

5.1 Advances in Automated Binary Analysis

5.1.1 Data Delineation Analysis

We developed a novel heuristic analysis that addresses the following question:

> *Given an arbitrary stripped executable, infer locations and sizes of objects suitable for buffer-overrun detection and protection.*

We call our approach *Data-Delineation Analysise* (DDA) (See Section 3.3.5). Accurate data delineation is critical for many analyses, not just buffer-overrun detection and protection, although that was a primary motivation for developing DDA in PEASOUP.

Our approach avoids the trap of choosing a "sound" technique that will infer objects that make buggy programs look correct. The inferred data layout information allows a number of existing techniques for ensuring memory safety to be applied effectively in the absence of source code.

Our evaluation of the analysis showed that it achieves good precision at low cost. Under separate funding, we integrated DDA in our commercial defect-detection tool, CodeSonar/x86. We have demonstrated that DDA significantly improves CodeSonar's buffer overrun detection.

In the future, we plan to further enhance DDA. The current approach focuses primarily on detecting the layout of top-level objects and ignores their internal structure. We believe that our approach can be extended in a straightforward fashion to infer information about the internal structure of program data, for example structure fields that are themselves structures. Once we have information about the internal structure, there is the potential for finding overruns of the subobjects itself. The CodeSonar back end has support for finding warnings of this type in source code, though doing the same thing for binaries would require non-trivial extensions even once the internal structure is known.

Another improvement to the analysis that we are considering is a more sophisticated tracking of offsets from base pointers. Currently, the analysis collects sets of constant offsets. Extending the tracking to support symbolic offsets will improve the precision of the analysis by allowing it to model naturally functions that are similar to memset and memcpy (e.g., program-specific wrappers of those functions).

Finally, most of our effort during the integration of DDA into CodeSonar/x86 (again, under separate fundiong) was focused on reducing the number of binary-only warnings (FP proxy) with little attention paid to the matched warnings (TP proxy) that we were dropping at the same time. We would like to return to the lost matched warnings and either make sure that they are being dropped for a good reason or determine what we can do to retain them without undermining the reduction in FPs.

5.1.2 Speculative Transformation

Another approach we developed for analysizing and protecting software binaries is called *speculative transformation.* This technique requires a robust test suite. The technique uses the following steps:

1. An initial, possibly incorrect IR is recovered from the binary (e.g., using DDA).
2. Multiple versions of the program are created, using different subsets of the IR.
3. Each version of the IR is run against the test suite.
4. Based on which versions pass and which fail, the IR is refined so that only facts used in passing versions are trusted.

As an example, one focus of PEASOUP was in identifying IR for use in Stack-Layout Transformation (SLX). The IR needed for SLX includes (a) identifying the boundaries of stack-allocated data objects and (b) identifying the instructions that reference each stack-allocated data object that is relocated or padded. This serves as a good challenge problem in IR recovery, even if the ultimate goal is a confinement-based defense, instead of a diversification-based defense (like SLX). Existing approaches (that we are aware of) to this IR recovery challenge are either unsound, imprecise, or suffer from poor scalability. In this case an unsound IR identifies object boundaries where there are none or misidentifies the instructions that access stack-allocated objects. In either case, a naïve application of a transformation based on an IR with these errors is likely to break the program. On the other hand, if the IR is overly coarse, than many functions may not be transformed at all and will remain unprotected.

We demonstrated the utility of BED and TSET in the context of SLX and its associated IR recovery challenges. PEASOUP implements SLX as follows:

- It uses an unsound analysis, such as DDA, to identify candidate IR facts, including possible boundaries between stack-allocated objects and associations between instructions and the stack objects they access.
- It uses automatic test-case generation to generate a high-coverage test suite; alternatively, a user may supply a high-coverage test suite.
- It uses run-time error detectors to classify each generated input as ‘good’ or ‘bad’ depending on whether or not the input triggers a detectable run-time error.
- TSET, the Test-Set Evaluation Technology, evaluates the quality of the set of tests that have been labeled ‘good’ for every function in the subject program. Currently, TSET uses simple coverage metrics for evaluation: if the ‘good’ tests achieve poor coverage for a

function, then PEASOUP will only apply conservative transformations to that function. On the other hand, if TSET determines that there is good coverage for a function, then PEASOUP will apply aggressive SLX to generate many variants of the function with significant differences in the relative locations and sizes of the stack objects.

- BED, the Behavioral Equivalence Detector, evaluates each variant function to determine whether or not it exhibits equivalent behavior to the original program. Variants that were based on faulty IR are likely to cause changes in behavior, and are therefore discarded. On the other hand, variants based on correct IR are likely to have equivalent behavior to the original.

Our experimental evaluation of SLX demonstrated the utility of this: PEASOUP succeeded in applying SLX to many functions that could not have been transformed if a more conservative IR had been used. The testing done by BED and TSET allows PEASOUP to use an aggressive, unsound analysis for IR recovery by compensating for any errors that result from the faulty IR prior to deployment of program variants. Furthermore, behavioral differences detected by BED can be used to provide feedback about IR facts that are unlikely to be true, leading to an improved IR.

While we demonstrated the utility of BED and TSET in the context of SLX (and, to a lesser extent with other transformations), we believe that the approach has some weaknesses. In particular, we require a high-quality test suite of *known* benign inputs. Our original idea was to automatically generate a high-quality test suite. However, the challenge is that when BED reports a difference in behavior, it is difficult to know whether the difference is due to (a) use of bad IR or (b) an error in the program that becomes apparent due to the transformation.

5.1.3 Limitations of Automated Test-Case Generation

One of the hypotheses of PEASOUP was that we could achieve precise IR recovery by using automatically generated tests to drive dynamic analyses. We have concluded that this approach has limited applicability. There are two challenges: the first is that it is difficult to automatically classify generated test inputs as begnign or harmful, that is, whether or not an input exercises a vulnerability. Without this knowledge, it is often difficult to derive precise information from the dynamic analyses, although even imprecise information learned this way can be useful [160].

The second limitation appears to be scalability. While we do not have precise numbers, we believe that high-coverage test suites for moderate sized programs are likely to be very large, with tens of thousands of inputs, at least. This makes it expensive to run all of the inputs while using a dynamic monitoring tool. One possible alternative is to use

5.2 Advances in Techniques for Building Binary-Hardening Tools

5.2.1 Secure Dynamic Code Generation (SDCG)

PEASOUP demonstrates that software dynamic translation is a useful technology for implementation software defenses. However, we also demonstrated that a code-cache injection attack is a viable exploit technique that can bypass many of the state-of-art defense mechanisms. To defeat this threat, we proposed SDCG, a new architecture that enforces mandatory W⊕X policy. To demonstrate the feasibility and benefit of SDCG, we ported two software dynamic translators, Google V8 and Strata, to this new architecture. Our development experience showed that SDCG is easy to adopt and our performance evaluation showed the performance overhead is small. We believe that SDCG is of foundational importance in building software protection technology.

5.2.2 Robust, Extensible Architecture

It is arguable that the most important results of PEASOUP are not the individual defensive innovations, but rather the development of infrastructure that will enable future research and development. PEASOUP was deliberately designed to support extensibility, including many innovative techniques that simplify extensibility:

- *Well-defined, relational schema for the IRDB.* The importance of picking the correct format for an IR cannot be overstated. The design of the IR affects performance and extensibility. In our experience developing our own IRs on previous projects and using the IRs of other tools, we have encountered many design decisions that frustrate tool development. Our choice of relational tables has two significant advantages: first, most of the data that we need to store in the IR is naturally expressed as a table. In contrast, other formats, such as XML, are best suited for ASTs, a small portion of the IR that can typically be generated on demand, rather than stored. Second, the relational tables can be documented using a standard relational schema. This allows any component writer to easily understand the interface to the IR.
- *Scalable, robust technology for storing the IRDB.* PEASOUP uses Postgres to store the IRDB. This allows PEASOUP to leverage the concurrency, scalability, and robustness of a mature database technology. In particular, PEASOUP can use the concurrency of the IRDB to run multiple analyses in parallel. We believe this will be important in achieving scalability in later phases of PEASOUP. Furthermore, the use of an industrial database for storing the IRDB enables a mode where analysis of a subject program could be run on a central, dedicated machine, with the results of the analysis available to any machine on the network.
- *Well-defined interfaces between components*. Each component of PEASOUP is designed so that it can be augmented or replaced by other technology as it becomes available. For example, Grace represents generated inputs using JSON and stores the generated inputs in the IRDB. Any other test-case generation technology could use the same format and provide additional inputs for use in Grace. As a second example, PEASOUP is prepared to make use of any run-time error detection tool that can be used with the replayer.
- *Flexible, automatable rewriting techniques*. One of the innovations in Phase 1 was the development of *sprockets* for representing the rewrites necessary to transform a program into a variant. There are many different approaches to executable rewriting, including other techniques that leverage software dynamic translation. However, we many of these frameworks are targeted at supporting manual generation of rewriting tools and are agnostic about the IR that is used to drive the rewriting. In contrast, sprockets are very convenient for representing rewriting algorithms that are based on the PEASOUP IRDB. One indication of the utility of sprockets is that they were useful for inventing ILR.

Together, we believe these design decisions will enable additional breakthroughs in software security in future work.

5.3 Advances in Automatic Exploit Prevention and Software Repair

We believe that PEASOUP's most lasting impact will come from its innovations in program analysis and transformation, as outlined in the previous sections. Unfortunately, it is difficult to provide a quantitative assessment of the importance of an IR recovery technique or the flexibility and robustness of a machine-code rewriting technique. In the case of PEASOUP, our results also advanced the state-of-the-art in automated prevention of software exploits, sometimes in ways that are quantifiable. These advances are important in their own right, but also in that they provide indirect evidence of the effectiveness, flexibility, and robustness of the architecture and the techniques used in PEASOUP.

PEASOUP advanced the state-of-the-art in automatic exploit prevention and program repair in the following ways:

- *Dramatically increases the entropy of software diversification*. As a defensive technique, diversification has significant drawbacks: it cannot guarantee exploit prevention, specific diversification techniques can often be sidestepped by cleverly crafted attacks, and its response to exploits is failure oblivious. However, diversification is extremely successful because it is easy to deploy and has very low overhead[20]. Using the ILR technique, PEASOUP demonstrated the ability to increase the entropy of code-layout diversification by 3.5 orders of magnitude beyond what is provided by systems today. On a 32-bit machine, ILR can relocate 99.7% of instructions to any on 2^{31} addresses, leaving very little room for improvement. ILR does this while maintaining similar efficiency and usability to diversification techniques that are widely employed.
- *Frustrates exploits based on arc-injection*. There are existing techniques that are theoretically effective against arc-injection attacks, including ASLR. Unfortunately, practical limitations (e.g., the need for backwards compatibility) mean that existing implementations of these defenses can usually be circumvented. Advanced attack techniques based on Return-Oriented Programming (ROP) allow attackers to perform arbitrary actions. PEASOUP solves these issues with ILR. Our evaluation demonstrates that ILR is as effective at preventing arc-injection as $W \oplus X$ is at preventing code injection. Furthermore, ILR does not suffer the same practical limitations that mar implementations of ASLR.
- *Enables many source-level defenses to be applied to directly to software binaries.* Source code provides a wealth of information—types, function entry points, code versus data, layout of data objects—that is not readily available in a software binary. For this reason, many techniques that are easy to implement in a compiler have been impractical to implement on machine code. This includes many techniques that modify the construction of a functions activation records, including: inserting stack canaries, laying out input buffers so they are isolated from other data, or randomizing the layouts of data objects. PEASOUP advanced the state-of-the-art in IR recovery sufficiently that transformations such as these can be applied effectively to binaries (that do not already have these protections).

[20] It is arguable that software diversification has stopped more attacks than otherwise superior techniques, such as control-flow integrity [24] or SELinux [130].

- *Provides a composable, layered defense.* No single defensive technique is suitable for preventing all types of exploits against all types of vulnerabilities. PEASOUP enables multi-faceted defenses from within a single framework. Furthermore, BED and TSET provide additional confidence that independent defenses do not interfere with one another, nor interact in a way that breaks program semantics. The list of defenses provided by PEASOUP includes: Secure In-VM Monitoring, program-counter confinement, incorrect heap-usage confinement (e.g., double frees), instruction layout randomization, stack-layout randomization, heap randomization, and instruction set randomization.
- *Automatic repair of vulnerabilities.* PEASOUP is also capable of repairing faulty programs. Most often, this happens when PEASOUP adds padding to the size of data object, effectively removing a buffer-overrun vulnerability. This was demonstrated during the independent test and evaluation when PEASOUP was able to correct the behavior of bzip2 and ngircd (2 of the 3 real-world test cases). Malicious inputs that previously crashed these programs no longer had any negative effect, and the programs appeared to behave as they should for a malformed input. In addition to increasing buffer sizes, PEASOUP is capable of implementing many other repair policies, including saturating arithmetic and terminating infinite loops.
- *Software DNA Shotgun Sequencing (S^3).* We developed S^3 a new, efficient, approach for detecting taint markings based on positive taint inference. Our findings indicate that S^3 can be effectively used to detect OS command injection attacks on binary programs. Furthermore, S^3 has demonstrated that it can be used in many real-world situations because it has negligible performance overhead and can be applied directly to binary programs without need for source code or compiler support. Under separate funding UVA expanded S3 to cover SQL injections and demonstrated its utility in protecting web applications.
- *Twitcher memory protections*. We developed a novel memory protection system called Twitcher. Twitcher is based on defining different classes of guard regions, each with different memory access permissions. Using these extended definitions of guard regions, Twitcher is able to defend against the infamous Heartbleed exploit, in many situations [13]. To our knowledge, Twitcher is the only tool capable of automatically defending against Heartbleed in a generic fashion, under any circumstances.
- *Number-handling defesnes*. We developed a novel technique for detecting number-handling errors based on recognizing benign number-handling weaknesses. The technique is effective for medium sized binaries, such as those used in Phase 2 T&E. The overhead of this technique is high, so it is most effective for error amplification.
- *Other defenses*. Finally, we developed a host of other promising defenses against exploits null-pointer dereferences, concurrency errors, and resource drains.

We believe that each of these advances is significant individually, but are particularly important when taken as a whole.

5.4 Transition and Future Work

The STONESOUP program required a high degree of technical readiness in order to pass the independent test and evaluation. Consequently, many components of PEASOUP are ready for immediate transition. Unfortunately, there is no single Transition Readiness Level (TRL) for

PEASOUP as a whole. The TRL varies depending on which features are needed, and the application. Some components and defenses are more mature than others, and some applications have more stringent requirements than others.

S^3 and Twitcher are among PEASOUP's most mature features. They can both be used for automatic hardening of binaries with very little fear of altered functionality and very low overhead. The number handling defenses still impose some risk of altered functionality on large binaries and they incur a heavy runtime overhead. Consequently, it is more appropriate for error amplification (see Section 2.2). The null-pointer defenses may be effective for some server applications. The defenses against concurrency errors and resource drains are the least mature. They also could be used for error amplification and input classification, but further development is likely necessary to make them more robust.

We intend to continue research and development of PEASOUP. Our near-term focus will be on applying some of PEASOUP's defenses using static binary rewriting, in order to eliminate the overhead of software dynamic translation. We are also interested in seeking transition opportunities based on automatic hardening, error amplification, or automatic input classification.

6.0 References

1. CVE-2003-0041: MIT kerberos FTP client remote shell commands execution, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0041.
2. PaX Team. PaX address space layout randomization (ASLR), http://pax.grsecurity.net/docs/aslr.txt.
3. *MISRA-C 2004 Guidelines for the Use of the C Language in Critical Systems*, 2004, The Motor Industry Software Reliability Association.
4. CVE 2008-2575: crbPager: Arbitrary command execution via shell metacharacters, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2575.
5. Design of Chrome v8, https://developers.google.com/v8/design.
6. LibTIFF TiffFetchShortPair Remote Buffer Overflow Vulnerability, http://www.securityfocus.com/bid/19283.
7. CVE-2010-1132: SpamAssasin mail filter:Arbitrary shell command injection, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1132.
8. Inside Adobe Reader Protected Mode -Part 3- Broker Process, Policies, and INter-Process Communication, http://blogs.adobe.com/security/2010/11/inside-adobe-reader-protected-mode-part-3-broker-process-policies-and-inter-process-communication.html.
9. IDA Pro, http://www.hex-rays.com/products/ida/index.shtml.
10. ROPgadget, http://shell-storm.org/project/ROPgadget/.
11. Web Workers: W3C Candidate Recommendation 01 May 2012, http://www.w3.org/TR/workers/.
12. CVE-2013-3568: Linksys CSRF + root command injection, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-3568.
13. The Heartbleed Bug, http://heartbleed.com/.
14. App capability declarations (Windows Runtime apps), https://msdn.microsoft.com/en-us/library/windows/apps/hh464936.aspx.
15. CVE vulnerabilities found in browsers, http://web.nvd.nist.gov/view/vuln/search-results?query=browser&search_type=all&cves=on.
16. CWE-416: Use After Free, http://cwe.mitre.org/data/definitions/416.html.
17. CWE-680: Integer Overflow to Buffer Overflow, http://cwe.mitre.org/data/definitions/680.html.
18. Ruby, www.ruby-lang.org.
19. seccompsanbox, https://code.google.com/p/seccompsandbox/wiki/overview.
20. The Chromium Projects: Sandbox Design Principles, http://dev.chromium.org/developers/design-documents/sandbox.
21. The perl programming language, www.perl.org.
22. V8 Benchmark Suite - Version 7, https://www.rstforums.com/forum/80945-mobile-pwn2own-autumn-2013-chrome-android-exploit-writeup.rst.
23. Wordpress, www.wordpress.org.
24. Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J., *Control-flow Integrity: Principles, implementations, and applications.* In *ACM Conference on Computer and Communications Security (CCS)*. 2005.
25. Aditya Kapoor, *An approach towards disassembly of malicious binary executables*. *University of Louisiana*. 2004.
26. Ammann, P. E. and Knight, J. C., *Data Diversity: An Approach to Software Fault Tolerance*. IEEE Transactions on Computers, 1988. **37**(4): pp. 418-425.

27. Andersen, S. and Abella, V., Part 3: Memory Protection Technologies, Data Execution Prevention, https://technet.microsoft.com/en-us/library/bb457155.aspx.
28. Anh Nguyen-Tuong, Andrew Wang, Jason D.Hiser, John C.Knight, and Jack W.Davidson, *On the effectiveness of the metamorphic shield.* In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. 2010. pp. 170-174.
29. Ansel, J., Marchenko, P., Erlingsson, U., Taylor, E., Chen, B., Schuff, D. L., Sehr, D., Biffle, C. L., and Yee, B., *Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code.* In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011. San Jose, CA: ACM. pp. 355-366.
30. Apala Guha, Kim Hazelwood, and Mary Lou Soffa, *Reducing exit stub memory consumption in code caches.* In *Proceedings of the 2nd international conference on High performance embedded architectures and compilers*. 2007. pp. 87-101.
31. Arjan van de Ven, *New Security Enhancements in Red Hat Enterprise Linux v.3, update 3.* In . 2004.
32. Armour-Brown, C., Fitzhardinge, J., Hughes, T., Nethercote, N., Mackerras, P., Mueller, D., Seward, J., Van Assche, B., Walsh, R., and Weidendorfer, J., Valgrind Home, http://valgrind.org/.
33. Austin, T. M., Breach, S. E., and Sohi, G. S., *Efficient Detection of All Pointer and Array Access Errors (extended version).* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1994. Orlando, FL: University of Wisconsin. pp. 290-301.
34. Aycock, J., *A Brief History of Just-in-time*. ACM Computing Surveys (CSUR), 2003. **35**(2): pp. 97-113.
35. Babic, D., Martignoni, L., McCamant, S., and Song, D. X., *Statically-Directed Dynamic Automated Test Generation.* In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2011.
36. Balakrishnan, G. and Reps, T., *DIVINE: DIscovering Variables IN Executables.* In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2007. pp. 1-28.
37. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S. H., Chen, C.-H., and Teitelbaum, T., *Model Checking x86 Executables with CodeSurfer/x86 and WPDS++, (tool-demonstration paper).* In *International Conference on Computer Aided Verification (CAV)*. 2005. Edinburgh, Scotland: Springer. pp. 158-163.
38. Balakrishnan, G., Reps, T., Melski, D., and Teitelbaum, T., *WYSINWYX: What You See Is Not What You eXecute.* In *IFIP Working Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*. 2005. Zurich, Switzerland: Springer.
39. Barrantes, G., Ackley, D. H., Palmer, T. S., Zovi, D. D., Forrest, S., and Stefanovic, D., *Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks.* In *ACM Conference on Computer and Communications Security (CCS)*. 2003. Washington, DC: ACM. pp. 281-289.
40. Beckman, N. E., Nori, A. V., Rajamani, S. K., and Simmons, R. J., *Proofs from Tests.* In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2008. Seattle, WA: ACM. pp. 3-14.

41. Bellard, F., *Qemu, a fast and portable dynamic translat.* In *Proceedings of the Annual Conference on USENIX Annual Techincal Conference*. 2005.
42. Bhatkar, S., DuVarney, D. C., and Sekar, R. C., *Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.* In *USENIX Security Symposium*. 2003. Washington, DC: USENIX Association. pp. 105-120.
43. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., and Boneh, D., *Hacking blind.* In *Security and Privacy (SP), 2014 IEEE Symposium on*: IEEE. pp. 227-242.
44. Bond, M. D. and McKinley, K.S., *Probabilistic Calling Context*. 2007, ACM.
45. Borisov, N., Johnson, R., Sastry, N., and Wagner, D., *Fixing Races for Fun and Profit: How to abuse atime*. 2005.
46. Bosman, E., Slowinska, A., and Bos, H., *Minemu: The World's Fastest Taint Tracker.* In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*. Berlin, Heidelberg. pp. 1-20.
47. Brumley, D., Wang, H., Jha, S., and Song, D., *Creating Vulnerability Signatures Using Weakest Preconditions.* In *IEEE Computer Security Foundations Symposium (CSF).* 2007. Venice, Italy: IEEE Computer Society. pp. 311-325.
48. Cadar, C., Dunbar, D., and Engler, D. R., *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* In *Symposium on Operating System Design and Implementation (OSDI).* 2008. San Diego, CA: USENIX Association. pp. 209-224.
49. Cadar, C. and Engler, D. R., *Execution Generated Test Cases: How to Make Systems Code Crash Itself.* In *International SPIN Workshop on Model Checking of Software.* 2005. San Francisco, CA: Springer. pp. 2-23.
50. Cai, X., Gui, Y., and Johnson, R., *Exploiting Unix File-system Races via Algorithmic Complexity Attacks*. 2009, IEEE. pp. 27-41.
51. Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R., *Fast byte-granularity software fault isolation.* In *ACM Symposium on Operating Systems Principles (SOSP).* 2009. Big Sky, Montana, USA: ACM. pp. 45-48.
52. Chen, X., ASLR Bypass Apocalypse in Recent Zero-Day Exploits, https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html.
53. Cheng, W., Zhao, Q., Yu, B., and Hiroshige, S., *TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting.* In *Proceedings of the 11th IEEE Symposium on Computers and Communications*. 2006. Washington, DC. pp. 749-754.
54. Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H., *ROPecker: A generic and practical approach for defending against ROP attacks.* In *Symposium on Network and Distributed System Security (NDSS).* 2014.
55. Chew, L. and Lie, D., *Kivati: Fast Detection and Prevention of Atomicity Violations.* 2010. pp. 307-319.
56. Chin, E. and Wagner, D., *Efficient Character-level Taint Tracking for Java.* In *Proceedings of the 2009 ACM Workshop on Secure Web Services*. 2009. Chicago, IL: ACM. pp. 3-12.
57. Chipounov, V., Kuznetsov, V., and Candea, G., *S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems.* In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2011. Newport Beach, CA, USA: ACM. pp. 265-278.

58. Cho, C. Y., Babic, D., Poosankam, P., Chen, K. Z., Wu, E. X., and Song, D., *MACE: Model-inference-Assisted Concolic Explration for Protocol and Vulnerability Discovery*. In *USENIX Security Symposium*. 2011. San Francisco, CA, USA: USENIX Association.
59. Chris Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. 2008: No Starch Press.
60. Christey, S., 2011 CWE/SANS top 25 most dangerous software errors, http://cwe.mitre.org/top25/.
61. Christiansen, A. S., Moller, A., and Scwartzbach, M. I., *Precise Analysis of String Expressions.* In *Proceedings of the 10th International Conference on Static Analysis*. 2003. San Diego, CA: Springer-Verlag. pp. 1-18.
62. Christodorescu, M., Kidd, N., and Goh, W.-H., *String Analysis for x86 Binaries.* In *6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*: ACM Press.
63. Clause, J., Li, W., and Orso, A., *Dytan: a generic dynamic taint analysis framework.* In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. 2007. London, UK: ACM. pp. 196-206.
64. Coffman, E., Elphick, M., and Shoshani, A., *System deadlocks*. 1971. pp. 67-78.
65. Connover, M., w00w00 on heap overflows, http://www.w00w00.org/.
66. Coppens, J., cbrPager: a simple comic book pager for Linux, http://jcoppens.com/soft/cbrpager/index.en.php.
67. Corbet, J., Seccomp and sandboxing, http://lwn.net/Articles/332974/.
68. Corbet, J., Yet another new approach to seccomp, http://lwn.net/Articles/475043/.
69. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P., *Vigilante: End-to-End Containment of Internet Worms.* In *ACM Symposium on Operating Systems Principles (SOSP)*. 2005. Brighton, UK: ACM Press. pp. 133-147.
70. Cowan, C., Beattie, S., Johansen, J., and Wagle, P., *PointGuard: Protecting pointers from buffer overflow vulnerabilities.* In *12th USENIX Security Symposium*. 2004: USENIX. pp. 91-104.
71. Cui, H., Simsa, J., Lin, Y.-H., Li, H., Blum, B., Xu, X., Yang, J., Gibson, G., and Bryant, R., *PARROT: A Practical Runtime for Deterministic, Stable, and Reliable Threads*. 2013, ACM.
72. Cui, H., Wu, J., Gallagher, J., Guo, H., and Yang, J., *Efficient Deterministic Multithreading through Schedule Relaxation.* In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*. 2011. pp. 337-351.
73. Daniel, M., Honoroff, J., and Miller, C., *Engineering Heap Overflow Exploits with JavaScript.* In *Proceedings of the 2Nd Conference on USENIX Workshop on Offensive Technologies*. 2008. San Jose, CA: USENIX Association.
74. DataRescue, The IDA Pro Disassembler, http://www.hex-rays.com/idapro/.
75. Davi, L., Sadeghi, A.-R., and Winandy, M., *ROPdefender: a detection tool to defend against return-oriented programming attacks.* In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 2011. pp. 40-51.
76. Dean, D. and Hu, A.J., *Fixing Races for Fun and Profit: How to use access(2)*. 2004. pp. 195-206.
77. Demsky, B., Ernst, M. D., Guo, P. J., McCamant, S., Perkins, J. H., and Rinard, M. C., *Inference and enforcement of data structure consistency specifications.* In *ACM*

SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). 2006. Portland, Maine: ACM. pp. 233-244.

78. Demsky, B. and Rinard, M. C., *Automatic detection and repair of errors in data structures.* In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 2003. Anaheim, CA: ACM. pp. 78-95.
79. Demsky, B. and Rinard, M. C., *Data structure repair using goal-directed reasoning.* In *International Conference on Software Engineering (ICSE)*. 2005. St. Louis, MO: ACM. pp. 176-185.
80. Dhurjati, D. and Adve, V., *Backwards-compatible Array Bounds Checking for C with Very Low Overhead.* In *Proceedings of the 28th international conference on Software engineering*. 2006. Shanghai, China: ACM.
81. Dickens, C., A Tale of Two Cities, www.gutenberg.org/files/98/98.txt.
82. Durden, T., Bypassing PaX ASLR Protection, http://phrack.org/issues.html?issue=59&id=9#article.
83. Edward J.Schwartz, Thanassis Avgerinos, and David Brumley, *Q: exploit hardening made easy.* In *Proceedings of the 20th USENIX conference on Security*. 2011.
84. Elena Gabriela Barrantes, David H.Ackley, Stephanie Forrest, and Darko Stefanovic', *Randomized instruction set emulation*. ACM Transaction Information System Security, 2005. **8**(1): pp. 3-40.
85. Emery D Berger and Benjamin G.Zorn, *DieHard: probabilistic memory safety for unsafe languages.* In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2006. pp. 158-168.
86. Flanagan, C. and Freund, S.N., *FastTrack: Efficient and Precise Dynamic Race Detection*. 2009. pp. 121-133.
87. Ford, B. and Cox, R., *Vx32: Lightweight User-level Sandboxing on the x86.* In *USENIX Anual Technical Conference*. 2008. Boston, MA, USA: USENIX Association. pp. 293-306.
88. Forrest, S., Somayaji, A., and Ackley, D. H., *Building Diverse Computer Systems.* In *Workshop on Hot Topics in Operating Systems (HotOS)*. 1997. Cape Cod, MA: IEEE Computer Society Press. pp. 67-72.
89. FSE, *Failures-Divergence Refinement: FDR2 Manual*. 1997, Formal Systems (Europe) Ltd. (FSEL).
90. Futoransky, A., Gutesman, E., and Wassbein, A., *A dynamic technique for enhancing the security and privacy of web applications.* In *Black Hat USA.*
91. Garfinkel, T., Pfaff, B., Rosenblum, M., and et.al., *Ostia: A Delegating Architecture for Secure System Call Interposition.* In *Proceedings of the Symposium on Network and Distributed System Security*. 2004.
92. Gene Novark and Emery D Berger, *DieHarder: securing the heap.* In *Proceedings of the 17th ACM conference on Computer and communications security*. 2010. pp. 573-584.
93. Gene Novark, Emery D Berger, and Benjamin G Zorn, *Exterminator: automatically correcting memory errors with high probability.* In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. 2007. pp. 1-11.
94. Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi, *Surgically Returning to Randomized lib(c).* In *Proceedings of the 2009 Annual Computer Security Applications Conference*. 2009. pp. 60-69.

95. Godefroid, P., *Compositional Dynamic Test Generation.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2007. Nice, France: ACM. pp. 47-54.
96. Godefroid, P., Klarlund, N., and Sen, K., *DART: Directed Automated Random Testing.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. Chicago, IL: ACM. pp. 213-223.
97. Godefroid, P., Levin, M. Y., and Molnar, D., *Automated Whitebox Fuzz Testing*. 2007, Microsoft MSR-TR-2007-58.
98. Gopan, D., Melski, D., nguyen, d., naydich, d., and Driscoll, E., *Data-Delineation in Software Binaries and its Application to Buffer-Overrun Discovery.* In *ICSE 2015*. 2015. Firenze, Italy.
99. Greve, G. C. F., Brown, M., and Nelson, D., SpamAssasin Milter Plugin, http://savannah.nongnu.org/projects/spamass-milt/.
100. Haldar, V., Chandra, D., and Franz, M., *Dynamic Taint Propagation for Java.* In *Proceedings of the 21st Annual Computer Security Applications Conference*. 2005. Washington, DC: IEEE Computer Society. pp. 303-311.
101. Halfond, W., Orso, A., and Manolios, P., *WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation*. IEEE Transacations on Software Engineering, 2008. **34**: pp. 65-81.
102. Halfond, W. G. J. and Orso, A., *AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks.* In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. 2005. Long Beach, CA: ACM. pp. 174-183.
103. Halfond, W. G. J., Orso, A., and Manolis, P., *Using Positive Tainting and Syntax-aware Evaluation to Counter SQL Injection Attacks.* In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2006. Portland, OR: ACM. pp. 175-185.
104. Hammer, C., Dolby, J., Vaziri, M., and Tip, F., *Dynamic Detection of Atomic-Set Serializability Violations*. 2008, ACM. pp. 231-240.
105. Hiser, J. W., Coleman, C., and Davidson, J. W., *MEDS: The Memory Error Detection System.* In *International Symposium on Engineering Secure Software and Systems*. 2009. pp. 164-179.
106. Hiser, J., Coleman, C., Co, M., and Davidson, J., *MEDS: The Memory Error Detection System*. 2008, University of Virginia.
107. Hiser, J., Nguyen-Toung, A., Co, M., Hall, M., and Davidson, J., *ILR: Where'd My Gadgets Go?* In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 2012. Washington, DC: IEEE Computer Society. pp. 571-585.
108. Hiser, J., Williams, D., Hu, W., Davidson, J. W., Mars, J., and Childers, B., *Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems.* In *IEEE International Symposium on Network Computing and Applications*.
109. Hiser, J. D., Williams, D., Filipi, A., Davidson, J. W., and Childers, B. R., *Evaluating Fragment Construction Policies for SDT Systems.* In *2nd International Conference on Virtual Execution Environments*. 2006. pp. 122-132.
110. Hovav Shacham, *The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).* In *Proceedings of the 14th ACM conference on Computer and communications security*. 2007. pp. 552-561.

111. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D., Knight, J. C., Nguyen-Tuong, A., and Rowanhill, J., *Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation.* In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*. 2006. Ottawa, Ontario, Canada: ACM. pp. 2-12.
112. Hu, W., Hiser, J., Williams, D., Filipi, A., Davidson, J. W., Evans, D., Knight, J., Nguyen-Tuong, A., and Rowanhille, J., *Secure and Practical Defense Against Code-Injection Attacks.* In *International Conference on Virtual Execution Environments (VEE).* 2006. pp. 2-12.
113. J.Hiser, A.Nguyen-Toung, M.Co, M.Hall, and J.Davidson, *IRL: Where'd my gadgets go?* In *In submission*. 2011.
114. Jackson, T., Salamat, B., Homescu, A., Manivannan, K., Wagner, G., Gal, A., Brunthaler, S., Wimmer, C., and Franz, M., *Compiler-Generated Software Diversity Moving Target Defense*. in . 2011, Springer New York. pp. 77-98.
115. James Newsome and Dawn Song, *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*. 2005, CMU Department of Electrical and Computer Engineering Paper 3.
116. Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram, *Defeating return-oriented rootkits with "Return-Less" kernels.* In *Proceedings of the 5th European conference on Computer systems*. 2010. pp. 195-208.
117. Jula, H., Tralamazza, D., Zamfir, C., and Candea, G., *Deadlock Immunity: Enabling Systems To Defend Against Deadlocks*. 2008.
118. Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg, *Slipstream Processors: Improving Both Performance and Fault Tolerance.* In *ASPLOS*. 2000.
119. Kc, G. S., Keromytis, A. D., and Prevelakis, V., *Countering Code-Injection Attacks With Instruction-Set Randomization.* In *ACM Conference on Computer and Communications Security (CCS).* 2003. Washington, D.C.: ACM. pp. 272-280.
120. Kemerlis, V. P., Portokalidis, G., Jee, K., and Keromytis, A. D., *Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems.* In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. 2012. London, England: ACM. pp. 121-132.
121. Kil, C., Jim, J., Bookholt, C., Xu, J., and Ning, P., *Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software.* In *Computer Security Applications Conference, 2006.ACSAC'06.22nd Annual*. 2006. pp. 339-348.
122. Kiriansky, V., Bruening, D., and Amarasinghe, S., *Secure Execution Via Program Shepherding.* In *USENIX Security Symposium*. 2002. San Francisco, CA: USENIX. pp. 191-206.
123. Kruegel, C., Robertson, W., Valeur, F., and Vigna, G., *Static Disassembly of Obfuscated Binaries.* In *USENIX Security Symposium*. 2004. San Diego, CA: USENIX. pp. 255-270.
124. Kupsch, J. and Miller, B., *How to Open a File and Not Get Hacked*. 2008. pp. 1196-1203.
125. Lal, A., Lim, J., and Reps, T., *McDash: Refinement-based property verification for machine code.* 2009, Computer Sciences Department, University of Wisconsin.
126. Lim, J., Lal, A., and Reps, T., *Symbolic Analysis via Semantic Reinterpretation*. 2009, University of Wisconsin.

127. Linda Torczon and Keith Cooper, *Engineering A Compiler*. 2nd ed. 2011: Morgan Kaufmann Publishers Inc.
128. Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea, *Cloud9: A Software Testing Service.* In *3rd SOSP Workshop on Large Scale Distributed Systems and Middleware (LADIS)*. 2009. Big Sky, MT. pp. 5-10.
129. Livshits, B., *Dynamic Taint Tracking in Managed Runtime*. 2012 MSR-TR-2012-114.
130. Loscocco, P. and Smalley, S., *Integrating Flexible Support for Security Policies into the Linux Operating System.* In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. 2001. pp. 29-42.
131. Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y., *MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs.* In *ACM Symposium on Operating Systems Principles (SOSP)*. 2007. Stevenson, WA: ACM. pp. 103-116.
132. Lu, S., Park, S., Seo, E., and Zhou, Y., *Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics*. ACM SIGARCH Computer Architecture News, 2008. **36**(1): pp. 329-339.
133. Lu, S., Tucek, J., Qin, F., and Zhou, Y., *AVIO: Detecting Atomicity Violations via Access-Interleaving Invariants.* IEEE Micro, 2007. **27**(1): pp. 26-35.
134. Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy, *ROPdefender: a detection tool to defend against return-oriented programming attacks.* In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. 2011. pp. 40-51.
135. Luk, C.-K., Cohn, R. S., Muth, R., Patil, H., Klauser, A., Lowney, P. G., Wallace, S., Reddi, V. J., and Hazelwood, K. M., *Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. Chicago, IL: ACM. pp. 190-200.
136. Majumdar, R. and Sen, K., *Hybrid Concolic Testing.* In *International Conference on Software Engineering (ICSE)*. 2007. Minneapolis, Minnesota: IEEE Computer Society. pp. 416-426.
137. Majumdar, R. and Sen, K., *LATEST: Lazy Dynamic Test Input Generation*. 2007, University of California at Berkeley, Berkeley, CA UCB/EECS-2007-36.
138. Mathias Payer and Thomas R.Gross, *Fine-grained user-space security through virtualization.* In *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2011: ACM. pp. 157-168.
139. Michael J.Voss and Rudolf Eigenmann, *A framework for remote dynamic program optimization.* In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*. 2000. pp. 32-40.
140. Michele, C., Jack, W. D., Jason, D. H., John, C. K., Anh, N., David, C., Denis, G., David, M., Wenke, L., Chengyu, S., Thomas, B., David, H., and Brian, M., *PEASOUP: preventing exploits against software of uncertain provenance (position paper).* In : ACM. pp. 43-49.
141. Mitre, CWE - Common Weakness Enumeration, http://cwe.mitre.org/.
142. Nagarakatte, S., Zhao, J., Matin, M., and Zdancewic, S., *CETS: Compiler Enforced Temporal Safety for C.* In *Proceedings of the 2010 International Symposium on Memory Management*. 2010. Toronto, Ontario, Canada: ACM. pp. 31-40.

143. Nagarakatte, S., Zhao, J., Martin, M. M. K., and Zdancewic, S., *SoftBound: highly compatible and complete spatial memory safety for C.* In *Programming Language Design and Implementation*: ACM. pp. 245-258.
144. Nergal, The Advanced Return-Into-Lib(C) Exploits, http://phrack.org/show.php?p=58&a=4.
145. Nethercote, N. and Seward, J., *Valgrind: A Program Supervision Framework*. Electronic Notes in Theoretical Computer Science (ENTCS), 2003. **89**(2): pp. 1-23.
146. Netzer, R. H. B. and Miller, B. P., *What Are Race Conditions?: Some Issues and Formalizations*. ACM Letters on Programming Languages and Systems (LOPLAS), 1992. **1**(1): pp. 74-88.
147. Newsham, T., Format string attacks.
148. Newsome, J. and Song, D., *Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software*. 2005, CMU Department of Electrical and Computer Engineering Paper 3.
149. Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D., *Automatically hardening web applications using precise tainting.* In *20th IFIP International Information Security Conference*. 2005: Springer. pp. 372-382.
150. Nguyen-Tuong, A., Hiser, J. D., Co, M., Davidson, J. W., Knight, J. C., Kennedy, N., Melski, D., Ella, W., and Hyde, D., *To B or not to B: Blessing OS Commands with Software DNA Shotgun Sequencing.* In *Proceedings of the 2014 European Dependable Computing Conference (EDCC '14)*. 2014. Washington, DC: IEEE Computer Society. pp. 238-249.
151. Nick Kolettis and N.Dudley Fulton, *Software Rejuvination: Analysis, Module and Applications.* In *Int.Symposium on Fault Tolerant Computing*. 1995.
152. Niranjan, H., Ashish, M., and R.Sekar, *Light-weight bounds checking.* In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO)*: ACM. pp. 135-144.
153. Niu, B. and Tan, G., *RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity.* In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014. Scottsdale, AZ: ACM. pp. 1317-1328.
154. Niu, B. and Tan, G., *Modular Control-flow Integrity.* In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2014. pp. 577-587.
155. Papgiannis, I., Migliavacca, M., and Pietzuch, P., *PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks.* In *Proceedings of the 2Nd USENIX Conference on Web Application Development*. 2011. Portland, Oregon: USENIX Association. p. 2.
156. Pappas, V., Polychronakis, M., and Keromytis, A. D., *Smashing the gadgets: Hindering return-oriented programming using in-place code randomization.* In *Security and Privacy (SP), 2012 IEEE Symposium on*: IEEE. pp. 601-615.
157. Park, S., Lu, S., and Zhou, Y., *CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places.* In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Washington, DC, USA.
158. Patil, H. and Fischer, C., *Low-cost, concurrent checking of pointer and array accesses in c programs*. Software-Practice & Experience, 1997. **27**(1): pp. 87-110.
159. PaX, PaX project, http://pax.grsecurity.net/.

160. Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G. W. W.-F., Zibin, Y., Ernst, M. D., and Rinard, M., *Automatically Patching Errors in Deployed Software.* In *SOSP '09*. 2009. Big Sky, Montana, USA: ACM. pp. 87-102.
161. Pie, P., Mobile Pwn2Own Autumn 2013 - Chrome on Android - Expolit Writeup, https://www.rstforums.com/forum/80945-mobile-pwn2own-autumn-2013-chrome-android-exploit-writeup.rst.
162. Pietraszek, T. and Berghe, C. V., *Defending Against Injection Attacks Through Context-sensitive String Evaluation.* In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*. 2006. Seattle, WA: Springer-Verlag. pp. 124-145.
163. Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie, *DROP: Detecting Return-Oriented Programming Malicious Code.* In *Proceedings of the 5th International Conference on Information Systems Security*. 2009. pp. 163-177.
164. Probst, M., *Dynamic binary translation.* In *UKUUG Linux Developer¡¯s Conference*: sn.
165. Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., and Wu, Y., *LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks*. 2006. pp. 135-148.
166. Ramalingam, G., Field, J., and Tip, F., *Aggregate Structure Identification and Its Application to Program Analysis.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1999. San Antonio, TX: ACM Press. pp. 119-132.
167. Ramos, D. A. and Engler, D. R., *Practical, low-effort equivalenc verification of real code.* In *International Conference on Computer Aided Verification (CAV)*. 2011. Snowbird, UT, USA: ACM. pp. 669-685.
168. Renieris, M., Chan-Tin, S., and Reiss, S. P., *Elided Conditionals.* In *ACM SIGPLAN-SIGSSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*. 2004. Washington, DC: ACM. pp. 52-57.
169. Reps, T. and Balakrishnan, G., *Improved Memory-Access Analysis for x86 Executables.* In *International Conference on Compiler Construction (CC)*. 2008. Budapest, Hungary. pp. 16-35.
170. Reps, T., Balakrishnan, G., Teitelbaum, T., and Lim, J., *A next-generation platform for analyzing executables.* In *Proc.3rd Asian Symposium on Programming Languages and Systems*. 2009. Tsukuba, Japan.
171. Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Beebe, W. S. Jr., *Enhancing server availability and security through failure-oblivious computing.* In *6th conference on Symposium on Operating Systems Design & Implementation*. 2004. San Francisco, CA. p. 21.
172. Roglia, G. F., Martignoni, L., Paleari, R., and Bruschi, D., *Surgically Returning to Randomized lib(c).* In *Proceedings of the 2009 Annual Computer Security Applications Conference*. 2009. pp. 60-69.
173. Ryan Glenn Roemer, *Finding the Bad in Good Code: Automated Return-Oriented Programming Exploit Discovery*. *University of California*. 2009.
174. S.Designer, *""return-to-libc" attack.* In *Bugtraq, Aug*. 1997.
175. Sandeep Bhatkar and R.Sekar, *Efficient techniques for comprehensive protection from memory error exploits.* In *Proceedings of the 14th conference on USENIX Security Symposium*. 2005. Baltimore, MD: USENIX Association. p. 17.

176. Sandeep Bhatkar and R.Sekar, *Data Space Randomization.* In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Paris, France: Springer-Verlag, 2008. pp. 1-22.
177. Saravanan Sinnadurai, Qin Zhao, and Weng-fai Wong, *Transparent Runtime Shadow Stack: Protection against malicious return address modifications.* In . 2008.
178. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. E., *Eraser: A Dynamic Data Race Detector for Multithreaded Programs*. ACM Transactions on Computer Systems (TOCS), 1997. **15**(4): pp. 391-411.
179. Saxena, P., Sekar, R., and Puranik, V., *Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking.* In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2008.
180. Schwarz, B., Debray, S. K., and Andrews, G. R., *Disassembly of Executable Code Revisited.* In *Working Conference on Reverse Engineering (WCRE)*. 2002. Richmond, VA: IEEE Computer Society. pp. 45-54.
181. Scott, K. and Davidson, J., *Strata: A software dynamic translation infrastructure.* In *IEEE Workshop on Binary Translation*: IEEE.
182. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., and Soffa, M. L., *Retargetable and reconfigurable software dynamic translation.* In *International Symposium on Code Generation and Optimization*: IEEE Computer Society. pp. 36-47.
183. Sehr, D., Muth, R., Biffle, C., Khimenko, V., Pasko, E., Schimpf, K., Yee, B., and Chen, B., *Adapting Software Fault Isolation to Contemporary CPU Architectures.* In *Proceedings of the 19th USENIX Conference on Security*. 2014. Washington, DC. p. 1.
184. Sekar, R., *An Efficient Black-box Technique for Defeating Web Application Attacks.* In *Network and Distributed Systems Security Symposium*. 2009.
185. Sen, K., Marinov, D., and Agha, G., *CUTE: A Concolic Unit Testing Engine for C.* In *European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 2005. Lisbon, Portugal: ACM. pp. 263-272.
186. Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D., *AddressSanitizer: A Fast Address Sanity Checker.* In *USENIX Annual Technical Conference*. pp. 309-318.
187. Serna, F., *The info leak era on software exploitation*. Black Hat USA, 2012.
188. Seward, J., bzip2, http://www.bzip.org/.
189. Seward, J. and Nethercote, N., *Using Valgrind to Detect Undefined Value Errors With Bit-Precision.* In *USENIX Technical Conference*. 2005. Anaheim, CA: USENIX. pp. 17-30.
190. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., and Boneh, D., *On the Effectiveness of Address Space Randomization.* In *ACM Conference on Computer and Communications Security (CCS)*. 2004.
191. Sharif, M., Lee, W., Cui, W., and Lanzi, A., *Secure In-VM Monitoring Using Hardware Virtualization.* In *ACM Conference on Computer and Communications Security (CCS).* 2009.
192. Sidiroglou, S., Giovanidis, G., and Keromytis, A. D., *A Dynamic Mechanism for Recovering from Buffer Overflow Attacks.* In *Proceedings of the 8th International Conference on Information Security*. 2005. Singapore: Springer-Verlag. pp. 1-15.

193. Sidiroglou, S., Laadan, O., Perez, C. R., Viennot, N., Nieh, J., and Keromytis, A. D., *ASSURE: Automatic Software Self-healing Using REscue points.* In *ASPLOS'09*. Washington, D.C.

194. Sidiroglou, S., Locasto, M., Boyd, S., and Keromytis, A., *Building a Reactive Immune System for Software Services.* In *Proceedings of the annual conference on USENIX Annual Technical Conference*. 2005.

195. Silberschatz, A., Galvin, P., and Gagne, G., *Operating System Concepts*. 1998, Addison-Wesley.

196. Sintsov, A., *Writing jit-spray shellcode for fun and profit*. Writing, 2010.

197. Snow, K., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A., *Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization.* In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. 2013. Washington, DC. pp. 574-588.

198. Song, C., Zhang, C., Wang, T., Lee, W., and Melski, D., *Exploiting and protecting dynamic code generation.* In *Proceedings of the 2015 Network and Distributed System Security (NDSS) Symposium*.

199. Standard Performance Evaluation Corporation, SPEC CPU 2006, http://www.spec.org/cpu2006/.

200. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy, *Return-oriented programming without returns.* In *Proceedings of the 17th ACM conference on Computer and communications security*. 2010. pp. 559-572.

201. Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens, *ValueGuard: protection of native applications against data-only buffer overflows.* In *Proceedings of the 6th international conference on Information systems security*. 2010: Springer-Verlag. pp. 156-170.

202. Su, Z. and Wassermann, G., *The Essence of Command Injection Attacks in Web Applications.* In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2006. Chrleston, SC: ACM. pp. 372-382.

203. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., and Reps, T., *Directed proof generation for machine code.* In *Proceedings of International Conference on Computer Aided Verification (CAV '10)*. 2010. Edinburgh, UK. pp. 288-305.

204. Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann, *A framework for automated architecture-independent gadget search.* In *Proceedings of the 4th USENIX conference on Offensive technologies*. 2010.

205. Tsafrir, D., Hertz, T., Wagner, D., and Da Silva, D., *Portably Solving File Races with Hardness Amplification*. 2008.

206. Vaziri, M., Tip, F., and Dolby, J., *Associating synchronization constraints with data in an object-oriented language.* In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2006. Charleston, South Carolina: ACM. pp. 334-345.

207. Viega, J., Bloch, J. T., Kohno, T., and McGraw, G., *ITS4: A Static Vulnerability Scanner for C and C++ Code.* In *Annual Computer Security Applications Conference (ACSAC)*. 2000. New Orleans, LA: IEEE Computer Society. p. 257.

208. Vijayakumar, H., Schiffman, J., and Jaeger, T., *STING: Finding Name Resolution Vulnerabilities in Programs*. 2012. pp. 585-599.

209. Vitaliy B.Lvin and Gene Novark, E. D. B. a. B. G. Z., *Archipelago: trading address space for reliability and security*. SIGOPS Operating Systems Review, 2008. **42**(2): pp. 115-124.

210. Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea, *S2E: a platform for in-vivo multi-path analysis of software systems.* In *ASPLOS '11*. 2011. Newport Beach, California: ACM. pp. 265-278.

211. Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L., *Efficient Software-Based Fault Isolation.* In *ACM Symposium on Operating Systems Principles (SOSP)*. 1993. Asheville, NC: ACM Press. pp. 203-216.

212. Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., and Mahlke, S., *Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs*. 2008. pp. 281-294.

213. Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z., *Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code.* In *Proceedings of the 2012 ACM conference on Computer and communications security*: ACM. pp. 157-168.

214. Wei, J. and Pu, C., *TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study*. 2005.

215. Wei, T., Wang, T., Duan, L., and Luo, J., *Secure Dynamic Code Generation Against Spraying.* In *Proceedings of the 17th ACM conference on Computer and communications security*. 2010. Chicago, IL. pp. 738-740.

216. Wikipedia, Shotgun Sequencing, http://en.wikipedia.org/wiki/Shotgun_sequencing.

217. Wilander, J. and Kamkar, M., *A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention.* In *Symposium on Network and Distributed System Security (NDSS)*. 2003. San Diego, CA: The Internet Society. pp. 149-162.

218. Will Dietz, Peng Li, John Regehr, and Vikram Adve, *Understanding Integer Overflow in C/C++.* In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*. 2012. Zurich, Switzerland.

219. Xu, M., Bodík, R., and Hill, M. D., *A serializability violation detector for shared-memory server programs.* In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005. Chicago, IL, USA: ACM. pp. 1-14.

220. Xu, W., DuVarney, D., and Sekar, R., *An Efficient and Backwards-compatible Transformation to Ensure Memory Safety of C Programs.* In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. 2004. Newport Beach, CA: ACM. pp. 117-126.

221. Xu, W., Bhatkar, S., and Sekar, R., *Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks.* In *USENIX Security Symposium*.

222. Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N., and Fullagar, N., *Native Client: A Sandbox for Portable, Untrusted x86 Native Code.* In *IEEE Symposium on Security and Privacy*. 2009. Oakland, CA, USA: IEEE Computer Society. pp. 79-93.

223. Yih Huang and Anup Ghosh, *Automating Intrusion Response via Virtualization for Realizing Uninterruptible Web Services.* In *IEEE International Symposium on Network Computing and Applications*. 2009.

224. Zeller, A., *Yesterday, my program worked. Today, it does not. Why?* In *European Software Engineering Conference/ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*. 1999. Toulouse, France.

225. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W., *Practical control flow integrity & randomization for binary executables.* In *IEEE Symposium on Security and Privacy*. 2013. San Francisco, CA.

226. Zhang, M. and Sekar, R., *Control Flow Integrity for COTS Binaries.* In *Proceedings of the 22Nd USENIX Conference on Security*. 2013. Washington, DC: USENIX Association. pp. 337-352.

227. Zhang, T., Zhuang, X., Pande, S., and Lee, W., *Anomalous path detection with hardware support.* In *2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, NY: ACM. pp. 43-54.

228. Zhang, W., de Kruijf, M., Li, A., Lu, S., and Sankaralignam, K., *ConAir: Featherweight Concurrency Bug Recovery Via Single-Threaded Idempotent Execution*. 2013, ACM.

List of Acronyms, Abbreviations, and Symbols

ACRONYM	**DESCRIPTION**
!RUE	Not Rendered UnExploitable
ACSAC	Annual Computer Security Applications Conference
ADM	Advanced Micro Devises
AF	Altered Functionality
AOI	All Offset Inference
ASLR	Address Space Layout Randomization
ASSURE	Automatic Software Self-healing Using REscue points
BAA	Broad Agency Announcements
BED	Behavior Equivalence Detector
CAV	Computer Aided Verification
CSS	Computer and Communications Security
CFI	Control flow integrity
CGO	Symposium on Code Generation and Optimization
CM	Configuration Management
COTS	Commercial Off-The-Shelf
COW	copy-on-write
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DARPA	Defense Advanced Research Projects Agency
DHCP	Dynamic Host Configuration Protocol
DLL	Dynamically Linked Library
DOI	Direct Offset Inference
DOS	Denial-of-Service
DNS	Domain Name System
DROP	Detecting Return-Oriented Programming
DVT	Disassembler Validation Tool
EMP/NPT	Extended/Nested Page Tables
ELF	Executable and Linkable Format
ESI	Extended Stack Index register
GB	Gigabyte
GHz	Gigahertz
gtSDT	GrammaTech’s framework for software dynamic translation
HTML	Hypertext Markup Language
IRAPA	Intelligence Advanced Research Projects Activity
IB	Indirect Branch
IBT	Indirect Branch Targets
ICSE	International Conference on Software Engineering
ILR	Instruction-Layout Randomization
I/O	Input/Output
IP	Instruction Pointer register
IR	Intermediate Representation
IRC	Internet Relay Chat
IRDB	Intermediate Representation Database
JITted	just-in-time compiled

ACRONYM	DESCRIPTION
JSON	JavaScript Object Notation
KB	Kilobyte
KVM	A type 1 hypervisor
LTS	Long Term Support
LVA	Live Variable Analysis
MACE	Model-inference Assisted Concolic Execution
MEDS	Memory Error Detection System
MB	Megabyte
NDSS	Network and Distributed System Security Symposium
ngIRCd	Next Generation Internet Relay Chat Daemon
OS	Operating System
PC	Program Counter
PDF	Portable Document Format
PEASOUP	Preventing Exploits Against Software of Uncertain Provenance
PEPM	Partial Evaluation and Semantic-Based Program Manipulation
PIC	Position Independent Code
PIE	Position Independent Executable
PLDI	Programming Language Design and Implementation
PVM	Parallel Virtual Machine
QEMU	Quick EMUlator
QCOW2	QEMU Copy on Write 2
ROP	Return-Oriented Programming
RTL	Register Transfer List
RUE	Rendered UnExploitable
S2E	Selective Symbolic Execution
SAGE	Scalable Automated Guided Execution
SBIR	Small Business Innovation Research
SDT	Software Dynamic Translation
SDK	Software Developer’s Kit
SIM	Secure In-VM Monitoring
SLR	Stack-Layout Randomization
SMT	Satisfiability Modulo Theories
SO	Shared Object
SOI	Scaled Offset Inference
SOUP	Software of Uncertain Provenance
SPEC	Standard Performance Evaluation Corporation
SPRI	Sprocket Program Rewriting Interface
SQL	Structured Query Language
SRP	self-randomizing programs
SSA	Static Single Assignment
SSD	Solid State Drive
STARS	STatic Analyzer for Reliability and Security
STONESOUP	Securely Taking On New Executable Software of Uncertain Provenance
SWYX	See What You eXecute
T&E	Test and Evaluation

ACRONYM	**DESCRIPTION**
TRUSS	Transparent RUntime Shadow Stack
TSET	Test-Suite Evaluation Technology
USENIX	USENIX: The Advanced Computing Systems Association
UVA	University of Virginia
VCPU	Virtual Central Processing Unit
VEE	Virtual Execution Environments
VM	Virtual Machine
VMM	Virtual Machine Monitoring