AFRL-RY-WP-TR-2015-0002

MINESTRONE

Salvatore Stolfo, Angelos D. Keromytis, Junfeng Yang, Dimitris Geneiatakis, Michalis Polychronakis, Georgios Portokalidis, Kangkook Jee, and Vasileios P. Kemerlis

Columbia University

Angelos Stavrou and Dan Fleck

George Mason University

Matthew Elder and Azzedine Benameur

Symantec

MARCH 2015
Final Report

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

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by IARPA Public Affairs Office and is available to the general public, including foreign nationals. Qualified requestors may obtain copies of this report from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2015-0002 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

TOD J. REINHART
Program Manager
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

//Signature//

DAVID G. HAGSTROM, Chief
Avionics Vulnerability Mitigation Branch
Spectrum Warfare Division

//Signature//

TODD A. KASTLE, Chief
Spectrum Warfare Division
Sensors Directorate

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)*
March 2015

2. REPORT TYPE
Final

3. DATES COVERED *(From - To)*
2 August 2010 – 30 November 2014

4. TITLE AND SUBTITLE
MINESTRONE

5a. CONTRACT NUMBER
FA8650-10-C-7024

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
695285

6. AUTHOR(S)

Salvatore Stolfo, Angelos D. Keromytis, Junfeng Yang, Dimitris Geneiatakis, Michalis Polychronakis, Georgios Portokalidis, Kangkook Jee, and Vasileios P. Kemerlis (Columbia University)
Angelos Stavrou and Dan Fleck (George Mason University)
Nathan Evans, Matthew Elder, and Azzedine Benameur (Symantec)

5d. PROJECT NUMBER
ST0N

5e. TASK NUMBER
RY

5f. WORK UNIT NUMBER
Y0LK

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Columbia University
Department of Computer Science
M.C. 0401
1214 Amsterdam Avenue
New York, NY 10027-7003

George Mason University
Symantec

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

Office of Safe and Secure Operations
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-2015-0002

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

13. SUPPLEMENTARY NOTES
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 cleared by Schira Madan of the IARPA Public Affairs Office, 23 March 2015. Supported by the Intelligence Advanced Research Projects Activity (IARPA) via contract Air Force Research Laboratory (AFRL) contract number FA8650-10-C-7024. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, AFRL, or the U.S. Government. Report contains color.

14. ABSTRACT

MINESTRONE is an architecture that integrates static analysis, dynamic confinement, and code diversification techniques to enable the identification, mitigation and containment of a large class of software vulnerabilities. These techniques protect new software, as well as legacy software, by transparently inserting extensive security instrumentation.

15. SUBJECT TERMS

MINESTRONE, vulnerability detection and mitigation, static analysis, dynamic confinement, code diversification

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**			
Unclassified	Unclassified	Unclassified	SAR	66	Tod Reinhart **19b. TELEPHONE NUMBER** *(Include Area Code)* N/A

TABLE OF CONTENTS

Section **Page**

LIST OF FIGURES

Figure **Page**

1.0 SUMMARY

MINESTRONE is a novel architecture that integrates static analysis, dynamic confinement, and code diversification techniques to enable the identification, mitigation and containment of a large class of software vulnerabilities. These techniques protect new software, as well as already deployed (legacy) software by transparently inserting extensive security instrumentation. They also leverage concurrent program analysis (potentially aided by runtime data gleaned from profiling software) to gradually reduce the performance cost of the instrumentation by allowing selective removal or refinement.

MINESTRONE also uses diversification techniques for confinement and fault-tolerance purposes. To minimize performance impact, this project also leverages multi-core hardware or (when unavailable) remote servers to enable the quick identification of potential compromises.

The developed techniques require no specific hardware or operating system features, although they take advantage of such features where available, to improve both runtime performance and vulnerability coverage.

2.0 INTRODUCTION

This work investigates the integration of static analysis, dynamic confinement, and code diversification techniques to enable the identification, mitigation and containment of a large class of software vulnerabilities from language class B (C/C++). The system enables the *immediate* deployment of new software and the protection of already deployed (legacy) software by transparently inserting extensive security instrumentation, while leveraging concurrent program analysis (potentially aided by runtime data gleaned from profiling actual use of the software) to gradually reduce the performance cost of the instrumentation by allowing selective removal or refinement. Artificial diversification techniques are used both as confinement mechanisms and for fault-tolerance purposes. To minimize performance impact, this work leverages multi-core hardware or (when unavailable) remote servers that enable quick identification of likely compromise. The approach requires no specific hardware or operating system features, although it seeks to take advantage of such features where available, to improve both runtime performance and vulnerability coverage beyond the specific Broad Agency Announcement (BAA) goals. MINESTRONE is an integrated architecture, which brings together the elements of this effort.

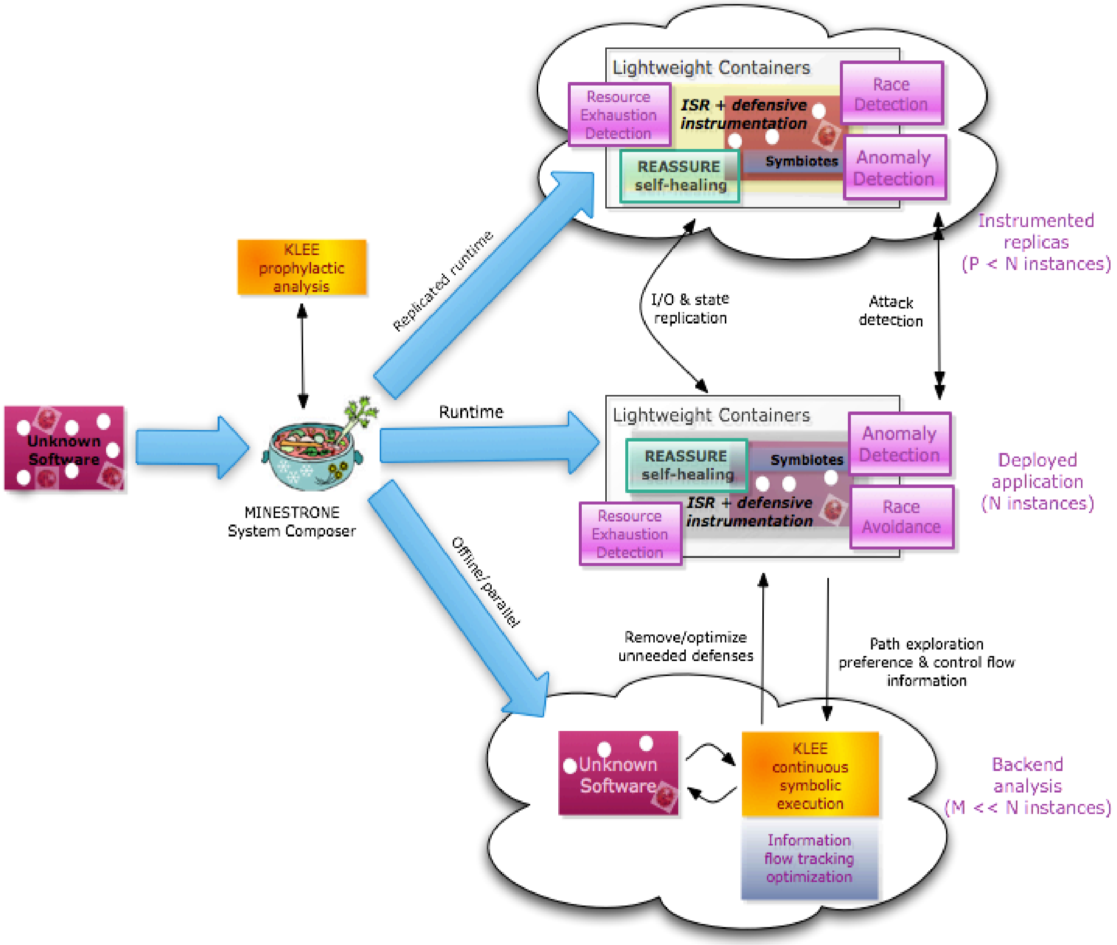

Lightweight Containers
Resource Exhaustion Detection
ISR + defensive instrumentation
Race Detection
REASSURE self-healing
Symbiotes
Anomaly Detection
Instrumented replicas (P < N instances)
KLEE prophylactic analysis
Replicated runtime
I/O & state replication
Attack detection
Unknown Software
MINESTRONE System Composer
Runtime
Lightweight Containers
REASSURE self-healing
Symbiotes
Anomaly Detection
Resource Exhaustion Detection
ISR + defensive instrumentation
Race Avoidance
Deployed application (N instances)
Offline/parallel
Remove/optimize unneeded defenses
Path exploration preference & control flow information
Unknown Software
KLEE continuous symbolic execution
Information flow tracking optimization
Backend analysis (M << N instances)

Figure 1. MINESTRONE Architecture

As part of the effort, stand-alone tools were developed for each of the research thrusts identified in Figure 2 below. MINESTRONE is an integrated architecture incorporated all of these thrusts. Figure 2 shows a Gantt chart of all tasks.

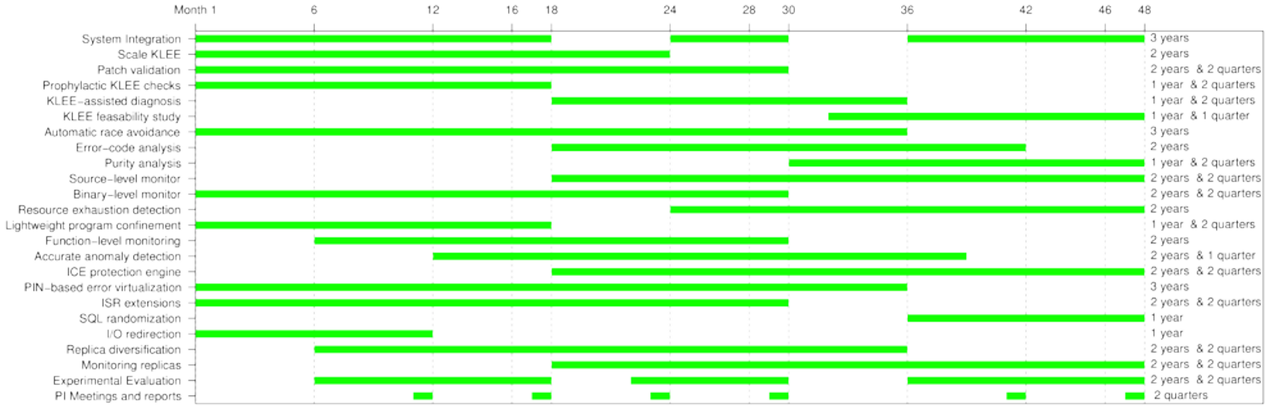
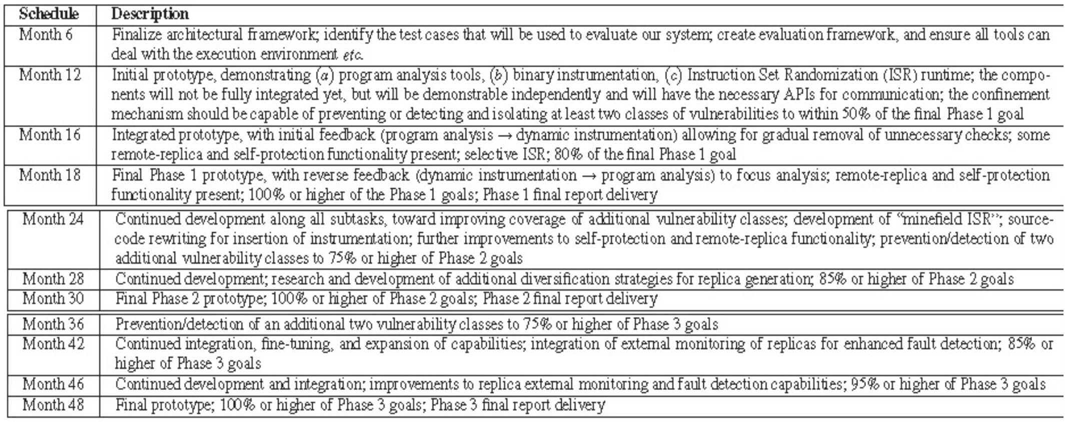


Figure 2: Gantt Chart of all Tasks

Figure 3 below shows the original schedule for these tasks.



Schedule	Description
Month 6	Finalize architectural framework; identify the test cases that will be used to evaluate our system; create evaluation framework, and ensure all tools can deal with the execution environment *etc.*
Month 12	Initial prototype, demonstrating (*a*) program analysis tools, (*b*) binary instrumentation, (*c*) Instruction Set Randomization (ISR) runtime; the components will not be fully integrated yet, but will be demonstrable independently and will have the necessary APIs for communication; the confinement mechanism should be capable of preventing or detecting and isolating at least two classes of vulnerabilities to within 50% of the final Phase 1 goal
Month 16	Integrated prototype, with initial feedback (program analysis → dynamic instrumentation) allowing for gradual removal of unnecessary checks; some remote-replica and self-protection functionality present; selective ISR; 80% of the final Phase 1 goal
Month 18	Final Phase 1 prototype, with reverse feedback (dynamic instrumentation → program analysis) to focus analysis; remote-replica and self-protection functionality present; 100% or higher of the Phase 1 goals; Phase 1 final report delivery
Month 24	Continued development along all subtasks, toward improving coverage of additional vulnerability classes; development of "minefield ISR"; source-code rewriting for insertion of instrumentation; further improvements to self-protection and remote-replica functionality; prevention/detection of two additional vulnerability classes to 75% or higher of Phase 2 goals
Month 28	Continued development; research and development of additional diversification strategies for replica generation; 85% or higher of Phase 2 goals
Month 30	Final Phase 2 prototype; 100% or higher of Phase 2 goals; Phase 2 final report delivery
Month 36	Prevention/detection of an additional two vulnerability classes to 75% or higher of Phase 3 goals
Month 42	Continued integration, fine-tuning, and expansion of capabilities; integration of external monitoring of replicas for enhanced fault detection; 85% or higher of Phase 3 goals
Month 46	Continued development and integration; improvements to replica external monitoring and fault detection capabilities; 95% or higher of Phase 3 goals
Month 48	Final prototype; 100% or higher of Phase 3 goals; Phase 3 final report delivery

Figure 3: Original Schedule

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

A systems-level approach was taken to designing, prototyping, and evaluating the individual components. The components are designed for use in Linux, and tested with the Ubuntu distribution (but should be portable across all major Linux releases). Most of the components are user-space resident; the integrated system makes use of the OpenVZ container functionality contained in recent Linux kernels, as well as a small kernel module for enabling I/O redirection and container management functionality. Most components can operate on pure binary executables, although they will operate more effectively if there are debug symbols compiled in the binary.

3.1 Evaluation Assumptions

In making these performance/overhead estimates, the following assumptions are made:

- A variety of applications will be used to test runtime overhead. Runtime overhead is highly dependent upon the application being tested, as well as the particular inputs provided to the application, and the runtime overhead of each detection technology will vary based upon that application and workload.
- All detection technology components can be executed in parallel, as opposed to sequentially. Measuring the runtime overhead in terms of increased running time would only make sense in the context of the parallel execution configuration; having to run each detection technology sequentially would result in prohibitive increased running times.
- Sufficient additional hardware resources are present in order to execute all detection technologies in parallel. This assumption is likely reasonable given the hardware capabilities of today's personal computers and datacenter resources. Personal computers now come with multiple processor cores, which are often underutilized – those existing multiple cores could be used by the MINESTRONE integrated prototype, one per container, in order to run multiple detection technologies in parallel. Alternatively, or in addition, many enterprises have local server and/or datacenter infrastructure, where MINESTRONE containers could be executed in parallel, without significant network latency.
- One-time initialization/configuration activities - actions that can be completed prior to execution of the application - are not included as part of the runtime overhead measurement. For example, in interactive applications with a graphical user interface where the UI/mouse actions must be redirected/replicated to the different detection technology containers, the one-time cost of initializing the VNC sessions between the No Security container and detection technology containers is not included in the runtime overhead measurement. Similarly, for the ISR component, the upfront cost of encoding the binary is not included in the runtime overhead.
- The MINESTRONE integrated prototype configuration can be tailored to the application. In other words, the detection technologies deployed within a particular instance of the MINESTRONE integrated prototype can be tailored, turned on or off as desired, in order to meet the performance and detection requirements of the particular environment.

4.0 RESULTS AND DISCUSSION

4.1 Phase 1 Developments

4.1.1 Scale KLEE

As was indicated in the initial proposal, the bulk of the work has focused on scaling KLEE. There have been two main thrusts:

- Developing techniques that handle the enormous number of code paths through real code. Such "path explosion" is the most challenging scalability problem for checking large systems.
- Vastly increasing the scope of programs KLEE can handle by building a version that can analyze x86 and ARM executables.

In addition, significant work has been done tuning the KLEE system, including developing many novel optimizations for analyzing constraints.

4.1.1.1 Handling Path Explosion

Two main techniques have been developed for handling path explosion:

- Path pruning, which detects when the suffix of a path must produce the same result as a previously explored path and skips it.
- Path merging, which merges multiple paths into a single state, thereby allowing the system to simultaneously execute many paths at once.

Path pruning is a much more mature technique, which gives order of magnitude performance improvements over a broad range of applications. Path merging is a more recent technique (developed in the past couple of months), which gives dramatic speedups on some applications, but performs indifferently on others.

Each is discussed below.

4.1.1.2 Path Pruning

KLEE uses path-based symbolic execution. In this context, each state follows a single unique path in the code, starting from the checked program's first instruction. States are created when a code path forks (*e.g.,* at an if-statement); different paths will run the same instruction when these paths rejoin (*e.g.,* the instruction after an if-statement). In this context, a naive state equivalence-checking algorithm is simple:

- Every program point records the set of all states that have reached it in a "state cache."
- Before running a state S on the instruction at program point P, check if S is in P's state cache. If so, terminate (since it will not do anything new). Otherwise insert it and continue.

This, of course, is the algorithm used in many model checkers.

The main problem with the naive approach is that it considers two states equal *iff* all locations have the same value. Most locations cannot change behavior, yet a difference in one of them will prevent a cache miss. This is especially true when checking implementation code as compared to reduced an abstract model since the number of locations will typically be dramatically larger (*e.g.,* tens of thousands in the programs checked). Further, for this flavor of path-based symbolic execution, this cache will almost never produce a hit --- since each state represents a unique path, typically at least one constraint in that state will differ from all other states, preventing a hit unless the location is dead.

There are two complementary ways to dramatically improve hit rates. The first determines when superficially different constraints will produce the same (or a subset) of previously seen behavior, such as a state that has a more specific set of constraints than what has already been explored (*e.g.,* "x = 10" versus "x > 0"). The second approach focuses on discarding as many irrelevant locations as possible in the state records recorded in the state cache, driving down the miss rate from spurious differences.

Both are good ways to go. Focus has been placed only on the second. Because it is desirable to generate inputs that get high code coverage, the strategy has been: remove locations that do not matter (cannot lead to new coverage) and then propagate these refinements as far as possible (*i.e.,* to as many state caches above the prune point as possible). As the results show, discarding irrelevant locations can improve the cache-hit rate by to one to two orders of magnitude.

One reason a location at l in a state S at program point P is irrelevant is if no subsequent instruction reachable from P using the values in S can read l, *i.e.,* l is dead rather than live. One of the main refinements is to compute the set of all live locations reachable from P (the ``live set") by taking the union of all locations read during an exhaustive exploration of all reachable instructions from P and then intersecting the live set with S in the state cache, discarding all irrelevant locations.

The number of relevant locations can be reduced further. Again since statement coverage is important, once an instruction is covered, reaching it again is uninteresting. Thus, if a location l only affects whether an already covered instruction can be reached, it can be ignored.

Conceptually, this check is implemented by tracking all control and data dependencies for all instructions reachable from P given state S. By definition these list the complete set of locations that could cause a change in path behavior. Thus, if a location l is not contained in any such set, then there is no value it can be assigned that could cause an uncovered instruction to be executed. It can thus be discarded from the state cache entry.

The main implementation challenge is the sheer number of these dependencies. This can be mitigated somewhat by exploiting the fact that statements are run whenever a the edge of the conditional branch controlling them is taken, and thus one need only track the dependencies of conditional expressions. However, scalability is still a challenge. One intuitive way to see why is to notice that security tools that try to do full tracking of tainted (untrusted) values have to do a similar analysis --- the explosion of transitively tainted locations caused when tracking control flow dependencies (``implicit information flow") has meant many tools simply give up on the problem and only track data dependencies. As a result, much of the work has been on devising techniques and data structures that can handle this enormous number of dependencies.

This optimization was evaluated by using it to execute the 70+ Unix utilities that comprise CoreUtils suite. These programs form the core user-level environment installed on almost all Unix systems. They are used daily by millions of people, bug fixes are handled promptly, and new releases are pushed regularly. The breadth of functions they perform means that this system cannot be a ``one trick pony" special-cased to one application class. Moreover, the heavy use of the environment stress tests this system where symbolic execution has historically been weakest.

When path pruning is added to KLEE it gets an average speedup of 182X (median is 12.8X), with a max speedup of 1143X. Statement coverage of the applications increases by an average of 6.8% (median is 5.9%). The two figures break these down on a per application basis.

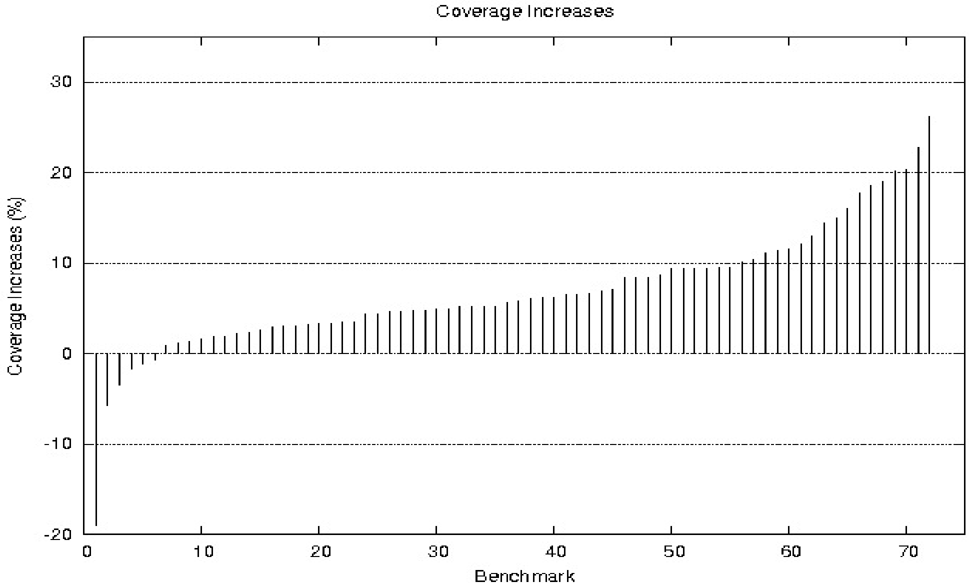


Figure 4: Coverage Increase in KLEE with Path Pruning

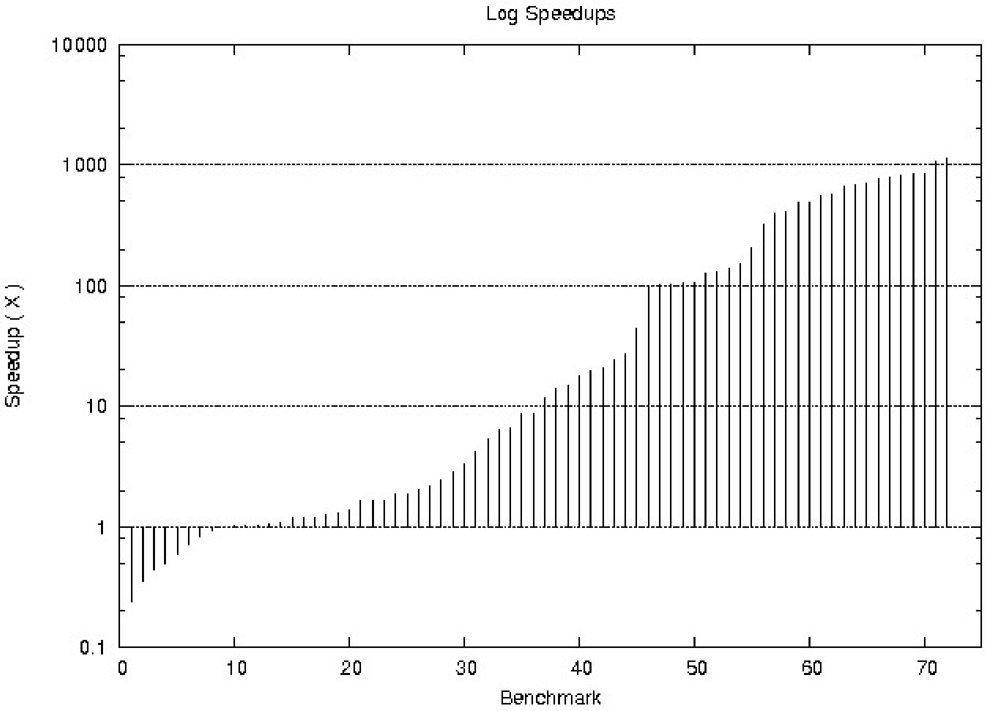


Figure 5: Speedup Increase in KLEE with Path Pruning

4.1.1.3 Path Merging

Assume there are two paths through code:

```
x = symbolic();
...
if(x)
        y = 1;
else
        y = 2;
```

In normal KLEE, if both paths are feasible, this will produce two states: {x != 0, y = 1} and { x = 0, y = 2 }. With merging one state results{x != 0 ==> y = 1 \/ x = 0 ==> y = 2}. This can be a big win, since many states can be executed simultaneously. It can also be a big loss for two main reasons: (1) many concrete locations become symbolic (y in the example above) and (2) the constraints generated are much more complex and different than in the past (in particular, disjunctions are expensive). A significant amount of time has been spent doing low-level optimization of the resultant constraints to try to beat them into something more reasonable. The result is nearly a 10x performance improvement in the best case, with more modest results on other programs.

4.1.2 Binary KLEE

At a high level, the binary symbolic execution system uses Valgrind's translation library, VEX, to translate x86-64 machine code a basic block at a time to the internal representation of a heavily modified copy of the KLEE symbolic execution system (LLVM bitcode from the LLVM compiler).

Like the base system KLEE, the tool represents each path through code as a separate address space. It extends this "guest state" with memory map bookkeeping, the current state of the guest's machine registers, symbol mappings, and environment details. The program it interprets (the guest) is loaded so that every pointer in the guest state matches the equivalent native process through low-overhead translations. Mostly these translations are the identity; for some, however, one must add a constant offset (specific cases are discussed later in the paper). Since both the guest program and KLEE occupy the same address space, conflicts of text addresses are avoided by using the common trick of linking the system's executable at an uncommon base address.

Shoehorning a dynamic binary translator (DBT) into KLEE requires an updated executor. This new executor is referred to as the DBT bridge. Originally, KLEE was designed to operate on static LLVM bitcode files; all the code is immediately available on initialization. On the other hand, DBTs are designed to incrementally uncover basic blocks on demand. Since precise disassembly of x86 code is an intractable problem, KLEE's execution model had to be updated to support a stream of basic blocks. The DBT bridge crosses the gap between the DBT and symbolic execution. The bridge dispatches basic blocks, wrapped as LLVM functions, to the KLEE LLVM symbolic interpreter that inspects the function's return value to find the next address (or system call) to execute.

The KLEE DBT uses VEX as a baseline CPU decoder, but ignores the translated host machine instructions in favor of the intermediate VEX basic block format. The VEX intermediate representation is a single assignment language with support for arbitrary expressions. Basic blocks in this representation readily translate into LLVM (which is also single assignment). Each basic block is translated into an LLVM function which takes a VEX register file as input and returns the next guest address to execute. At this stage, a concrete DBT executor can be implemented by using the built-in LLVM just-in-time compiler to emit the blocks as (binary) machine code.

4.1.2.1 Memory Translation

If address space layout randomization is enabled on the host machine, most of the memory mappings within the guest process will not conflict with the host process. With no conflicts, the native process may be safely copied in-place into the host process to form the guest address space. Two notable exceptions to layout randomization are executable text sections and the virtual system call page.

On a typical system, the fixed mapping from the guest's text section conflicts with the host process. Without intervening at compile-time, executables are at the mercy of the default linker script, which links all executables to the same base address. This conflict is resolved through the popular (*e.g.,* used by Pin and QEMU) binary translator linker trick of changing the host text section's base address to an unusual memory location.

Linux optimizes several system calls through a special kernel-owned, user-visible *vsyscall* page. The vsyscall page is read-only mapped at a fixed kernel address that changes from run to run; relocating and copying the vsyscall page is a challenge. As an aside, this mapping has properties similar to the "u-area" found on other Unix systems. Although the data in the page may change from run to run, forking off and ptrace'ing the guest program preserves the vsyscall page of the host process. Hence, reconciling distinct vsyscall pages at this stage is unnecessary.

4.1.2.2 Robustness

A key goal of this work is to ensure binary-KLEE is a robust tool suitable for real world use, rather than a cobbled-together, hodge-podge of hacks that can (barely) eek out some results for a publication. As part of this, focus has been placed on running real programs from the beginning. The current nightly regressions use roughly 100 programs from "/usr/bin" and "/usr/sbin" from a current Linux distribution. Larger programs such as firefox have also been executed (albeit with low coverage). The results of the most recent nightly regressions can be seen at:

http://bingularity.org/kleemc-bot/2012-01-09/err.xml
http://bingularity.org/kleemc-bot/2012-01-09/report.xml

Much of the work on binary KLEE has been on coming up with ways to turn KLEE on itself so that errors in the code can be found/verified. These checks have found tens of errors in the Valgrind code that can be used for instruction decoding along with a multitude of errors in KLEE itself (and in the constraint solvers it uses).

4.1.3 Patch validation

This work item was to verify that bug patches only removed crash-causing errors, rather than changed functionality. The intuition is that one can use symbolic execution to verify on a path-by-path basis that all non-crashing paths in the old code have identical behavior in the new.

The initial work on this item has been to verify "functional equivalence" where it has been verified that two functions that purport to implement the same interface do so. While this sounds different than patches, it is largely the same since you can view a patch as a function. The infrastructure built is the same needed for patches.

Historically, code verification has been hard. Thus, implementers rarely make any effort to do it. UC-KLEE, a modified version of KLEE designed to make it easy to verify that two routines are equivalent, has been built. This ability is useful in many situations, such as checking: different implementations of the same (standardized) interface, different versions of the same implementation, optimized routines against a reference implementation, and finding compiler bugs by comparing code compiled with and without optimization.

Previously, cross checking code that takes inputs with complex invariants or complicated data structures required tediously constructing these inputs by hand. From experience, the non-trivial amount of code needed to do so can easily dwarf the size of the checked code (*e.g.,* as happens when checking small library routines). Manual construction also leads to missed errors caused by over-specificity. For example, when manually building a linked list containing symbolic data, should it have one entry? Two? A hash table should have how many collisions and in which buckets? Creating all possible instances is usually difficult or even impossible. Further, manually specifying pointers (by assigning the concrete address returned by *malloc*) can limit paths that check relationships on them, such as when an if-statement checks whether one pointer is less than another. In general, if input has many constraints, a human tester will miss one.

In contrast, using the tool is easy: rather than requiring users to manually construct inputs or write a specification to check code against, they simply give the tool two routines (written in raw, un-annotated C) to cross check. The tool automatically synthesizes the routines' inputs (even for rich, nested data structures) and systematically explores a finite number of their paths using sound, bit-accurate symbolic execution. It verifies that the routines produce identical results when fed identical inputs on these explored paths by checking that they either (1) write the same values to all escaping memory locations or (2) terminate with the same errors. If one path is correct, then verifying equivalence proves the other is as well. If the tool terminates, then with some caveats it has verified equivalence up to a given input size.

Because UC-KLEE leverages the underlying KLEE system to automatically explore paths and reason about all values feasible on each path, it gives guarantees far beyond those of traditional testing, yet it often requires less work than writing even a single test case. It is shown that the approach works well even on heavily-tested code, by using it to cross check hundreds of routines in two mature, widely-used open-source LIBC implementations, where it:

- Found numerous interesting errors.
- Verified the equivalence of 300 routines (150 distinct pairs) by exhausting all their paths up to a fixed input size (8 bytes).
- Got high statement coverage: The lowest median coverage for any experiment was 90% and the rest were 100%.

A final contribution is a simple, novel trick for finding bugs in the compiler and checking tool by turning the technique on itself, which is used to detect a serious LLVM optimizer bug and numerous errors in UC-KLEE.

These results are described in more detail in the 2011 CAV paper "Practical, low-effort equivalence verification of real code".

4.1.4 Prophylactic KLEE Checks

This deliverable has been de-emphasized in order to throw more resources at building the binary version of KLEE. This goal was deemed more relevant for the Phase 1 effort than building prophylactic checks.

4.1.5 Automatic Race Avoidance

Two trends have caused multithreaded programs to become pervasive and critical. The first is a hardware trend: the rise of multicore computing. For years, sequential code enjoyed automatic speedup as computer architects steadily made single-core multiprocessors faster. Recently, however, this "free lunch is over": power and wire-delay constraints have forced microprocessors into multicore designs, and adding more cores does not automatically speed up sequential code. Thus, developers, including those working for various government agencies, are writing more and more multithreaded code.

The second trend is a software one: the coming storm of cloud computing. More and more services, including many traditionally offered on desktops (e.g., word processing), are now served from distributed "clouds" of servers to meet the current computing demands for high scalability, always-on availability, everywhere connectivity, and desirable consistency. These services are also getting ever richer and more powerful---and thus computation and data intensive. To cope with this massive workload, practically all services today employ threads to increase performance.

Unfortunately, despite the system's increasing reliance on multithreaded programs, they remain extremely difficult to write. This difficulty has led to many subtle but serious *concurrency vulnerabilities* such as race conditions in real-world multithreaded programs. Some of these errors have killed people in the Therac 25 incidents and caused the 2003 Northeast blackout. Multithreaded programs are the most widespread parallel programs, yet many luminaries in computing consider parallel programming one of the top challenges facing computer science. As John Hennessy once put: "*when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced.*"

Just as vulnerabilities in sequential programs can lead to security exploits, concurrency vulnerabilities can similarly compromise security and lead to *concurrency attacks*. The recent study of real concurrency vulnerabilities in the CVE database shows that these vulnerabilities are very dangerous: they allow attackers to corrupt arbitrary program data, inject malicious code, and escalate privileges. Worse, in addition to being directly exploited by attackers, concurrency vulnerabilities also compromise key defense techniques that were once trusted. For instance, consider an information flow tracking mechanism that tracks whether each piece of program data is classified or not using a metadata tag. An attacker may exploit a race condition on program data to make the data and the tag inconsistent, thus evading the information flow tracking mechanism.

A key reason that multithreaded programs are so difficult to get right is *non-determinism*: different runs of a parallel program may compute different results and show different behaviors, depending on how the parallel threads of executions interleave. The interleaved parallel execution is called a schedule. A typical parallel program exhibits many possible schedules across runs, due to variations in hardware, OS scheduling, input, and timing factors. Ideally all schedules will lead to a single correct result, but in practice, these schedules lead to different results, some of which are incorrect. Non-determinism makes it difficult to write, understand, maintain, test, debug, and verify parallel programs. It deprives parallel computation of the most essential and appealing properties of sequential computation: understandability, repeatability, predictability, and determinism. For instance, non-determinism makes it virtually impossible to rerun a large parallel computation and reproduce the exact results as the original run. Thus, it is impossible for third-party researchers to independently verify results from parallel computation, which is crucial for fields such as nuclear, biological, and medical sciences where correctness is a life-or-death matter. Similarly, testing, the predominant industrial method to assure software quality tends to cover few schedules, leaving many untested, buggy schedules running in the wild. Debugging is quite challenging because developers have to reproduce these exact buggy schedules on their machines.

As part of the thrust to automatically avoid races, a number of techniques and prototypes exploring and demonstrating automatic race avoidance were investigated and developed.

4.1.5.1 Tern

Tern uses *schedule-memoization* to memoize past working schedules and reuse them on future inputs, making program behaviors stable across different inputs. For instance, it can make programs to hang on to the good schedules, and avoid potential errors in the unknown schedules. A second novelty in Tern is the idea of windowing that extends schedule memoization to server programs by splitting continuous request streams into windows of requests. The Tern implementation runs on Linux. It operates as user-space schedulers, requiring no changes to the OS and only a few lines of changes to the application programs.

Tern was evaluated on a diverse set of 14 programs, including two server programs Apache and MySQL, a parallel compression utility PBZip2, and 11 scientific programs in SPLASH2. The workload included a Columbia CS web trace and benchmarks used by Apache and MySQL developers. Results show that *(1)* Tern is easy to use. For most programs, only a few lines were modified to adapt them to Tern. *(2)* Tern enforces stability across different inputs. In particular, it reused 100 schedules to process 90.3% of a 4-day Columbia CS web trace. Moreover, while an existing DMT system made three bugs inconsistently occur or disappear, depending on minor input changes, Tern always avoided these bugs. *(3)* Tern has reasonable overhead. For nine out of fourteen evaluated programs, Tern has negligible overhead or improves performance; for the other programs, Tern has up to 39.1% overhead.
(4) Tern makes threads deterministic. For twelve out of fourteen evaluated programs, the schedules Tern memoized can be deterministically reused barring some assumptions.

4.1.5.2 PEREGRINE

PEREGRINE improved upon Tern by removing manual annotations and by efficiently making threads deterministic. The key insight is that races tend to occur only within minor portions of an execution, and a dominant majority of the execution is still race-free. Thus, one can resort to a mem-schedule only for the “racy” portions and enforce a sync-schedule otherwise, combining the efficiency of sync-schedules and the determinism of mem-schedules. These combined schedules are called hybrid schedules. Based on this insight, PEREGRINE has been built, an efficient deterministic multithreading system. When a program first runs on an input, PEREGRINE records an execution trace. It then relaxes this trace into a hybrid schedule and reuses the schedule on future compatible inputs efficiently and deterministically. PEREGRINE further improves efficiency with two new techniques: determinism-preserving slicing to generalize a schedule to more inputs while preserving determinism, and schedule-guided simplification to precisely analyze a program according to a specific schedule.

PEREGRINE was evaluated on a diverse set of 18 programs, including the Apache web server; three desktop programs, such as PBZip2, a parallel compression utility; implementations of 12 computation-intensive algorithms in the popular SPLASH2 and PARSEC benchmark suites; and racey, a benchmark with numerous intentional races for evaluating deterministic execution and replay systems. Results show that PEREGRINE is both deterministic and efficient (executions reusing schedules range from 68.7% faster to 46.6% slower than nondeterministic executions); it can frequently reuse schedules for half of the programs (e.g., two schedules cover all possible inputs to PBZip2 compression as long as the number of threads is the same); both its slicing and simplification techniques are crucial for increasing schedule-reuse rates, and have reasonable overhead when run offline; its recording overhead is relatively high, but can be reduced using existing techniques; and it requires no manual efforts except a few annotations for handling server programs and for improving precision.

4.1.5.3 RACEPRO

RACEPRO is a tool for detecting process races. Process races occur when multiple processes access shared operating system resources, such as files, without proper synchronization. The first study of real process races and the first system designed to detect them has been presented. A study of hundreds of applications shows that process races are numerous, are difficult to debug, and constitute a real threat to reliability. To address this problem, RACEPRO was created, a system for automatically detecting these races. RACEPRO checks deployed systems in-vivo by recording live executions then deterministically replaying and checking them later. This approach increases checking coverage beyond the configurations or executions covered by software vendors or beta testing sites. RACEPRO records multiple processes, detects races in the recording among system calls that may concurrently access shared kernel objects, then tries different execution orderings of such system calls to determine which races are harmful and result in failures. To simplify race detection, RACEPRO models under-specified system calls based on load and store micro-operations. To reduce false positives and negatives, RACEPRO uses a replay and go-live mechanism to distill harmful races from benign ones. RACEPRO was implemented in Linux, shown that it imposes only modest recording overhead, and used it to detect a number of previously unknown bugs in real applications caused by process race.

4.1.6 Binary-level monitor

As part of this thrust, a number of techniques and prototypes exploring and demonstrating a binary-level monitor were investigated and developed.

4.1.6.1 libDFT

Dynamic data flow tracking (DFT), also referred to as information flow tracking, is a well known technique that deals with the tagging and tracking of “interesting” data as they propagate during pro- gram execution. DFT has many uses, such as analyzing malware behavior, hardening software against zero-day attacks (e.g., buffer overflow, format string), detecting and preventing information leaks, and even debugging software misconfigurations. From an architectural perspective, it has been integrated into full system emulators and virtual machine monitors, retrofitted into unmodified binaries using dynamic binary instrumentation, and added to source codebases using source-to-source code transformations. Proposals have also been made to implement it in hardware, but they had little appeal to hardware vendors.

Previous studies utilized DFT to investigate the applicability of the technique into a particular domain of interest, producing their own problem-specific and ad hoc implementations of software-based DFT that all suffer from one or more of the following issues: high overhead, little reusability (i.e., they are problem specific), and limited applicability (i.e., they are not readily applicable to existing commodity software). For instance, LIFT and Minemu use DFT to detect security attacks. While fast, they do not support multithreaded applications (the first by design). LIFT only works with 64-bit binaries, while Minemu only with 32-bit binaries, featuring a design that requires extensive modifications to support 64-bit architectures. More importantly, they focus on a single problem domain and cannot be easily modified for use in others.

More flexible and customizable implementations of fine-grained DFT have also failed to provide the research community with a practical and reusable DFT framework. For example, Dytan focuses on presenting a configurable DFT tool that supports both data and control flow dependencies. Unfortunately, its versatility comes at a high price, even when running small programs with data flow dependencies alone (control flow dependencies further impact performance). For instance, Dytan reported a 30x slow- down when compressing with gzip, while LIFT reports less than 10x. Although the experiments may not be directly comparable, the significant disparity in performance suggests that the design of Dytan is not geared towards low overhead.

A practical dynamic DFT implementation needs to address all three problems listed above, and thus it should be concurrently fast, reusable, and applicable to commodity hardware and software. libdft was developed, a meta-tool in the form of a shared library that implements dynamic DFT using Intel’s Pin dynamic binary instrumentation framework. libdft’s performance is comparable or better than previous work, incurring slowdowns that range between 1.14x and 6.03x for command-line utilities, while it can also run large server applications like Apache and MySQL with an overhead ranging between 1.25x and 4.83x. In addition, it is versatile and reusable by providing an extensive API that can be used to implement DFT-powered tools. Finally, it runs on commodity systems. The current implementation works with x86 binaries on Linux, and it will be extended to run on 64-bit architectures and the Windows operating system (OS). libdft introduces an efficient, 64-bit capable, shadow memory, which represented one of the most serious limitations of earlier works, as flat shadow memory structures imposed unmanageable memory space overheads on 64-bit systems, and dynamically managed structures introduce high performance penalties. More importantly, libdft supports multi-process and multithreaded applications, by trading off memory for assurance against race, and it does not require modifications to programs or the underlying OS.

A novel optimization approach to DFT was developed, based on combining static and dynamic analysis, which significantly improves its performance. This methodology is based on separating program logic from taint tracking logic, extracting the semantics of the latter, and representing them using a Taint Flow Algebra. Multiple code optimization techniques were applied to eliminate redundant tracking logic and minimize interference with the target program, in a manner similar to an optimizing compiler. Drawing on the rich theory on basic block optimization and data flow analysis, done in the context of compilers, the safety and correctness of the algorithm using a formal framework can be argued.

The correctness and performance of the methodology was evaluated on libdft, and showed that the code generated by this analysis behaves correctly when performing dynamic taint analysis. The performance gains achieved by the various optimizations were evaluated using several Linux applications, including commonly used command-line utilities (bzip, gzip, tar, scp, etc.), the SPEC CPU 2000 benchmarks, the MySQL database server, the runtimes for the PHP and JavaScript languages, and web browsers. Results indicate performance gains as high as 2.23×, and an average of 1.72× across all tested applications.

4.1.6.2 Virtual Application Partitioning

Virtual partitioning was developed, a technique that enables application of diverse defensive mechanisms to manage the attack surface of applications. This approach is based on the observation that applications can be separated into parts that face different types of threats, or suffer dissimilar exposure to a particular threat, because of external events or innate properties of the software. The goal is to use these virtual partitions to apply a multitude of security techniques without inflicting cumulative overheads, deploying what is needed, when it is needed. As a starting point, focus was placed on virtually partitioning applications based on user authentication, and selectively applying distinct protection techniques on its partitions. A methodology was introduced that enables automatic determination of the authentication point of an application with little or no supervision, and without the need for source code. A virtual partitioning tool that operates on binary-only software, and at runtime splits the execution of an application in its pre- and post-authentication segments, based on the identified authentication point was also developed. Different protection mechanisms, such as dynamic taint analysis and instruction-set randomization, can be applied on these partitions.

This is the first work on virtual partitioning. The approach was applied on well-known server applications, such as OpenSSH, MySQL, Samba, Pure-FTPd, and more. These services were set up to use different authentication schemes, and demonstrated that it is possible to effectively and automatically determine their authentication points. Moreover, a particular security management scenario was run to demonstrate the applicability of the solution on existing software. DTA and ISR were enforced on the pre-authentication part of the servers, and switch to ISR, or disable all protection, after successful authentication. This not only minimizes the attack surface to otherwise unprotected software, but also does so with significantly lower performance cost. Note that the same mechanisms can be applied in the reverse order, which enables protection of applications against different type of threats (e.g., code-injection attacks and sensitive information leakage). Results show that, in the first set up, one can greatly reduce the user-observable overhead of DTA, compared with having it always operational, up to 5x for CPU-bound applications and with negligible overhead for I/O intensive applications. However, other configurations (i.e., combinations of mechanisms) may not enjoy the same improvements in performance.

4.1.6.3 SecondWrite

A prototype tool for inserting security features against low-level software attacks into third party, proprietary or otherwise binary-only software was developed. This was motivated by the inability of software users to select and use low-overhead protection schemes when source code is unavailable to them, by the lack of information as to what (if any) security mechanisms software producers have used in their toolchains, and the high overhead and inaccuracy of solutions that treat software as a black box. This approach is based on SecondWrite, an advanced binary rewriter that operates without need for debugging information or other assist. Using SecondWrite, a variety of defenses into program binaries are inserted. Although the defenses are generally well known, they have not generally been used together because they are implemented by different (non-integrated) tools. Such mechanisms were developed without source code availability for the first time. The effectiveness and performance impact of this approach were experimentally evaluated. This showed that it stops all variants of low-level software attacks at a very low performance overhead, without impacting original program functionality. However, because SecondWrite works as a static binary rewriter, it is currently limited with respect to the size and complexity of the programs that it can handle.

4.1.6.4 Symbiotes

A large number of embedded devices on the Internet, such as routers and VOIP phones, are typically ripe for exploitation. Little to no defensive technology, such as AV scanners or IDS's, is available to protect these devices. A host-based defense mechanism was developed, called Symbiotic Embedded Machines (SEM), which is specifically designed to inject intrusion detection functionality into the firmware of the device. A SEM or simply the Symbiote may be injected into deployed legacy embedded systems with no disruption to the operation of the device. A Symbiote is a code structure embedded *in situ* into the firmware of an embedded system. The Symbiote can tightly co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The Symbiote is stealthily embedded in a randomized fashion within an arbitrary body of firmware to protect itself from removal. The operation of a generic whitelist-based rootkit detector Symbiote injected in situ into Cisco IOS with negligible performance penalty and without impacting the routers functionality was demonstrated. A MIPS implementation of the Symbiote was ported to ARM and injected into a Linux 2.4 kernel, allowing the Symbiote to operate within Android and other mobile computing devices. The use of Symbiotes represents a practical and effective protection mechanism for a wide range of devices, especially widely deployed, unprotected, legacy embedded devices.

4.1.7 Lightweight Program Confinement

In Phase I, the automation of the execution and data collection of binary programs inside the lightweight containers was completed. To achieve that, an architecture that enables the efficient and completely automated execution of software, collection of the observable information, and the post processing of the produced logs for analysis was created. Figure 1 below shows the overall automation architecture.

Each application contained in the database is run once (or possibly more than once) inside a separately instantiated container. The container offers a standard and clean execution environment. In addition, it enables tracking the application behavior and provides a uniform and robust execution environment for the data collection because the starting state, application configuration, and input are recorded in the form of a system snapshot. All operations triggered by the application - system calls, all memory usage, all communications towards other applications or via the network - are recorded and form the so called *standard profile* of the application. Moreover, this approach allows running several application containers in parallel on the same machine, thus leveraging the high computational power of servers to produce a quicker and more responsive analysis.

As part of this effort, the coding of the Framework core and the design of the database schema for the Application Database were completed. In addition, the system snapshot architecture using UnionFS was automated. UnionFS is a stackable unification file system, able to merge the contents of several directories (branches), while keeping their physical content separate. UnionFS allows any mix of read-only and read-write branches, as well as insertion and deletion of branches.

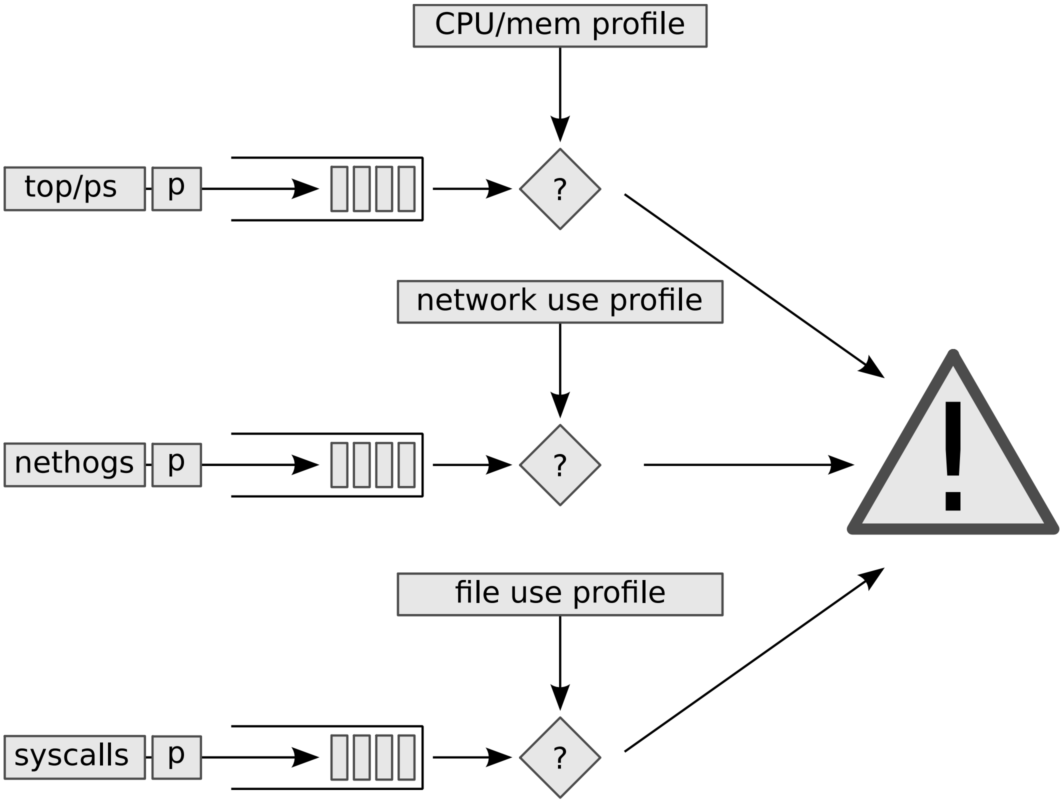


Figure 6: Resource Monitor & Policing

4.1.7.1 Resource Monitoring

Resource monitoring aims to collect resource usage information from different sources (see attached image) and process them in a homogeneous way that allows constant profile checking. In this way anomalies or policy infringement can be detected at runtime.

The basic monitoring structure needed to collect data was implemented. The queuing system is now able to gather data from any source through a file interface. Moreover, the full structure minimizes CPU utilization and a buffering system avoids dynamic memory allocation. In this way it was possible to implement a lightweight interface that will be reused for all data sources. Moreover, the parsing mechanism necessary for CPU and memory monitoring was implemented. It is thus now possible to dynamically track usage for the two resources and do simple runtime statistics (min, max, avg).

A preliminary study on *nethogs* and other network analysis tools revealed that network-monitoring time granularity is bound to hit a lower limit at around 2-3 seconds. In other words, reading data faster than that introduces a lot of noise and can lead to imprecise measurements. On the contrary, this system can track CPU and memory with a granularity well under the single second. This discrepancy is still under study**.**

4.1.7.2 Resource Monitoring - Network Traffic

After several testing rounds, already existing network traffic monitoring tools (e.g., nethogs) showed intrinsic limits in their implementation. Typically, the allowed granularity in traffic measuring was never appropriate to the requirements of the current project. Furthermore, the statistical analysis provided by existing tools was too limited, compared to the required insights. For these reasons the implementation of an ad-hoc network filter was begun. The tool will behave like a daemon, running in background and monitoring the traffic coming from several containers. It will be built on top of *iptables*, and will also integrate the capability of injecting new packets when appropriate.

4.1.7.3 Resource Monitoring - Memory and CPU

The problem of tracking the child processes that are spawned by the original monitored process was addressed by implementing a per-user filtering mechanism. Each container hosts a dummy user, created only for monitoring purposes. The monitored process is executed with permission elevation into the dummy user, and statistics are then extracted by filtering and adding all the activity generated by it. This way, even if a malicious attacker is able to exploit an existing vulnerability to download and execute a binary, the resource count will still keep track of the new process. We have implemented and tested the harness that is needed for the user filtering capability. Also, the memory and CPU monitoring tools have undergone pervasive testing to verify their stability and scalability.

4.1.7.4 Synchronizing Outgoing Packets among Containers

MINESTRONE’s design inherently poses a significant network synchronization problem. Since each container will be running the same program, they will all produce the same network traffic. However, the rate at which traffic is produced will vary with the particular technology employed on each container. Additionally, external stimuli will expect to communicate with a single host, not each of the containers. Thus, one must be able to reduce outgoing traffic to a single stream, and then replicate incoming traffic to all of the containers.

To address the issue of outgoing traffic, NFQUEUE was used, targeting with *iptables*. A user-space program continuously reads new packets as they enter the queue. Then, the payload of each packet is hashed and stored in a table. Next, when another container sends a packet with the same payload hash, a 'times sent' variable is incremented. When that variable reaches the number of containers (i.e., when all containers have sent the same packet), the first packet is let through and the rest of the packets are dropped. However, because each technology may alter network behavior, a one-second window is established in which to receive packets from all containers. If a container does not send a packet in that window, the packet is sent anyway. This prevents network deadlock since a particular container may unexpectedly stop network traffic while we are waiting for it to send a packet. There are obvious performance implications with this approach, since it essentially limits network capacity to that of the slowest container.

A hash table has been implemented using the MD5 hash to generate keys, and includes a dynamically allocated array in which to store them. When the load factor becomes too large, the hash table increases in size and rehashes data. A user-space program is also used that can read packets sent to a queue. The next step is to hash each packet, check for the correct number of copies, and then send the packet through.

4.1.7.5 Replicating Incoming Packets to Containers

Attempts at being able to replicate packets incoming to a container to the other containers are being made. Research was done into the quickest/easiest way to implement this functionality and later work towards a kernel module in order to transition more easily towards keyboard and mouse control as well. Iptables does not seem to be able to provide the functionality needed. The main issue is being able to properly NAT each packet to its rightful destination separately. The easiest way to reach this goal is likely through a packet injection program. This program will receive all incoming packets destined for the "main" container (the one driving the conversation), rewrite the destination information and inject the new packet onto the wire.

The first thing investigated was Pycap, a Python module for sniffing/injecting of packets. It provided the exact functionality that was needed but it was discovered that the module would not build on the system, most likely due to the fact that it is from 2003 and has not been updated since. Investigation then began on Scapy. This program also has the ability to sniff, rewrite packet data, and inject packets into the network. A version to be able to replicate the packets using Scapy was also prototyped. The module will listen on the host at all packets coming in and check to see if they are destined for the openvz container. If it is, the destination information (MAC and IP) will be changed for each container and subsequently injected onto the wire.

4.1.8 Function-level Monitoring

When a new application needs to be profiled this framework executes it inside an isolated lightweight containers. Each container is an instance of an OpenVZ virtual machine, and guarantees a common and clean starting point: every container starts from a standardized snapshot of the system, and the execution flows through a controlled path.

The usage of containers creates a powerful vantage point for:

- IO operations: all accessed or modified files are tracked. Moreover, all created files are saved for further analysis.
- Network operations: a complete dump of the network operation inside the container is easily performed through standard tools like tcpdump.

The usage of separate containers exposes the application to a set of different asynchronous events: simulated user interactions, network responses, and naturally occurring interrupts to cite a few. This was leveraged to observe the behavior of the application in several, different scenarios. Once all data are gathered, the common behavior is extracted, and it is used to create the standard profile.

Furthermore, a complete environment was set up able to run in a streamlined fashion several applications and automatically create the logs of the observed operations. So far focus was placed on the completeness of the system from the elaboration point of view. For each application the framework goes through the following steps:

- A new instance of a container is created from a saved template. If the system is already at its full capacity, the request is put on hold until the number of running containers decreases;
- The initial image of the system is mounted, through the use of UnionFS;
- The application and the scripts necessary for its execution are copied in the mounted directory;
- The application code is executed within the container(s);
- The container is stopped after a timeout is reached or if the application terminates, whatever happens first;
- All the logs are copied in a result directory;
- All the recorded network activities are copied in a result directory;
- All the modified files are copied in a result directory;
- The result directory is compressed;
- The image of the system is unmounted

4.1.8.1 Preliminary Results -Generating Normality Models

The container-based and session-separated architecture not only enhances the security but also provides the isolated information flows that are separated in each container session. It allows identification of the mapping between different components For instance, in typical 3-tiered web server architecture, the web server receives HTTP requests from user clients and then issues SQL queries to the database server to retrieve and update data. These SQL queries are causally dependent on the web request hitting the web server. It is desirable to model such causal mapping relationships of all legitimate traffic so as to detect abnormal/attack traffic.

In practice, it is impossible to build such mapping under a classic 3-tier setup. Although the web server can distinguish sessions from different clients, the SQL queries are mixed and all from the same web server. It is impossible for a database server to determine which SQL queries are the results of which web requests, much less to find out the relationship between them. Even if one knew the application logic of the web server and were to build a correct model, it would be impossible to use such a model to detect attacks within huge amounts of concurrent real traffic unless one had a mechanism to identify the pair of the HTTP request and SQL queries that are causally generated by the HTTP request. However, within the container-based web servers, it is a straightforward matter to identify the causal pairs of web requests and resulting SQL queries in a given session. Moreover, as traffic can easily be separated by session, it becomes possible to compare and analyze the request and queries across different sessions.

To that end, sensors were placed at both sides of the servers. At the web server, the sensors are deployed on the host system and cannot be attacked directly since only the virtualized containers are exposed to attackers. The sensors will not be attacked at the database server either, as it is assumed that the attacker cannot completely take control of the database server. In fact, it is assumed that the sensors cannot be attacked and can always capture correct traffic information at both ends.

Once the mapping model is built, it can be used to detect abnormal behaviors. Both the web request and the database queries within each session should be in accordance with the model. If there exists any request or query that violates the normality model within a session, then the session will be treated as a possible attack.

4.1.9 Accurate Anomaly Detection

Recent advances in offensive technologies targeting embedded systems have shown that the stealthy exploitation of high-value embedded devices such as router and firewalls is indeed feasible. However, little to no host-based defensive technology is available to monitor and protect these devices, leaving large numbers of critical devices defenseless against exploitation. A method of augmenting legacy embedded devices, like Cisco routers, with host-based defenses in order to create a stealthy, embedded sensor-grid capable of monitoring and capturing real-world attacks against the devices which constitute the bulk of the Internet substrate was devised. Using a software mechanism that is called the Symbiote, a white-list based code modification detector is automatically injected *in situ* into Cisco IOS, producing a fully functional router firmware capable of detecting and capturing successful attacks against itself for analysis. Using the Symbiote-protected router as the main component, a sensor system was designed which requires no modification to existing hardware, fully preserves the functionality of the original firmware, and detects unauthorized modification of memory within 150 ms. It is believed that it is feasible to use the techniques described in this paper to inject monitoring and defensive capability into existing routers to create an early attack warning system to protect the Internet substrate.

Recent studies suggest that large populations of vulnerable embedded devices on the Internet are ripe for exploitation [8]. However, examples of successful exploits against such devices are rarely observed in the wild, despite the availability of proof-of-concept malware, known vulnerabilities and high monetization potential. It is posited that the inability to monitor embedded devices for malware installation is a factor in this phenomenon. When deployed throughout the Internet substrate, the sensor system discussed in this paper will provide visibility into black-box embedded devices, allowing capture and analysis of the exploitation of embedded devices in real time.

As a first step to show feasibility, a general method of transforming existing legacy embedded devices into exploitation detection sensors was demonstrated. Cisco firmware and hardware were used as the main demonstrative platform in this paper. However, the techniques described are not specific to any particular operating system or vendor, and can be directly applied to many other types of embedded devices.

In order to detect and capture successful attacks against Cisco routers for analysis, a system that automatically injects generic whitelist-based anti-rootkit functionality into standard IOS firmwares was engineered. Once injected, the augmented router firmware can be loaded onto physical Cisco routers, essentially transforming such devices into highly interactive router honeypots. The resulting devices are fully functional, and can be deployed into production environments. The main challenge of creating an embedded device honeypot rests with the difficulties of injecting arbitrary detection code into proprietary, closed-source, embedded devices with complex and undocumented operating systems. A Symbiote, along with its payload, is injected *in situ* into an arbitrary host binary, in this case, Cisco IOS. The injection is achieved through a generic process that is agnostic to the operating environment of the host program. In general, Symbiotes can inject arbitrary host-based defenses into black-box embedded device firmwares. The unique capabilities of the Symbiote construct allow overcoming the complexities of injecting generic exploitation detection code into what is essentially an unknown black-box device. The original functionality of resulting Symbiote-injected embedded device firmware remains unchanged. A portion of the router's computational resources is diverted to a proof of concept Symbiote payload, which continuously monitors for unauthorized modifications to any static areas within the router's memory address space, a key side effect of rootkit installation. The portion of the CPU diverted to the Symbiote's payload is a configurable parameter, and directly affects the performance of the Symbiote payload: in this case, the detection latency of any unauthorized modification.

A monitoring system is constructed around the main component of the system, the Symbiote-injected IOS image. The Symbiote within the IOS firmware simultaneously performs checksums on all protected regions of the router's memory while periodically communicating with an external monitor via a covert channel. In the event of an unauthorized memory modification within the router, the Symbiote will raise an alarm to the external monitor, which then triggers the capture and analysis component of the system.

The sensor system has three components; a Symbiote-protected router, a monitoring station, and a capture and analysis system that automatically collects and analyzes forensics data once an alarm is triggered. The Symbiote within the IOS firmware simultaneously performs checksums on all protected regions of the router's memory while periodically communicating with an external monitor via a covert channel. In the event of an unauthorized memory modification within the router, the Symbiote will raise an alarm to the monitor, which then triggers the capture and analysis component of the system.

The proposed exploitation detection sensor can be deployed in one of at least three ways; natively, emulated within a general-purpose computer, or as a shadow replica for a production device. The implementation of the monitoring station and capture and analysis engine changes depending on how the Symbiote-injected router firmware is executed natively on embedded hardware or emulated on a general-purpose computer.

When deployed natively, the monitor and capture components are integrated into the Symbiote payload and injected directly into Cisco hardware, producing a standalone sensor. When the detection payload raises an alarm, the Symbiote immediately triggers the core dump functionality from within IOS. This causes the bulk of the router's execution state to be captured and transferred via FTP or TFTP.

When deployed as an emulated sensor, using Dynamips for example, the monitoring and capture components of the sensor are implemented within the emulator. This reduces the footprint of the Symbiote and allows performance of more sophisticated capture and analysis on the server running the emulation. For example, Dynamips was modified to continuously monitor a region of the router's memory for an encoded marker, which is set by the Symbiote payload only when an alarm is raised.

For testing purposes, a portion of the text that is printed when the "show version" command is invoked was modified. In practice, many better covert channels can be used to communicate between the Symbiote and the router emulator.

In order to transform large populations of embedded devices into massive embedded exploitation sensor grids, the native deployment is the most efficient and practical. For the purposes of testing and validation of this approach, the emulated deployment scenario is most appropriate.

4.1.10 Pin-based Error Virtualization

Program errors or bugs are ever-present in software, and especially in large and highly complex code bases. They manifest as application crashes or unexpected behavior and can cause significant problems, like limited availability of Internet services, loss of user data, or lead to system compromise. Many attempts have been made to increase the quality of software and reduce the number of bugs. Companies enforce strict development strategies and educate their developers in proper development practices, while static and dynamic analysis tools are used to assist in bug discovery. However, it has been established that it is extremely difficult to produce completely error-free software.

To alleviate some of the dangers that bugs like buffer overflows and dangling pointers entail, various containment and runtime protection techniques have been proposed. These techniques can offer assurances that certain types of program vulnerabilities cannot be exploited to compromise security, but they do not also offer high availability and reliability, as they frequently terminate the compromised program to prevent the attacker from performing any useful action.

In response, researchers have devised novel mechanisms for recovering execution in the presence of errors. ASSURE, in particular, presents a powerful system that enables applications to automatically self-heal. Its operation revolves around the understanding that programs usually include code for handling certain anticipated errors, and it introduces the concept of rescue points (RPs), which are locations of error handling code that can be reused to gracefully recover from unexpected errors. In ASSURE, RPs are the product of offline analysis that is triggered when a new and unknown error occurs, but they can also be the result of manual analysis. For example, RPs can be identified by examining the memory dump produced when a program abnormally terminates. Also, they serve a dual role; first they are the point where execution can be rolled back after an error occurs, and second they are responsible for returning a valid and meaningful error to the application (i.e., one that will allow it to resume normal operation).

Regrettably, deploying RPs using ASSURE is not straightforward, but demands that various complex systems be present. For instance, to support execution rollback, applications are placed inside the Zap virtual execution environment, while RP code is injected using Dyninst. Zap is a considerably complex component that is tightly coupled with the Linux kernel, and requires maintenance along with the operating system (OS). In practice, RPs are a useful but temporary solution for running critical software until a proper solution, in the form of a dynamic patch or update, is available. It is likely that RPs have not been widely used mainly because of the numerous requirements, in terms of additional software and setup, of previous solutions like ASSURE.

REASSURE was developed, a self-contained mechanism for healing software using RPs. REASSURE assumes that a RP has already been identified, and needs to be deployed quickly and in a straightforward manner. It builds on Intel's Pin dynamic binary instrumentation (DBI) framework to install the RP and provide the virtual execution environment for rolling back execution. As Pin itself is simply an application, installation is simple and very little maintenance (or none at all) is necessary. Furthermore, REASSURE does not need to be continuously operating or even present, but can be easily installed and attached only when needed. Disabling it and removing it from a system is equally uncomplicated, since it can be detached from a running application without interrupting its operation. Combined with a dynamic patching mechanism, applications protected with REASSURE can be run and eventually patched without any interruption.

REASSURE was implemented as a Pin tool for Linux. Evaluation with popular servers, like Apache and MySQL, that suffer from well-known vulnerabilities shows that REASSURE successfully prevents the protected applications from terminating. When no faults occur, the performance overhead imposed by REASSURE varies between 1% and 115% depending on the application, while in the presence of errors there is little effect on the protected application until the frequency of faults surpasses five faults per second. Note that Pin supports multiple platforms (e.g., Windows and Mac OS), and REASSURE can be extended to support them with little effort.

4.1.11 ISR extensions

4.1.11.1 PinISR

ISR is a general approach that defeats all types of remote code-injection regardless of the way they were injected into a process. It operates by randomizing the instructions that the underlying system "understands", so that "foreign" code such as the code injected during an attack will fail to execute. It was initially proposed as a modification to the processor to ensure low performance overheads, but unfortunately this proposal has had little allure with hardware vendors. Instead, software implementations of ISR on x86 emulators have been created, mainly to demonstrate the effectiveness of the approach, as they incur large runtime overheads. Software-only implementations of ISR using dynamic binary translation have been also proposed, but have seen little use in practice as they cannot be directly applied to commodity systems. For instance, they do not support shared libraries or dynamically loaded libraries (i.e., they require that the application is statically linked), and increase the code size of encoded applications.

A fast and practical software implementation of ISR for commodity systems was developed. The implementation is based on Intel's dynamic instrumentation tool called Pin, which provides the runtime environment. Application code is randomized using the XOR function and a 16-bit key, which is randomly generated every time the application is launched to make it resistant against key guessing attacks. Multiple keys can be used to randomize different parts of the application. For instance, every shared library used by the system can be randomized using a different key, creating a randomized copy of each library. While additional disk space will be required to store the randomized versions, during runtime all binaries running under ISR will be using the same randomized copy. Also, original (native) code can be combined with randomized code. The keys used to encode the various libraries are managed using SQLite, a self-contained and server-less database engine. Libraries can be randomized once and reused by multiple applications, while frequently re-randomizing them also protects them against key guessing attempts. Finally, it is assumed that the attacker does not have access to the randomized code (i.e., it is a remote attacker), so a known ciphertext attack against the key is not possible.

Instruction-set randomization for commodity systems using Intel's Pin framework was implemented. This implementation of ISR is freely available from https://sourceforge.net/projects/isrupin/
This implementation operates on currently deployed binaries, as it does not require recompilation, or changes to the underlying system (i.e., the operating system and hardware). This system supports dynamically linked executables, as well as dynamically loaded libraries. A key management scheme for storing and keeping track of the keys used to randomize shared libraries and applications is also introduced. This is the first to apply ISR on shared libraries. Executables are re-randomized every time they are launched, and shared libraries are re-randomized at custom intervals to protect the key from guessing attacks.

The overhead of this implementation can be as low as 10% compared with native execution. It is able to run popular servers such as the Apache web server, and the MySQL database server, and show that running Apache using ISR has negligible effect on throughput for static HTML loads, while the overhead for running MySQL is 75%. The cost of completely isolating the framework's data from the application was also evaluated. This memory protection (MP) requires more invasive instrumentation of the target application, and it has not been investigated by previous work on software-based ISR, since it incurs significant overhead. It was shown that adding MP over ISR does not reduce Apache's throughput, while it imposes an extra 57% overhead when running MySQL.

4.1.11.2 In-place Code Randomization

In-place code randomization was also developed, a mitigation technique against return-oriented programming (ROP) attacks that can be applied directly on third-party software. This method uses narrow-scope code transformations that can be applied statically, without changing the location of basic blocks, allowing the safe randomization of stripped binaries even with partial disassembly coverage. These transformations effectively eliminate about 10%, and probabilistically break about 80% of the useful instruction sequences found in a large set of Windows executable (PE) files. Since no additional code is inserted, in-place code randomization does not incur any measurable runtime overhead, enabling it to be easily used in tandem with existing exploit mitigations such as address space layout randomization. Evaluation using publicly available ROP exploits and two ROP code generation toolkits (Q and Mona) demonstrates that this technique prevents the exploitation of vulnerable Windows 7 applications, as well as the automated construction of alternative ROP payloads that aim to circumvent in-place code randomization using solely any remaining unaffected instruction sequences. Although quite effective, in-place code randomization is not meant to be a complete prevention solution against ROP exploits as it offers probabilistic protection. However, it can be applied in tandem with existing randomization techniques to increase process diversification. This is facilitated by the practically zero overheads of the code transformations, and the ease with which they can be applied on existing third-party executables.

4.1.12 I/O redirection

The goal of the I/O redirection task within MINESTRONE is to enable multiple, diverse replicas of a program being tested to be run concurrently, possibly in an alternate, remote execution environment, such as a data center or cloud computing environment. This alternate execution environment enables the exploration of diversification strategies other than ISR (discussed in Section 5.12) while also providing a sensor for *a posteriori* detection of attacks/vulnerabilities.

Within the MINESTRONE system, I/O redirection involves (1) capturing inputs and outputs from a “canonical” version of the program being tested on a “local” computer, and (2) communicating those inputs and outputs to diverse replicas of the program being tested running on possibly “remote” computers. This local versus remote distinction could be conceptual – the diverse replicas could be run on the same computer as the canonical program given sufficient computing power, isolation capabilities between the program instances, etc. (understanding that the decision on how to distribute replicas could impact the diversification strategies that can be employed).

Examples of high-level I/O types to capture and replay include keyboard, file system, network, and mouse/graphics. Multiple options for I/O redirection in both user and kernel space across the various I/O types of concern were investigated and developed. Some investigations were specific to particular input types (e.g., keyboard or network), while others have been more generally applicable. The techniques that were investigated specific to particular input types included the following:

- Capture of keyboard input by listening to keyboard device events, which are mapped to /dev/input/event* on a Linux system.
- Capture of keyboard input by listening to /dev/input/uinput.
- Capture of keyboard input using pseudo-tty and expect.
- Capture of keyboard input using the Linux notifier chain facility, a mechanism provided for kernel objects to register for notifications upon the occurrence of asynchronous events, in this case keyboard events. The keyboard notifier was registered using register_keyboard_notifier(), defined in the Linux headers keyboard.h, and defined the function to be called upon occurring keyboard events.
- Investigation of X11 events for mouse/graphical input, given that in a Linux environment, generally speaking one can assume the X Window System is present and then sniff X11 events to track mouse events and input. (X11 events are also applicable to keyboard input.)

Alternately, one could capture mouse device events in an analogous manner to keyboard device events.

- Capture of network input using the commonly available network packet analyzer/sniffer, tcpdump. Replay of the network traffic using iptables (the packet filtering capability provided by the Linux kernel firewall), redirection of traffic to a QUEUE, and transformation of that traffic using NFQUEUE (the interface to packets that have been queued for the kernel packet filter) was investigated.

One generally applicable technique for I/O redirection that was investigated is library interposition, in which a subset of function calls from glibc is intercepted using the LD_PRELOAD command. Library interposition is a well-understood technique. One advantage of library interposition, in contrast with the other approaches that were investigated, is that it captures only the input associated with the particular program being tested. (All other approaches will receive all the input from the instrumented system, and then the input associated with the particular program must be sorted out for capture.) Another advantage of library interposition is that it operates in user space, so it is a simple change to the container environment. The primary issue with the library interposition approach is that there are a great many functions within glibc that would have to be intercepted in order to capture all input.

Another technique that was investigated for I/O redirection that is applicable to multiple I/O types is system call interception. System call interception is another well-understood technique for capturing I/O to a system. There are multiple ways to intercept system calls. The first method investigated was hooking the system call table. However, in version 2.6 of the Linux kernel, the system call table is no longer exported. A proof-of-concept kernel module was developed that locates the system call table based on the address of the Interrupt Descriptor Table (IDT), stored in a machine register, which points to the system call handler. From the system call handler the address of the system call table can be found. Using that address the system call table was hooked, demonstrating capture of an example system call, the open call for files; this technique is easily extended to the read calls associated with input. However, given control over the configuration of the platform in the MINESTRONE system, it is possible to intercept system calls in a more straightforward manner.

The MINESTRONE system is built upon lightweight containers as described in Section 5.6, and the system enabling that container-based virtualization technology has been extended to enable system call interception and logging using the Linux kernel debugging facilities of Kprobes, Jprobes, and Return probes. The extensions enable system calls to be be tracked on a per-container basis. This method of system call interception was investigated for I/O redirection as well. In addition its advantage of being applicable to many I/O types, system call interception has the advantage that many of those input types end up using read system calls. One disadvantage of intercepting system calls is that each high-level operation can result in a large number of system calls, and some form of aggregation is required.

Based on these investigations, I/O redirection in the MINESTRONE system uses a combination of methods for capturing input, including system call interception at the core and X11 events for mouse/keyboard input.

4.1.13 Replica Diversification

Building upon the capabilities provided by I/O redirection within the MINESTRONE system, additional diversification strategies were investigated beyond ISR that can be implemented for remote execution of program replicas.

The ability to redirect I/O to lightweight containers running replicas in a remote computing environment (e.g., an enterprise data center or a cloud computing environment) enables alternate, "heavyweight" strategies beyond traditional diversification techniques like ASLR. In particular, it provides control over the platform and environment for replica execution in such a remote computing environment, enabling exploration of diversification strategies using those elements. Exploits for vulnerabilities are often very dependent upon the specific characteristics of the environment to execute properly, and the purpose of these diversification strategies would be to provide many variations on environment in order to trigger these environment-dependent exploits.

A number of additional diversification strategies for implementation in the MINESTRONE system were identified, including the following:

- When testing a program for which source code is provided, one can diversify during the compilation process, including generating replicas using different compilers and different versions of a single compiler.
- When building lightweight containers to execute program replicas, one can diversify the software running within each container, including providing different versions of libraries for dynamic linking and different versions of other software upon which the program replica might depend.
- When building physical or virtual machines that host the lightweight containers, one can diversify the platform itself, including varying characteristics of the operating system such as its version or even distribution (for Linux). One could also diversify the underlying hardware, for example running replicas on hardware with different processor architectures or characteristics (e.g., 32-bit versus 64-bit).

Given this initial set of additional diversification strategies, the characteristics of each are being explored. Some of the platform diversification strategies might be difficult to implement depending upon the particular I/O redirection methodology chosen for certain input types – for example, where using system call interception for I/O redirection, some system calls might not translate across different operating system versions.

A container was developed that provided replica diversification in terms of the build process and container environment (Linux loader and libc), primarily, including templates that enabled different compilation parameters (CFLAGS) to be used and different versions of the C compiler (gcc and clang). An initial evaluation was conducted of some of these diversification strategies along with an LLVM-based multi-compiler approach (from UC-Irvine) using the Phase 2 T&E dataset, and it was found that these diversification strategies do provide some limited detection/protection capabilities (approximately 10% of weaknesses prevented), but there are issues with alternate functionality.

4.1.14 Experimental Evaluation

Each component has been experimentally evaluated by itself. Some of the high-level results are supplied in the text describing each component; for full details, see the papers accompanying this report.

The Test & Evaluation process conducted by MITRE also covered this work. An integrated system against vulnerability classes 7 (Memory Errors) and 8 (NULL Pointer Errors) was used. For the former, the libDFT, PinISR, REASSURE and KLEE components were used, each within a container. For the latter, REASSURE and KLEE were used, each within a container. A rough prototype of a system that was worked on more fully during Phase 2, named DYBOC, was also used.

4.1.15 System Integration

The component technologies of the MINESTRONE project were integrated into a single prototype system, building upon the lightweight containers described previously.

For the MINESTRONE system prototype at the end of Phase 1 of the STONESOUP program, a lightweight container was built for each of the following prototypes: KLEE, PinISR, REASSURE, and libDFT. A container for the DYBOC prototype was also built, which was used during the test and evaluation process but did not include in the Phase 1 system prototype.

The integration of MINESTRONE components involved configuring a container for each of the detection technologies with the specific prerequisites for the particular technology being installed. Each container was also configured with access to a shared file system using UnionFS, enabling each container to see the same files but maintain its own copy of each. Finally, each container was configured with its own network interface and X window display.

The integrated MINESTRONE system relies upon the I/O redirection component to distribute inputs to each detection container for testing of a program. A “canonical” version of the program runs in its own container without any instrumentation, and this is the container with which a user interacts.

The MINESTRONE system prototype can operate in one of two modes:

- Off-line: Input to the program in the canonical container is captured until program termination (either user initiated or the result of a crash), and the input is replayed within each detection container after the fact.
- On-line: Input to the program in the canonical container is captured, transmitted to, and replayed within each container in “real time”. (In this mode, it is preferable to run the lightweight containers for the detection technologies in a separate machine – either a physical or virtual machine – from the canonical container due to the replay of X windows events in each of the containers. A user might find the concurrent replay of X windows events in multiple containers distracting or disconcerting.)

The MINESTRONE system composer (see Figure 1) ensures that each container has a copy of the program being tested and the arguments used for execution. The results from each container are presented upon completion of execution.

4.1.16 Miscellaneous Items

This section describes work done as part of the project in supporting functions (i.e., enables but does not "cleanly" fit in any of the tasks described above).

4.1.16.1 Vulnerable Host Scanner

One "out of band", activity directly related to the effort on vulnerable embedded devices that are being provided to the agency is the embedded system scanner.

A quantitative lower bound on the number of vulnerable embedded devices on a global scale was determined. Over the past year, large portions of the Internet have been systematically scanned to monitor the presence of trivially vulnerable embedded devices. At the time of writing, over 540,000 publicly accessible embedded devices configured with factory default root passwords have been identified. (As of December 2011, the number has increased to 1.4 million.) This constitutes over 13% of all discovered embedded devices. These devices range from enterprise equipment such as firewalls and routers to consumer appliances such as VoIP adapters, cable and IPTV boxes, to office equipment such as network printers and video conferencing units. Vulnerable devices were detected in 144 countries, across 17,427 unique private enterprise, ISP, government, educational, satellite provider as well as residential network environments. Preliminary results from a longitudinal study tracking over 102,000 vulnerable devices revealed that over 96% of such accessible devices remain vulnerable after a 4-month period. The data provides a conservative lower bound on the actual population of vulnerable devices in the wild. By combining the observed vulnerability distributions and their potential root causes, a set of mitigation strategies was proposed. Employing this strategy, a partnership with Team Cymru to engage key organizations capable of significantly reducing the number of trivially vulnerable embedded devices currently on the Internet was made.

4.1.16.2 ROP Payload Detection Using Speculative Code Execution

The exploitation of memory corruption vulnerabilities in server and client applications has been one of the prevalent means of system compromise and malware infection. By supplying a malicious input to the target application, an attacker can inject and execute arbitrary code, known as shellcode, in the context of the vulnerable process. Fortunately, the wide adoption of non-executable memory page protections like Data Execution Prevention (DEP) in recent versions of popular OSes has reduced the impact of conventional code injection attacks.

In turn, attackers have started adopting a new exploitation technique, widely known as return-oriented programming (ROP), which allows the execution of arbitrary code on a victim system without the need to inject any code. In the same spirit as in the return-to-libc exploitation technique, return-oriented programming relies on the execution of code that already exists in the address space of the process. In contrast to return-to-libc though, instead of executing the code of a whole library function, return-oriented programming is based on the combination of tiny code fragments, dubbed gadgets, scattered throughout the code segments of the process. The execution order of the gadgets is controlled through a sequence of gadget addresses that is part of the attack payload. This means that an attacker can execute arbitrary code on the victim system by injecting only control data.

Besides the effective circumvention of non-executable page protections, return-oriented programming also poses significant challenges to a broad range of defenses that are based on shellcode detection. The main idea behind these approaches is to execute valid instruction sequences found in the inspected data on a CPU emulator and identify characteristic behaviors exhibited by different shellcode types using runtime heuristics. Besides the detection of code injection attacks at the network level, shellcode identification has been used for in-browser detection of drive-by download attacks, as well as malicious document scanning.

In a ROP exploit, however, in place of the shellcode, the attack vector contains just a chunk of data, referred to as the ROP payload, comprising the addresses of the gadgets to be executed along with any necessary instruction arguments. Since there is no injected binary code to identify, existing emulation-based shellcode detection techniques are ineffective against ROP attacks. At the same time, return-oriented programming is increasingly used in the wild to broaden the targets of exploits against Acrobat Reader and other popular applications, extending the infection coverage of recent exploit packs.

As a step towards filling this gap, a new technique was developed for the detection of ROP exploits based on the identification of the ROP payload that is contained in the attack vector. ROPscan, the prototype implementation, uses a code emulator to speculatively execute code fragments that already exist in the address space of a targeted process. The execution is driven by valid memory addresses that are found in the injected payload, and which could possibly point to the actual gadgets of a malicious ROP code. ROPscan was evaluated using an array of publicly available ROP exploits against Windows applications, as well as with a vast amount of benign data. Results show that ROPscan can accurately detect existing ROP exploits without false positives, while it achieves an order of magnitude higher throughput compared to Nemu, an existing shellcode detector with which ROPscan shares the code emulation engine.

Current exploits use ROP code only as a first step to bypass memory protections and to enable the execution of a second-level conventional shellcode, which is included in the same attack vector and thus can be identified by existing shellcode detectors. However, the embedded shellcode can easily be kept unexposed through a simple packing scheme, and get dynamically decrypted by a tiny ROP-based decryption routine, similarly to simple polymorphic shellcode engines. It has also been demonstrated that return-oriented programming can be used to execute arbitrary code, and thus future exploits may rely solely on ROP-based malicious code.

In any case, the ability to identify the presence of ROP code can increase the detection accuracy of current defenses that rely only on shellcode detection. ROPscan can inspect arbitrary data, which allows its easy integration into existing detectors—two case studies are presented in which ROPscan is used as part of a network-level attack detector and a malicious PDF scanner.

4.1.16.3 kGuard

Return-to-user (ret2usr) attacks are control-flow violation attacks against the kernel that enable local users to hijack privileged execution paths and run arbitrary code with elevated privileges. Such attacks have become increasingly frequent since they were first demonstrated in 2007. Normally, they require local access to the system, and operate by exploiting vulnerabilities in kernel code that implement facilities like system calls. When successful, they manage to overwrite some, or all, bytes of a function or data pointer in kernel space. Since most OSs keep user memory mapped and accessible when the kernel is invoked, privileged execution paths can be hijacked and redirected to user space, leading to arbitrary code execution and, oftentimes, privilege escalation. In fact, Intel has recently announced a new CPU feature, named SMEP, which mitigates ret2usr by preventing privileged code from branching to pages without the supervisor bit set.

There are numerous reasons why attacks against the kernel are becoming more common. First, processes running with administrative privileges have become harder to exploit, due to the various defensive mechanisms adopted by modern OSs, such as W^X memory pages, address space layout randomization (ASLR), and stack-smashing protection. Second, NULL pointer dereference errors have not received significant attention, exactly because they were thought impractical and difficult to exploit. However, in the kernel setting, where there is unrestricted access to all memory and system objects, such assumptions do not hold. As a matter of fact, some security researchers dubbed 2009 as “the year of the kernel NULL pointer dereference flaw.” Third, the separation wall between kernel and user space is not symmetrical. Kernel entrance is hardware-assisted and facilitated by a considerable amount of protection logic, including user argument validation and system call parameter sanitization. However, the opposite (i.e., kernel-to-user crossing) is not always policed, allowing the kernel to freely cross the boundary between kernel and user space, and when abused, execute user-provided code in kernel mode.

Current defenses have proven to be inadequate, as they have been repeatedly circumvented, incur considerable overhead, or rely on extended hypervisors and special hardware features. The most popular approach has been to disallow user processes to memory-map the lower parts of their address space (i.e., the one including page zero). Unfortunately, this scheme has several limitations. Firstly, it does not address the root cause of the problem, which is the weak separation between kernel and user space. Secondly, it has been circumvented through various means. Thirdly, it breaks compatibility with various applications that depend on having access to low logical addresses. A proposed system named UDEREF offers comprehensive protection, but incurs considerable overhead, requires kernel patching, and works only on specific architectures. On x86, it utilizes the segmentation unit to isolate kernel from user space, incurring overheads between 5.6% and 257% in system call and I/O latency. On x86-64, where segmentation is not available, the overhead is even larger.

On the other hand, recent advances in virtualization have prompted a wave of research that employs custom virtual machine monitors (VMMs) for attesting or assuring the integrity of privileged software. SecVisor and NICKLE are two hypervisor-based systems that can prevent ret2usr attacks by leveraging memory virtualization and VMM introspection. However, running the whole OS as a VM guest incurs notable performance penalty, additional management cost, and simply buries the issue to a lower level.

kGuard was developed, a compiler plugin that augments the kernel with compact inline guards, namely Control-Flow Assertions, which prevent ret2usr with low performance and space overhead. kGuard can be used with any operating system that features a weak separation between kernel and user space and is vulnerable to such attacks, requires no modifications to the source code of the OS, and is applicable to both 32- and 64-bit architectures. kGuard identifies all indirect control transfers during compilation, and injects compact dynamic checks to attest that the kernel remains confined. When a violation is detected, a user-defined fault handler is invoked. The default handler reports the error and halts the system. kGuard is able to protect against attacks that overwrite a branch target to directly transfer control to user space, while it also handles more elaborate, two-step attacks that overwrite data pointers to point to user- controlled memory and, hence, hijack execution through tampered data structures. Finally, kGuard protects itself from being subverted. Evaluation demonstrates that Linux kernels compiled with kGuard become impervious to a variety of control-flow hijacking exploits, while at the same time kGuard imposes on average an overhead of 11.4% on system call and I/O latency on x86 OSs, and 10.3% on x86-64. The size of a kGuard-protected kernel grows between 3.5% and 5.6%, due to the inserted checks, while the impact on real-life applications is minimal (~1.03%).

4.1.16.4 Taint Exchange

A generic cross-process and cross-host taint tracking framework was developed, called Taint-Exchange. This system builds on the libdft open-source data flow tracking (DFT) framework, which performs taint tracking on unmodified binary processes using Intel's Pin dynamic binary instrumentation framework. libdft was extended to enable transfer of taint information for data exchanged between hosts through network sockets, and between processes using pipes and unix sockets. Taint information is transparently multiplexed with user data through the same channel (*i.e.*, socket or pipe), allowing marking of individual bytes of the communicating data as tainted. Additionally, users have the flexibility to specify which communication channels will propagate or receive taint information. For instance, a socket from HOST A can contain fine-grained taint information, while a socket from HOST B may not contain detailed taint transfer information, and all data arriving can be considered as tainted. Similarly, users can also configure Taint-Exchange to treat certain files as tainted. Currently, entire files can be identified as a source of tainted data.

Most real-world services consist of multiple applications exchanging data, which in many cases run on different hosts, e.g., Web services. Taint-Exchange can be a valuable asset in such a setting, providing transparent propagation of taint information, along with the actual data, and establishing accurate cross-system information flow monitoring of interesting data. Taint-Exchange could find many applications in the system security field. For example, in tracking and protecting privacy-sensitive information as it flows throughout a multi-application environment (e.g., from a database to a web server, and even to a browser). In such a scenario, the data marked with a "sensitive" tag, will maintain their taint-tag throughout their lifetime, and depending on the policies of the system, Taint-Exchange can be configured to raise an alert or even restrict their use on a security-sensitive operation, e.g., their transfer to another host. In a different scenario, a Taint-Exchange-enabled system could also help improve the security of Web applications by tracking unsafe user data, and limiting their use in JavaScript and SQL scripts to protect buggy applications from XSS and SQL-injection attacks. The performance overhead imposed by Taint-Exchange was evaluated, showing that it incurs minimal additional overhead over the libdft baseline.

4.2 Phase 2 Developments

The key component technologies have been successfully developed and tested and are poised for integration into a single MINESTRONE system.

4.2.1 Source-level Monitor Based upon DYBOC and Number Handling Tool

In preparation for Phase 2 evaluation, DYBOC and the number handling tool were migrated from TXL to CIL and LLVM compiler framework respectively to support applications with large source base.

4.2.2 Binary-level Monitor Based Upon Pin

The prototype for binary-level DFT is libdft. It is complete and mature, and can apply taint analysis to binaries to prevent control-flow diversion attacks. Since such attacks are part both of code-injection and ROP payloads, it can detect and prevent a broad range of attacks.

4.2.3 Automatic Race Avoidance

A prototype system was built. Technology was created to reduce the set of schedules of a parallel program for avoiding races.

4.2.4 Pin-based Error Virtualization

The component for error virtualization is REASSURE. The prototype was completed and mature, and can deploy RPs both on Windows and Linux binaries with or without symbols. It is also able to operate on 32-bit and 64-bit systems.

4.2.5 Instruction Set Randomization (ISR)

One of the run-time protection modules that was created implements instruction-set randomization (ISR) for processes. ISR can be used to defend against code-injection (CI) attacks that can be still performed against legacy software that does not adhere to W^X semantics. That is, code that includes memory pages that are both writable and executable. It can also protect against the second stage of attacks that utilize code-reuse attacks to facilitate code injection. In this context ISR prevents the introduction of new code into a process, essentially operating as a code integrity solution. The solution uses a 16-bit XOR for randomizing binaries in disk and dynamically de-randomizing them before execution. It supports dynamic libraries and it is the first approach that allows for randomized libraries to be shared among processes, lessening memory pressure to the system. The approach is not vulnerable to brute-force attacks, as requests to execute injected code are detected and result in de-randomization with a random key to avoid divulging any information about used keys. However, it remains vulnerable to attacks that leak code memory of known binaries to learn the key. Its use is recommended for non W^X binaries or for reducing the attack surface of a binary. Two publications were written associated with ISR and the majority of this work was done at Columbia.

4.2.6 Replica Diversification

A number of diversification strategies for replicas were investigated and prototyped, including diversification of the build process (e.g., compiler, options) and environment (e.g., libraries, platform). These strategies were experimented with and both the detection capabilities and altered functionality using the test and evaluation dataset were evaluated.

4.2.7 Replica Monitoring

The xpc prototype (formerly process checker) was researched and prototyped for monitoring the MINESTRONE prototype platform and associated replicas, using either virtual machine introspection or monitoring from within the host. Experimentation and initial evaluation of xpc's detection capabilities for integrity checking of the kernel, loaded modules, processes, and libraries as well as unlinked kernel modules, privilege escalation, and resource monitoring of memory utilization was conducted.

Replica monitoring explores alternate monitoring strategies that are enabled by the MINESTRONE architecture and MINESTRONE/containers running in off-host computing environments. These off-host computing environments (again, data center or cloud) enable different, more expensive monitoring strategies, which enable not only replica monitoring but also monitoring of the MINESTRONE platform itself. Virtual machine introspection (VMI) techniques were explored and prototyped for monitoring the integrity of the platform kernel and kernel modules, as well as processes and libraries running in containers. Non-VMI techniques were also explored, using Linux kernel module extensions for reading memory (e.g., /dev/mem, /dev/fmem) and developing a new kernel module implementation. Proof-of-concept detections of various attacks on the platform were also demonstrated, such as user and kernel rootkits, privilege escalation, and unlinked kernel modules.

4.2.8 TheRing

As many of the above technologies are built on top of runtime binary instrumentation using Intel's Pin framework, theRing aims to unify different compatible protection mechanisms into a single program hardening tool. Currently, theRing supports REASSURE, ISR, and libdft, while it can also parse and run program binaries compiled with DYBOC, which allows it to intercept and apply self-healing to vulnerabilities prevented by DYBOC. Currently, the applied defenses are selected at load time, and we plan to add support for dynamic reconfiguration.

4.2.9 KLEE

KLEE was scaled to programs of larger sizes.

4.2.10 In-Code Engine (ICE) (aka Symbiote)

The proposed ICE concept became the Symbiote technology. ISA- and OS-agnostic prototypes of randomly injected symbiote defenses has been completed and demonstrated for Cisco IOS routers, Cisco IP phones and HP printers. FRAK technology automating the “random” reorganization of arbitrary firmware in addition to the random injection of Symbiotes was separately developed.

4.3 Phase 3 Developments

4.3.1 MINESTRONE

An integrated prototype was created and demonstrated for the kickoff meeting, including both a Linux and Windows implementation. Several applications of MINESTRONE internal to Symantec were investigated. A new baseline virtual machine was created and all base test programs were imported for use in evaluating MINESTRONE, which was set up within Amazon EC2. It was further extended with a CLANG source-to-source transformation module to better handle function variables by creating a new block of memory per module and address format string vulnerabilities. Work was done on improving replica diversification by evaluating multicompiler technology with corpora from phase 2. An article on results of this phase was written for a penetration-testing magazine, and phase 2 MINESTRONE results were written up and submitted to CSET.

4.3.2 PaX/SMEP/SMAP Kernel Exploits

Attacks were demonstrated on systems using PaX/SMEP/SMAP which break isolation guarantees of ret2usr protection schemes by allowing implicit sharing of memory pages between kernel and user space. A proof-of-concept translator to generate BPF/seccomp filter from x86 shell code to bypass validation checks was developed, which uses a “spraying” phase where kernel memory is flooded with BPF filters containing a malicious payload.

4.3.3 ROP Exploits

A protection technique against return-oriented-programming (ROP) exploits was investigated by observing that legitimate code rarely needs to perform read operations from code sections, so memory access operations to code sections from instructions fetched from the same section can be treated differently.

4.3.4 Parrot System

This system was released as opensource. It was tested on a benchmark suite constructed with 50 programs representing many parallel computing idioms. The system was integrated with TheRing (currently some issues with compatibility with ASSURE rollback). Concurrent programs using Parrot will have enforced schedules race-checked using the Parrot-Tsan detector. A system using ideas from STONESOUP to help find bugs in mobile apps called AppDoctor was also designed and implemented, for which a paper was accepted at EuroSys. AppDoctor was successfully used to find real bugs in Android apps made by large companies, and demonstrated an over 10x improvement in bug detection and app testing. An open source release is being considered, and a patent application is in progress.

4.3.5 OS-assisted Dataflow Tracking

Developed a performance improvement by using (base, length) segment registers to efficiently track tagged memory locations, and setting pages that are beyond the storage limits of the registers to fault on every access. This was implemented as a Linux kernel patch. A MMU simulator to evaluate DFT configurations has also been designed and tooled with various management policies for segment registers: eviction of LRU segment to page-level tracking, eviction of LRU segment that has the largest number of segments in same page, dynamically monitoring false positive page faults to move back to register tracking. A tool for detection of false positives in propagating information flows was prototyped, and a new input generation system to get more coverage in evaluation was created.

4.3.6 Hybrid Code Instrumentation Framework

A system that allows both source code and executables to support third-party software composed of multiple modules and parts was designed. The framework maximizes performance benefits by applying instrumentation at both source code level and through dynamic binary instrumentation at runtime.

4.3.7 Resource Exhaustion Capture Tool

4.3.7.1 KLEE Evaluation

KLEE was evaluated as a component for enumerating all input classes, generating resource profiles, and generating symbolic outputs for all flow paths for malicious code classification. However, being designed only as a coverage tool, not for complete flow path monitoring, KLEE may not be the best way to get exhaustive symbolic inputs. This work identified four problem aspects: KLEE needs recompilation in C86 which can't be done since only binary is available, KLEE needs inputs characterized in source code, KLEE produces noisy output, and finally KLEE produces far too much output for large programs. It was finally concluded that KLEE has limited use for small programs but is not robust enough for general use in the system.

4.3.7.2 Development

A vulnerability was successfully inserted into XPDF demonstrating the ability of the tool to capture resource exhaustion in real applications. It was also evaluated using the MITRE test cases, for which it detected 569/596 (96.9%) of the cases. DynInst was finally chosen as a component for injecting binary instructions and function calls into small programs for resource monitoring tools. A shadow libC was initially investigated as an alternative to DynInst (which would require labor-intensive instrumentation of C libraries), however concluded DynInst could be used to meet all needs. This will allow clustering KLEE flow paths by resource usage, and providing typical usage profiles for threat analysis. Further data-mining tools were developed to generate classes of typical resource usage to be used in conjunction with KLEE-style classes to help with malicious detection, including tools to generate resource usage classes, do multivariate time series data mining, and compare code usage to known usage profiles.

4.3.7.3 Evaluation

These tools were successfully used to differentiate four sets of test runs generated from the GREP program, some modified with code that would cause CPU and memory exhaustion attacks. Similar tests were also run with Wget. Upon request, this approach was evaluated and it was determined that it would not detect Heartbleed.

4.3.8 CFI Attacks

It was demonstrated that practical CFI implementations are unsafe because they allow too broad a set of control-flow transfers, thus allowing code-reuse attacks. Attacks were demonstrated that were extended to break some assumptions made by tools like kBouncer and ROPecker and identify small chains of gadgets that can be attacked. This work was published to USENIX Security. Some initial work was done on providing tools to evaluate defenses to these attacks.

4.3.9 DynaGuard

The tool was improved with detection of Feeb-like attacks on stack canaries and tested on SPEC benchmarks. A paper was submitted to DIMVA.

4.3.10 Pintool

Pintool was re-implemented from scratch. In doing so, it was investigated and demonstrated that it now has very low overhead, with what overhead exists being mostly due to context switching.

4.3.11 ShadowReplica

A project for efficient parallelized data flow tracking. The system was improved by generalizing the communication channel implementation using a n-way buffering scheme to minimize on-chip cache interference, and the prototype was made open source. A paper was published in ACM CCS.

4.3.12 IntFlow

IntFlow is a compile time arithmetic error detection tool using information flow tracking and static code analysis to improve detection and prevention of integer overflow. It was improved to reduce false positives via whitelisting of certain known benign sources, blacklisting of correlated untrusted sources and possible integer error locations, and correlation of integer locations (sources) and input arguments of sensitive functions (sinks). The stable implementation was installed into MINESTRONE. It was evaluated on actual exploitable vulnerabilities in real-world programs as well as the SPEC2000 benchmark suite. False positives were reduced by 89% compared to IOC. A paper was published in the ACSAC.

4.3.13 TheRing

The tool was improved to allow customization of activated components with compile-time and run-time configurations instead of multiple trees, and was ported it to the CMake software build system.

Integration of pmalloc to enhance the tool and enable continued execution of overflow an underflow errors was explored.

4.3.14 SQLRand

A tool was created to defend against SQL injection with techniques inspired by instruction set randomization (combination of static code and data flow analysis). This idea was implemented as LLVM addon. De-randomization was done by intercepting ODBC driver API calls from the application side.

4.3.15 Rescue Point Auto-configuration

A Clang plug-in was created to gather data about pre-processor macros during compilation, particularly those relating to return values, which can now be mapped to errors to help generate more RPs. This involves a new post-fault phase that causes unrescued faults to reconfigure REASSURE and TheRing and attempt to find a RP to rescue future similar faults.

4.3.16 Learning Approach to Reduce Resource Bounds

A system was developed to profile executable's usage using early-decision time-series analysis. This involved implementing Symbolic Aggregate Approximation (SAX) to transform the time series into a symbol sequence, and a Probabilistic Suffix Automaton (a Markov model that trains using a Suffix Tree Model) that can help detect anomalies in test strings. It can do so either by using a sliding window approach to compute probabilities of incoming symbols and detect anomalies (as unlikely symbols). Alternatively, unlikely states can also be identified using the PSA’s Markovian statistics.

4.3.17 Dynamic Taint Analysis

Work was done using dynamic taint analysis (DTA) to detect control-hijacking attacks thatlead to code-reuse and code-injection attacks. The methodology followed is well-established and involved the tainting of user inputs (e.g., inputs from the network and user files) and tracking their propagation as the program executes. Attacks are prevented by enforcing that no tainted inputs ever directly affect the program’s control flow. That is, tainted data are not loaded in the program counter (PC) through a call, jump, return, etc. Data tracking is done by a reusable library, namely libdft, which tracks data movement and direct dependencies using shadow memory (1-bit of shadow memory corresponds to each byte of regular memory) and instrumenting program instructions. While the overhead of doing DTA over a VM, like Pin, can be considerable, libdft has been carefully engineered to incur relatively small overhead (compared with similar approaches). Multiple publications were written related to libdft and novel methodologies for accelerating it using static analysis and parallelization. While the optimizations are not mature enough to be included in the delivered prototype, additional engineering effort could deliver speedups up to 4x. The majority of this work was done at Columbia.

4.3.18 REASSURE

REASSURE delivers software self-healing support and protects against null pointer errors by allowing the program to handle unexpected errors, preventing fatal crashes due to the dereferencing of null pointers. Self-healing operates by using certain program functions as rescue points (RP). When entering a rescue point, all the modifications made to memory by the program are logged, essentially taking a kind of checkpoint when entering a RP function. An unexpected error occurring within a RP is captured, reverting memory state to what it was upon entry to the function. More importantly, the RP returns to its caller, so that code does follow the same execution path, returning an error value that can be checked by the program to gracefully handle the error. Each program thread can enter a separate RP, ensuring that a rollback occurring in one thread does not affect other threads in the program. Furthermore, the application of REASSURE using Pin allows one to easily deploy it when a new fault is discovered and until a patch is released. The entire solution is self-contained and it does not require any modifications to the application or the operating system. However, it does require that a configuration including the RPs to be deployed for an application is present. We have made two publications associated with REASSURE. This work was initiated at Columbia and continued at Stevens.

4.3.19 Compiler Support for Self-healing

The operation of the self-healing module greatly depends on the ability to define the proper RPs for a program. To this end, an LLVM-based module was developed that analyzes an application at compile time to automatically extract the functions that can server as RPs. The process involves generating a function-call graph (FCG) for the application and detecting all possible return values for each function in the application. Return values passed between functions are also correlated with each other and a set of heuristics is used to identify the values that most likely correspond to errors. The extracted information can be used in two ways. First, one can automatically configure an application for maximum coverage. That is, deploy as many as RPs as possible to ensure that any unexpected fault will be captured and handled immediately. This mode, which was also used in the T&E, offers maximum zero-day coverage, but incurs the largest overhead because applications checkpoint very frequently. Second, one can use the pool of identified RPs to dynamically activate them as needed. For example, when one first identifies a fault that causes the application to crash, one can analyze the crash information, pick an appropriate RP, and deploy it. This process may lead to a cycle of crashes, as it may be necessary to deploy various RPs to locate the one that completely encapsulates the error.

4.3.20 LLVM-based Checkpoining

In collaboration with other researchers, work was done toward developing fast checkpointing performed by code introduced during the compilation of an application. The approach relies on splitting memory space in two parts, the program’s memory and checkpoint memory. Checkpointing is done by copying the updated memory into checkpoint memory, the first time a particular address is written. While other approaches were experimented with, like using a log to preserve original memory contents, the split memory model was the fastest. This type of checkpointing was not pursued in the prototype for two reasons. First, it requires that a binary and all of its shared libraries are compiled with LLVM and our extension, and, second, checkpointing in this fashion can only be supported for single threaded programs. While many high performance servers are single threaded, many applications that MINESTRONE protects are not.

4.3.21 Buffer Overflow Protection with libpmalloc (DYBOC)

libpmalloc offers safe versions of standard libc memory management routines, like malloc, free, etc. It operates by placing guard pages around allocated buffers. These guard buffers are mapped in memory without any access permissions, so any overread, underread, overwrite, or underwrite touching the guard buffers is detected. There are mainly two different versions of libpmalloc provided, one that places the user buffer close to the guard page near the end of the buffer, and one close to the guard near the beginning of the buffer. The purpose is to ensure that memory errors are detected as soon as possible, however, it is worth mentioning that deploying one of the versions is sufficient, because guards are placed on both ends of a buffer. During the third phase T&E, modifications were made so that libpmalloc strictly adhered to the alignment requirements of the libc versions of the functions. These alignment requirements no longer allowed placement of the buffer right before a guard page, as some padding might be introduced. As a result, while security is not compromised, an off-by-one error may go undetected, even though it is prevented (i.e., the attacker cannot “escape” buffer boundaries).

4.4 Evaluation Components

The MINESTRONE integrated prototype produced for Phase 3 of the STONESOUP program relied on OpenVZ lightweight virtualization containers for isolation and parallel execution of different detection technologies. The MINESTRONE integrated prototype consisted of the following containers and their associated detection components:

- No Security container
- ISR container
- REASSURE container
- libDFT container
- DYBOC overflow container
- DYBOC underflow container
- Number Handling container
- Resource drain container
- Parrot/xtern race condition container

The MINESTRONE integrated prototype can run in one of two modes: on-line/real-time execution or off-line/capture-and-replay mode. For the purposes of this document, focus is placed on the runtime overhead of the on-line/real-time execution mode. In on-line/real-time execution mode, a user interacts with a “No Security” container in which input is captured and redirected to the remaining containers, each of which is executing a detection component. If any of the detection components detect a vulnerability, an alert is provided to the user.

Performance and runtime overhead measurements are highly dependent upon the particular applications and workload used. The target applications for Phase 3 will be 64-bit applications for the x86-64 platform of approximately 500,000 lines of code (LOC). For the purposes of this document estimates are based on previous experiments, often using real-world applications of varying sizes.

4.5 MINESTRONE Overhead Summary

Performance and runtime overhead measurements are highly dependent upon the particular applications and workload used. The target applications for Phase 3 will be 64-bit applications for the x86-64 platform of approximately 500,000 lines of code (LOC). For the purposes of this document estimates are based on previous experiments, often using real-world applications of varying sizes.

The runtime overhead of the MINESTRONE integrated prototype at the beginning of Phase 3 was estimated to be approximately 115%, dependent upon application and workload.

Given the parallel execution of the detection technologies, the runtime overhead will be dominated by the slowest component. For example, if one detection technology imposes a 20% runtime overhead and the rest of the containers have lower overheads than that, then all of the other detection technologies will complete execution of a task prior to the slowest detection technology, and thus the lower overhead technologies will not impact the overall MINESTRONE integrated prototype overhead (again, assuming sufficient hardware resources to run all detection technology containers in parallel). At a high level, the different detection technologies impose the following overheads:

- ISR: 0-75%
- REASSURE: 1-115%
- libDFT container: 24%-14.52x
- DYBOC overflow: 2x
- DYBOC underflow: 2x
- Number Handling container: 2x
- Resource drain container: 1.7x – 14.2x
- Parrot/xtern race avoidance container: 20%

For most applications the libDFT container incurs too high a runtime overhead and as such would be configured as an optional component, turned off by default. Many of the vulnerabilities detected by the libDFT technology could be detected by other components as well.

Given these numbers, the worst-case parallel execution runtime overhead for the MINESTRONE integrated prototype would be dominated by the REASSURE component at approximately 115% for an application such as MySQL. For other applications, the worst-case parallel execution runtime overhead could be imposed by other components, at a lower rate.

4.6 Evaluation Cases

For their performance evaluation real-world applications like the following are used:

- The **Apache** HTTP server https://httpd.apache.org/
- The **MySQL** open source database https://dev.mysql.com/
- The **CoreHTTP** web server http://corehttp.sourceforge.net/
- The **Firefox** web server http://www.mozilla.org/en-US/firefox/new/
- The **SAMBA** file sharing service for Linux https://www.samba.org/

Benchmark suites like CPU Spec 2006 are also used.
The size of the software used ranges from small to large:

- Apache HTTP server ~200K. Large.
- MySQL database ~1M. Very large.
- CoreHTTP web server server ~700. Small.
- SAMBA Windows interoperability suite ~2M. Very large.

These solutions build on the Pin DBI framework, so part of the overhead is due to the VM used.

4.7 MINESTRONE Integrated Prototype for Test and Evaluation

In the first two phases, the overall detection capabilities of the MINESTRONE integrated prototype on memory corruption and null pointer weakness (in Phases 1 and 2) and number handling and resource drain weakness (in Phase 2) were positive and largely achieved the program goals. For the final T&E in Phase 3, a number of integration challenges arose due to having to integrate with a new version of the test infrastructure (TEXAS) as well as the compressed nature of the dry runs and the introduction of changes in TEXAS and the test programs every dry run and before final T&E. Most significantly, a configuration error due to a change introduced in dry run 3 caused complete failure of execution, requiring a re-run of dry run 3 in order to test the MINESTRONE integration with TEXAS, and this two-week delay effectively caused the final T&E to become another dry run, with expectedly poor results. Debugging integration issues during the final T&E enabled execution of a limited test suite of only the CTREE test program against a final set of changes for the MINESTRONE integrated prototype, and the results of that CTREE test suite largely matched prior expectations for the detection capabilities of the MINESTRONE integrated prototype, summarized in the table below.

Weakness Class	TCs Passed	TCs Failed	Total TCs	Score %
Injection	16	17	33	48%
Memory Corruption	135	25	160	84%
Number Handling	43	1	44	98%
Null Pointer	24	4	28	86%
Concurrency	24	35	59	41%
Resource Drain	39	9	48	81%

Figure 7: Testing and Evaluation results for MINESTRONE

In summary, the MINESTRONE integrated prototype is capable of detecting and mitigating memory corruption, number handling, null pointer, and resource drain weaknesses. The MINESTRONE integrated prototype only handles SQL injection and not command injection weaknesses. The race condition components of the MINESTRONE prototype only handled a subset of concurrency weaknesses, and they had to be disabled in the final configuration due to false positive and component conflict issues. Overall, the runtime overhead of the MINESTRONE detection components is prohibitively high, especially for interactive, GUI-based applications, and more engineering effort would be required to develop and deploy a scaled-down version of the MINESTRONE integrated prototype for transition purposes.

4.8 Evaluation on CTREE

The results were analyzed on the CTREE program, which offers the best insights as they exhibited the least interference from integration issues. The table below summarizes the results reported by the T&E team and the results according to analysis of the data following the T&E. Briefly, in some cases good I/O pairs appeared to fail because bugs in the injected code got triggered even when they were not supposed to, while in other cases some errors were caused due to the complexity of Texas and MINESTRONE. However, ~8% of bad I/O pairs in the memory corruption category did indeed not trigger an alert. Some were expected. For example, libpmalloc does not handle overflows within a structure allocated in a single malloc statement, so it cannot detect such an overflow on its own. Some others did not trigger an alert, but were essentially implicitly handled by our solution. For example, a double free can no longer cause harm because we no longer manage buffers in a list. If it were to cause harm, it would be detected by the protected version of free(). Similarly, due to alignment tricks, freeing a buffer with a slightly different pointer than the one supplied by malloc also works correctly without corrupting any memory. With moderate engineering such errors can also be captured and forbidden.

Fault	Good I/O pairs passed		Bad I/O pairs passed (mitigated)	
	T&E	MINESTRONE	T&E	MINESTRONE
Null pointer (CWE-476)	120/140 (85.71%)	140/140 (100%)	56/56 (100%)	56/56 (100%)
Memory Corruption	633/660 (95.91%)	660/660 (100%)	225/264 (85.23%)	244/264 (92.42%)

Figure 8: Evaluation on CTREE

4.9 Component Evaluation

4.9.1 REASSURE

For MySQL the overhead is between 18%-115%. Note that the costly checkpointing and rollback are not performed all the time, but only for the code that requires it. For Apache and the CoreHTTP server the overhead is lower between 40%-60%. Transferring data over a REASSSURE-protected SAMBA server only incurs 1% overhead. So the worst case scenario is 115%.

4.9.2 libDFT

The overhead of libdft on the Apache web server is between 24%-64%. On MySQL the overhead is on average 3.36x. On Firefox the overhead is between 7x-8x. Running JavaScript has large overhead between 13.9x-14.52x

4.9.3 ISR

ISR imposes 75% overhead on average with MySQL. It imposes negligible overhead on Apache.

4.9.4 TheRing

The overhead of TheRing compound tool is primarily due to the DFT component. Also, since a large part of the overhead is caused by Pin, it is not expected that the total overhead will be cumulative. Instead, it is expected to get ISR for free (no additional overhead), and REASSURE will only add overhead when recovery from a previously detected bug/vulnerability is required.

Using TheRing to combine the three above detection technologies is one configuration option within the MINESTRONE integrated prototype. The primary advantage of the TheRing would be in a configuration where none of the other detection technologies are required, or where the MINESTRONE integrated prototype's usage of lightweight virtualization containers is not acceptable or possible. In such a configuration, TheRing could be used to provide the protection of three detection technologies on an application directly, without the use of OpenVZ, a No Security, and I/O redirection.

4.9.5 DYBOC Overflow/Underflow Containers

The runtime overhead for containers running DYBOC protection against buffer overflows and underflows varies greatly depending upon the application and workload, just like for the other components. During Phase 1 test and evaluation, DYBOC demonstrated approximately 2x slowdown over native execution.

Using the real-world programs of the Phase 2 test and evaluation, for most of the base programs, the runtime overhead caused by DYBOC is negligible because the memory allocations are pretty light. Wget, nginx, cherokee, and wwwx show almost no performance impact when used with DYBOC. On the other hand, tcpdump when used to process a pcap file, can result in significant performance impact. This is because for each packet a very small memory allocation is done; whereas malloc can optimize by allocating a single and splitting it up on each allocation, DYBOC allocates a whole page every time, slowing things down. However, if tcpdump is used on actual network traffic, again the runtime overhead is minimal. The usage pattern (e.g., frequency and size of memory allocations) of an application should be understood and taken into account in order to manage the runtime overhead of the DYBOC containers.

4.9.6 Number Handling Container

In testing with the Phase 2 Test and Evaluation (T&E) applications, the number handling tool incurred a slowdown of approximately 2x over native execution. The Phase 2 T&E applications included web servers, shell engines, Wget clients, etc. The number handling tool is designed to defend against different classes of integer errors - overflow/underflow, sign extension, conversion errors, divide by zero and so on. Therefore, it requires most integer operations to be instrumented/replaced with conditions blocks.

4.9.7 Resource Drain Container

The current overhead of the resource drain container ranges from 1.7x - 1103x with typical overheads of 1.7x – 14.2x depending on:

a) The granularity of verification (i.e. frequency calls of the Pin tool),
b) The number of types of resource exhaustion attacks being tracked and defended against
    a. CPU usage,
    b. Memory allocation rate and utilization
    c. I/O allocation rate and utilization
    d. The specific complexity of the code being monitored.

In Phase 3, using the current implementation of the Pin tool, the typical overhead can be reduced to 100% - 500% (1x - 5x) depending on the level of granularity of the code verification desired and the resource protection.

Using static binary rewriting as an alternate implementation to the Pin tool, it is estimated that the overhead in Phase 3 will be reduced to 10% - 150% for the same ranges of granularity and number of simultaneous processes used to estimate current overhead.

In general, the overhead incurred from the resource drain tool is not dependent on the overhead of the other tools in the suite and a parallel container can be used for its implementation and policy enforcement.

4.9.8 Parrot/xtern Race Avoidance Container

Parrot works by intercepting synchronization calls, such as pthread_mutex_lock, which incurs overhead. Parrot also enforces a particular synchronization order, which has some overhead as well.

Parrot was integrated with a popular race detector, the Google ThreadSanitizer. This integrated tool reduced the performance overhead for ThreadSaniziter by 43% on a diverse set of 16 popular multithreaded programs on representative workloads.

During the STONESOUP T&E, this integrated tool was studied on all the 23 CWEs on a base program called CTREE, which were divided into two categories: (1) eight of these CWEs involve data races and have source code tar bar, so they are supposed to be handled by the tool; (2) eight CWEs do not have source tar bar, and seven CWEs are inter-process races, so the tool is not designed to handle these fifteen CWEs. For the eight CWEs in category (1) that the tool is designed to handle, the tool correctly detected data races on 5 CWEs (414, 543, 663, 765, 833) in both good and bad IO pairs, and safely avoided the races in the other 3 CWEs (609, 820, 821) via PARROT's schedules. These T&E results show that the tool can effectively detect races in CWEs or safely avoid races. However, one caveat is that currently the tool is designed to simply detect races so that developers can fix them, and it is not designed to fix these bugs automatically.

4.9.9 Resource Monitor Tool

Figure 9 below shows the runtime for WireShark without monitoring, with dynamic (Pin) instrumentation and then static (Dyninst) instrumentation.

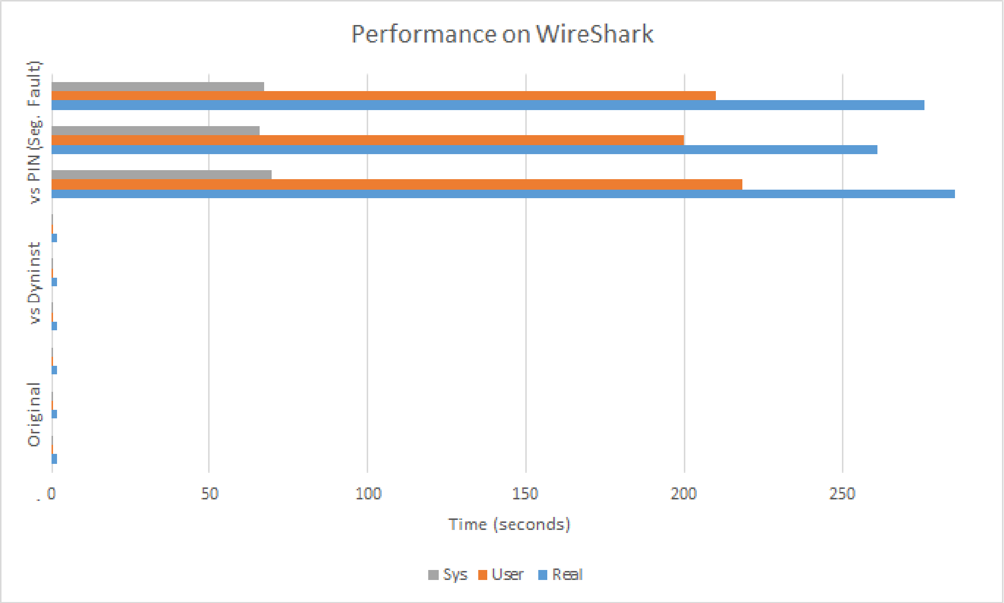


Figure 9: Wireshark Performance Comparison between Instrumentation Types

The resource monitoring system (resmon) effectively instrumented and monitored all base programs attempted. The following C programs were all instrumented using the static binary instrumentation approach: GIMP, WireShark, Subversion, FFMPEG, OpenSSL, Postgresql. Due to issues with the testing system, detailed IV&V results are not available for OpenSSL and Postgresql. Each of the tests for the remaining programs has been reviewed to determine how the resmon system functioned. The following issues were encountered which caused errors reported at IV&V.

GIMP – The plugins were patched in addition to the base program. Two of the plugins had extremely high CPU usage which triggered our CPU exhaustion monitor. This is the expected and desired behavior when CPU usage reaches 100% which was not seen during basic usage of the program. The remedy is to either train with all plugins enabled so the system learns 100% CPU is acceptable or disable the CPU monitor altogether for GIMP.

Subversion – Some subversion results were stored in the “run” directory which was not seen by the test system. The evaluation shows the technology should catch the faults injected into the program.

FFMPEG – One test set did fail to be recognized by our system due to it using mkstemp() which was not monitored. This is a capability lacking within the system which can be addressed in the future. Additionally, some tests exited with a segfault, however these were not in the resmon system. Once the segfaulting code is corrected, the resmon system should detect the exhaustion.

Wireshark - There were exhaustions in the Wireshark test cases which were not detected correctly by resmon. These tests did use a large amount of resources, but not enough to cross the detection threshold.

Overall, in lab testing the resource usage was detectable for all programs being tested in the final T&E. In some cases the resources being used did not cross the alert boundary and were not reported. The following table summarizes the internal pass/fail metrics based on evaluating the IV&V results available.

	Total	Passed	Failed
GIMP	144	142	2
WireShark	56	52	4
Subversion	20	20	0
FFMPEG	25	24	1
Total:	245 (100%)	238 (97%)	7 (3%)

Figure 10: Testing results for resmon

Another key metric is the overhead injected using the resmon system. The T&E results will provide overhead numbers for the entire MINESTRONE system so metrics are provided here for only the resmon subsystem.

The following table shows the average real time for programs when running the baseline (no monitoring) and the percent increase when running static binary instrumentation using DynInst and dynamic binary instrumentation using Pin.

	Baseline time	DynInst % increase	Pin % increase
GREP	0.589s	69.10%	19667%
GIMP	2.49s	178%	17494%
WireShark	1.704s	1.27%	15987%
FFMPEG	0.15s	576.17%	27156%

Figure 11: Instrumentation performance overhead

The FFMPEG overhead for static instrumentation is large; however, this is also a very fast test. The actual time delay is less than 1 second. As described before, Pin consistently adds a large amount of overhead versus static instrumentation. This is due to the context switch into and out of the monitoring code. While the static instrumentation is substantially better, gradually reducing the monitoring calls’ frequency can further reduce the overhead incurred when monitoring an executable.

4.9.10 KLEE

Over the past year the GMU team evaluated the effectiveness of KLEE for resource exhaustion monitoring. The typical goal of KLEE is to provide inputs that result in 100% test coverage for an executable. Our goal was to use these inputs to drive the training of our system to develop a comprehensive normal model. Additionally, KLEE would be used to generate symbolic constraints for the inputs. The constraints could be used to “bin” new unseen inputs to determine which resource flow the executable would process. The final result is a simple way to predict resource usage for an executable based on the inputs.

The theoretical approach is sound, however the investigation found severe limitations in using KLEE. KLEE, in its present form, has limited use for small programs, but is not robust enough to be used generally as part of our resource monitoring tool-based anomaly detection system. KLEE requires the target program to be compatible with C89 and compilable in llvm. This means that none of the tests programs proposed for STONESOUP can be used with KLEE without extensive work modifying KLEE as shown in the table below.

	Compiles in C89?	**Compiles to llvm?**	**KLEE coverage**
Cherokee	No	No	-
Grep	No	No	-
Vim	No	No	-
Wget	No	No	Many invalid paths
Imagemagick	No	No	-
Mono	No	No	-

Figure 12: KLEE Evaluation

Without modifying KLEE’s source code to identify and characterize all the inputs, KLEE does not provide complete path coverage. For example, KLEE doesn’t identify paths or symbolic constraints for: *unconditional indirect branches, non-command line inputs, calls to non-libC libraries, threads, inline assembly, some simple conditions*. If an input is not characterized, KLEE either identifies only one path associated with that input or generates a large number of invalid paths. Using KLEE as intended, in the merge path mode, KLEE will generate valid paths for any identified inputs, but since KLEE uses random path selection, it will not define the unique set of paths but with some paths identified as multiple distinct paths and some paths not identified at all.

Due to these problems, it was concluded that KLEE is not useable when building a monitoring system to handle a corpus of large, real-world programs.

5.0 CONCLUSIONS

The contract terms were successfully executed in full, including the delivery of testable technology along with evaluation results that demonstrate its utility. The work on MINESTRONE encompassed many branches of research in multiple areas and was encapsulated into a encompassing suite that was comprehensively evaluated to demonstrate the effectiveness of the system. These results have led to several papers published in top operating systems and programming languages conferences.

6.0 PUBLISHED PAPERS

Many of the publications can be found in PDF form at:

http://nsl.cs.columbia.edu/projects/minestrone/?p=2

- “Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading” Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, Gang Hu. CACM '14.
- "The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines", To appear in Proceedings of the Network and Distributed System Security (NDSS) Symposium, San Diego, CA, USA, February 2015
- "IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking" Kangkook Jee, Theofilos Petsios, Marios Pomonis, Michalis Polychronakis, and Angelos D. Keromytis. To appear in Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC). December 2014, New Orleans, LA.
- "The power of amnesia: Learning probabilistic automata with variable memory length" D. Ron, Y. Singer, and N. Tishby, ," Machine learning, vol. 25, no. 2-3, pp. 117–149, 1996.
- "Approximate variable-length time series motif discovery using grammar inference", .Li, Y. and Lin, J. (2010). In Proceedings of the 10th International Workshop on Multimedia Data Mining, in conjunction.Xing, Zhengzheng, Jian Pei, and S. Yu Philip. "Early Prediction on Time Series: A Nearest Neighbor Approach" IJCAI. 2009.
- "Early classification of multivariate time series using a hybrid HMM/SVM model", Ghalwash, Mohamed F., Dusan Ramljak, and Zoran Obradovic. Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on. IEEE, 2012.
- "Extraction of Interpretable Multivariate Patterns for Early Diagnostics", Ghalwash, Mohamed F., Vladan Radosavljevic, and Zoran Obradovic. Data Mining (ICDM), 2013 IEEE 13th International Conference on. IEEE, 2013.
- "Large-Scale Evaluation of a Vulnerability Analysis Framework", Nathan Evans, Azzedine Benameur, Matthew Elder, USENIX CSET ’14, August 2014.
- "Dynamic Reconstruction of Relocation Information for Stripped Binaries", Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. To appear in Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID). September 2014, Gothenburg, Sweden.
- "ret2dir: Rethinking Kernel Isolation" Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. To appear in Proceedings of the 23rd USENIX Security Symposium. August 2014, San Diego, CA.
- "Size Does Matter - Why Using Gadget-Chain Length to Prevent Code-reuse Attacks is Hard",Enes Goktas, Elias Athanasopoulos, Herbert Bos, Michalis Polychronakis and Georgios Portokalidis, Usenix Security, 2014.
- "Time Randomization to Thwart Concurrency Bug Exploitation", David Tagatac, Sal Stolfo 2014 IEEE S&P Poster Reception on May 19, 2014
- "Out Of Control: Overcoming Control-Flow Integrity", Proceedings of the 35th IEEE Symposium on Security and Privacy, San Jose, CA, USA, May 2014 (13.6%)

- "The Best of Both Worlds. A Framework for the Synergistic Operation of Host and Cloud Anomaly-based IDS for Smartphones", Proceedings of the 2014 European Workshop on System Security (EUROSEC), Amsterdam, The Netherlands, April 2014 (42.9%)
- "The Other Side of the Fence: How to Protect Against Code Injection Attacks", Azzedine Benameur, Nathan Evans, Matthew Elder, PenTest Magazine, November 2013.
- "ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking" Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. To appear in Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS). November 2013, Berlin, Germany.
- "Practically Making Threads Deterministic and Stable", Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Junfeng Yang, Garth A. Gibson, Randal E. Bryant. Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP). Farmington, PA, November, 2013.
- "An Accurate Stack Memory Abstraction and Symbolic Analysis Framework for Executables", Kapil Anand, Khaled Elwazeer, Aparna Kotha, Matthew Smithson, Rajeev Barua and Angelos D. Keromytis. To appear in the Proceedings of the 29th IEEE International Conference on Software Maintenance (ICSM). Eindhoven,Netherlands, September 2013.
- "Effective Dynamic Detection of Alias Analysis Errors", Jingyue Wu, Gang Hu, Yang Tang, Junfeng Yang. Proceedings of the Ninth joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE). August 2013, Saint Petersburgh, Russia.
- "MINESTRONE: Testing the SOUP", Azzedine Benameur, Nathan S. Evans, Matthew C. Elder. Proceedings of the 6th Workshop on Cyber Security Experimentation and Test (CSET). August 2013, Washington, DC.
- "Determinism Is Overrated: What Really Makes Multithreaded Programs Hard to Get Right and What Can Be Done about It", Junfeng Yang, Heming Cui, Jingyue Wu. To appear in the Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HOTPAR '13). June, 2013, San Jose, CA.
- "Redundant State Detection for Dynamic Symbolic Execution", Suhabe Bugrara and Dawson Engler. To appear in the Proceedings of the USENIX Annual Technical Conference (ATC). June 2013, San Jose, CA.
- "Transparent ROP Exploit Mitigation using Indirect Branch Tracing", Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. To appear in Proceedings of the 22nd USENIX Security Symposium. August 2013, Washington, DC.
- "Expression Reduction from Programs in a Symbolic Binary Executor", Anthony Romano and Dawson Engler. To appear in Proceedings of the International SPIN Symposium on Model Checking of Software. July 2013, Stony Brook, NY.
- "SPECTRE: A Dependable Introspection Framework via System Management Mode", Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. To appear in the Proceedings of 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Performance and Dependability Symposium(PDS). June 2013, Budapest, Hungary.
- "Verifying Systems Rules Using Rule-Directed Symbolic Execution", Heming Cui, Gang Hu, Jingyue Wu, Junfeng Yang. To appear in the Proceedings of the Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS). March 2013, Houston, TX.

- "When Firmware Modifications Attack: A Case Study of Embedded Exploitation", Ang Cui, Michael Costello, Salvatore J. Stolfo. In Proceedings of NDSS 2013, February 2013, San Diego, CA.
- "kGuard: Lightweight Kernel Protection", Vasileios P. Kemerlis, Georgios Portokalidis, Elias Athanasopoulos, and Angelos D. Keromytis. In USENIX;login: Magazine, November 2012.
- "Self-healing Multitier Architectures Using Cascading Rescue Points", Angelika Zavou, Georgious Portokalidis, and Angelos D. Keromytis. In the Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC). December 2012, Orlando, FL. (Acceptance rate: 19%)
- "Adaptive Defenses for Commodity Software Through Virtual Application Partitioning", Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. In the Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS). October 2012, Raleigh, NC. (Acceptance rate: 18.9%)
- "Practical Software Diversification Using In-Place Code Randomization", Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. In "Moving Target Defense II: Application of Game Theory and Adversarial Modeling", Sushil Jajodia, Anup K. Ghosh, V. S. Subrahmanian, Vipin Swarup, Cliff Wang, and X. Sean Wang (editors), pp. 169 - 196. Springer, 2012.
- "kGuard: Lightweight Kernel Protection against Return-to-user Attacks", Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. In the Proceedings of the 21st USENIX Security Symposium. August 2012, Bellevue, WA. (Acceptance rate: 19.4%) (co-sponsored by DARPA)
- "Concurrency Attacks", Junfeng Yang and Ang Cui and Salvatore J. Stolfo, and Simha Sethumadhavan. In the Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar). June 2012, Berkeley, CA. (co-sponsored by DARPA, NSF, and ONR)
- "A Dependability Analysis of Hardware-Assisted Polling Integrity Checking Systems", Jiang Wang, Kun Sun, Angelos Stavrou. In the Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (IEEE DSN). June 2012, Boston, MA. (co-sponsored by the NSF and ARO)
- "Sound and Precise Analysis of Multithreaded Programs through Schedule Specialization", Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, Junfeng Yang. In the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). June 2012, Beijing, China. (co-sponsored by the NSF)
- "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization", Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. In the Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P), pp. 601 - 615. May 2012, San Francisco, CA. (co-sponsored by DARPA)
- "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems", Vasileios Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. In Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE). March 2012, London, UK. (co-sponsored by the NSF and DARPA)
- "SecureSwitch: BIOS-Assisted Isolation and Switch between Trusted and Untrusted Commodity OSes", Kun Sun, Jiang Wang, Fengwei Zhang and Angelos Stavrou. In Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed

Systems Security (SNDSS). February 2012, San Diego, CA. (Acceptance rate: 17.8%) (co-sponsored by the NSF and ARO)

- "A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware", Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. In Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed Systems Security (SNDSS). February 2012, San Diego, CA. (Acceptance rate: 17.8%) (co-sponsored by the NSF and DARPA)
- "From Prey To Hunter: Transforming Legacy Embedded Devices Into Exploitation Sensor Grids", A. Cui, Jatin Kataria, Salvatore J. Stolfo. In Proceedings of the Annual Computer Security Applications Conference (ACSAC), December 2011, Orlando, FL. (co-sponsored by DARPA)
- "ROP Payload Detection Using Speculative Code Execution", Michalis Polychronakis and Angelos D. Keromytis. In Proceedings of the 6th International Conference on Malicious and Unwanted Software (MALWARE). October 2011, Fajardo, PR. (co-sponsored by the NSF and DARPA)
- "Killing the Myth of Cisco IOS Diversity: Recent Advances in Reliable Shellcode Design", Ang Cui, Jatin Kataria, Salvatore J Stolfo. In Proceedings of the USENIX Workshop on Offensive Technology (WOOT), August 2011, San Francisco, CA. Also presented at Black Hat 2011.) (co-sponsored by DARPA)
- "Practical Software Model Checking via Dynamic Interface Reduction", Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, Lintao Zhang, Proceedings of the 23rd ACM Symposium on Operating Systems Principles(SOSP '11), October, 2011. (co-sponsored by the NSF)
- "Efficient Deterministic Multithreading through Schedule Relaxation", Heming Cui, Jingyue Wu, John Gallagher, Junfeng Yang. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), October 2011, Cascais, Portugal. (co-sponsored by the NSF)
- "Pervasive Detection of Process Races in Deployed Systems", Oren Laadan, Chia-che Tsai, Nicolas Viennot, Chris Blinn, Peter Senyao Du, Junfeng Yang, Jason Nieh. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), October 2011, Cascais, Portugal. (co-sponsored by the NSF)
- "Taint-Exchange: a Generic System for Cross-process and Cross-host Taint Tracking", Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. In Proceedings of the 6th International Workshop on Security (IWSEC). November 2011, Tokyo, Japan. (co-sponsored by the NSF)
- "REASSURE: A Self-contained Mechanism for Healing Software Using Rescue Points", Georgios Portokalidis and Angelos D. Keromytis. In Proceedings of the 6th International Workshop on Security (IWSEC). November 2011, Tokyo, Japan. (co-sponsored by the NSF and AFOSR)
- "Defending Legacy Embedded Systems with Software Symbiotes", Ang Cui and Salvatore J. Stolfo. In Proceedings of the 13th Symposium on Recent Advances in Intrusion Detection (RAID). September 2011, Menlo Park, CA. (co-sponsored by DARPA)
- "DoubleGuard: Detecting Intrusions In Multi-tier Web Applications", Meixing Le, Angelos Stavrou, Brent ByungHoon Kang. In the IEEE Journal on Transactions on Dependable and Secure Computing (TDSC), 2011. (co-sponsored by the NSF)

- "Finding Concurrency Errors in Sequential Code --- OS-level, In-vivo Model Checking of Process Races", Oren Laadan, Chia-che Tsai, Nicolas Viennot, Chris Blinn, Peter Senyao Du, Junfeng Yang, and Jason Nieh. In Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS). May 2011, Napa Valley, CA. (co-sponsored by the NSF)
- "The MINESTRONE Architecture: Combining Static and Dynamic Analysis Techniques for Software Security", Angelos D. Keromytis, Salvatore J. Stolfo, Junfeng Yang, Angelos Stavrou, Anup Ghosh, Dawson Engler, Marc Dacier, Matthew Elder, and Darrell Kienzle. In Proceedings of the 1st Workshop on Systems Security (SysSec). July 2011, Amsterdam, Netherlands.
- "Practical, low-effort verification of real code using under-constrained execution", David A. Ramos and Dawson Engler. In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV). July 2011, Snowbird, UT.
- "Retrofitting Security in COTS Software with Binary Rewriting", Padraig O'Sullivan, Kapil Anand, Aparna Kothan, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. In Proceedings of the 26th IFIP International Information Security Conference (SEC), pp 154 - 172. June 2011, Lucerne, Switzerland. (co-sponsored by DARPA and NSF)
- "Firmware-assisted Memory Acquisition and Analysis tools for Digital Forensic (short paper)", Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. In Proceedings of the Sixth International Workshop on Systematic Approaches to Digital Forensic Engineering (IEEE SADFE 2011). In conjunction with IEEE Security and Privacy Symposium. May 2011, Oakland, CA.
- "Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution", Georgios Portokalidis and Angelos D. Keromytis. In Proceedings of the ARO Workshop on Moving Target Defense. October 2010, Fairfax, VA. (co-sponsored by NSF and AFOSR)
- "Stable Deterministic Multithreading through Schedule Memoization", Heming Cui, Chia-che Tsai and Junfeng Yang. In Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 207 – 222. October 2010, Vancouver, Canada. (co-sponsored by the NSF)
- "Bypassing Races in Live Applications with Execution Filters", Jingyue Wu, Heming Cui and Junfeng Yang. In Proceedings of the 9th USENIX Symposium Operating Systems Design and Implementation (OSDI), pp. 135 - 150. October 2010, Vancouver, Canada. (co-sponsored by the NSF)
- "Fast and Practical Instruction-Set Randomization for Commodity Systems", Georgios Portokalidis and Angelos D. Keromytis. In Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC), pp. 41 - 48. December 2010, Austin, TX. (Acceptance rate: 17%) (co-sponsored by the NSF)

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Acronym	Nomenclature
ACM	Association for Computing Machinery
ARM	Advanced RISC Machine
ASLR	Address Space Layout Randomization
BAA	Broad Agency Announcement
CCS	Computer and Communications Security
CPU	Central Processing Unit
CTREE	Conditional Inference Tree
CWE	Common Weakness Enumeration
DBI	Dynamic Binary Instrumentation
DBT	Dynamic Binary Translator
DEP	Data Execution Prevention
DFT	Dynamic Flow Tracking
DIMVA	Detection of Intrusions and Malware & Vulnerability Assessment
DTA	Dynamic Taint Analysis
FCG	Function Call Graph
FTP	File Transfer Protocol
HTTP	Hyper Text Transfer Protocol
IARPA	Intelligence Advanced Research Projects Activity
IDT	Interrupt Descriptor Table
IOS	Internetwork Operating System
ISR	Instruction Set Randomization
I/O	Input/Output
LOC	Lines Of Code
MD5	Message-Digest 5
MIPS	Millions of Instructions Per Second
MMU	Memory Management Unit
MP	Memory Protection
NAT	Network Address Translation

Acronym	**Nomenclature**
OS	Operating System
PC	Program Counter
PDF	Portable Document Format
QEMU	Quick Emulator
RISC	Reduced Instruction Set Computer
ROP	Return-Oriented Programming
RP	Rescue Point
SAX	Symbolic Aggregate Approximation
SEP	Symbiotic Embedded Machines
SQL	Structured Query Language
SSH	Secure Shell Host
SPEC	Standard Performance Evaluation Corporation
TEXAS	Test & Evaluation Execution and Analysis System
TFTP	Trivial File Transfer Protocol
TXL	Turing eXtender Language
T&E	Test and Evaluation
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
VOIP	Voice Over Internet Protocol
XOR	Exclusive Or