Table of Contents

Section **Page**

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This report describes the research that our team has performed during the STONESOUP (Securely Taking On New Executable Software Of Uncertain Provenance) Program. This section summarizes the motivation, goal, approach, and results of this research. Sections 2, 3, and 4 provide details about the techniques that our solution is based on. Testing methods and results are discussed in Section 5, while Section 6 describes future work. The Appendices provide detail on how we addressed each CWE (Common Weakness Enumeration).

1.1 Motivation

The security weaknesses that plague today's applications provide relatively easy avenues of exploitation by malicious parties, posing a significant risk for users of the software. Due to market pressures and geographical distribution of development teams, even applications from reputable companies cannot be trusted to be immune from security weaknesses, as attested from the frequently issued security patches.

SQL (Structured Query Language) injection is either the #1 or the #2 security exploit, depending on which "Top *n*" list is read. Numerous government and commercial web sites have been compromised by this exploit, causing losses of millions of dollars. Other kinds of injection attacks (e.g. OS (Operating System) command injection) work in a very similar way to SQL injection, and their occurrence is expected to increase. Injection attacks are a special case of tainted data attacks, where unexpected data is sent to applications that do not properly validate their input data.

1.2 Goal

The goal of this research, and of the STONESOUP Program in general, is to give end users tools that can be used to automatically find and fix security weaknesses in applications. This enables users to run the (thus hardened) applications with more assurance.

Within the broad goals of the STONESOUP Program, our research focused on the following weakness classes in Java applications:

1. Injection (Phase 1).
2. Tainted data (Phase 1).
3. Numeric (Phase 2).
4. Error handling (Phase 2).
5. Resource drains (Phase 3).
6. Concurrency (Phase 3).

The goal of our research is to provide end users with a tool that automatically finds and fixes these weaknesses in Java applications.

The tool must operate automatically, with little or no user intervention. End users are often not trained in software development, and therefore any input to the tool from the user must be very high-level and expressed in terms of easy-to-understand policies in the domain of the application. This requirement for automation poses significant challenges on the tool: since a specification of

the application (or even just of its security policy) is not available, the tool must infer from the application code the *intended* specification of the application, finding and fixing any mismatches between code and (inferred) specification.

1.3 Approach

The root cause of the weakness classes mentioned above is a failure of the application to validate input data that may come from an attacker. The application may operate correctly under (possibly extensive) "friendly" testing, but may be susceptible to security or other failures when given "unexpected" inputs. Examples of vulnerable code can be found on MITRE's CWE web site at *cwe.mitre.org*, e.g. the page on CWE-89 shows examples of SQL injection.

Input validation checks are often completely missing. Other times they are incomplete or have some other shortcoming, such as occurring at the wrong time, e.g. before decoding (for instance, code that looks for single quotes in an HTTP (Hypertext Transfer Protocol) request is easily fooled by the encoding *%27* of the single quote character).

Automatically adding or fixing validation logic to an application's code is very difficult because the needed validation checks depend on how the input data is used, which may occur in code locations that are distant from the code locations that receive the data. A simpler and more effective approach, which we follow in VIBRANCE (Vulnerabilities in Bytecode Removed by Analysis, Nuanced Confinement, and Diversification), is to track the provenance of each piece of data, and defer the validation checks until the data is actually used by the application for potentially dangerous operations (e.g. calling an SQL API (Application Programming Interface)).

The provenance of data is tracked by a synergy of static analysis and run-time instrumentation:

- static tracking reduces the overhead of run-time instrumentation;
- run-time instrumentation improves the precision of static tracking.

Data that originates from within the application (e.g. string constants) is always regarded as trusted (i.e. not tainted). Data that originates from the external world (e.g. from files or from sockets) is regarded as untrusted (i.e. tainted) or trusted, depending on their source.

At the time in which data is about to be used in a dangerous operation (e.g. the data is passed to a vulnerable API method, such as a method that executes an SQL statement contained in a string that is passed as parameter to the method), appropriate validation checks are performed, by code that VIBRANCE adds to the application code. These validation checks take into account both the provenance of each part of the data (e.g. each character of the SQL string) and the semantics of the operation (e.g. the semantics of SQL). The validation checks enforce "best practices" (e.g. that there should be no unescaped quotes in untrusted SQL text that is within trusted quotes)—recall that no specification of the application is available, and that the tool must infer the intended policy.

If the validation checks fail, a corrective action is taken, which attempts to fix the data and continue execution, to the extent possible. For instance, in the SQL example above, an unescaped quote in untrusted text that is within trusted quotes is fixed by escaping it with another quote.

Tracking data provenance and applying validation checks when the data is about to be used suffices to prevent many weaknesses (e.g. injection and tainted data). Other weaknesses require the tracking of additional metadata (besides provenance) in order to apply appropriate confinement actions at the right time without disrupting legitimate functionality.

For example, numeric overflow is not always an error, and may in fact be intended functionality (e.g. in a hash computation). Thus, when a numeric overflow occurs, VIBRANCE (augments the application with code that) merely records the fact that the result of the operation is an overflowed value. If/when the overflowed value is used in a potentially dangerous operation (e.g. to index an array), VIBRANCE takes a corrective action (e.g. it prevents the array access).

As another example, excessive resource consumption can be thwarted only after a certain threshold of consumed resources is reached. In order to detect the threshold, VIBRANCE tracks suitable metadata, e.g. the number of iterations of a loop whose bound is controlled by external inputs, the number of file descriptors used by each thread, and so on. When certain configurable thresholds are reached, VIBRANCE takes corrective actions (e.g. break out of the loop, or interrupt the thread).

As a third example, in order to detect TOCTOU (Time-Of-Check-to-Time-Of-Use) concurrency errors, VIBRANCE tracks, for each file path (represented by a Java File object), the corresponding inode. Thus, if an attacker causes a file path to resolve to a different node after a check is made (e.g. by replacing a file with a symbolic link), VIBRANCE compares the current inode with the previously recorded inode and blocks the erroneous access if the two inodes are not equal.

In summary, VIBRANCE tracks a variety of metadata, using a combination of static analysis and run-time tracking. This metadata enables VIBRANCE to take confinement actions at the appropriate times with the necessary precision to avoid disrupting legitimate functionality.

Static analysis is not merely used to optimize run-time tracking. Static analysis plays a more essential role in VIBRANCE, by inferring information that would be very difficult or impossible to infer at run time. An example is the identification and extent of loops (with tainted bounds) and of server loops (which are protected by VIBRANCE's server guard). Another example is the identification of potential data races, which VIBRANCE prevents by adding appropriate synchronization.

The combination of static analysis, run-time tracking, and run-time confinement should suffice to prevent all attacks (from the six weakness classes listed above), in principle. However, for injection attacks, an additional, independent line of defense can be taken. Injection attacks work by introducing unexpected keywords into strings. Therefore, if (expected) keywords are randomized, an attacker could not easily guess the correct randomization for the keywords to inject (provided the randomization space is sufficiently large). Based on this observation, the second line of defense consists in randomizing the keywords that occur in constant strings in the application code, and wrapping the API methods that execute the commands to check and de-randomize the keywords. In addition, other API methods calls (e.g. of string manipulation

methods) must be wrapped to make the randomization transparent and to keep it from the attacker's view.

The idea of keyword randomization is not new, but previously published keyword randomization approaches require manual modification of the application code. VIBRANCE's novel contribution to research on keyword randomization defense is that all the necessary code modifications are carried out in a completely automatic way.

1.4 Results

Based on the approach described above, we have built a tool, called VIBRANCE, whose architecture is shown in Figure 1.

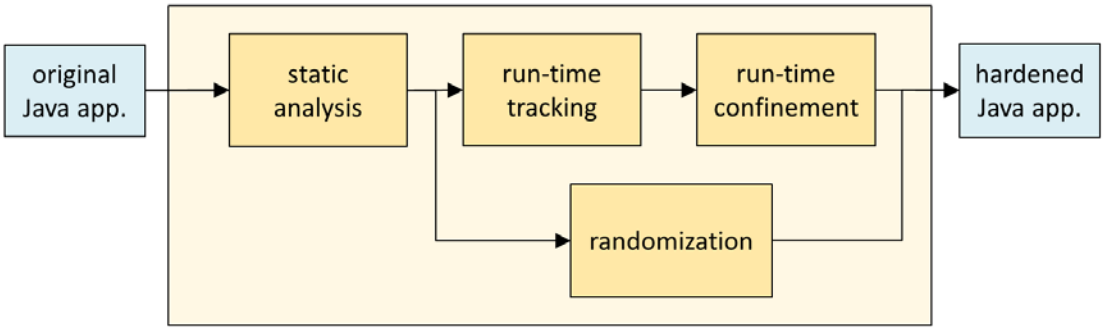

Figure 1: Architecture of VIBRANCE

VIBRANCE uses static analysis, run-time tracking, confinement, and randomization, to automatically harden Java bytecode with comprehensive and precise protection from the attacks enables by the six weakness classes listed earlier. For a large class of attacks, the protection added by VIBRANCE blocks the attacks and safely continues execution. Besides addressing specific classes of attacks, VIBRANCE can conservatively handle some unforeseen attacks, using a backstop confinement mechanism.

Most of VIBRANCE's protections are not hard-wired in the implementation, but can be customized via a configuration file (not shown in Figure 1) that enables fine-tuning of the checks to be performed on tainted data, of the actions to be taken when those checks fail, as well as of the designation of trusted and untrusted data sources. The default policy in the configuration file should be adequate to most users and applications.

VIBRANCE has been tested extensively, both by our team and by the STONESOUP T&E teams (MITRE and TASC). The test results have been quite positive—see Section 6 for details. VIBRANCE is robust and ready for end users.

VIBRANCE manipulates applications in their Java bytecode format, and not in their Java source format. This choice is motivated by the fact that bytecode is always available, while source may

not be available. Furthermore, bytecode is a simpler language to manipulate than source. There is a slight information/abstraction loss when going from source to bytecode, but the loss is minimal, can be often recovered, and is amply counter-balanced by the two advantages of bytecode mentioned just above (namely, universal availability and greater simplicity).

2 STATIC ANALYSIS

The static analysis component of VIBRANCE is based on the CodeHawk tool (Venet, 2007). In the VIBRANCE project, we have extended CodeHawk with the following capabilities:

- Java front end that parses and translates Java class files into the CodeHawk internal language to enable analysis with the CodeHawk engine;
- Taint tracking of numeric and symbolic variables;
- Server loop inference;
- Tainted loop bound inference.
- Integer overflow/underflow and truncation analysis
- Analysis of error-return values, null-return values
- Syntactic analyses to determine the extent of catch blocks and return instructions in finally blocks
- Resource analysis
- Concurrency analysis

This CodeHawk extension is the static analysis component of the VIBRANCE tool, shown in Figure 1.

2.1 Background on CodeHawk

The CodeHawk static analysis technology has been in development for several years (Venet, 2007). CodeHawk is a customizable static analysis engine based on the mathematical theory of abstract interpretation (Cousot & Cousot, Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, 1977) (Cousot & Cousot, 1979), implemented in OCAML. It computes over-approximations of the program behavior in decidable domains including numerical domains such as intervals, linear equalities and polyhedra, and symbolic domains.

CodeHawk consists of a programming-language independent abstract-interpretation engine and three language front ends for Java, C, and x86 executables. The abstract-interpretation engine provides the following components:

- CHIF (CodeHawk Internal Form), the internal language in which programs are represented and on which analysis is performed;
- Abstract domains, that provide semantics for all constructs in the language possibly augmented with custom operations relative a particular decidable theory;
- Fixpoint iterators, highly optimized algorithms that perform flow-sensitive forward and backward propagation of the semantics encoded in the domains.

The fixpoint iterators are completely transparent to the user and do not need further elaboration. CHIF and the abstract domains are described in some more detail below.

2.1.1 CodeHawk Internal Form

CHIF is an imperative language. Data types include integers, structs and arrays. Expressions include arithmetic expressions, boolean expressions, and expressions on sets. The language

supports both structured and unstructured control flow. Control flow constructs can be arbitrarily nested, e.g., loops may contain arbitrary control flow graphs constructed from jumps, as long as the control flow graphs stay contained within the loops. Breakout blocks are provided to enable representation of otherwise structured loops with exits to the statement immediately following the loop.

The language provides assignment statements for all data types. Generic, named operations are provided to enable assignments with user-defined semantics for operators not directly supported by CHIF expressions, e.g., bitwise operations.

The language also includes analyzer directives, including commands to activate or de-activate domains, assert the validity of expressions, transfer values from one domain to another, designate certain regions of the code to establish summary transfer relations, and operations directed at custom domains to inject constraints at particular locations in the program.

2.1.2 Abstract domains

An abstract domain is a decidable theory. It consists of a finite or infinite set of elements that form a lattice, with well-defined meet and join operations and a bottom (inconsistent, or unreachable) and top (no information) element. Furthermore, it provides forward and backward transformers for all dataflow constructs in the language as well as for assert statements. The transformers are guaranteed to be an over-approximation of the concrete semantics of the operations being modeled. Custom domains may include an arbitrary number of domain operations that define transformers for custom constraint generation.

The core system provides standard numerical and symbolic domains, including intervals (ranges over integers), linear equalities, and linear inequalities (polyhedra), and symbolic sets. New domains can be added relatively easily.

2.2 Java Front End

2.2.1 Overview

The CodeHawk Java Front End, developed as part of the VIBRANCE project, provides the interface between the byte code and the CodeHawk engine. It abstracts the byte code into CHIF, directs the analysis, interprets the analysis results, and translates the result back in terms of byte code to be consumed by the runtime system. The Java Front End consists of the following components:

- **Parser**: parses Java byte code into OCAML data structures;
- **Hierarchy analysis**: determines which classes need to be loaded and which methods need to be parsed;
- **Translator**: transforms the OCAML data structures created by the parser into CHIF;
- **Preprocessor**, transforms the CHIF into SSA (static single assignment) form;
- **Taint analysis:** intra- and inter-procedural propagation of taint on primitive values and references;

- **Loop-bound analysis:** analysis to identify loops whose number of iterations may be directly or indirectly under the influence of the user.
- **Error-return value analysis:** analysis that follows values identified by method summaries as error-values, and determines if the value is correctly checked for;
- **Null-value analysis:** analysis that follows return values from methods that may return null and determines if the return value is properly checked before dereferencing;
- **Resource analysis:** analysis that identifies arguments passed to library methods that denote resources
- **Concurrency analysis:** analysis that identifies static and object fields that may be read and written by threads that run concurrently, thus creating the possibility of data races.

The following sections will describe each of these components in more detail.

2.2.2 Java Byte Code Parser

Many Java byte code parsers exist. Therefore the first decision to make was whether to build upon an existing parser or implement our own. We evaluated several front ends, including ASM, BCEL (Bytecode Engineering Library), and JavaLib, but ultimately decided to create our own parser. ASM and BCEL would have required a Java-OCAML language interface, complicating the design of the front end. Using JavaLib, written in OCAML, seemed attractive initially, but required extensive conversion of data structures to fit with standard data structures used in CodeHawk, such that a complete rewrite of the parser was more efficient and provides a better foundation in the long run.

The Java class file format is precisely specified and thus the implementation of a parser consists of a careful design of the target data structures and a rather straightforward engineering effort to cover all constructs of the class file.

2.2.3 Hierarchy Analysis

Virtually all Java applications rely heavily on library classes and methods. Due to virtual and interface method invocations it is a nontrivial (undecidable, in fact) task to statically determine which library methods may be called at runtime. Therefore the first task in the static analysis of a Java application is to construct an over-approximation of the call graph, that is the set of classes and methods to be considered in the analysis.

Much research has been performed into call graph construction for object-oriented languages, also called hierarchy analysis, especially in the context of compiler optimization. Execution of a direct call instruction takes much less time than a virtual call instruction that requires method lookup. Therefore compilers try to statically resolve as many virtual calls at compile time to increase runtime performance. Virtual calls can be converted to direct calls when it can be statically determined that either the set of receiver object class types is a singleton set, or none of the subclasses overrides the method called.

The context in most of the research in call graph construction is a closed-world assumption: all classes and their relationships are known and available to the analysis. Furthermore, call graph

construction is pessimistic: the starting point is the largest possible call graph in terms of type compatibility and then edges are pruned based on constraints generated. One of the first approaches to class hierarchy analysis, generally referred to as CHA (Dean, Grove, & Chambers, 1995), assumes that any subclass of a declared type can be the receiver of a virtual method call. Reduction in the set of target methods is obtained by pruning receiver types that do not override the method. Rapid Type Analysis (referred to as RTA) (Bacon & Sweeney, 1996), improved on this approach by pruning all classes that are not explicitly instantiated within the application. Tip and Palsberg investigated several propagation-based algorithms for call-graph construction and compared their performance with RTA (Tip & Palsberg, 2000). Although some improvement in precision was observed, analysis times increased by an order of magnitude. They allowed for library classes to be represented by a single "superclass". A similar comparison performed by DeFouw (DeFouw, Grove, & Chambers, 1998) reached the same conclusion. Variable Type Analysis (VTA) and Declared Type Analysis (DTA) (Sundaresan, et al., 2000) refine RTA by creating variable dependency graphs to determine the set of types that can reach a particular object reference. All of these approaches are compared in lattice-theoretic framework in (Grove & Chambers, 2001).

The goals and context of hierarchy analysis in our analysis environment are somewhat different from that in compiler optimization. Our primary goal is to construct the smallest possible over-approximation of the execution call graph; we are less concerned with creating singleton sets specifically. The context is different in that we make extensive use of function summaries for library functions that summarize parts of the call graph, but at the same time may hide the existence of instantiated types in the system. Furthermore, all SQL database front end applications use dynamic loading of classes that implement the *java.sql* interfaces, a situation not considered in any of the above studies.

Our initial implementation was a straightforward RTA-type algorithm using the pessimistic approach in call graph construction. It quickly became clear that this approach was far too imprecise: too many methods were included in the analysis, thereby increasing the number of instantiated types, increasing even more the number of methods, resulting in a vicious cycle that was difficult to recover from. We then turned to an optimistic approach, interleaving call graph construction with hierarchy analysis and dynamically updating the set of edges emanating from methods processed earlier as new instantiated types appeared. This approach has a much more complicated implementation but yields considerably smaller call graphs.

The implementation starts with a set of "main" methods, which may include all methods in a particular application, or a set of designated methods that can start an execution tree, such as "main", "run", or "doGet", "doPut", "doPost", "doDelete" in servlet applications. The methods are added to a worklist. A simplified description of the implementation follows.

Define the following sets:
C: set of potentially instantiated types
M: methods to be analyzed by the taint analyzer
C_{wl}: worklist of classes to be loaded
M_{wl}: worklist of methods the processed by hierarchy analysis
V_c: set of virtual call sites
I_c: set of interface call sites

Initially: add all main methods to M_{wl}.

For each class *c* in C_{wl}:
    load and parse *c*;
    add *c* to **C**;
    add c.<clinit> to M_{wl};
    add superclass to C_{wl};
    for each method *c.m*:
        if *c.m* can be a target of any call site in V_c or I_c then add *c.m* to M_{wl}

For each method *m* in M_{wl}:
    add *m* to **M**;
    for all types c in the signature of *m* add c to C_{wl};
    for each of the following instructions in *m* do:
        new/checkcast/instanceof/newarray/multiarray, field operations: add target type to C_{wl};
        invokespecial (c,m)/invokestatic(c,m): add c to C_{wl}, add m to M_{wl};
        invokevirtual (c,m): add c to C_{wl}; for each subclass s of c, add s.m, if defined, to M_{wl};
            add instruction to V_c;
        invokeinterface(i,m): for each class s in **C** implementing i, add s.m, if defined, to M_{wl};
            add instruction to I_c.

The actual implementation is more complex due to the presence of summarized classes and methods. Methods that are summarized are not added to the method worklist. Classes are only loaded if a method call is encountered that is not summarized. We currently have close to 10,000 method summaries encoded in XML (Extensible Markup Language) format. Each summary is identified by its signature and states whether taint is generated by the method and how taint is transmitted among its arguments and return value, as well as many other properties of the method (see section 2.3).

A further complication in class hierarchy is dynamic class loading and reflection. The studies mentioned above do not deal with these. Some more recent work presents partial solutions for these cases. In (Livshits, Whaley, & Lam, Reflection analysis for Java, 2005) reflection is handled by propagating string constants and resolving names of classes and methods in reflective calls. In (Bodden, Sewe, Sinchek, Oueslati, & Mezini, 2011) dynamic analysis is used to collect targets of reflective calls dynamically and this information is then added to the class files to enable static analysis tools to perform a sound (relative to the completeness of the data collected) analysis in the presence of reflective calls and dynamically loaded classes.

Our implementation has limited support for dynamic class loading, targeted at this time specifically at the *java.sql* API, which contains mostly interfaces and relies on dynamically identified implementations, specific to the database being used, for maximum flexibility. The specific drivers are identified by the class *java.sql.DriverManager*, which, as part of its initialization attempts to load the drivers identified by a certain system property that is set in the user's system environment. As SQL were a major part of our target application, we had to handle at least this case. We chose to let the user identify the jars related to the database front end. All methods in classes implementing the *java.sql* interfaces were assumed to be possible targets in the corresponding interface calls encountered.

2.2.4 Translator

The translator is a combination of a semantics-preserving transformation and a semantic over-approximation. The transformation converts the stack-based representation of Java byte code into a register-based representation. The over-approximation abstracts all byte code instructions into CHIF instructions such that all behaviors of the original byte code are included in the behaviors of CHIF.

The Java Virtual Machine is stack-based, that is, all byte-code operations take their arguments from an operand stack and push the result back on the same stack. Most static analysis engines, including CodeHawk, are register-based. To enable analysis with the CodeHawk engine, the stack-based byte code operations must first be converted into a stack-less representation. This conversion is achieved by simulating the stack explicitly. The same approach is described and formally proved correct in (Demange, Jensen, & Pichardie, 2010). At each instruction the stack is copied from the previous instruction and modifications are made according to the specification of the byte code instruction. A potential concern about this approach, raised in (Logozzo & Fahndrich, 2008), is the large increase in the number of variables, which negatively affects the scalability of all subsequent analyses, especially those on relational domains such as polyhedra. We mitigate this problem by subsequent simplification of the generated CHIF code by copy propagation and static single assignment (see next section), in which all redundant variables are eliminated, resulting in a manageable representation. The advantage of our approach it its simplicity: each step is simple and can therefore easily be validated. The combination of steps achieves similar code sizes as a more complicated one-pass approach would have achieved.

All byte-code instructions must be translated into CHIF. The expressiveness of CHIF, however, is less than that of the byte code instructions. For example, CHIF does not have any floating-point operations. To ensure sound analysis, byte-code instructions that cannot be represented precisely must be over-approximated, such that all behaviors of the original program are guaranteed to be represented in the set of behaviors of the CHIF representation. The simplest over-approximation is to assume that any variables written may have any value, which is appropriate for floating-point operations. An alternative method for over-approximation is offered with uninterpreted operations, provided in CHIF to enable user-defined, analysis-specific semantics. These operations are used for field reads and field updates, method invocations, and special operations such as `new` and `newarray`, where the semantics can be assigned according to the analysis that is performed. In each case care is taken that no behaviors of the original program are removed by the chosen semantics.

All Java byte code instructions except JSR are covered in our translator—note that JSR cannot appear in Java 6. Conversion to a stack-less representation is semantics-preserving, over-approximation is used where CHIF lacks expressiveness, or analysis information is needed to resolve the actual semantics. Exception control flow is over-approximated by assuming a potential edge in the control flow graph from any instruction that can throw an exception to an exceptional-exit of the method.

2.2.5 Preprocessing: Static Single Assignment

The two principal approaches in static program analysis are *path-sensitive* analysis and *flow-sensitive* analysis. Path-sensitive analysis duplicates the program state at each decision point and computes subsequent states for each branch independently, resulting in a symbolic execution tree. Flow-sensitive analysis operates on the control-flow graph, performing a join of states whenever branches merge.

Path-sensitive analysis is often more precise than flow-sensitive analysis, as it "remembers" all decisions made so far. With flow-sensitive analysis past decisions are lost at each join point. The main drawback of path-sensitive analysis, however, is that paths can only be analyzed to a finite depth, that is, paths need to be truncated, which severely limits its application in sound static analysis, in particular in the presence of loops, as most approaches for truncation compromise soundness.

Analysis based on abstract interpretation has traditionally been flow-sensitive. Some attempts have been made, for example in the ASTREE analyzer (Cousot, et al., 2005), to introduce limited path sensitivity by keeping branches artificially apart after join points with the use of boolean variables. These approaches, however, have been ad-hoc and application-specific. The use of SSA is a more generic approach at introducing path sensitivity on a per-variable basis that has, to the best of our knowledge, not been tried before in combination with abstract interpretation.

The main problem with introducing SSA in an abstract-interpretation framework is giving semantics to the ϕ operator. Consider the following example

```
...
if (a > 0) {
  x = 14 ;
} else {
  x = 42;
}
y = x + 3;
...
```

which is transformed into SSA as follows

```
...
if (a > 0) {
  x1 = 14;
} else {
  x2 = 42;
} ;
x3 = ϕ ( x1; x2) ;
y = x3 + 3;
...
```

The variable x3 will take the value of x2 if branch 1 is executed and it will take the value of x2 if branch 2 is executed.

Most abstract interpretation frameworks, including the CodeHawk abstract interpretation engine, assume a pure register-based system representation, that is, it is assumed that there is a one-to-one correspondence between variables and their denotations. Phi nodes express a disjunction of values at the level of single variables using references to the values of other variables at different control points. It is hard to adapt existing relational domains to using this form of disjunction. The implementation of the join would need to keep track of the information on variables that would otherwise be discarded because they are no longer in scope. This significantly complicates memory management in the abstract-interpretation engine.

To solve this problem we perform a number of preprocessing steps: we first transform the system into full SSA. Then we substitute the ϕ operators by assignments to a common variable just before the join is performed, as illustrated below for the simple example shown earlier.

```
...
if (a > 0) {
  x1 = 14;
  x3 = x1;
} else {
 x2 = 42;
 x3 = x1;
}
y = x3 + 3;
...
```

By using the *dominance frontier approach*, as described by Cytron et. al. (Cytron, Ferrante, Rosen, Wegman, & Zadeck, 1991), to position the ϕ operators, we minimize the number of new variables that must be introduced.

The next step in the preprocessing reduces the number of variables in the system. The simulation of the stack operations in CHIF creates many stack variables and many assignments between these at each point in the program. Transforming the program into an SSA multiplies the variables by creating multiple versions of each and adding ϕ-variables. However, many of these variables represent the same quantity or reference, so our solution to the problem was to compute equivalence classes with respect to equality (as inferred from the byte-code, as actual equality cannot be determined statically) and replace all the equivalent variables with one representative. The code is then simplified by removing redundant assignments.

After preprocessing, there is an almost one-to-one correspondence between variables and sources of information, such as arithmetic operations, invocation, constant assignments, etc. This allows for increased precision in analysis, with the penalty of an increased number of variables kept to a minimum.

2.3 JDK Summaries

Virtually every Java application uses JDK (Java Development Kit) library methods. Even a small application can pull in a large number of JDK methods. To limit the number of methods to be analyzed we have created close to 10,000 JDK method summaries, represented in XML. The method summaries are created automatically in template form, which includes signature, checked exceptions thrown, and possibly callback functions. These automatically created summaries are then completed manually, based on the API documentation provided by Oracle (http://docs.oracle.com/javase/7/docs/api/), with the following information, if applicable:

- Taint: information on how taint is transmitted from or among arguments to the object itself and to the return value; this information is provided exhaustively for all summarized methods.
- Exceptions: a list of all exceptions that may be thrown by the method, including runtime exceptions. Individual exceptions may be augmented with a safety condition, providing a precondition that rules out that the exception is thrown; exceptions are provided exhaustively (as far as correctly specified by the API documentation); safety conditions have been added where the condition can be expressed in terms of the arguments.
- Post-conditions: constraints on the return value, possibly in relationship to the arguments. Examples include a specification that the return value is never null, or that the return value is always nonnegative, or within a certain range, or bounded by one of the arguments.
- Error-post-conditions: a constraint on the return value that indicates an error-value. The full post-condition is the disjunction of the conjunction of all post-conditions and the conjunction of all error-post-conditions.
- Side effects: addition of elements and union of collections is recorded for all collection classes and methods that manipulate arrays to enable the tracking of aggregate numerical information on collections and arrays.
- String sinks: information on how strings passed as arguments are used by the function. Examples include the specification of class names for dynamic loading, or method names used in reflection, which are directly used in the analysis. Other examples are names that are used in environment requests such as system properties or file names. This latter information is currently used for information only to get more insight in the behavior of the system.
- Dynamic dispatch: calls made to other public methods that could possibly be sub-classed or methods on objects passed as arguments.
- Resources: arguments to methods can be identified as denoting a particular kind of resource, such as memory to be allocated, time to be delayed, number of threads to be created.
-

JDK classes covered	1147
JDK methods summarized	9739
JDK fields (with fixed value)	1602 (897)
Post-conditions specified	2409
Error-post-conditions specified	480
Side-effects specified	251
String sinks identified	360
Resource sinks identified	103
Exception safety conditions specified	1139
Dynamically dispatched methods specified	2680

Table 1 Information on JDK summaries provided with the CodeHawk Java Analyzer

Table 1 provides some information on the number of summaries provided with the CodeHawk Java Analyzer and the elements included.

Application methods analyzed	JDK methods analyzed	Total methods analyzed	Summaries used	Analysis time (sec)
9102	1604	10,706	2092	358
8900	32,130	41,030	0	3508

Table 2 jfreechart: methods analyzed with and without JDK summaries presentThe scalability of the analysis heavily relies on the presence of JDK summaries.

As an example, Table 2 shows the effect of the presence of JDK summaries on the number of methods analyzed and the analysis time for the application jfreechart (www.jfree.org/jfreechart).

2.3.1 Challenges

The JDK summaries are vital to the ability to scale to large programs. They also introduce a number of problems that to date have only been partially solved. Below we summarize some of these problems.

- *Incorrect or incomplete API documentation*: Summaries are constructed based on the published API documentation. This approach is attractive, because the resulting summaries are independent of the particular JDK used, as all JDK's are supposed to be compliant with the published API documentation. This documentation, however, is not always complete. Most commonly the specification of runtime exceptions that can be thrown is missing or inconsistent. To mitigate this problem we have in some cases inspected the source for OpenJDK to obtain this information, with the disadvantage that the summary may now be JDK-dependent.
- *Callbacks and Dynamic Dispatch*: The use of summaries hides the presence of callbacks to application methods, which may as a result not be included in the set of methods

analyzed. To mitigate this problem we automatically extract from the JDK virtual and interface methods being called to a certain depth and these to the summary. However, this is not guaranteed to be complete. In addition we also add to the methods analyzed all application methods that implement JDK interfaces.

2.4 Basic Analyses

2.4.1 Numeric Analysis

The intra-procedural numeric analysis is done via abstract interpretation over a domain that combines:

- an interval domain in which the possible values of each variable are approximated by closed intervals,
- a linear equality domain that captures linear relationships between variables but keeps aside the constant numeric variables for performance improvement,
- other numeric information such as excluded values.

The choice of using a complex domain rather than the product of an interval domain and a linear equality domain was made because of scalability concerns. A tightly integrated domain leads to a more efficient implementation and allows for different levels of precision. For instance, a method can be analyzed using the full precision of the domain but, if a limit on resources is reached, the relational constraints can be dropped and the analysis can continue with just the range information. Currently we are using a timeout of 20 seconds for the full analysis, followed by a transition to an interval-only analysis (which happens for about 0.1% of the methods).

For Phase 2, we used a more powerful domain. We found a polyhedral domain to be the most suitable for the analysis of loops and overflow/underflow. The overflow/underflow analysis requires tracking ranges for all numeric variables. For each variable, there is always some range information, if nothing else, at least the range derived from the type. This makes the polyhedral analysis very expensive because:

- Large methods can have hundreds of numeric variables and the dimension of the polyhedra equals the number of numeric variables.
- There are at least two constraints per numeric variable.
- The polyhedra used in the analysis are the intersection of a hypercube with the polyhedron defined by the non-range constraints.

This is potentially the worst-case scenario as some of the essential polyhedra algorithms are exponential in the dimension and hypercubes are a known corner case. Our solution was to use a hybrid polyhedron + intervals domain. In this domain, all the constraints of the type $x \geq a$ and $x \leq b$, where a and b are constants, are moved out of the polyhedra into the intervals. The only constraints left in the polyhedra are the few that have at least two variables. By also eliminating the variables that do not participate in a constraint, we reduced the dimension and kept the polyhedra as small as possible. The trade-off was that the implementation for some operations on the domain, especially joins, become more complicated and could still be as expensive as in the original polyhedron domain. Here again, we opted for scalability over precision by using over-approximations. Overall, our hybrid polyhedron-intervals domain:

- is more precise than intervals but less precise than a pure polyhedral domain.

- leads to the analysis of more methods in shorter time with the exception of small, simple methods.
- allows for a two-level precision analysis, in which if the polyhedral analysis takes too long, the polyhedral component is dropped and the analysis continues with just the interval information.

In Phase 3 we abandoned the polyhedra – interval domain in favor of a linear equalities – interval domain.

The analysis gathers information on all the numeric variables, but also aggregates information on arrays and collections of numeric values. As one of the main goals of the analysis is to detect loops that have tainted bounds and could iterate a very large number of times, we introduced an artificial loop counter for each loop. If we can infer that there is a finite interval of values for the loop counter, then the loop is safe. Using abstract interpretation over the interval domain alone will not suffice, as there is no information in the program about the artificially introduced loop counter. The linear equality domain, however, can establish relationships between the loop counter and program variables and the bounds on these variables can then be used to find bounds on the loop counter as well. For instance, consider the case of a loop such as:

```
int i = 10;
while (true) {
  ...
  i = i + 2;
  ...
  if (i > n) break;
  ...
  if (i > 200) break;
}
```

The domain analysis results for `i` and the loop counter `lc` are respectively [10,200] and [1, inf) and the linear equality inferred for the beginning of the loop is 2 * `lc` – `i` + 8 = 0. These are used to get the interval [1, 96] for the loop counter.

The intra-procedural analyses on individual methods are combined to obtain the inter-procedural flow of information. To make this possible, the following information is maintained:

- Intervals for fields. These represent over-approximations of the sets of values of all the fields with the same signature.
- Summaries of methods. Aside from the summaries of the methods analyzed, the analysis has thousands of library function summaries.
- Summaries of calls. For each method, the information on all the calls to that method is joined to give an overall context for each method.

To be able to use this information, the numeric analysis makes several passes of intra-procedural analysis:

- In the first pass, the methods are analyzed in call graph bottom-up order using top intervals for the fields and the parameters. However, the bottom-up order ensures that, most of the time, summaries are available when a call is analyzed. If not, the numeric return values are set to top. In this pass, intervals for fields and calls are also recorded.

- In the second pass, the methods are analyzed in a top-down order. Starting from main, the flow of information through the calls is recorded and is available to be used when the called methods is analyzed. The fields' values from the previous pass are used.

This strategy results in a much-increased level of precision while remaining scalable.

2.4.2 Taint Analysis

Our first approach to tracking taint was to use abstract interpretation on a finite, specialized domain combined with the same inter-procedural strategy that we use for numeric analysis. However, our SSA internal representation allowed us to go a different route altogether. Our current approach is to create variable dependency graphs and to analyze the flow of information on these graphs.

The nodes are variables, static fields, and calls. The edges connect operands to the result, arguments to the call, the call to the return value, and so on. These edges alone are not sufficient as the following example illustrates:

```
public static main (int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    sum = sum++;
    mk_array(sum);
  }
}

public static mk_array (int len) {
  int[] a = new int[len];
  ...
}
```

The variable `n` is tainted but this taint does not transfer to any other variables. In particular, variable `sum` is not tainted but it becomes as large as n and can be used in the same type of attacks as n. Our solution is to taint the loop if the exit condition is tainted. More precisely, we add edges from the variables in an exit condition of a loop to the loop counter and from that loop counter to the variables that are assigned a value in the loop. In this case, the taint on `n` transfers to the taint on the loop counter (signaling the number of times the loop iterates could be under the control of the user), and from there it transfers to `i` and `sum`. If the loop counter was found by the numeric analysis to have a finite range, then these edges are not added.

Rather than working with one graph of the entire system, we create many smaller graphs that give a view of a part of the system, which we transform and analyze. The fact that the graphs share the nodes guarantees if taint is set on a variable in one graph, the taint of that variable is visible in all the other graphs that contain it.

We first create:

- A graph for each method, to track transmission of taint intra-procedurally.
- A context graph for each method, to track transmission of taint from the arguments of all the calls to the parameters of the method.

- Graphs for the library methods that are invoked by the application and for which we have summaries.

To further reduce the complexity of the graphs, edges are added only if necessary and they are pruned if not needed. For instance, once a static variable has been determined to be tainted, edges to it are not added any more.

The methods in the call graph are analyzed bottom-up, and each methods' taint graph is reduced to a taint-transmission summary between arguments, return variables and static fields. These summaries along with the JDK summaries are then used to reduce the graphs of the calling functions. Once all the methods are analyzed, the summaries are integrated into a graph for the entire application. This graph for the entire application is used to propagate taint from the arguments of main and other methods, such as methods called by dynamic dispatch, and to determine the taint of all the static fields.

In a top-down pass through the call graph, the context graphs are used and taint is propagated from the calling variables and static fields to the internal variables and the taint for the arguments of each invocation is recorded. The analysis of each graph is linear in the number of edges but the graphs, especially the graph skeleton graph for the entire application can theoretically have a node for each static field and for each method argument. In practice, our approach succeeds in keeping the graphs small enough.

2.5 Property-driven Analysis

2.5.1 Loop-bounds Analysis

The loop-bounds analysis (CWE 606) relies on a combination of numerical analysis and taint analysis. The numerical analysis attempts to establish an upper bound on the number of iterations of the loop; the taint analysis attempts to determine whether the number of iterations is under the influence of the user. If the number of iterations is found to be tainted, the loop is reported to the runtime system, so its number of iterations can be monitored at runtime and possibly cut short when it exceeds a certain, configured number of iterations. The information provided to the runtime system includes the program counter to jump to, to resume execution after terminating the loop, and information about variables that may have to be initialized.

Section 2.7.1.2 presents the results for loop bounds analysis for the Juliet Test Suite (CWE 606). Sections 2.7.2, 2.7.3, and 2.7.4 present the results for larger applications, including the Phase 2 and Phase 3 T&E applications. The results on the Juliet Test Suite show near 100% precision. For larger applications, however, the percentage of loops that can be proven to be untainted is only 10-25%, and the percentage of loops that can be proven to be bounded is 40-50%. It is difficult to determine the precision in this case, as, unlike with the Juliet Test Suite, we do not have ground truth for these applications and thus cannot easily determine the number of false positives.

2.5.2 Number-handling Analysis

The number handling analysis relies on both the numerical analysis and taint analysis, as well as on the numerical post-conditions and side effects provided by the JDK summaries. Ranges of all numerical variables present on the expression stack are reported for every instruction in the analysis results, as well as the possibility of overflow, underflow, truncation, and division by zero for all relevant arithmetic instructions.

Section 2.7.1 reports the results of the numerical analysis for several of Juliet Test Suites for the number-handling CWE's. Sections 2.7.2, 2.7.3, and 2.7.4 report the results for the number handling analyses for larger applications, including the Phase 2 and Phase 3 T&E applications. Again, the CodeHawk analysis for the Juliet Test Suite has a very high precision. The ability to prove the safety of numerical operations in larger program is much lower, 30-65% for integer overflow (CWE 190), 60-95% for integer underflow (CWE 191), 0-50% for truncation (CWE 197), and 50-100% for division by zero (CWE 369). Again, we do not know how many of the unproven operations are actually safe (false positives), as we do not have ground truth for these applications.

2.5.3 Error-handling Analysis

The CodeHawk analysis provides four types of analysis results to the runtime system related to error handling. Two of these are syntactic: the presence of the return instructions in finally blocks, and the extent of catch blocks, and two of them are semantic: null returns from method calls that may be dereferenced without checking, and error-values returned by method calls that are not checked correctly.

The null analysis relies on symbolic analysis (propagation of reference values that may be null) and on the JDK summaries, in particular the post conditions that specify whether the return value of a JDK method may be null.

The error-return value analysis relies on the SSA analysis to track the flow of error-return values and determine how and where they are used and checked. The JDK summaries are used, in particular the error-post-conditions, to determine what return values from JDK methods are considered error values and should be checked for, before being used, or before being discarded.

Sections 2.7.3 and 2.7.4 present the results for the error-return value analysis. The analysis partitions the handling of return values into five classes:

1. Properly checked: there is no need for the runtime system to instrument these values
2. Returned: the error-return value is returned from the application method without checking; we have determined that the vast majority of these cases involve wrapper methods, and thus these values need not be instrumented either
3. Passed as an argument: the value is passed as an argument to another method. This is considered an error and the runtime system is informed to throw an exception when the error-value occurs in an execution;

4. Popped: the value that is returned is immediately popped from the expression stack without any check. This is considered an error and the runtime system is informed to throw an exception when the error-value occurs in an execution.
5. Unknown: the returned value is used in a different way than any of the above. At present we still consider this an error, and an exception is thrown when the error value occurs during an execution.

2.5.4 Resource Analysis

The resource analysis makes use of JDK summaries to identify arguments that denote resources. The types of resources currently supported are: memory, number of threads, time delay, and file size. This information is propagated and included in the analysis results, but not currently used by the runtime system.

2.5.5 Concurrency Analysis

The concurrency analysis is based on a call graph search for methods that write a field and could be executed concurrently with methods that read that field and then either update it or use its value in a conditional. The analysis does not distinguish between same-signature fields of different objects. To reduce the number of false positives caused by this over-approximation, a graph analysis, similar to the taint analysis described in Section 1.2, is used to find the variables that are shared by multiple threads. Fields of such variables are not reported. As the call graph search can be very costly, the depth of the search is currently limited so that only about 100 methods are flagged.

2.6 Interface with Dynamic Analysis

CodeHawk assists the dynamic analysis by:

- Determining whether the application is a server and finding the blocks that could fail in worker threads and need to be guarded to ensure continuous execution
- Detecting the loops that could potentially be executed a very large number of times under user control and providing information used to instrument for confinement.
- Providing information on the presence of return instructions in finally blocks
- Providing information on the extent of catch blocks
- Providing information on potential null dereference
- Providing information on error-return values that are not checked correctly by the application
- Providing information on static and object fields that may be accessed by multiple threads, thus having the potential to be involved in data races.

In Phase 1 this information was provided in the form of text files with a fixed format, specifying class, method, and the piece of information of interest. In Phase 2 these text files were complemented with an xml file that contained data at the level of individual instructions on numeric data types.

In Phase 3 it became clear that this way of communication was rather restrictive in terms of the information that could be conveyed, it was inflexible, and it didn't scale that well. Therefore, we adopted a new means of communication that provides all analysis results in xml, organized by class file and saved according to package name, similar to how class files are stored. On the instrumentation side we provide a Java API to extract the results from the xml files and make them available programmatically to the instrumentation code. This approach has a number of advantages:

- It scales to arbitrarily large systems, as analysis results are saved on a class by class basis
- It allows pre-analysis of classes provided in library jars
- It provides more bandwidth between static analysis and runtime analysis: in principle all analysis results are always available to the runtime system.
- It allows a two-tier supply of analysis results: the first tier results are the results obtained from context-sensitive whole-program analysis started from one or more main methods; the second tier results are the results obtained from a context-insensitive analysis of all application classes and methods provided on the class path. For all methods that were missed by the context-sensitive analysis due to dynamic loading, reflection, or other means of introducing methods and classes into the application, the runtime system always has access to the context-insensitive results.

2.7 Evaluation

Throughout the project we have evaluated the performance of the analyzer on a wide variety of test cases, including both benchmark programs and real-world (large) applications. In this section we describe the results of those evaluations.

2.7.1 Juliet Test Suite

The Juliet Test Suite is designed to measure the performance of static analysis tools, in terms of both false positives (indicating a weakness where none exists), and false negatives (missing a weakness). The terminology of false positives and false negatives is appropriate for bug finders, which, typically, have large numbers in each category, but does not apply in the same way to sound static analysis.

The approach in sound static analysis is to identify all potentially vulnerable (syntactic) constructs in a certain (well-defined) class of weaknesses and try to prove them safe. A construct is considered "guilty" until proven "innocent." That is, every potentially unsafe construct is considered unsafe unless we can construct a proof of its safety. A false negative in this context means an erroneous proof (or possibly erroneous assumptions) that can be identified (unlike false negatives from bug-finders, which are just ``misses'' for no particular reason). A false positive in a sound static analysis context is the inability of the analyzer to construct a proof, perhaps because of relationships that cannot be represented or that cannot be discovered in reasonable time and space.

In the discussion of the CodeHawk results on the Juliet Test Suite we focus on proving safety. Our goal is to prove that no harm can be done and the code does not need to be instrumented, thus reducing overhead. We do not try to prove presence of vulnerabilities, although we may

occasionally conclude that a vulnerability is present, based on the nature of the proof obligations (e.g. proof obligations that are inconsistent indicate definite vulnerabilities). These cases are common in the Juliet Test Suite, by the nature of their design and purpose, but are rare in real-world application code.

Metrics. Measuring the precision of static analysis tools requires metrics that define precision. Traditionally precision of bug finders is expressed by false positive rate, that is, the number of tool reports that turn out not to be bugs, divided by the total number of tool reports. The reason for the popularity of this measure is that it gives an indication of how much effort is wasted in triaging tool reports. The disadvantage of this measure of precision is that it represents only half of the story: it does not provide any information on what was actually checked and how many bugs were missed (the false negatives). A tool may achieve a low false positive rate by simply screening out all but the most blatant errors, but would probably have a high number of missed bugs, severely reducing its utility for the ultimate goal of static analysis: providing assurance.

Sound static analysis allows a metric that better reflects the goal of assurance and also better fits the role of static analysis in STONESOUP, which is to reduce runtime overhead. Given a particular property, for example, absence of integer overflow, sound static analysis starts by identifying (syntactically) all locations in a program that have an instruction that could possibly violate this property. For example, an *add* instruction can violate this property, but a *putfield* operation cannot violate this property. The next step is to attempt to create proofs, based on the semantics of the language, that each of these locations is in fact safe with respect to integer overflow. If such a proof can be found, the proof provides the evidence that the given instruction cannot lead to overflow under any possible scenario. If such a proof cannot be found, it may sometimes be possible to demonstrate an actual input sequence that violates the property, but more often we do not know whether the inability to find a proof results from proof methods that are too weak or from the presence of an actual vulnerability.

The metric for precision that we will use in this report is defined as follows. Given a property φ and program P let $I_{\varphi,P}$ be the set of program locations (instructions or constructs) in program P that could possibly violate property φ. Let *S* be the subset of $I_{\varphi,P}$ of locations that are actually safe, that is no possible program execution exists in which the property is violated at that location. Our goal is to provide proofs for all instructions in *S*. Let *C* be the subset of $I_{\varphi,P}$ of locations for which our analysis is able to construct a proof. The analysis is sound (no false negatives) if *C* is a subset of *S*. The analysis is complete (no false positives) if *S* is a subset of *C*. Sound static analysis does not compromise on soundness. The most appropriate metric for precision then is to measure how close we get to completeness, that is, we can define precision as

$$precision = \frac{|C|}{|S|} = \frac{number\ of\ locations\ proven\ safe}{number\ of\ locations\ that\ are\ safe}$$

This measure of precision is adequate for benchmarks like the Juliet Test Suite where we actually know the set of instructions that are safe. This measure is not practical, however, for real-world programs, because the set *S* is not known (if it were known, we did not have to analyze the program). For real-world programs, however, we may assume (or at least hope) that the number of locations that actually can violate the property is small compared to the total

number of locations that can potentially violate the property, that is, most locations are actually safe, expressed by $|I| \approx |S|$, and thus we can approximate our measure of precision by

$$precision = \frac{|C|}{|I|} = \frac{number\ of\ locations\ proven\ safe}{number\ of\ all\ relevant\ locations}$$

Using the above notation, the traditional false positive rate is expressed as

$$false\ positive\ rate = 1 - \frac{|I \backslash S|}{|I \backslash C|} = \frac{|S \backslash C|}{|I \backslash C|} = \frac{number\ of\ safe\ locations\ not\ proven}{number\ of\ locations\ not\ proven}$$

The false positive rate, of course, breaks down under the assumption we make for real-world programs (it always is 100% under the assumption that all instructions are in fact safe). This reflects a more general problem with false positive rate as a measure of precision for sound static analysis. Consider a program with 10,000 locations that could (syntactically) violate a property, of which 9990 are safe. If an analysis is able to prove 9980 of these safe, this analysis should be considered very high precision (99.9%). The false positive rate, however, is 10/20 = 50%, which is generally considered a bad rate. In general, one would expect to see high false positive rates in systems with a low number of bugs.

2.7.1.1 Summary of Results

Table 3 presents a summary of the precision achieved by CodeHawk analysis on the Juliet Test Suites relevant to Phase 1 and Phase 2 properties. Analysis times indicate the time it takes to analyze the entire test suite. The sections below describe the tests and the CodeHawk analysis in more detail for each CWE.

CWE	#tests	construct	#reachable constructs	#safe constructs	#constructs proven safe	precision	Time (sec)
606	444	loop	1716	1272	1248	98.1%	208
191	1702	subtraction multiplication	6578	4876	4738	97.2%	635
190	2553	add multiply square	9867	7314	7107	97.2%	1063
369	1702	division modulo	6578	4876	4646	95.3%	745
197	1221	type conversion	4719	3498	3399	97.2%	196
129	2664	array access	10,512	7848	7632	97.2%	1132
193	51	array access	147	96	64	66.7%	12

Table 3 Juliet Test Suites: CodeHawk precision and timing

2.7.1.2 CWE 606: Unchecked Loop Condition

CWE 606 describes the weakness that *a product does not properly check inputs that are used for loop conditions, potentially leading to a denial of service because of excessive looping* (cwe.mitre.org).

The Juliet Test Suite for CWE 606 consists of twelve groups, shown in Table 4. Each group of tests has a different source for the "bad" data. Each of the twelve groups has 37 control/data-flow variants, shown in Table 5, resulting in 444 test cases.

Name	Source of bad data
Environment	Read data from an environment variable
File	Read data from a file
Property	Read data from a system property
PropertiesFile	Read data from a properties file
URLConnection	Read data from a web server with URL connection
Connect_tcp	Read data using an outbound connection
Console_readLine	Read data from the console using ReadLine
database	Read data from a database
getCookies_Servlet	Read data from the first cookie using getCookies
getParameter_Servlet	Read data from a query string using getParameter
getQueryString_Servlet	Parse parameter from a URL query string (without using getParameter)
Listen_tcp	Read data by listening to a tcp connection

Table 4 CWE 606: Different sources of "bad" data

Each test case consists of a "bad test", that is, a weakness is present, and several "good tests", that is, the program provides sufficient protection against potentially bad data or the sink of the data is such that no damage can be done by the data. The bad test has both a bad source for the data and a bad sink. The good tests may have a bad source, but a good sink (goodB2G), or a good source and a bad sink (goodG2B). The bad sources of data are those listed in Table. The bad sinks are loops that could execute as many as MAX_VALUE times; the good sinks are loops that execute five times. The bad and goodB2G tests both have data from a bad source that control a loop; in the goodB2G test, however, the loop has an additional check that ensures that the loop is executed only if the loop bound is between 0 and 5. In the goodG2B tests, the source for the loop upper bound is the string "5", but there are no checks.

1	`baseline`
2	`If (true), if (false)`
3	`If (5==5), if (5!=5)`
4	`If (PRIVATE_STATIC_FINAL_TRUE), if (PRIVATE STATIC FINAL FALSE)`
5	`If (privateTrue), if (privateFalse)`
6	`If(PRIVATE_STATIC_FINAL_FIVE==5),if (PRIVATE STATIC FINAL FIVE!=5)`
7	`If (privateFive==5), if (privateFive!=5)`
8	`If (privateReturnsTrue()), if (privateReturnsFalse())`
9	`If (IO.STATIC_FINAL_TRUE), if (IO.STATIC_FINAL_FALSE)`
10	`If (IO.STATIC_TRUE), if (IO.STATIC_FALSE)`
11	`If (IO.staticReturnsTrue()), if (IO.staticReturnsFalse())`
12	`If (IO.staticReturnsTrueOrFalse())`
13	`If (IO.STATIC_FINAL_FIVE==5), if (IO.STATIC_FINAL_FIVE!=5)`
14	`If (IO.staticFive==5), if (IO.staticFive!=5)`
15	`Switch(6), switch(7)`
16	`While (true)`
17	`For (int j=0; j<1; j++)`
21	Boolean private variable if (badPrivate), if (goodG2BPrivate), if (goodB2G)
22	Boolean public static variable if (badPublicStatic)
31	Copy of data
41	Data passes as argument to another method in the same class
42	Data returned from one method to another in the same class
45	Data passed as private field from one method to another in the same class
51	Data passed as an argument from one method to another (same package)
52	Data passed as an argument from one method to another in 3 different classes (same package)
53	Data passed as an argument from one method to another in 4 different classes (same package)
54	Data passed as an argument from one method to another in 5 different classes (same package)
61	Data returned from one method to another in different classes (same package)
66	Data passed in an array from one method to another in a different class (same package)
67	Data passed in a class from one method to another in a different class (same package)
68	Data passed as a public stat field (same for bad, goodB2G, and goodG2B)
71	Data passed as an object argument
72	Data passed in a Vector<String>
73	Data passed in a LinkedList<String>
74	Data passed in HashMap<String>
75	Data passed in a serialized object
81	Data passed in parameter to an abstract method

Table 5 CWE 606: Different control variants

2.7.1.2.1 CodeHawk Analysis

The CodeHawk analysis of loops consists of two main steps: a numeric analysis and a taint analysis. The numeric analysis uses the CodeHawk abstract-interpretation engine to generate invariants that over-approximate the sets of possible values of each variable at each location in the program. The taint analysis uses graphs to capture the flow of information during the execution of an application and determines which variables could be influenced by the user of the application. Variables that are potentially influenced by the user are called *tainted variables*.

As mentioned before, our primary goal is to prove the good tests (goodB2G and goodG2B) safe. Numeric and taint analysis contribute to this goal in two different ways. Numeric analysis is able to prove that all loops in the goodB2G tests have small upper bounds. The taint analysis is able to prove that the number of loop iterations, given by **numberOfLoops** in

```
String data = "5" ;
int numberOfLoops = Integer.parseInt(data);
```

is in fact a program constant (determined by the programmer). The analysis cannot determine a numerical value from a string, but it can determine that the value is *untainted*, that is, the value cannot be influenced by the user, and therefore the value can be considered safe.

Below we elaborate on the numeric and taint analysis performed to prove safe the good tests.

Numeric Analysis. The goal of the numeric analysis is to identify a variable whose value is related to the number of times a loop is executed, and to prove that the possible values of this variable are in a safe range.

All of the goodB2G tests include the following construct as the “good sink”:

```
if (numberOfLoops >= 0 && numberOfLoops <= 5)
{
  for (int i=0 ; i < numberOfLoops ; i++)
  {
     IO.writeLine("hello world");
  }
}
```

Before the if statement the variable numberOfLoops originating from a bad source may be totally unconstrained. Within the “then” block, however, the variable numberOfLoops clearly lies within the interval [0, 5]. Abstract interpretation using the interval domain suffices to determine this interval. For a simple loop like the one shown above, the variable i can be used as the counter for the number of times the loop is executed. In the general case, however, it is hard to determine which program variable, if any, can be used in this role. To deal with loops in a general, uniform way, we add a loop counter for each loop that is initialized before the loop and incremented for each iteration. Abstract interpretation using an interval domain produces the interval [1,∞), for the range of the loop counter: after a few symbolic executions of the loop, intervals are widened to infinity. The interval obtained for the variable i is initially similarly widened to [0,∞), but then is narrowed to [0,4] by the assertion i < numberOfLoops. It remains to

relate the loop counter to the variable i to obtain an upper bound on the loop counter. A more powerful domain than the (non-relational) intervals is required to establish this relationship: abstract interpretation using the domain of linear equalities could be used to infer

```
loop_counter = i + 1
```

Taint Analysis: Sources and Flows. Taint generally relates to the user's ability to affect the computation. Specifically in the context of the CWE 606 weakness, it relates to the user's ability to cause a system to consume large amounts of computing resources by increasing the number of loop iterations to a number much larger than reasonably intended by the program creators. Taint, and the flow of taint, however, is not a well-defined notion. For example, if the variable t is controlled by the user in the statement

```
x = (t>0) ? 3 : 5
```

does that make the variable x tainted? An argument can be made for both cases. The more aggressive approach would say yes, because the user has an influence on the value of x. On the other hand, all of the possible values of x were explicitly put in the program by the programmer and thus should be considered acceptable values. More generally, if a tainted value controls a branch in the program, should we taint every variable assigned on either branch? The ultimate consideration, of course, is the utility of the analysis results. The more aggressive approach of tainting all variables that can be directly or indirectly influenced by the user by either data flow or control flow is likely to result in almost all variables being tainted. For this reason we decided to not take into account influence via control flow, with one notable exception related to loops. When the number of iterations of a loop is influenced by the user, variables that get assigned within the loop may grow in relation to the loop counter, even if there is no direct information flow (via assignment) between those variables and the tainted variable that controls the number of iterations. To capture this indirect influence on the value of variable our analysis considers all variables that are assigned a value within a tainted loop to be tainted.

Another question in taint analysis is: what constitutes a source of taint and how do we identify these sources? We decided to take the more aggressive approach in this case. We consider tainted:

1. the arguments of main;
2. the arguments of listener methods, and, in general, all methods that are dynamically dispatched and for which we do not know the source of their arguments;
3. public and protected methods of application libraries that could be called by an application using the library;
4. a large collection of JDK methods, which we have explicitly identified as taint sources in our library of JDK method summaries;
5. a small number of static fields in the JDK.

The sources of taint for different groups of tests in the CWE 606 Juliet Test Suite as identified by our analysis are shown in Table 6.

Name of test case	Source of taint
Environment	Return value of java.lang.System.getenv(…)
File	Return value of java.io.FileInputStream(…)
Property	Return value of java.lang.Syste.getProperty(…)
PropertiesFile	Return value of java.io.FileInputStream(…)
URLConnection	Return value of URLConnection.getInputStream(…)
Connect_tcp	Return value of java.net.Socket.getInputStream()
Console_readLine	Field java.lang.System.in
database	Return value of java.sql.DriverManager.getConnection()
getCookies_Servlet	Argument to javax.servlet.http.HttpServlet.doPost(…)
getParameter_Servlet	Argument to javax.servlet.http.HttpServlet.doPost(…)
getQueryString_Servlet	Argument to javax.servlet.http.HttpServlet.doPost(…)
Listen_tcp	Return value of java.net.Socket.getInputStream()

Table 6 Source of taint identified for the test cases in Juliet Test Suite CWE 606

Taint Analysis: Implementation. Our taint analysis uses graphs to represent the flow of taint. Nodes in the graph are variables, static fields, method calls and constant taint values. Edges in the graph represent possible flow of taint. For example, the instruction

```
z = x + y
```

adds edges from `x` to `z` and from `y` to `z` to the graph.

Taint analysis is performed in two passes. The first phase is a bottom-up pass through the call graph that creates local taint transfer relations between arguments, static fields, and return values. The second phase is a top-down pass through the call graph in which taint is passed down from the arguments to main and to other root-level methods to the methods lower in the call graph.

The bottom-up pass starts with a detailed intra-procedural taint graph for each individual method. These graphs are reduced to summary graphs that only represent the transfer of taint between arguments, return values, and static fields. An inter-procedural taint graph is then created starting from methods that do not call any other methods or only call JDK methods for which we have method summaries. We eliminate the calls to the summarized JDK methods and reduce the graph to the signature variables and static fields. These summary graphs are in turn used to produce summary graphs for other methods higher up in the call graph. Cycles in the call graph, a common occurrence in Java programs, are resolved conservatively: method arguments and return values of methods that are part of a cycle are set to top (unknown value). At the completion of the bottom-up traversal of the call graph the taint on all static fields is known, as well as how taint is transmitted through methods.

The top-down pass starts with assigning taint to the arguments of main and other methods that may be dynamically dispatched. This taint and the taint from static fields is transmitted to all variables of all methods and all object fields in a top-down traversal of the call graph.

2.7.1.2.2 Results and Discussion

Recall that the test suite consists of 12 groups of 37 tests: 12 different types of input sources for bad data and 37 control/data flow variants for each input source. Once the input source has been identified, the analysis is independent of the input source, so we focus on the control/data flow variants only. The goal is to correctly prove safe all loops that iterate only a small number of times, thereby eliminating the need for the runtime system to check these loops, and to report all other loops to the runtime system for checking (if this were an application).

Control-flow variants 1 through 22 from Table 5 are straightforward for our analysis: the baseline case is easily handled and the control-flow variants resulting in dead code are easily recognized, thus eliminating any spurious tainted loops. We correctly prove all "good" test cases safe: no false positives, no false negatives.

Data-flow variants 31 through 81 from Table 5 pose more of a challenge, but all of them except for two cases are handled precisely (no false positives, no false negatives) by our analysis. The exceptions are 45 (data passed in private field from one method to another) and 68 (data passed in a static public field), which result in false positives, that is, we are not able to prove that the "good" test cases are safe, which gives rise to 24 false positives (no false negatives).

Test 45 passes bad data via a private field: the private field `dataBad` is assigned a tainted value. In our analysis this assignment results in setting the "this" object to tainted. All tests within a class (bad, goodG2B, goodB2G) are called on the same object and therefore they share "this". The method goodG2BSink() obtains its data from a different private field, `dataGoodG2B`, but we aggregate taint from all fields, and thus the taint from `dataBad` (although not read by GoodG2BSink) is still transmitted to the variable `numberOfLoops` resulting in 12 false positives (one for each bad source).

Test 68 passes data from one method to another via static public field `data`. The static field is assigned a tainted value in the method bad(), and subsequently an untainted value before the method goodG2BSink() is invoked. Our graph representation of flow of taint does not capture order of execution of methods. Therefore, once a static field is tainted it will always be tainted; it cannot become untainted. The taint again is transmitted by goodG2BSink() to `numberOfLoops` and thus we fail to prove that the loop is in fact safe, resulting in 12 false positives.

The 444 CWE606 tests have a total of 1716 reachable loops, 444 of which are not safe (that is, they may have as many as `Integer.MAX_VALUE` iterations); the remaining 1272 loops are safe and would not have to be instrumented if this were an application. We are able to prove 1248 out of the 1272 loops safe (98.1%), and we would report 468 loops to the runtime system to be instrumented, with 24 false positives (4.2%).

Remark The 444 CWE606 tests also contain 93 loops that are unreachable. Our analysis correctly identifies these loops as unreachable (dead code) and does not report these loops to the runtime system. The instrumentation of dead code, however, is mostly irrelevant to the performance of the runtime system (it may add some to the code footprint, but would not add

significantly to the running time), and therefore we generally omit dead code from the total numbers of operations in reporting percentages of operations safe.

2.7.1.3 CWE 191: Integer Underflow

CWE 191 describes the weakness that *a product subtracts one value from another such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result* (cwe.mitre.org).

In Java, unlike in C and C++, integer overflow and underflow are well defined in the language semantics (wrap-around) and do not cause an exception to be thrown. In some cases, integer overflow or underflow may have been intended by the programmer (for example, in hash functions). In most cases, however, integer overflow or underflow indicates an error. In the CWE191 Juliet Test Suite all underflow are considered an error.

The CWE 191 Test Suite is organized along four axes:

1. 37 control/data flow variants, the same as shown in Table 5
2. 4 data type: byte, int, long, or short
3. 14 input sources, as shown in Table 4, with two additional sources:
    a. min: Set data to the `MIN_VALUE` of the respective data type;
    b. rand: Set data to a random value
4. 2 types of operations: subtract and multiply

The total number of tests is 1702 (not all combinations are represented). As for CWE 606 the control/data flow variants are the most interesting axis from an analysis point of view, and we will focus on the different challenges posed by some of them.

2.7.1.3.1 CodeHawk Analysis

The goal in these test cases is again to prove the “good” cases safe. This was achieved purely by numerical analysis. For the goodB2G (bad source, good sink) test cases both operations (subtraction and multiplication) were proven safe for all test cases. For the goodG2B (good source, bad sink) test cases, all but three data flow variants were proven safe. Below we describe the tests in some more detail and how we approach the different variants.

The goodB2G test cases all have a check that guards the potentially unsafe operation and ensures the operation is executed only if it is safe. In the subtraction tests the check in the goodB2G tests is (for integers; they are similar for the other data types):

```
if (data > Integer.MIN_VALUE) {
  int result = (int) (data - 1) ;
  .....
}
```

In the multiplication tests the check in the goodB2G tests is

```
if (data < 0 ) {
  if (data > (Integer.MIN_VALUE / 2)) {
    int result = (int) (data * 2) ;
    .....
  }
}
```

Both of these tests are easily handled by analysis by abstract interpretation with a (computationally cheap) non-relational domain of intervals.

The goodG2B test cases all have a source of 2, but do not perform any checks before the operation itself. The challenge is thus to correctly propagate this value to the unguarded sink.

The different control flow variants (1 through 22) are easily handled by our analysis. In some tests (5-15, 21, 22), data gets different values on different paths; all values, however, are harmless (either 0 or 2) so the join of the different paths is still safe. Some tests have dead code, which is easily recognized by our analysis. In variants 16 and 17 the target operations are inside loops, which are executed only once. In variant 31, data is copied before it is used. All of these tests are easily handled by our analysis.

In the data flow variants the challenge is to correctly capture the transmission of data from method to method through arguments. One possible way to approach this problem is to perform full in-lining, that is, to do a full expansion of the called method (the callee) in the calling method (the caller). This is certainly a feasible solution for programs of the size of the Juliet tests. This approach, however, does not scale up to even medium-sized programs. For this reason, we decided to adopt a more general approach that scales up to much larger programs. The analysis is performed in two phases. The first phase is a bottom-up pass through the call graph, in which we analyze each method and summarize the transfer relation between arguments and return values. These summaries are used in the analysis of methods higher-up in the call graph that call these methods. The second phase is a top-down pass through the call graph in which we record how each method is called, thereby establishing a maximal context for the arguments passed to the function. Loops in the call graph are resolved in a conservative manner: if a summary of a called method is not yet available we make no assumptions about its return value; the context of methods in a call-graph loop is set to all possible values. These Juliet tests do not have cycles in the call graph, so this scenario does not arise in these tests. This approach accurately captures the transmission of data on any straight path through the call graph in tests 41, 42, 51-55, and 81.

Constructing the call graph itself sometimes is the challenge, as we deal with virtual and interface method calls. Consider, for example, tests 81, which have an abstract class 81_base with a method action(String). This base class has three subclasses: 81_bad, 81_goodG2B, and 81_goodB2G, all of which implement action(String). The methods 81_bad.action and 81_goodG2B.action have a bad sink, while 81_goodB2G includes a check. In all three tests, bad(), goodG2B(), and goodB2G(), an 81_base object is created (from the byte code perspective) and the method action(String) is invoked on this object. The objects are then initialized with the respective implementing class (81_bad in test bad and so on), so the invocation of action(String)

has different targets in each case. During the loading of classes CodeHawk performs an intra-procedural data type propagation in the construction of the call graph. This analysis is sufficient to correctly identify the unique target for each case, and therefore allows accurate discrimination between the good and the bad tests.

In some tests numeric values are encapsulated in objects before being passed as an arguments. The challenge is to keep sufficient information on these objects to allow some inference about the values when the numeric values are eventually retrieved from the object. In addition to keeping numeric relationships between primitive numeric variables, our analysis also maintains numeric relationships between wrapped numeric variables, and aggregate range information on numeric arrays and collections of wrapped numbers. In the test cases where good data is not mixed with bad data, tests 66 and 71-74, our approach is able to maintain sufficient information to correctly distinguish between the good and bad case and prove the good tests safe. In tests 75, however, data is serialized before transmission and de-serialized afterwards, which our analysis is not able to capture, resulting in false positives for this group of tests.

In some tests numeric values are transmitted through fields, for example, in tests 45, 67, and 68. In our analysis fields are treated as global variables. Our analysis is field-aware, but not field-sensitive, that is, the analysis keeps range information on each distinct field, but as an aggregate for all objects of a particular object type. Furthermore, the analysis is insensitive to order of execution of methods: all assignments to a particular field are aggregated. Field-aware analysis is adequate when a field either gets good data or bad data, as in tests 45. In these tests, the classes have three fields: `dataBad, dataGoodG2B`, and `dataGoodB2G`. In the first pass of our analysis, the interval of the field `dataGoodG2B` is inferred to be [0,2] (the field is initialized with zero, so zero will always be part of the range), and this interval is used in the second pass of the analysis when it is passed to the bad sink, but the inferred range allows us to prove this case safe. Field-aware analysis is not good enough for tests 67. In tests 67, bad(), goodG2B(), and goodB2G() use the same field. Although the fields in each of the methods belongs to different objects and hence, different “global variables”, our analysis cannot make this distinction and therefore joins the ranges of the good and bad sources, resulting in false positives for goodG2B(). In tests 68, the methods bad(), goodG2B(), and goodB2G() store the data in the same public static field, which, as the field is static, is the same global variable. Our analysis is not sensitive to the order of execution, and thus aggregates the good and bad data, resulting in false positives for goodG2B.

2.7.1.3.2 Results and Discussion

The CWE 191 test suite has 1702 tests with 6578 reachable subtractions and multiplications. Of these operations 1702 can result in underflow and 4876 are safe. Of the safe operations we are able to prove 4738 safe, thus we have 138 false positives.

Many of the tests for CWE191 evaluate how accurately the analysis captures the transmission of data. Our solution to inter-procedural transmission of data deals well with the tested situations and is linear in the number of methods. This ensures that when analyzing large applications, time is not an issue. Our analysis covers not only numeric variables, but also many other types that can be thought of as holding a numeric value (such as objects of type java.lang.Integer) or a set

of values (such as arrays). Increasing the number of variables has an impact on the performance, but we have many strategies in place that allow us to analyze even methods that are very large, have a very large number of variables, or have a very complex control flow. For instance, based on time-outs, we switch to simpler domains or take a more conservative approximation.

2.7.1.4 CWE 190: Integer Overflow

CWE 190 describes the weakness that *software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value* (cwe.mitre.org).

The CWE 190 test suite is, like the one for CWE 191, organized along four axes:

1. 37 control/data flow variants, the same as shown in Table 5
2. 4 data types: byte, int, long, or short;
3. 14 input sources, as shown in Table 4, with two additional sources:
    a. max: Set data to the `MAX_VALUE` of the respective data type
    b. rand: Set data to a random value
4. 3 types of operations: add, multiply, and square

The total number of tests is 2553 (not all combinations are represented). As for CWE 191 the control/data flow variants are the most interesting axis from an analysis point of view, and we will focus on the different challenges posed by some of them.

2.7.1.4.1 CodeHawk Analysis

The tests for CWE 190 for addition and multiplication are, from an analysis perspective, identical to those for CWE 191, and the same discussion of results applies. The new operation in the CWE 190 test suite is the square operation, which poses new challenges to the analysis because it is non-linear. All our relational numerical domains are linear (linear equalities, and linear inequalities). To be able to prove safety of the square goodG2B test we designed a new analysis to be able to infer sufficient information from the guard to prove absence of overflow.

The guard for the square operation (shown for the integer case) is

```
if ((data != Integer.MIN_VALUE) && (data != Long.MIN_VALUE) &&
      (Math.abs(data) <= (long) Math.sqrt(Integer.MAX_VALUE)))
   {
      int result = (int) (data * data) ;
      .....
   }
```

The checks for dis-equality with `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are included to ensure that `Math.abs` returns the expected absolute value. Because of the asymmetric nature of two's complement representation the absolute value of a 32-bit representation of `Integer.MIN_VALUE` is equal to `Integer.MIN_VALUE`, because its positive counterpart overflow, and similarly for 64-bit representations and `Long.MIN_VALUE`.

Translation of this code fragment into the CodeHawk internal form (CHIF) produces the following code (shown as pseudo code):

```
 1:  if (data != Integer.MIN_VALUE) {
 2:    if (data != Long.MIN_VALUE) {
 3:      int abs_int = Math.abs(data) ;
 4:      long abs_long = (long) abs_int ;
 5:      double max_sqrt_double = Math.sqrt(Integer.MAX_VALUE) ;
 6:      long max_sqrt_long = (long) max_sqrt_double ;
 7:      if (abs_long <= max_sqrt_long) {
 8:        int result = data * data ;
 9:        ......
10:      }
11:    }
12:  }
```

The goal is to infer from the checks in the guard that the multiplication on line 8 does not overflow. Clearly a non-relational domain like the intervals is not sufficiently expressive to establish the required invariant. We need to be able to express relationships between variables, both equalities and inequalities. Furthermore, we need to be able to go back and forth between integers and floating-point numbers.

Before the `if` block is entered the variable `data` is constrained only by the interval associated with its data type,

```
1: data ∈ [-2147483648, 2147483647]
```

After the conditional on line 1 the constraint becomes

```
2: data ∈ [-2147483647, 2147483647]
```

The conditional on line 2 does not constrain the interval, and so we keep

```
3: data ∈ [-2147483647, 2147483647]
```

Line 3 applies the library function `Math.abs(int)` to the variable `data`. We have constructed a large number of method summaries for library methods. These summaries provide an over-approximation of the transition relation of the method, as well as side effects, if any. The transition relation provided by the summary of `Math.abs` is

$$return \geq 0 \wedge return \geq arg_1 \wedge -return \leq arg_1$$

Remark This transition relation for `Math.abs` may seem somewhat unusual, as one would expect a relation of the form $return \geq 0 \wedge (return = arg_1 \vee return = -arg_1)$, but in many situations disjunctions are not useful, because most of our domains are conjunctive, and thus disjunctions will be joined as soon as the two branches join, which for the post-condition of a library function would happen almost immediately.

Applying the transition relation for `Math.abs` to the actual argument and return variable produces

```
4: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
```

The up-cast in line 4 results in an equality

```
5: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   abs_long ∈ [0, 2147483647]
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
   abs_int – abs_long = 0
```

Line 5 invokes the library method `Math.sqrt(double)`, for which the only reasonable general constraint on the post-condition is

$$return \geq 0$$

which does not contribute any new constraints. It is possible, however, for a constant input value, to actually compute an interval on the return value. Taking the square root of `Integer.MAX_VALUE` produces a value between 46340 and 46341, so we could take this interval as a constraint on $\texttt{max\_sqrt}_{\texttt{double}}$, but we know that the value is neither 46340 nor 46341, and we add this information explicitly as a set of excluded values:

```
6: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   abs_long ∈ [0, 2147483647]
   max_sqrt_double ∈ [46340, 46341]  with excluded values {46340,46341}
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
   abs_int – abs_long = 0
```

The cast to a long on line 6 rounds down the value of $\texttt{max\_sqrt}_{\texttt{double}}$. Because of the excluded value of 46341 we can infer the exact value of $\texttt{max\_sqrt}_{\texttt{long}}$:

```
7: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   abs_long ∈ [0, 2147483647]
```

```
   max_sqrt_double ∈ [46340, 46341] with excluded values {46340,46341}
   max_sqrt_long = 46340
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
   abs_int – abs_long = 0
```

The final check on line 7 constrains the value of $\texttt{abs}_{\texttt{long}}$ and thereby, via the relationships collected, the range of `data` as required:

```
8: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   abs_long ∈ [0, 2147483647]
   max_sqrt_double ∈ [46340, 46341] with excluded values {46340,46341}
   max_sqrt_long = 46340
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
   abs_int – abs_long = 0
   abs_long – max_sqrt_long ≤ 0
```

and we can infer in line 9 that the multiplication is safe with respect to integer overflow:

```
9: data ∈ [[-2147483647, 2147483647]
   abs_int ∈ [0, 2147483647]
   abs_long ∈ [0, 2147483647]
   max_sqrt_double ∈ [46340, 46341] with excluded values {46340,46341}
   max_sqrt_long = 46340
   result ∈ [0, 2147395600]
   data + abs_int ≥ 0
   -data + abs_int ≥ 0
   abs_int – abs_long = 0
   abs_long – max_sqrt_long ≤ 0
```

2.7.1.4.2 Results and Discussion

The CWE 190 test suite has 2553 tests with 9867 reachable additions and multiplications. Of these 2553 could result in overflow and 7314 are safe. Our analysis is able to prove 7107 of these safe, and hence has 207 false positives.

For the square tests, we exercise more of our capabilities: expressive summaries of JDK methods and an abstract interpretation analysis over a powerful domain. In the polyhedral-based domain that we are using, we can express all the inequalities arising from the properties of the `Math.abs` and `Math.sqrt` as well as the conditionals in the guards. Without these, our analysis would not be able to prove this many test cases safe. We carefully designed our analysis such that it will scale up. Thus, our domain might behave as a polyhedral domain, but under the hood it is structured in a way to avoid the known complexity problems of working with polyhedra. The main innovation was to remove all the inequalities that contain only one variable (such as `x ≥ 3`) from the polyhedra and express them using intervals (`x ∈ [3,Integer.MAX_VALUE]`)(x in [3. This was especially important for an analysis of

over/underflow where we are interested in the range constraints. In Phase 2 we were using the Parma Polyhedra Library (PPL), which is a fast and efficient C++ library, but some operations on polyhedra are exponential in the dimension (number of variables) with one of the worse cases being a bounded polyhedron in all dimensions. By removing the range inequalities, we avoid the worst case, we greatly reduced the number of constraints and we can minimize the dimension by only keeping the variables in the multi-variable constraints in the polyhedra. The trade-off is more complex code and a performance penalty in the case of small applications. In Phase 3 we eliminated our reliance on PPL altogether and replaced it fully with the efficient combination of linear equalities and intervals.

2.7.1.5 CWE 369: Division by Zero

CWE 369 describes the weakness that *the product divides a value by zero* (cwe.mitre.org).

The CWE 369 test suite is organized along four axes:

1. 37 control/data flow variants, the same as shown in Table 5
2. 2 data types: float or int
3. 14 input sources, as shown in Table 4, with two additional sources:
    a. rand: Set data to a random value
    b. zero: Set data to zero
4. 2 types of operations: divide and modulo

There are 46 groups of 37 tests for a total number of 1702 tests (not all combinations are represented).

2.7.1.5.1 CodeHawk Analysis

Proving absence of division by zero introduces two new difficulties in the analysis. Guards that protect against division by zero typically test for a dis-equality, which can be represented precisely only by a disjunctive domain. Almost all abstract numerical domains are conjunctive; disjunctive domains are unattractive because of their high computational complexity. A second challenge in proving absence of division by zero is that zero is the default value of any numeric field before initialization, and hence we need to be able to ascertain whether a field has been initialized before it is used in a division operation.

goodB2G: integers The guard for the integer goodB2G (bad source, good sink) tests is

```
1: if (data != 0) {
2:   int result = 100 / data ;
3:    ….
4: }
```

Before entering the `if` block the variable `data` is constrained only by the interval associated with its data type:

```
1: data ∈ [-2147483648, 2147483647]
```

After the conditional on line 1 the true range is

`[-2147483648,-1] ∪ [1,2147483647].`

None of our numeric domains, however, can express unions of ranges. The reason is that the complexity of disjunctive sets is exponential in the number of joins, and intractable in practice for the general case. To prove absence of division by zero, however, we do not need the expressiveness of the general case. A much more targeted and efficient solution suffices to express the constraints required to prove the necessary conditions. Instead of maintaining a general disjunction of sets, we maintain an interval augmented with a set of excluded values. Using this approach we represent the disjunctive range `[-2147483648,-1] ∪ [1,2147483647]` as

`2: data ∈ [-2147483648, 2147483647]` with excluded values: `{0}`

from which we can infer that the division is well defined. Note that the propagation of such intervals through arithmetic operations is more complex than those for regular intervals as all excluded values have to be specifically accounted for.

goodB2G floats The guard for the goodB2G tests for floats is

```
1: if (Math.abs(data) > 0.000001) {
2:   int result = 100.0 / data;
3:    ….
4: }
```

In the CodeHawk internal representation this becomes (in pseudo code)

```
1: float abs_float = Math.abs(data) ;
2:  if (abs_float > 0.000001) {
3:  int result = 100.0 / data
4:  …..
5: }
```

We do not keep the type range for float and double variables, so at line 1, the interval for the variable `data` is unconstrained:

`1: data ∈ (-∞,∞)`

Instantiating the post condition of the summary for `Math.abs` produces

```
2: data ∈ (-∞,∞)
   abs_float ∈ [0,∞)
   data + abs_float ≥ 0
   -data + abs_float ≤ 0
```

and after the conditional on line 2, zero is excluded from the values of $\texttt{abs}_{\texttt{float}}$. To enable to back-propagate this constraint to the variable `data` we record that $\texttt{abs}_{\texttt{float}}$ is the absolute value of `data`:

```
2': data ∈ (-∞,∞)
    abs_float ∈ [0,∞) with equal(abs(data))
    data + abs_float ≥ 0
    -data + abs_float ≤ 0
```

which produces the following constraints at the location of the division:

```
3: data ∈ (-∞,∞) excluded values {0}
   abs_float ∈ [0,∞) with equal(abs(data)) and excluded values {0}
   data + abs_float ≥ 0
   -data + abs_float ≤ 0
```

which are sufficient to prove the safety of the division operations in the goodB2G tests for floats.

goodG2B tests For division zero is the “bad" value. Zero also is the default value of uninitialized fields and array elements. To be able to prove absence of division by zero we need to be able to exclude the uninitialized case. In tests 45 and 66, our analysis is not able to do so, resulting in false positives for these tests.

In tests 45 data is passed as a field. Although there are separate fields `dataBad`, `dataGoodG2B`, and `dataGoodB2G`, these fields are not initialized in constructors and therefore, to be sound, we have to add the default value zero as a possible value for these fields. The fields are actually set before they are used, but our analysis is not able to capture this.

In tests 66, data is passed in an array. The interval that we associate with an array is an over-approximation of the set of values of all array elements. As we do not know if all elements have been assigned a value, we always add zero the aggregate interval. In tests 66 only array element 2 has a value and only this element is used, but our analysis does not track array elements individually and thus is not able to prove this case safe.

In addition to the false positives for tests 45 and 66 we have false positives for 67 and 68 for the same reason as explained for CWE 190 and CWE 191.

2.7.1.5.2 Results

The CWE 369 test suite has 1702 tests with 6578 reachable operations of interest. Of these operations 1702 could results in a division by zero, and 4876 are safe. Our analysis is able to prove 4646 of these safe, and hence has 230 false positives.

2.7.1.6 CWE 197: Numeric Truncation Errors

CWE 197 describes the weakness that *a truncation error occurs when a primitive is cast to a primitive of a smaller size and data is lost in the conversion* (cwe.mitre.org).

The CWE 197 test suite is organized along three axes:

1. 37 control/data flow variants, the same as shown in Table 5
2. 11 inputs sources, as shown in Table 7
3. 3 types of conversions: int – byte, int – short, short – byte

There are 33 groups of 37 tests for a total number of 1221 tests.

Name	Source of bad data
Environment	Read data from an environment variable
File	Read data from a file
Property	Read data from a system property
PropertiesFile	Read data from a properties file
URLConnection	Read data from a web server with URL Connection
Connect_tcp	Read data using an outbound tcp connection
Console_readLine	Read data from the console using ReadLine
database	Read data from a database
Listen_tcp	Read data by listening to a tcp connection
large	Sets data to Short.MAX_VALUE
random	Sets data to a random value

Table 7 CWE 197: Different source of "bad" data

2.7.1.6.1 CodeHawk Analysis

The range analyses performed in these tests are analogous to those described for CWE 190 and CWE 191 without any additional challenges. The false positives for this test suite are the same as those for CWE 190: tests 67, 68 (good and bad data stored in the same field), and tests 75 (data is serialized).

2.7.1.7 CWE 129: Improper Validation of Array Index

CWE 129 describes the weakness that *the product uses untrusted input when calculating or using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array* (cwe.mitre.org).

The CWE 129 test suite is organized along the following axes:

1. 37 control/data flow variants, the same as shown in Table 5
2. 15 inputs sources, as shown in Table 4, with three additional sources:
    a. large fixed: Set data to 100

   b. negative fixed: Set data to -1
   c. rand: Set data to a random number
3. 5 types of sinks:
   a. array_read_check_max
   b. array_read_check_min
   c. array_read_no_check
   d. array_write_no_check
   e. array_size

2.7.1.7.1 CodeHawk Analysis

To be able to analyze conditions on array lengths we added more information to the extra information we keep on each variable: the interval of values for the size, the variable that was used as size when the interval was created, and other variables that were assigned the array length as value. For each array, during the abstract interpretation analysis (and thus at each location in the program) we may have some or none of this information.

For the goodB2G tests in array_read_check_max, array_read_check_min, array_read_no_check, and array_write_no_check, the access to an array of size 5 is guarded by a check

```
1: int array[] = { 0, 1, 2, 3, 4}
   …..
2: if (data >= 0 && data < array.length) {
3:   IO.writeLine(array[data]) ;
4:   ….
5: }
```

Before the conditional we have the following information

`2: array ∈ [0,4]` array length interval: `[5,5]`

and after the conditional

```
3: array ∈ [0,4] array length interval: [5,5]
   data ∈ [0,4]
```

from which we can infer that the access is safe.

In the goodG2B and bad test the check `data ≥ 0` or the check `data < array.length` or both are missing. Except for tests 67, 68, and 75, which result in false positives for the same reason as discussed earlier, we correctly infer the safe source for all of the goodG2B tests.

Tests 66 have other array accesses besides the ones tested: `data` is passed from one method (bad(), goodG2B(), or goodB2G()) to a sink method (badSink(int dataArray[]), goodG2BSink(int dataArray[]), or goodB2GSink(int dataArray[]), respectively) in an array. The array is created with fixed size of 5, `data` is stored at index 2 and is read in another method. In the first analysis

pass through the call graph (bottom-up), there is no information on the parameter `dataArray`. In the second pass (top-down), however, the analysis starts with

`dataArray` ∈ `[…]` array length interval: `[5,5]`

and therefore

```
int data = dataArray[2]
```

will be resolved as a safe access.

The tests with the sink array size have an additional (probably unintended) weakness that renders the goodB2G cases unsafe. In the goodB2G tests `data` is the size of an array to be created. A guard is provided to ensure that the array has at least one element, so that the subsequent array access with index 0 is safe:

```
1:   int array[] = null ;
2:   .....
3:   if (data > 0) {
4:     array = new int[data] ;
5:   } else {
6:     IO.writeLine("Array size is negative") ;
7:   }
8:   array[0] = 5 ;
   .....
```

This code, however, has the problem that the array may not be created at all, resulting in a NullPointerException at line 8. Our analysis therefore is (correctly) not able to prove this group of 742 goodB2G tests safe, as they are in fact unsafe.

2.7.1.7.2 Results and Discussion

The CWE 129 test suite has 2664 tests with 10,512 reachable array accesses, of which 2664 can result in an ArrayIndexOutOfBoundsException and 742 can result in a NullPointerException, which leaves 7106 safe accesses. Of the 7106 safe accesses we are able to prove 6890 safe, and thus we have 216 false positives.

2.7.1.8 CWE 193: Off by one Error

CWE 193 describes the weakness that *a product calculates or uses an incorrect maximum or minimum value that is 1 more, or 1 less than the correct value* (cwe.mitre.org).

The CWE 193 test suite is organized along two axes:

1. 17 control flow variants (control flow variants 1-17 in Table 5)
2. 3 types of loops used as sink: for, while, and do

2.7.1.8.1 CodeHawk Analysis

In all of these tests, the array `intArray` of size 10 is accessed inside a loop using a loop counter as the index. In the good tests this loop counter is strictly less than the length of the array, while in the bad tests it is less than or equal to the length of the array.

The CWE 193 tests expose a problem with the abstract-interpretation based analysis of loops when a bounded domain is used. CodeHawk precisely discriminates between the good and bad cases for all the tests with `for` and `while` loops, but is not able to do so for the tests with `do-while` loops. To illustrate the problem we show the main steps of the analysis of a loop similar to the `for`-loop used in the test, and one similar to the `do-while` loops used in the test.

Consider the fragment

```
    int[] intArray = new int[10];
    int i=0 ;
1:  while(true) {
2:    if (i >= intArray.length) break ;
      ......
3:    intArray[i] = i ;
      ......
4:    i++ ;
```

This loop is a decomposed version of a for-loop. At the entry of the loop we have

$i \in [0,0]$

During the first symbolic execution of the loop, we get the following interval for `i`:

$1\text{-}4\text{: } i \in [0,0]$

At the head of the loop, the starting values are joined with the values obtained after the first pass:

$[0,0] \cup [1,1] = [0,1]$

This interval is an approximation of the set of values of `i` after at most one iteration of the loop, and is the start of the second pass. During the second symbolic execution of the loop we get

$1\text{-}4\text{: } i \in [0,1]$

At the head of the loop, after joining the values obtained previously to those obtained in this pass we get

$[0,1] \cup [1,2] = [0,2]$

which approximates the set of values after at most two iterations. At this point our analysis widens the set of values by keeping only the constraints that were true at the head of the loop in the previous iteration ([0,1]) and the current iteration ([0,2]), resulting in the interval [0,`Integer.MAX_VALUE`]. During the next symbolic execution of the loop we get

```
1-2: i ∈ [0,Integer.MAX_VALUE]
3-4: i ∈ [0,9]
```

Joining the intervals at the head of the loop, we get again

```
i ∈ [0,Integer.MAX_VALUE]
```

which means that we reached a fixed point. Thus at line 3 the analysis establishes the invariant

```
i ∈ [0,9]
```

from which the analysis infers that the array access is safe.

The situation for a do-while loop is different. A do-while loop can be represented as follows:

```
    int[] intArray = new int[10] ;
    int i = 0 ;
1:  while(true) {
    .....
2:    intArray[i] = i ;
      .....
3:    i++ ;
      .....
4:    if (i >= intArray.length) break;
    }
```

At the beginning of the loop we have again

```
i ∈ [0,0]
```

During the first symbolic execution of the loop, we get the following intervals for `i:`

```
1-3: i ∈ [0,0]
4:   i ∈ [1,1]
```

At the head of the loop the staring values are joined with the values obtained from the first pass:

```
[0,0] ∪ [1,1] = [0,1]
```

which is the starting point for the second pass. During the symbolic execution of the loop we get

```
1-3: i ∈ [0,1]
```

```
4:   i ∈ [1,2]
```

After the join at the head of the loop we get

```
[0,0] ∪ [1,2] = [0,2]
```

and after widening the interval becomes

```
i ∈ [0,Integer.MAX_VALUE]
```

During the next symbolic execution we get

```
1-3: i ∈ [0,Integer.MAX_VALUE]
4:   i ∈ [Integer.MIN_VALUE, Integer.MAX_VALUE]
```

The reason for the change in the lower bound is that `i` can overflow when it has the range `[0,Integer.MAX_VALUE]` and is then incremented. So at line 4, the value of `i` is now unknown.

After the conditional at the end of the loop, we get

```
i ∈ [Integer.MIN_VALUE,9]
```

Joining the intervals at the head of the loop, we get again

```
i ∈ [Integer.MIN_VALUE, Integer.MAX_VALUE]
```

and another pass through the loop will establish that this is the fixed point

```
4:   i ∈ [Integer.MIN_VALUE, Integer.MAX_VALUE]
```

which does not allow us to prove that the array access is safe.

Notice that the inability to establish a precise bound for the do-while loops is a consequence of modeling overflow and potential wrap-around. With a regular interval domain that widens to infinity, we would have been able to prove this array access safe.

2.7.1.8.2 Results and Discussion

The CWE 193 test suite has 51 tests, which contain 147 reachable array accesses. Of these array accesses, 51 can result in an ArrayIndexOutOfBoundsException and 96 are safe. We are able to prove 64 of these 96 safe, and hence have 32 false positives (33%).

The high false positive rate results from using an abstract domain that incorporates wrap-around. We are researching alternative widening strategies that do not suffer from the loss of precision caused by the regular widening strategy.

2.7.2 Internal Testing on Large Applications

During Phase 2 we used nine medium to large, real-world test applications for internal testing. The applications ranged from relatively small libraries and applications (ASM and BCEL) to very large applications (geotools). All applications except BCEL were analyzed starting from a main class and main function, loading classes and methods as they are referenced.

Loading classes and methods as they are referenced may miss classes and methods that are introduced via dynamic loading or called via dynamic dispatch or reflection. Our analyzer makes a best effort to incorporate these classes and methods using the following mechanisms:

- Class string literals: all string literals that can be interpreted as class names are considered as classes that are potentially dynamically loaded, and are loaded if they are available in any class files appearing on the class path;
- Method string literals: all string literals that are passed to methods that are known to invoke methods reflectively are considered names of methods that can be called; for all classes in the system that have methods with that name, those methods are added to the system
- Dynamic dispatch: all methods appearing in library interfaces are considered callable from within library functions, and thus all application function belonging to classes that implement these interfaces are added to the system, even if they are never explicitly called in the application itself.

These strategies, however, are not exhaustive. Class names for dynamic loading may be obtained via system properties, or via the network, or may be constructed from several substrings via StringBuilders. These classes will be missed in the analysis.

An alternative approach to circumvent the potential incomplete loading problem is to load and analyze all methods of all classes and jars that appear on the class path. For small applications this may be feasible and the CodeHawk analyzer has the option to do so (one of the test cases, BCEL, is analyzed in this way), but for large applications, this is generally not practical.

2.7.2.1 Description of Test Cases

1. ASM (version 4.1)
    - Description: all-purpose Java byte code manipulation and analysis framework
    - URL: asm.ow2.org
    - Test case: asm.test.ClassWriterTest.class (special-purpose main program, written in-house)
2. BCEL (version 5.2)
    - Description: library to analyze, create, and manipulate Java class files
    - URL: commons.apache.org/proper/commons-bcel
    - Test case 1: all nine main methods included in the package
    - Test case 2: all classes and methods included in the package
3. Cassandra (version 1.2.0)
    - Description: open-source distributed database management system

- URL: Cassandra.apache.org
- Test case: org.apache.cassandra.service.CassandraDaemon.class (server accepting requests)

4. Cayenne (version 3.0.2)
   - Description: open-source persistence framework providing object-relational mapping (ORM) and remote services
   - URL: cayenne.apache.org
   - Test case: cayenne.test.CayenneModelerTest.class (special-purpose main program, written in-house, that exercises the modeling framework)
5. Geotools (version 8.4)
   - Description: open-source Java library that provides tools for geospatial data
   - URL: www.geotools.org
   - Test case: four tutorial programs provided with the system:
     i. Org.geotools.tutorial.quickstart.Quickstart.class
     ii. Org.geotools.tutorial.feature.Csv2Shape.class
     iii. Org.geotools.tutorial.crs.CRSLab.class
     iv. Org.geotools.tutorial.filter.QueryLab.class
6. Lucene (version 4.0.0)
   - Description: open-source information retrieval software library
   - URL: lucene.apache.org
   - Test case: two test programs that exercise the search engine, written in-house
     i. Lucene.test.emailsearch.Main.class
     ii. Lucene.test.hotelsearch.Main.class
7. Pdfbox (version 1.7.1)
   - Description: open-source Java tool to create and manipulate pdf documents
   - URL: pdfbox.apache.org
   - Test case: seven example programs that are distributed with the system:
     i. Rg.apache.pdfbox.examples.pdmodel.CreateBlankPDF.class
     ii. Org.apache.pdfbox.examples.pdmodel.CreateLandscapePDF.class
     iii. Org.apache.pdfbox.examples.pdmodel.RemoveFirstPage.class
     iv. Org.apache.pdfbox.examples.pdmodel.ReplaceString.class
     v. Org.apach.pdfbox.examples.pdmodel.AddMessageToEachPage.class
     vi. Org.apache.pdfbox.examples.pdmodel.AddImageToPDF.class
     vii. Org.apache.pdfbox.examples.pdmodel.ImageToPDF.class
8. Tomcat (version 7.0.33)
   - Description: open-source implementation of the Java Servlet and JavaServer Pages technologies
   - URL: tomcat.apache.org
   - Test case: org.apache.catalina.sratup.Bootstrap.class (bootstrap loader of the server)
9. Xerces (version 2.11)
   - Description: open-source software providing robust full-featured, commercial-quality XML parsers and related technologies
   - URL: xerces.apache.org
   - Test case: xerces.test.DomWriterTest.class (test program, written in-house)

Table 8 lists the number of methods analyzed for each of the nine test programs. The methods are divided into application methods, auxiliary methods, and JDK methods. The application methods are typically those methods that get compiled when the application is downloaded and installed. The auxiliary methods are those methods that are provided in (precompiled) auxiliary jars that come with the application. We have observed that the methods in these jars have often been compiled with very old versions of Java, often going back to Java 1.1. The JDK methods designate all methods that are provided by the JRE standard libraries (rt.jar, jce.jar, jsse.jar, tools.jar, etc.). The column "method summaries" indicates the number of method summaries that were used in the analysis.

program	Application methods	Auxiliary methods	JDK methods	Total methods analyzed	Method summaries	Analysis time (sec)
asm	439	0	450	889	242	83
bcel	2020	0	1108	3128	1119	191
Bcel(library)	2903	0	1113	4016	1214	215
cayenne	4206	3198	678	8082	1498	648
cassandra	8065	10,801	2348	21,214	2098	1717
geotools	10,192	0	6795	16,987	2907	3070
lucene	4317	0	1181	5498	1178	561
pdfbox	2048	150	363	2561	1049	342
tomcat	5024	98	967	6089	1684	545
xerces	528	0	8	536	288	36

Table 8 Programs analyzed as part of internal testing

2.7.2.2 CodeHawk Analysis Results

All of the tests were run with a MacBook Pro with a 2.6 GHz Intel Core i7 processor with 16GB of 1600MHz DDR RAM. Table 9 through Table 13 report the number of potentially unsafe constructs proven safe for the individual CWEs.

Table 14 aggregates the result for all CWEs. The tables show the number of relevant (that is, potentially unsafe) constructs, the subset of these that were proven safe by numeric analysis and an additional set that was proven safe by taint analysis. The ratio of the number of proven safe constructs to the total number of constructs is a measure for the reduction in runtime overhead, assuming that each of these constructs is equally likely to be executed in any run, as these instructions do not have to be instrumented and checked at runtime.

program	#loops	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	205	5	8	13	6%
bcel	333	16	12	28	8%
Bcel (library)	430	16	10	26	6%
cayenne	633	15	75	90	14%
cassandra	1348	47	98	145	10%
geotools	2187	169	238	407	18%
lucene	1029	45	136	181	17%
pdfbox	392	29	35	64	16%
tomcat	1211	70	241	311	25%
xerces	89	17	9	26	25%

Table 9 Loops proven safe (eliminating the need for instrumentation) (CWE606)

program	#arithmetic operations	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	1121	325	20	345	30%
bcel	575	405	6	411	71%
Bcel (library)	751	522	5	527	70%
cayenne	512	281	9	290	56%
cassandra	2221	1020	70	1090	49%
geotools	4795	2475	43	2518	52%
lucene	3823	1095	55	1150	30%
pdfbox	1040	544	21	565	54%
tomcat	1587	849	33	882	55%
xerces	96	76	7	83	86%

Table 10 Arithmetic operations proven safe from integer overflow (CWE190)

program	#arithmetic operations	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	1121	881	1	882	78%
bcel	575	460	0	460	80%
Bcel (library)	751	611	0	611	81%
cayenne	512	397	4	401	78%
cassandra	2221	1143	46	1189	53%
geotools	4795	3205	17	3222	67%
lucene	3823	2783	30	2813	73%
pdfbox	1040	786	5	791	76%
tomcat	1587	1158	17	1175	74%
xerces	96	88	0	88	91%

Table 11 Arithmetic operations proven safe from integer underflow (CWE 191)

program	#down cast operations	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	66	4	0	4	6%
bcel	9	7	0	7	77%
Bcel (library)	20	12	0	12	60%
cayenne	23	9	3	12	52%
cassandra	238	14	29	43	18%
geotools	341	25	17	42	12%
lucene	188	24	4	28	14%
pdfbox	232	26	7	33	14%
tomcat	52	9	4	13	25%
xerces	14	0	0	0	0%

Table 12 Down cast operations proven safe (CWE 197)

program	#division operations	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	29	29	0	29	100%
bcel	15	15	0	15	100%
Bcel (library)	17	17	0	17	100%
cayenne	7	5	0	5	71%
cassandra	104	77	1	78	75%
geotools	126	104	0	104	82%
lucene	91	65	0	65	71%
pdfbox	82	77	0	77	93%
tomcat	18	18	0	18	100%
xerces	1	1	0	1	100%

Table 13 Division operations proven safe from division by zero (CWE 369)

program	#constructs/ operations	Numeric analysis	Taint analysis	Total	Reduction in runtime overhead
asm	2536	1244	29	1273	50%
bcel	1507	903	18	921	61%
Bcel (library)	1969	1178	15	1193	61%
cayenne	1687	707	91	798	47%
cassandra	6132	2301	244	2545	42%
geotools	12,244	5978	315	6293	51%
lucene	8954	4012	225	4237	47%
pdfbox	2786	1462	68	1530	55%
tomcat	4455	2104	295	2399	54%
xerces	296	182	16	198	67%

Table 14 Constructs and operations proven safe (summary)

2.7.3 Phase 2 T&E Applications

The tables below show the performance of CodeHawk on the Phase 2 T&E applications on the same metrics as described above.

program	Application methods analyzed	JDK methods analyzed	Total methods analyzed	Method summaries used	Analysis time (sec)
7-FTPS	3230	0	3230	1451	102
9-BARC	913	50	963	757	41
10-FIND	10,414	3411	13,825	2932	813
11-HTML	357	0	357	448	17
12-JMET	5048	1855	6903	2434	248
15-PMDX	14,956	287	15,243	1531	1543
16-SCHE	869	888	2757	1282	74

Table 15 Size and analysis times for Phase 2 T&E applications

programs	loops	% proven trusted	%proven bounded
7-FTPS	396	10%	39%
9-BARC	189	26%	62%
10-FIND	1440	5%	38%
11-HTML	93	6%	23%
12-JMET	729	8%	43%
15-PMDX	2562	11%	36%
16-SCHE	247	1%	12%

Table 16 Percentage of loops proven untainted and bounded

program	CWE 190		CWE 191		CWE 197		CWE 369	
	operations	%safe	operations	%safe	Operations	%safe	operations	%safe
7-FTPS	544	52%	544	77%	14	0%	2	50%
9-BARC	883	64%	883	76%	42	31%	88	93%
10-FIND	2484	44%	2484	69%	114	31%	63	73%
11-HTML	152	55%	152	95%	3	0%	0	n/a
12-JMET	1215	53%	1215	77%	67	21%	12	50%
15-PMDX	6547	46%	6547	71%	599	26%	274	92%
16-SCHE	125	40%	125	73%	2	50%	11	100%

Table 17 Percentage of arithmetic operations proven safe

program	Error-returns	%checked	%returned	%argument	%popped	%unknown
7-FTPS	92	65%	2%	7%	13%	13%
9-BARC	43	37%	5%	14%	5%	39%
10-FIND	241	56%	2%	13%	16%	13%
11-HTML	22	46%	0%	0%	14%	40%
12-JMET	91	53%	4%	9%	12%	21%
15-PMDX	239	63%	1%	10%	6%	20%
16-SCHE	70	34%	0%	0%	60%	6%

Table 18 Status of error return values

2.7.4 Phase 3 T&E Applications

The tables below present some of the analysis results for the Phase 3 T&E applications and for jfreechart, a graphics application used for internal testing.

program	Application methods analyzed	JDK methods analyzed	Total methods analyzed	JDK method summaries used	Analysis time (sec)
CMUD	5275	290	5565	1545	1067
ELAS	39,810	290	40,100	2320	8776
JENA	18,156	40	18,196	1761	1866
LENY	15,458	2320	17,778	2407	1661
LUCE	4735	74	4809	1072	298
POIX	12,382	88	12,470	1488	649
JFREECHART	9102	1604	10,706	2092	358

Table 19 Size and Analysis Times for Phase 3 T&E Applications and Internal Testing Programs

program	loops	Percent trusted	Percent bounded
CMUD	1279	9%	40%
ELAS	7134	5%	41%
JENA	2651	11%	40%
LENY	2769	10%	43%
LUCE	1110	5%	45%
POIX	1653	11%	52%
JFREECHART	1004	6%	57%

Table 20 Percentage of loops proven untainted and bounded

program	CWE 190		CWE 191		CWE 197		CWE 369	
	operations	%safe	operations	%safe	operations	%safe	operations	%safe
CMUD	3455	48%	3455	75%	385	19%	55	84%
ELAS	21,568	37%	21,568	67%	1393	28%	386	65%
JENA	5861	49%	5861	76%	480	49%	104	75%
LENY	6631	50%	6631	76%	603	35%	137	75%
LUCE	4287	31%	4287	73%	211	19%	101	62%
POIX	6287	46%	6287	72%	579	23%	163	84%
JFREEC	2521	54%	2521	60%	447	9%	110	73%

Table 21 Arithmetic operations proven safe

program	Error returns	%checked	%returned	%argument	%popped	%unknown
CMUD	178	55%	3%	18%	6%	17%
ELAS	399	44%	7%	8%	24%	17%
JENA	380	51%	3%	11%	25%	10%
LENY	590	66%	3%	13%	3%	15%
LUCE	36	61%	6%	-	19%	14%
POIX	215	45%	-	-	9%	46%
JFREEC	61	33%	30%	2%	16%	20%

Table 22 Status of error return values

2.8 Future Work

Static analysis of Java programs is a complex task. Four years of development has resulted in a quite powerful analyzer. However, there are of course always many areas that can be improved. Two of those areas are (1) validation of JDK summaries, and (2) architecture analysis. We briefly describe each of these below.

2.8.1 Validation of JDK Summaries

As described above, the analysis relies heavily on (partially manually constructed) JDK summaries for its scalability. Although we were able to partially automate the construction and inspection of these summaries, and succeeded in creating almost 10,000 summaries, it would be desirable to have a means by which these summaries can be validated against JDK implementations. Often the API documentation of JDK methods is incomplete or under-specified, perhaps leaving room for certain features to be implementation-defined. On the other hand, certain JDK implementations may actually not be compliant with the API documentation. Having a means of validating JDK summaries would benefit both sides of the API: the users of the API and the implementers of the API, to establish consistency and completeness.

In addition to checking direct compliance, it would also be desirable to determine to what extent the implementations of JDK methods depend on the “correctness” of application code, to be compliant with their APIs. For example, if an application passes a List object to a JDK method

with a size() method that may return a negative number, how does this affect the behavior of the JDK method?

These questions require in-depth analysis of the JDK, including its many native methods.

2.8.2 Architecture Analysis

As described above, the main analysis starts from a main class/main method and includes classes in the analysis as they are encountered in the methods parsed. String sink analysis helps to include classes that are loaded dynamically and methods that are executed by reflection. Keeping track of possible callbacks and methods that can be dynamically dispatched helps to include other methods that are not directly referenced. As applications get more complex, however, more needs to be known about the application to ensure that all methods that could possibly be executed are included in the analysis.

The Phase 3 T&E application Lenya provides a good example of an application that requires more knowledge of the application structure to adequately analyze it. The application is a web applications and it reads xml files to determine which classes to load and which methods to execute in response to actions performed by the user on the user interface. We augmented the analyzer to include these sitemap.xml files as input to extract the possible classes that can be executed. It is not clear, however, whether this is a general technique for web applications, or whether many more frameworks exist that may load classes in yet different ways. More architecture analysis is required to determine what is the best way to ensure all application classes and methods are included in the analysis.

3 DYNAMIC TRACKING AND CONFINEMENT

3.1 Primitive Metadata

Metadata is tracked for each primitive value in the program. This tracking includes all uses of primitives including arrays, system library calls, etc. Each original primitive as a matching metadata value that tracks it through the program.

A key part of this approach is that each byte-code operation has a direct translation that is largely independent of context. This is possible because the value-metadata pair is maintained everywhere including on the Java stack. This results in a robust translation that can handle large complex programs and the Java system libraries without errors.

3.1.1 Metadata components

Metadata is used to track a variety of information about each value.

- **Data Source**. The source of the data. A source can be either trusted, untrusted, or unknown. Unknown sources can occur when a value is returned from uninstrumented code. For debugging purposes, VIBRANCE also tracks more precise sources such as properties, environment variables, files, sockets etc. Program constants are considered to be trusted. The configuration file specifies how to determine what sockets and files are trusted.
- **Data Type**. Integer types can be manipulated as numbers or as bit fields. When casting between integer types (e.g., byte to long), the type of the integer influences what checks are performed. Truncating casts (larger to smaller) are considered to be an error on numeric values (because the value is lost) but are considered acceptable on bit fields. The type is determined by the most recent operation on the value. A numeric operations (add, subtract, multiply, divide) indicates a numeric value. A bit operation (and, or, negate) indicates a bitwise value.
- **Data Status**. This specifies the status of the value. A value can be ok, overflowed, underflowed, or infinite. Each operation checks its result and sets the bits accordingly.

Metadata is stored in a 32 bit integer. This could be reduced by limiting specific source information. But in most cases, the space overhead is manageable.

3.1.2 Tracking Metadata

Metadata is maintained each time a primitive is manipulated. Each copy of a primitive copies its metadata. Each primitive operation calculates new metadata for the result based on the metadata of each of its operands.
Metadata thus must be maintained for each primitive variable in the program. This includes fields (both instance and static), method arguments, locals, stack temporaries, arrays, and return

values. In most cases, the metadata is simply a variable that is paired with original variable. For example, the method declaration

```
void method (int1, int2) {...}
```

becomes

```
void method (int1, int1_metadata, int2, int2_metadata) {...}
```

In a similar fashion, a metadata variable is paired with each field and local.

When a primitive value is pushed on the stack, both the original value and its metadata are pushed on the stack as a pair. Each primitive operation is translated to operate on the pair and return a value-metadata pair as a result.

3.1.3 Return values

Unfortunately, Java only allows a single return value from a function. It is thus not possible to return a value-metadata pair directly. VIBRANCE addresses this by adding a thread specific `VibranceData` parameter to each method signature. The `VibranceData` object is instantiated once per thread and then passed to each method called in that thread. Amongst other things, the VibranceData object has a field for returned metadata that is filled in when the method returns.

The caller then retrieves the metadata from `VibranceData` and pushes it on the stack to complete the value-metadata pair on the stack.

3.1.4 Arrays

Arrays are handled in a similar fashion to primitives. Each array is stored in an class that contains both the original array and a matching array of taint information. Each time an array element is read, both the value and its metadata are pushed on the stack. Similarly each time an array element is written, both the value and the metadata are written.

3.1.5 Instrumenting the System Libraries

For correct operation, Metadata must be maintained across calls to the Java system libraries (JDK). To achieve this, the JDK must be instrumented in a similar fashion to the application itself. This is difficult for a number of reasons:

1. Instrumenting primitives requires changing the signatures of methods and classes. These signatures may be presumed by the JVM and native routines. Changing signatures arbitrarily will cause crashes and incorrect behavior.
2. The instrumentation code itself uses the JDK. If it were to call instrumented code an infinite recursion would result.

VIBRANCE addresses the problem of method signatures by copying each JDK method. The signature change is made to the copied method leaving the original method unchanged. Native/JVM and VIBRANCE instrumentation calls to the original method will work correctly. The application, however, will be modified to call the instrumented version of each method (with the modified signature), ensure that metadata is properly propagated.

VIBRANCE also adds fields to classes. Since it leaves the original fields alone, this works in most cases. But in some core classes (such as `Integer`and `Long`) the JVM presumes the layout of the class and adding field result in a JVM crash. VIBRANCE resolves this problem by creating subclasses of each of those core classes and using/instrumenting the subclass instead.

3.1.6 Native/JVM Field access

Native code and JVM routines may access specific fields in objects or parameters. Largely this works correctly as is because the original fields are not modified by VIBRANCE's instrumentation.

Arrays are modified however (they are converted to an object that contains both value and metadata arrays). When arrays are passed as parameters to native routines, the value array is extracted from the object and passed to the routine. Note that if the array is being set by the native routine (as occurs in reading data from files or the network), the corresponding metadata is also set based on the source of the data.

3.2 Server Detection and Exception Handling

The rules of engagement define different security policies for server and user programs. It defines a server as:

A server application refers to a program that is designed to perform some function for multiple users during one invocation.

This is a somewhat vague definition for the purposes of program analysis. A more concrete definition that we can use would be a program that acts on inputs read over a socket. Such sockets are defined in `java.net` as `ServerSocket`. When a program attempts to attach to the server, a `Socket` is returned. Obviously, servers can be either single or multi-threaded.

VIBRANCE detects `ServerSockets` in the program and to determine some facts about how the program uses them:

1. Does the program support multiple connections via the socket? More precisely, is the `ServerSocket.accept()` call in a loop that does not include the creation of the `ServerSocket` itself.
2. Is each command (client socket) handled in a separate thread?

3. Is a separate thread created for each command or is a pool of worker threads used? If worker threads are used, is it possible to exhaust the pool of threads? Is there a simple way to create new threads?
4. Is there exception handling around the code executed with the client socket? If so, what exceptions are handled?

Given the above information, VIBRANCE modifies the program to ensure that errors that occur during one command do not stop the processing of other commands (both concurrent and subsequent). VIBRANCE can also enforce limits on how long the program will wait for input and also perhaps on the maximum amount of time a particular command should take.

3.2.1 Multi-threaded Servers

If an error occurs in a thread, it should not affect the server's ability to process other commands. VIBRANCE modifies multi-threaded servers as follows:

1. If there is code outside of the thread involved in processing the command, it is wrapped in an error handler in the same fashion as are single-threaded servers.
2. Any errors thrown back to the run() method of the thread are caught to ensure that the thread does not exit. This ensures that errors will not deplete the threads available in a thread pool.
3. All sockets are set to time out after reasonable amount of time (.5 seconds?). See single-threaded server for details.
4. Threads that consume too many resources (CPU, memory, file handles, etc) are terminated.

3.2.2 Single-threaded Servers

We need to ensure that single threaded servers don't hang and that any errors are handled and the server loops around to process the next command. VIBRANCE modifies a single threaded server as follows:

1. Adds an exception handler (if there is not one already there) around all of the code that operates on the client socket. Any exceptions that are caught should close the client socket and wait for new commands.

   Normally the server code will look something like:

```
ServerSocket serverSocket = null;
try {
  serverSocket = new ServerSocket(4444);
} catch (IOException e) {
  System.err.println("Could not listen on port: 4444.");
  System.exit(1);
}

while (true) {
  Socket clientSocket = null;
```

```
    try {
      clientSocket = serverSocket.accept();
    } catch (IOException e) {
      System.err.println("Accept failed.");
      System.exit(1);
    }

    //read/write to client socket
    ...
    clientSocket.close();
  }
```

VIBRANCE finds the server loop (the while loop above) and then adds an exception handler over all of the operations on the socket. As the above is written, there is not much point in having the exception handler include the creation of the client socket, but it can't hurt either. So VIBRANCE changes the code to look like:

```
  ServerSocket serverSocket = null;
  try {
    serverSocket = new ServerSocket(4444);
  } catch (IOException e) {
    System.err.println("Could not listen on port: 4444.");
    System.exit(1);
  }

  while (true) {
    try{
      Socket clientSocket = null;
      try {
        clientSocket = serverSocket.accept();
      } catch (IOException e) {
        System.err.println("Accept failed.");
        System.exit(1);
      }

      //read/write to client socket
      ...
      clientSocket.close();
    } catch (Throwable t) {
      if ((clientSocket != null) && !clientSocket.isClosed())
        clientSocket.close();
    }
  }
```

To do this we need to find the enclosing loop. Using a control flow graph of the application program (excluding the JDK), it is straightforwad to follow that graph upward from each serverSocket.accept() call to the enclosing loop. The exception handler can cover the entire loop (as shown above).

If the loop is not found in the current method, then VIBRANCE looks at all of the methods that could call the current method and performs essentially the same operation in the caller. In this case, we need access to the client socket to close it. The easiest way to do that is probably to add a thread specific reference to the client socket when it is created. That reference can be used to close the socket in the exception handler. Or we can just not worry about closing the socket in those cases for now.

2. VIBRANCE sets up client sockets to timeout after a reasonable amount of time (0.5 seconds by default). This is implemented by simply finding everywhere that a `Socket` is returned from `ServerSocket.accept()` and making the `setSoTimeout()` call. We should consider whether other mechanisms of creating a socket (e.g., client sockets) should also have this added. This is done directly by the runtime instrumentation.

3.2.3 Non-socket based servers

There are other ways that servers could be created in Java without the use of server sockets. The most obvious approach would be RMI. RMI should, however, not have many of the problems that sockets have, but we should look into it to see if there are any issues. CORBA is similar. It's also possible to use files and named pipes for communication.

VIBRANCE currently does not check for non-socket based servers.

3.2.4 Non-server applications

Some non-servers act very much like servers in practice. For example, browsers and other GUI programs have a loop over user input that is very similar to the server loops described above. VIBRANCE's techniques could be applied to these programs as well confining the effects of one command that accessed malicious data to that command only.

VIBRANCE currently does not support the server protections in GUI programs.

3.3 Repairs/Continued Operations

Optimally the program would continue correctly in the presence of errors. For the test and evaluation this is required for servers but not for non-server applications.

Whenever possible, VIBRANCE repairs the problem and continues to execute. This allows the normal code path to execute which is more likely to leave the program in a good state. For example, if quotes are added SQL data (as is commonly done in SQL attacks), they can be escaped. This negates the attack and allows the program to function normally (essentially adding the correct fix to the program).

In some cases, however, the problem cannot be repaired in place. For example, if an overflowed value is used in an `if` statement, it is not possible to know which branch should be taken

(without calculating an arbitrary precision result which would be prohibitively expensive). In cases, such as these VIBRANCE throws an exception.

If the application has its own exception handling, it will catch the problem and deal with it as it does other errors. This is likely to leave the program in a good state. If there is no application exception handling (which is often the case), VIBRANCE's server protections will catch the error at the level of a client request. That request will be terminated, but concurrent and subsequent requests will be handled correctly.

It is possible that the program could be left in an unexpected state as a result of terminating the user request. Our examination of real world servers, however, indicates that in most cases the server will continue to act correctly.

Other approaches were considered as well.

- We could force execution back to a known point. This could be implemented by adding our own exception handle at an appropriate point. The obvious difficulty with this approach is finding the appropriate point. The obvious approach would be to discard this input and go onto the next one. If we can determine the stream from which the bad input originated, we could perhaps go back to there. Could static analysis help figure this out? Or should we determine this dynamically (perhaps by tracking the source of unsafe data).

  Perhaps we should jump back to the nearest point on the current call stack where there is a call to the read that obtained the data. This would require obtaining a call stack on each read (which could be expensive). Though its possible that a heuristic based on the possible call graph would be good enough. We could just back up the call stack until we find a method that could possibly call the read that returned the unsafe data. It seems likely that would be the right place.

- We could just ignore the call and continue. In cases where the call doesn't return anything (eg, deleting a file) this might work very well.

3.4 Data Races (program variables)

There are a number of different scenarios of data races that we can address. The following subsections describe these.

One thing that we need to consider that applies to all of these scenarios is how to handle any existing synchronization. We don't want to add synchronization if it is already present. One simple approach that we could use to start would be to simply ignore any functions that contained synchronization. Or at least synchronization that surrounded the code of interest. Certainly methods that are already synchronized should be ignored.

3.4.1 Data Race - variable update

An update data race occurs when a variable can be updated from two different threads. If the read/write does not occur atomically for each thread, an incorrect result can occur.

One possible approach would be to add synchronization code where appropriate. To do this we would need to identify code that could possibly be accessed from multiple threads that might have race conditions. The easiest fix would be to add synchronized annotations to methods. For other situations, we could add individual locks.

We need to statically identify two conditions that are required for a update data race.

1. **Methods callable from multiple threads**. This can be determined from a call graph for each possible thread. One obvious case would be multiple threads started on the same class (a common case).

2. **Methods that perform updates**. These are methods that update (read and then write) the same field.

For methods that fit both conditions, we would add a synchronized annotation if the field was a direct instance field. We would add a lock otherwise. The simplest way to do the lock would be over the entire method (though in some simple cases they could be only over the update itself (eg, `count++`)). Because they will have to force the race condition to occur, we should expect delays or other code between the read and the write.

We also need to check to see if the added lock could cause a deadlock. To determine that, we need to calculate a *lock order graph*. This is a graph of all of the locks acquired by the program and the order in which they are acquired. Cycles in the graph indicate possible deadlocks. If the new lock causes a cycle, then it can cause a deadlock. The new lock would need to be added to the graph which requires a list of the locks that may already be held when entering the method and the locks that the method itself may acquire. Note that many locks are at a bottom level where they never acquire other locks within them (for example, the locks in StringBuffer or Vector). If only these locks are acquired under the new lock, then there cannot be deadlock (because there can be no cycle).

If the adding the lock can cause a deadlock, we can either not add it or just add it with a timeout (perhaps a small one). Another alternative would be to utilize our deadlock protections to handle any problems that arise.

To add locks other than synchronized method annotations, it will be necessary to identify the byte codes involved in the update. The monitor entry/exit would need to surround these byte codes. A try/catch block would also need to be added to ensure that exceptions don't cause the exit not to be executed. This could be implemented in phases, with easier implementations occurring first. The first step in any approach is to identify the start and end of the update code (eg, the location of the read and the location of the write). In order of increasing complexity, possible approaches are:

1. **Modified basic blocks**. We would only add the lock if there were no jumps into or out of the code from start to end. We would allow method calls.

2. **Support internal jumps**. In this case we would allow jumps between start and end whose targets are also between start and end.

3. **Support exits**. In this case we would allow a jump that exited the block. This would require a monitor exit at each jump that exited the block.

4. **Expand the block** If the block has incoming jumps, we could expand the block until the problem is resolved. Worst case, the block could be expanded to the entire method. It then becomes easy to synchronize but perhaps more subject to deadlock.

As a very first step, we could add only synchronized annotations (where appropriate) to entire methods that did not acquire any additional locks. Note that while not really correct, this might work for at least some cases where the item being updated is not a field within the class. For example, if a static field were updated and we synchronized the method that updated it, that would work as long as there were not similar updates in different methods.

We will also want to consider uses of JDK methods. In most cases, we presume that the JDK is thread-safe, but there are obvious exceptions. For example if a shared string is updated, we want to make sure that StringBuffer was being used rather than StringBuilder. Similarly for container classes.

Note that this approach will not be able to detect (easily) when an error was prevented. We will not be issuing a CWE. To issue a CWE we would have to add a test on the lock first and issue a CWE if the lock was already held. This does not seem worth the effort. We should, however, communicate this to the T&E team.

3.4.2 Class Based Data Race - Add Synchronization

Synchronization is also required anywhere that a group of variables needs to be dealt with atomically. A simple example would be a token class where the contents of certain fields depends on the type of the token. If the token type is integer then the `intvalue` field is set, otherwise it is not.

Obviously, this could be true of any arbitrary set of variables, but a lot of real world examples would be fields in the same class. This also encompasses data races over a single variable, and TOCTOU on variables.

Similar to data races, we would want to find classes whose methods are called from multiple threads. Our presumption would be that any method in the class should see a consistent set of values for all of the fields it accesses in the class.

In the simplest case, we could just add a synchronized flag on all methods of a class that is called from multiple threads. We should also check for possible deadlocks and either use a timeout or avoid the synchronization on classes that could deadlock.

There are a number of optimizations that could be used:

- Don't synchronize methods that only read a single variable.
- Don't synchronize methods that only read a single *active* variable, where an active variable is one that is written in a non-constructor method
- Create multiple locks for sets of variables that are used disjointly from one another.
- Use a locking scheme with multiple readers and a single writer.

Note that the key issue here is how well the additional synchronization works on the base programs. We want to make sure that it doesn't cause slowdowns or deadlocks. Once we can run the system in its simplest version we should be able to look at the classes it is synchronizing and determine what optimizations will work the best.

3.4.3 Time of check, Time of Use (variable)

This problem occurs when a variable is checked for some attribute and then used presuming that attribute is still true. If the attribute has changed since the check, the code will fail. For example:

```
if (p != null)
  p.m1();
```

This is very similar to the update data race with the check replacing the read and the use replacing the write. The key is that any writes to the variable in question have to be locked as well. Thus the lock approach would require:

```
synchronized (p_lock) {
  if (p != null)
    p.m1();
}
```

and

```
synchronized (p_lock) {
  p = null;
}
```

Seemingly this should be similar in complexity to data races.

One interesting question is whether or not there are *attributes* other than null. The `instanceOf` operator is similar to checking for null.

```
  if (x instanceof MyClass)
    MyClass y = x;
```

Note that as long as the variable in question is an instance variable the the class based data race approach (section) should handle this (as long as we consider the check and the use to be two accesses of the variable)

Another approach which might work well and at low cost would be to attempt to remove the problem. This can be done most easily when the check and the use are over the same variable (as in the examples above) Rather than operating directly on the variable, simply make a copy of it before performing the check and use. That guarantees that it cannot change. For example:

```
  MyClass localp = p;
  if (localp != null)
    localp.m1();
}
```

It is not clear whether or not this would always actually work. If another thread is setting p to null, it may also make changes to the internals of p that will cause it to fail. But given garbage collection, there is little motivation to do that (unless p has external descriptors of some sort that need to be closed).

To make this change would require us to identify the check/use and add the appropriate local variable. One key is to ensure that any check/use of a field is always using the same object to refer to the field (otherwise our change would break the program). This is easy for instance variables and static variable, the object is obvious. For variables referred to by a variable, we would have to ascertain that the variable was the same and held the same value.

In simple cases (instance variables and statics), this could probably be done without a complex analysis. In the method we'd need to note any instance/static fields that are checked, used, and not updated. The value of the field will be stored in a local and use of the local substituted for that of the field. Optimally the store would be at the first check, but it could possibly be done at method entry.

Optimally, though, this would be driven by the static analysis which could be extended to more complex cases and take advantage of the data race analysis results.

Yet another alternative would be to try to catch the error. For example:

```
  if (p != null) {
    try {
      p.m1();
    } catch (NullPointerException) {
       ...
    }
```

We could either just throw the error when it occurred (and count on the server protections to work) or we could attempt to end up in the else branch. That latter seems difficult if the contents of the if block are at all complex. Or we could simply swallow the error and continue.

This would require static analysis to determine what type of check should be performed. But that might be pretty straightforward for null checks and instanceof checks.

3.4.4 Singleton Pattern

The singleton pattern is very similar to variable TOCTOU. For example:

```
  private static NumberConverter singleton;
  public static NumberConverter get_singleton() {
    if (singleton == null) {
      singleton = new NumberConverter();
    }
    return singleton;
  }
```

In some cases (see CWE-609) the program may attempt double check locking to avoid the overhead of the synchronized call, but this does not consistently work either.

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
    }
  }
```

In both of these cases, the method performs an if-check on a variable for null and then later sets the variable. This should be straightforward to identify. If the variable in question is an instance variable, then the method can be synchronized. If the variable is static, then the method can be synchronized on the class.

3.4.5 Use of Non-thread-safe Classes

Code that is shared between multiple threads may access non-thread-safe classes (e.g., `StringBuilder`, `ArrayList`). This is, of course, only dangerous unless objects of the class are shared between threads.

There are safe versions of these classes (e.g., `StringBuffer`, `Vector`) that can be used instead. But these classes do not (in general) have a subtype relationship with one another. That means that not only does the creation of the unsafe object need to be changed, but all uses need

to be changed as well. Including passing the object to other methods (which might require duplicating the method to support both types). For most collections there is an interface which both the safe and unsafe versions implement, but there is no guarantee that the application always uses this interface (though it is common).

Rather than modifying the code to use the safe classes, we can add synchronization around the unsafe calls. For example:

```
class A {
  StringBuilder sb = new StringBuilder();
  void m1(String str) {
    synchronize (sb) {
      sb.append (str);
    }
  }
}
```

This approach is easier than using the safe class. It is easy to add synchronization around a call. We just need to know which calls need synchronization added. Determining that is the same as in the other data race scenarios. Any class whose methods are callable from multiple threads that contains a shared unsafe object should have synchronization added to all calls to the object.

Note that it is much easier to solve this problem on direct references to the shared field. If the shared object is passed as a parameter, this may be more difficult to find. That would require any methods passed the unsafe object to be changed to add synchronization around uses of the parameter. Alternatively, anytime an unsafe object is passed as a parameter, synchronization over the object could be added at that point. This could be both expensive and prone to deadlock.

3.4.6 Multi-threaded I/O

One possible problem is multiple threads writing output to the same stream. This is essentially the same problem as non-thread-safe classes (Section).

The difference is that this is perhaps more common and the expense of adding synchronization possibly higher. Also, if the application is a server, socket output is likely to look like an example of this (unless the analysis can determine that the client socket is not shared)

The next question would be whether or not a single call is all that is necessary to achieve synchronization on write. A method might make several print call and expect all of them to show up together. This is different from the other unsafe classes where synchronizing method calls would always seem to be sufficient.

Optimally we would synchronize around all shared stream writes in a method. This could possibly be expensive depending on how good a job we do of identifying shared streams. It could also be deadlock prone as writes can occur in odd places and putting the lock around blocks of code increases the chance that other locks will be involved. Perhaps we should not add a write lock around anything that can obtain another lock.

3.5 Time of Check Time of Use Details (Files)

The classic TOCTOU problem occurs in a setuid program. The program needs to check that the real user has access to a file. They use the access call to do that. Unfortunately, the file they check may have changed by the time that they open it.

Seemingly these sorts of problems are not applicable to Java. Java does not lend itself to setuid programs (it is very difficult to run it safely) and does not contain an equivalent to the `access()` system call.

There are, however, a number of TOCTOU file problems that are applicable to Java. Java programs can run directly as root or as special users (servers for example, often run as a special user) Consider for example a program that is trying to clear older temp files (including directories and their contents) from `/tmp`. To ensure that only files that reside in /tmp are deleted, symbolic links are checked for and not processed. But the check for symbolic links and the file deletion occur at different times and an attacker may change the file between those two events. Consider the following simple example:

```
    Privileged                  Attacker
    --------------------        ---------------------------
1                               mkdir(“/tmp/etc”)
2                               creat(“/tmp/etc/passwd”)
3   readdir(“/tmp”)
4   lstat(“/tmp/etc”)
5   readdir(“/tmp/etc”)
6                               rename(“/tmp/etc”,“/tmp/x”)
7                               symlink(“/etc”,“/tmp/etc”)
8   unlink(“/tmp/etc/passwd”)
```

In this case, the `lstat()` call is the check and the use is the `readdir()` call.

VIBRANCE’s approach is to ensure that the file checked and the file used are the same. The inode can be used to uniquely identify the file to ensure that the check and use are the same. At each check, the filename and its inode are stored in map. At each use, the filename and inode are checked against the map to ensure that they match. In addition to matching the inode any check whose result may change on the same inode (such as a permission) must be rechecked at use time to ensure that it has not changed.

Note that the inode will stay the same even if the content of the file changed. For example, if the file were written and then read back in, an attacker could change the contents of the file without changing the inode. In addition to the inode we check the time of the last status change (ctime) as well. That field will change anytime any change is made to the file (including meta-data such as permissions). It also cannot be reset by a user. The initial version of ctime had a granularity of a second, which is imprecise enough to miss changes. However, more recent kernels (> 2.5.48) support nanosecond resolution. This is supported by the ext4 file-system but not by ext3 (and

others). The file system on the current T&E machine is ext4 and has been tested to support nanosecond resolution.

For this to work each check and use must be implemented such that the matching inode is known. The trick is use an operation that returns all of the results at once (ie, is atomic with respect to the file system). As long as that is possible, the check/use and the inode should correlate.

Each check and use might require a slightly different approach. The easiest and most common method will be to use lstat or stat for the check. Both return both the inode and other pertinent information about the file in one operation. When the use is an open, the approach is to non-destructively open the file and then use fstat to obtain the inode and ensure that it matches any previous checks.

A list of check methods and the approach for each is below. This list is not comprehensive. We should check the Wei and Pu paper (FAST 2005) which lists all combinations of check/use to make sure we haven't missed anything.

`File` **check methods:**

- **canExecute(), canRead(), canWrite()**. Use stat to perform the check. This sort of check is known to have a problem if the directory containing the file is unreadable. But since stat itself will fail in this case, that would not seem to be a problem. The result of the check should be placed in the map and any future uses or checks should ensure that the bit has not changed.

- **exists()**. Use stat to check to see if the file exists and add the result to the map. If the file does not exist, add that fact to the map.

- **getAbsoluteFile(), getAbsolutePath(), getCanonicalPath(), getCanonicalFile**. Use stat on both the original filename and the result of the method. It is an error if the inodes are different. Put both filenames in the map.

- **getName(), getParent(), getParentFile(), getPath()**. Nothing needs to be done as these don't interact with the file system, they only manipulate the filename string.

- **isDirectory(), isFile(), lastModified()**. Use stat to check and add the file and the result to the map.

- **delete()**. The Pei/Wu paper points out that unlink can be thought of as checking to see if a file doesn't exist. Thus we should add to the map that this file doesn't exist.

`Files` **check methods:**

- **CreateDirectory(), CreateDirectories()**. These might be proceeded by a symlink check on any of the parent directories (ie, one may only want to create a directory if is located where expected. This doesn't seem that likely as one would expect any directory in the

path not to be accessible to a possible attacker. The correct check, however, would be to check each path component to make sure it still passes any checks. I think this can be safely skipped though.

- **delete(), deleteIfExists()**. Same as `File.delete()`.

- **exists()**. Same as `File.exists()`.

- **getAttribute(), getAttributes()**. Use stat to get the attributes. Compare any matching attributes with the result of the method call. It is an error if any are different. The result is stored in the map. Note that one of the supported attributes is the fileKey() which is essentially the inode. That means that we can use these calls directly if that is easier than using stat. Essentially this is a call to stat.

- **getLastModifiedTime(), getOwner(), getPosixFilePermissions()**. Use stat or `getAttributes()` to get the information and the inode. Put the results in the map.

- **isDirectory(), isExecutable(), isHidden(), isReadable(), isRegularFile(), isWritable()**. Same as the corresponding `File` routines.

- **isSameFile()** Stat both files and put the results in the map. Throw an exception if the stat result doesn't match the isSameFile() results.

- **isSymbolicLink()** Use lstat or `getAttributes()` to get the answer and inode. Put both in the map.

- **move**. The same as `delete()` for the source file and `exists()` == true for the destination file.

- **notExists()**. Inverse of `File.exists()`.

- **probeContentType()**. This call evidently performs something similar to the unix `file` command. Optimally we would obtain the inode of the file that was checked and check that if the file was later opened. It is not clear, however, how best to accomplish this short of re-implementing the command. Since we have already changed to use a native implementation for our own purposes, perhaps we could do that. An insecure shortcut would be to do a stat on either side of this call and check that the inodes were the same each time and use that inode for future checks. Or we could not worry about this as having the wrong type is unlikely to cause a security problem.

- **getFileAttributeView()**. Returns basic attributes about a file. This includes `fileKey()` which includes the inode of the file. These should be added to the map.

- **readSymbolicLink()**. The link itself and its destination should be added to the map. If the file is later opened, the check should ensure that the link has not changed.

- **setAttribute()**. This sets a file attribute. After performing the set, we should execute a readAttribute(), ensure that the attribute matches, and store the results in the map. Note that this also shows up in the use list.

- **size()**. Use stat to get the size and fill the information in the map. Of course, it might be reasonable for a file size to change. Perhaps this should only ensure that the file itself doesn't change.

Read/Write check methods. Reading or writing a file creates some implicit checks. Any file write creates a file that is both expected to be present and have the same content as was written to it. For example, a script could be written out and then executed. thus for any write, the inode and ctime must match on any later use. Checking ctime ensures that the contents of the file have not changed. Files reads don't seem to imply as many checks. There is no reason to believe that a file that is read twice should contain the same data. Reading a file does imply that the file exists (in the same way as the explicit `File.exists()`.

Use Methods:

- **Runtime.exec(), ProcessBuilder.start()**. Executing a program uses both the command itself and any file arguments to the command. Neither is easy to check. There is an `fexecve()` call that executes a program that is specified by its file descriptor. This could be used to make any checks with fstat. This would require, however, modifying the native library used to implement exec to use `fexecve()` or re-implementing the various `exec()` calls. If, we are instrumenting Runtime itself, it might be possible to just exchange the native call normally made to `execve` to our own that would call `fexecve`.

  There doesn't appear to be any good way to check file based arguments to the command. Those are opened by the specified program which we can't easily modify (unless we added binary instrumentation).

- **File.delete(), deleteOnExit()**. Optimally, we would check any delete to ensure that the file being deleted matched any checks previously done on that file. But there doesn't seem to be any way to do this. It does not seem possible to delete a file through its file descriptor. However, this may not be a significant hole. The main problem with delete happens if a program is deleting all of the files in a directory. In that case, it is the directory that gets changed between the check and the use. It is not clear what sort of check would occur on a filename itself. Particularly because deleting a link will just delete the link and not what it points to.

- **setAttribute()**. This should be treated just like a write. We need to open the file, perform any checks, and then use fchmod to make the actual change.

- **setLastModifiedTime()**. This should be treated just like a write. Open the file, perform any checks and then use `futimes()` to update the modified time.

- **setOwner()**. This should be treated just like a write. Open the file, perform any checks and then use `fchown()` to update the owner. The only issue here is that 'owners' and

'groups' are handled in a special Java way that would have to be translated back to unix before making the call. For simplicity the shortcut of using stat to check that the file matches before call `setOwner()` may be better. Another check that may make sense here is to ensure that the owner is not specified with tainted data.

- **setPosixFilePermissions()**. Same as setAttribute().

- **FileWriter.init(), FileOutputStream.init(), Files.newBufferedWriter(), Files.newOutputStream()**. Each of these opens a file for writing, writing over any existing data in the file. If the file has previously been checked to not exist, the open must be changed to ensure that it does not overwrite an existing file. This can be done by using the static Files methods to create a stream and passing the `CREATE_NEW` option. That ensure that the open will fail if the file already exists. Otherwise the file can be opened in a non-destructive fashion (by opening it as a random access file) and any additional checks performed using fstat on the open file.

- **File.write()**. These should be treated in the same way as the output streams above. They will have to be re-implemented using the output streams.

- **FileReader.init(), FileInputStream.init(), Files.newBufferedReader(), Files.newInputStream()**. Each of these opens a file for reading. Since the read is non-destructive, the file can be opened in the normal fashion and then any checks can be performed with fstat.

- **Files.readAllBytes(), Files.readAllLines()**. These should be treated the same was as the various input streams above. They will have to be re-implemented using the input streams.

- Files.newByteChannel(). These open random access files. Since the open is non-destructive, the file can be opened as normal and then any checks can be performed with fstat. As with the write calls above, if the file has been checked to not exist, the file should be opened with the `CREATE_NEW` option.

- **Files.newDirectoryStream()**. These calls need to be checked to ensure that the directory being read matches any previous checks on that directory. This can be done by converting the returned `DirectoryStream` to `SecureDirectoryStream` and then using the `FileAttributeView` for the directory. The `FileAttributeView` can return fstat information on the directory and the `fileKey()` can be used to extract the inode.

- **File.List(), listFiles()**. As with `newDirectoryStream()`, we need to check to ensure that the directory being read matches any previous checks on that directory. We can do this by re-implementing these calls by using `Files.newDirectoryStream()` and implementing the same checks as are specified there.

- **Files.copy()**. Like File.List(), we need to re-implement these and perform the check on the file open. If we instrument `Files` ourselves, the `open()` within the `copy()` method should get checked like any other open.

- **Files.walkFileTree()**. This is similar to newDirectoryStream() except that it can read multiple levels of directories. Presuming that walkFileTree() is implemented using newDirectoryStream and that we instrument Files, this should work as is.

3.6 Related Work

There has been a fair amount of work in the area of taint tracking in Java. Early work only tracked taint at a string level and was not able to track taint information for each character (Chess & West, 2008) (Halder, Chandra, & Franz, 2005). Because requests are usually built by concatenating strings together, string level taint is not sufficient to address most injection or tainted data vulnerabilities.

More recent work has extended taint tracking to each character within a string. Chin and Wagner (Chin & Wagner, 2009) track taint by providing an alternative implementation of the *String*, *StringBuilder*, and *StringBuffer* classes. Methods within those classes that create and/or modify strings are instrumented to track character-level taint. Because the standard *String* class is modified, this approach only works with IBM JDK 6. More common JDKs (OpenJDK) do not allow the *String* class to be modified. Halfond et al. (Halfond, Orso, & Manolios, 2008) use byte code rewriting to convert instances of *String*, *StringBuilder*, and *StringBuffer* to instrumented versions. They require a custom *rt.jar* library because they modify *String* to not be final.

Both of these implementations are limited to the string manipulations that use the *String*, *StringBuilder*, and *StringBuffer* classes. While this covers many common cases, characters can be manipulated in a number of ways that are not covered by these approaches, as discussed below.

Character and byte arrays. Arrays are also used to manipulate strings. This occurs not only in user code, but within common Java I/O classes (such as *ByteArrayInputStream*, *CharArrayReader* in *java.io* and *ByteBuffer* and *CharBuffer* in *java.nio*) and the character encoding classes. VIBRANCE maintains meta-data over all array operations.

Character primitives. Individual characters are manipulated both in conjunction with arrays and the various *String* objects. Some common operations that operate at the level of characters are character encoding, tokenization, sanitization routines and regular expression evaluation. VIBRANCE's character heuristic is very effective at tracking the character level manipulations that occur in the JDK and complex applications.

Intern and equality. Strings in Java can be interned (multiple strings that contain the same content can be combined into a single reference). This both saves space and allows object identity comparisons (implemented with ==) to return true (normal use of *String.equals()* would be needed. If meta-data is included when calculating intern, then strings that are otherwise equal will not compare as equal (using ==). This will lead to problems with back compatibility. If meta-data is not included, then the differing meta-data will be lost. VIBRANCE addresses this problem by only sharing references when the meta-data matches. To handle cases where meta-data does not match, VIBRANCE overrides the comparison operator on objects to check for interned strings whose contents match, but whose meta-data does not match.

Reflection. Chin and Wagner's system works with reflection (because they replace the existing libraries), but Halfond's does not. VIBRANCE fully supports reflection by adding necessary instrumentation to any reflective calls.

VIBRANCE's complete implementation enables it to handle complex real world programs that cannot be handled by previous systems. VIBRANCE has been tested on Apache Tomcat, Daikon, and OpenCMS and correctly maintains taint on each.

Taint tracking has been previously used to detect injection attacks. Xu et al. (Xu, Bhatkar, & Sekar, 2006) developed a system for C programs that provided protection against directory traversal, SQL injection, and command injection attacks. Halfond et al. (Halfond, Orso, & Manolios, 2008) system detected SQL injection attacks in Java.

VIBRANCE is, however, the first system we know of to provide comprehensive protection against tainted data attacks in Java. VIBRANCE has comprehensive protection against SQL injection, command injection, XPath/XQuery injection, relative/absolute path traversal, path equivalence, upload of dangerous files, reflection, redirection and loop bounds. VIBRANCE's backstop protection of unspecified socket communications also provides a unique additional check for unusual and unforeseen tainted data/injection attacks.

Other systems have been developed to provide recovery from errors. Sidiriglou's rescue point system (Sidiroglou, Ladan, & Keromytis, 2007) retrofits error handling capabilities into existing C programs. When unhandled errors occur it attempts to follow the path taken by errors that are handled. To discover rescue points, applications are profiled and monitored during tests that bombard the program with bad/random inputs.

In contrast, VIBRANCE's server protections do not require any test runs, they are completely discovered statically. They are also based on the detection of server loops guaranteeing that the error recovery is to the correct location. VIBRANCE also adds timeouts to socket communications to ensure that code does not hang waiting for remote input.

4 RANDOMIZATION

Injection attacks add keywords (and separators etc.) to the tainted portions of the string, thus causing the structure of the overall expression to change. Keywords are well known, so attackers can easily use them. However, if keywords were unknown to the attacker, the attacker could not inject them. In order to make keywords unknown to the attacker, they can be randomized in a different way each time the application is run. This is related to ASLR (Address Space Layout Randomization), but is at a higher level. A sufficiently large randomization space makes it virtually impossible for attackers to guess the randomized keywords or to try all possibilities by brute force.

This keyword randomization approach is carried out by instrumenting the application bytecode as explained below. This instrumentation takes place at the same time as the instrumentation for dynamic tracking (Section 3) and confinement (Section 4), but it is logically separate. In fact, randomization is an independent line of defense. VIBRANCE provides command line options to turn randomization on or off.

The current version of VIBRANCE uses this randomization approach for SQL keywords only. Future versions of the tool may use the approach on keywords of other injectable languages.

4.1 Instrumentation

VIBRANCE can apply randomization dynamically (using the Java agent infrastructure) or statically (transforming the whole application code, producing a hardened version of the application). The dynamic approach handles code that may not be available in advance (e.g. class files retrieved remotely, or even created on the fly) but has a startup performance overhead due to the instrumentation being applied to each class as it is loaded. The static approach to instrumentation, on the other hand, has no such overhead (because the class files have been already transformed) but requires all the class files to be available in advance.

The randomization instrumentation changes the value of string constants in class files; this is how it modifies query templates, replacing standard SQL keywords with randomized keywords. The randomization instrumentation also replaces certain method calls in class files with calls to methods that we have written. These method call replacements allow us to insert our own code between the program and the Java libraries, which is done for two reasons. First, it serves as a transparent query verification layer, where we can decide whether a given query string contains an injection or not. Second, it allows us to de-randomize strings in places where automated keyword randomization has mutated a string used outside of the context of a database query.

4.1.1 Bytecode Manipulation Tool

We use the ASM bytecode manipulation tool from the OW2 consortium to implement our bytecode modifications. ASM reads the raw bytecode, and provides a simple interface for

iterating over each instruction in each method in each class file. It also provides an easy interface for inserting and removing instructions from a method's bytecode.

4.1.2 Premain

In the dynamic instrumentation approach, the Java agent infrastructure gives us the ability to execute some code before the target program's `main` method begins. This allows us to register our bytecode transformers and specify which classes to transform. In some cases, our code depends on classes we would like to transform, which means they are loaded before our transformer is registered with the Java agent. In these cases, we use the `Instrumentation.retransformClasses` method to ask the infrastructure to reload these classes, giving our transformer a chance to modify their code.

4.2 String Mutation

Whenever a Java program uses a string constant, its corresponding bytecode contains an `ldc` or `ldc_w` (load-constant or load-constant-wide) instruction, which pushes a reference to the string constant onto the stack. For example, the compiled code for the statement in Figure 2 would contain the three instructions in Figure 3. Note that the string data are not contained directly within the instruction. Rather, the instruction contains an index into the constant table of the class file.

```
query = "SELECT * FROM users WHERE name = '" + name +
        "' AND password = '" + password + "'";
```

Figure 2: Program fragment using string constants

```
ldc #1; //String SELECT * FROM users WHERE name = '
...
ldc #2; //String ' AND password = '
...
ldc #3; //String '
```

Figure 3: Compiled bytecode referencing string constants

As each class is loaded, our instrumenter examines all of the `ldc` and `ldc_w` instructions in the bytecode. When it finds one that loads a string constant, it mutates the value, replacing any SQL keywords within the string with randomized versions. It adds the mutated string to the class's constant table, and updates the `ldc` or `ldc_w` instruction's operand. The bytecode from Figure 3 might be transformed into the bytecode in Figure 4. This transformation does not introduce additional instructions and thus incurs no overhead when the mutated application is run.

```
ldc #7; //String SELECT123 * FROM123 users WHERE123 name = '
...
ldc #8; //String ' AND123 password = '
...
ldc #9; //String '
```

Figure 4: Compiled bytecode with mutated string constants

4.2.1 Keyword Randomization

Our approach to mutation appends a random string of digits to the end of each SQL keyword. It has the disadvantage that it changes the final length of the string constant, as well as the positions of data within the string. For example, in Figure 3, string #1 is originally 34 characters long, but after the mutation in Figure 4, string #7 has 43 characters. Additionally the keyword `FROM` originally extended from positions 9 to 12 in the string, but after mutation it extends from positions 12 to 18. These changes can cause challenges in maintaining program correctness, as we will discuss later.

A different approach to mutation would be to replace each keyword with a same-length string of random characters. This would have the advantage of maintaining overall string length and data positions, but we chose not to use this approach because we wanted to be able to make the possibility of collisions arbitrarily small. For example, consider the keyword `AND`. In order to mutate this keyword, we have to choose a random three-character string of characters to replace it. Imagine that we chose the replacement string `ZMC`. If a legitimate user enters his password, `baZMC8tW`, then the query verification layer will produce something like Figure 5 to send to the database. Note that the user's password has been garbled, and as a result he will not be able to log in.

```
RDKDBS id EQNL users VIDQD name = 'john' ZMC password = 'baZMC8tW';

SELECT id FROM users WHERE name = 'john' AND password = 'baAND8tW';
```

Figure 5: Randomized and de-randomized query with an alternative approach

It might seem that we could reduce the probability of collision by trying to pick character combinations that are unlikely to occur in a normal program. However, it may be very difficult to pick such strings, as we can see from the password example. Additionally, to maintain security, we have to change the replacement keywords regularly; possibly even as often as every day, or every time the program is loaded. Each time we do so, it becomes more likely that we will cause a collision.

Our randomization solution, on the other hand, always allows us to reduce the chance of a collision. Clearly, the chances that we will encounter another string that collides with a randomized keyword decrease as the randomized keyword gets longer. Even if we only use

digits as the key character set, we can always reduce the probability of a collision by making the key longer. For example, it may be likely that the string `123` will occur in other parts of a program, but it is much less likely that the string `829357038497025` will occur elsewhere. By allowing ourselves flexibility on string length, we gain the ability to make the probability of a collision arbitrarily small, even if we choose new randomization keys daily.

The current implementation of VIBRANCE randomizes each SQL keyword by appending a randomization key consisting of 10 ASCII letters (upper case or lower case) or digits. For example, `SELECT` could become `SELECTa2831jfy6`. To minimize the possibility that an attacker could generate the key by chance, we use a large space consisting of 62^{10} (i.e., over 800 quadrillion) possible keys. This corresponds to about 60 bits, which is small for cryptographic keys, whose threat model is offline brute force search. However, VIBRANCE's keys have a different threat model, namely an attacker attempting injections over the network, whose latency limits the rate at which keys can be tried. Nonetheless, VIBRANCE's key length is configurable and could be easily increased. We have run experiments and found that increasing the key by 10 characters increases the overhead by only about 0.73%.

A different randomization key is chosen by VIBRANCE every time that an application is instrumented (dynamically or statically). For each such instrumentation instance, the same randomization key is applied to every SQL keyword in the string constants in the application.

4.2.2 Keyword Matching and Replacement

Our string mutation implementation is based on regular expressions. In the instrumenter, we construct a regular expression that matches any SQL keyword, as long as it is a whole word. For example, we match the characters `OR` in `"a OR b"` but not in `"MORK FROM ORK"`. Our transformer uses ASM to scan each instruction in each class file. Each time it encounters an `ldc` or `ldc_w` instruction, it matches it with this regular expression. Any matches are replaced with the matched text, followed by the randomization key.

The set of tokens that VIBRANCE regards as SQL keywords is easily configurable. The current default configuration protects against injection of standard SQL as well as non-standard SQL extensions for popular databases. Since SQL keywords are case-insensitive, the VIBRANCE tokenizer is case-insensitive.

4.3 Method Call Replacement

Whenever a Java program calls a method on an object, it contains instructions that load that object and all of the method arguments onto the stack, followed by an `invoke`… instruction. The exact method call instruction depends on the kind of method called. For example, methods specified in a regular class definition are called using the `invokevirtual` instruction, whereas methods specified in an interface definition are called using the `invokeinterface` instruction. Figure 6 shows the bytecode for a JDBC query execution method (`s.execute ("SELECT * FROM users");`).

```
aload 2 //Statement s
ldc #14; //String SELECT * FROM users
invokeinterface #15, 2; //InterfaceMethod
                java/sql/Statement.execute:(Ljava/lang/String;)Z
```

Figure 6: Bytecode fragment calling `Statement.execute` method

Notice that this code first loads the `Statement` instance that is the target of the `execute` method, followed by the `String` argument to the method. For static methods, there is no target object to load onto the stack, so a static method call simply loads the arguments before executing the `invokestatic` instruction.

We take advantage of this difference to easily replace method calls in the compiled code with calls to our own methods. For each method we want to replace, we define a static method that takes, as its first argument, an instance of the class whose method we are replacing. The static method's second argument is the same as the replaced method's first argument, the third argument is the same as the replaced method's second argument, and so on. This way, when our instrumenter encounters an `invokevirtual` or `invokeinterface` instruction referring to one of the methods we wish to replace, it simply replaces that instruction with an `invokestatic` instruction referring to our static replacement method.

4.3.1 Instrumentation Annotations

Unlike the string mutation pass, which examines and potentially modifies every `ldc` and `ldc_w` instruction in each class file, we only want to replace certain `invoke`… instructions. To make it easier to determine which method calls to replace, we devised a system of annotations with which we mark our replacement methods.

When we want to replace calls to methods in a particular class or interface, we create an instrumentation class to contain them. For example, to replace methods in `java.sql.Statement`, we define a new class, `StatementInstrumentation`. We apply the `@InstrumentationClass` annotation to `StatementInstrumentation`, as shown in Figure 7. This annotation requires the name of the class or interface owning the original method. If the original method owner is an interface, we also set `isInterface = true`, so that our transformer replaces `invokeinterface` instructions instead of `invokevirtual` instructions.

```
@InstrumentationClass(value = "java/sql/Statement",
      isInterface = true)
    public class StatementInstrumentation {

      @InstrumentationMethod
      public static boolean execute(Statement s, String sql)
          throws SQLException {

        sql = SQLRandomizer.getInstance().intervene(sql);
        Return s.execute(sql);
      }
```

Figure 7: Part of `StatementInstrumentation`

In `StatementInstrumentation`, we implement methods designed to replace calls to the `Statement` methods. These replacement methods are all static, but for non-static calls to `Statement`, each replacement method takes an instance of `Statement` as its first argument, followed by the normal parameters for that call. We mark these methods using the `@InstrumentationMethod` annotation. Using this annotation, we can specify everything about the target `invoke`… instruction we would like to modify. This includes the type of `invoke`… instruction, the name of the target method, and its type signature. By default, we assume that it is invoked with `invokeinterface` or `invokevirtual`, depending on the value of `isInterface` in the `@InstrumentationClass` annotation. The default name is the same as the replacement method's, and the default type signature depends on the invocation instruction. If we are replacing an `invokeinterface` or `invokevirtual` instruction, the parameters are the same as the replacement method's without the first parameter. If we are replacing an `invokestatic` method, then the targeted type signature is the same as the replacement method's.

4.3.2 Replacement

Our transformer uses reflection to scan all of the instrumentation classes' annotations, producing a `Map` from target method invocations to the instrumentation invocations that should replace them. Then, as classes are loaded, the transformer scans their bytecode. When it finds a method invocation, it checks whether it is a call to a targeted methods. If it is, then the transformer replaces that invocation instruction with a call to the instrumentation method.

4.4 Instrumentation Methods

We replace method calls in the target program for two reasons. First, prior to SQL query execution, we need to insert the logic that determines whether the provided query text might contain an injection attack. Secondly, in other places where strings are used, we may need to derandomize the string data or otherwise correct the operation of the replaced methods, so that the program continues to work as expected on benign inputs.

4.4.1 Query Verification

The JDBC (Java Database Connectivity) interface is the API most Java programs use to interact with a database. Using JDBC, a program establishes a connection to its database, represented in the program by an instance of the `Connection` interface. By calling methods on this object, the program obtains `Statement` objects that represent queries it can send to the database. All of the SQL query text the program sends to the database passes through these method calls, so they are a natural place to install the transparent software layer that intercepts queries. To implement our query verification layer, we replace calls to `Statement.execute()` and `Statement.executeUpdate()`, as well as `Connection.prepareCall()` and `Connection.prepareStatement()`. Our replacement calls perform the query verification before calling the original method to send the query to the database.

Note that even though a prepared statement is not vulnerable to injections when the template is instantiated, the creation of the prepared statement itself is vulnerable to injection when the string (e.g., "`SELECT * FROM users WHERE username=? AND password=?`") is assembled from parts that are not all trusted.

Tokenizer

To implement query verification, we initially modified an existing SQL92 parser, written in Java, called JSqlParser. Unfortunately, there are many different dialects of SQL, because nearly every DBMS (database management system) has extended the language, usually to support DBMS-specific features. The plain SQL92 parser failed to recognize several syntactic structures from our initial tests.

One possible solution to this problem would be to extend the parser until it supports each individual SQL dialect completely. We found, though, that we do not necessarily need a full parser to determine whether a query string contains a potential injection attack or not. Instead, it is sufficient to tokenize the string, breaking it up into program elements such as identifiers, keywords, numbers, strings, and operators. In the example query string in Figure 8, there are eight tokens, including the keyword `FROM123`, the identifier `users`, the string `'Dave'`, and the operator `=`.

```
SELECT123 * FROM123 users WHERE123 name = 'Dave'
```

Figure 8: Example query string containing eight tokens

Verification

Once the query string is tokenized according to SQL syntax, we examine its keywords. For each one, we check whether it is a plain keyword, without the random mutation applied. This would

indicate that it did not come from one of the program's string constants. If we find such a keyword, we reject the query, reporting a database error to the program.

If the query text contains no un-mutated keywords, we remove all instances of the randomization key from the query. Since we only remove instances of our key in this step, an attacker cannot break the system by using his own random key. If he did so, his random key would not be removed, and the database itself would reject the query as invalid.

4.4.2 Program Correctness

One issue with a fully automated approach is that it cannot benefit from the careful eye of a programmer when it makes its changes to the program. When it mutates the program's strings, for example, it cannot tell whether a particular string is part of a query template or not. This indiscriminate modification will cause the help text `"Select whether you want guacamole or not"` to be changed to `"Select123 whether you want guacamole or123 not123"` (assuming that the randomization key is `123`). These modifications can cause user confusion, unexpected behavior, and program crashes. In general, we need a way to preserve program correctness even when its string data have been indiscriminately modified.

Fortunately, all operations that modify or examine string data in Java are implemented in method calls from the string API (which consists of the `String`, `StringBuffer`, and `StringBuilder` classes). Even the Java source `+` concatenation operator is compiled into a call to the `StringBuilder.append` method in Java bytecode. This allows us to use the same method-call replacement technique to insert our own code whenever we want to correct the outcome of an operation on mutated string data.

String Comparison

Many of the cases we had to correct involved comparisons. We wanted to preserve the meaning of string equality, even when strings that occurred as constants have been mutated whereas strings that came from user input have not. For example, the comparison `"ROW".equals(str)` should return `true` whenever `str` contains `"ROW"` or `"ROW123"`. To implement these semantics, we instrument calls to `String.equals`, `String.hashCode`, and `String.compareTo`. The instrumented methods pass both values to the same routine that removes the randomized key when a query verifies as safe, before calling the original comparison method.

Any classes on which our own instrumentation code depends on, do not get transformed automatically. In some cases we are able to retransform these classes so that any time they call one of the string comparison methods, our code is invoked instead. In other cases, we were unable to retransform the dependencies. For example, `HashMap` is loaded and used extensively before our transformer can start. In this case, we replace calls to `HashMap`'s methods with calls to our own methods, so that we can undo the mutations before entering `HashMap`'s logic. Other

types of `Map` and `Set` are able to use our implementations of `String.hashCode` and `String.equals` to produce correct results.

Because Java `String` objects are immutable, the string library is able to provide a faster way to test for string equality. By calling the `String.intern` method, a program can obtain a reference to a canonical ("interned") representation of the string. This reference can then be compared to other interned strings, and if these references are equal then the strings must be equal. This can provide a performance increase over standard string comparison, which must iterate over every character in the strings if they are indeed equal.

Testing references for equality does not call a method, so we cannot apply our method call replacement strategy in this case. Instead, we observe that the primary use of interned strings is for comparison. Also, there are no performance benefits to interning a string constant that is part of a query template, i.e. interning does not provide any benefits for concatenation or value interpolation. These reasons make it unlikely that an interned string constant will ever be used in a query context (user-entered strings may be interned and used, but these are not randomized anyway). So, in this case, instead of replacing the reference equality instruction, we replace calls to `String.intern`, inserting our own code that removes the random key before calling the real `intern` method.

String Length and Positions

Other string operations may also be affected by our mutations. Because we have elected to append our key to each keyword in the statement, rather than picking same-length replacements, our randomization pass will change both the overall length of string constants and the positions of characters within them. Consider the original and mutated strings in Figure 9. The original string has length 19, while the mutated string contains 25 characters. The `*` symbol has moved from (zero-indexed) position 7 to position 10. The keyword `FROM`, which originally extended from positions 9 to 12, now reads `FROM123` and extends from positions 12 to 18. All of these changes affect the correctness of string operations such as `String.length`, `String.charAt`, `String.substring`, and `String.indexOf`, as well as the correctness of operations that build on these primitives, like regular expression matching and string splitting.

```
S E L E C T   *   F R O M   u s e r s
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8

S E L E C T 1 2 3   *   F R O M 1 2 3   u s e r s
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
```

Figure 9: Original and mutated string with character positions

Because these changes affect methods that return string positions as well as methods that use string positions as parameters, it is not an unreasonable assumption that many programs will remain correct. For example, a program that uses `indexOf` to find the spaces in our mutated example string will correctly find them at positions 9, 11, and 19. If it subsequently uses `substring` to extract the text between these positions, it will correctly obtain the strings `SELECT123`, `*`, `FROM123`, and `users`. The randomized keywords obtained may be de-randomized when they are used in some other context later. Only if the program tries to find and operate on characters in the random key will it become incorrect. Note also that only string constants are randomized. All of the string-indexing operations continue to operate the same way on strings obtained as input. Many programs do not attempt to use indexing operations on their own string constants, or even on strings built by concatenating them with input. So, it may be possible to simply ignore the string length and position changes, and expect that many programs will still be able to operate properly.

Nevertheless, it may be desirable to maintain the consistency of these operations for programs as much as possible. For example, mutation might make a program's regular expressions match incorrectly—the expression `([A-Za-z]+ [A-Za-z]+)`, which matches two sequences of letters separated by a space, will no longer match the string "`FROM users`", because it has been mutated to "`FROM123 users`". In addition, programs that supply string positions as constants will no longer work properly. For example, a program which expects to retrieve `"FROM users"` by calling `substring(9, 18)` will instead receive `" * FROM12"`. (Note that the arguments to `substring` are the beginning index and the ending index, and that the ending index is not inclusive.)

For example, the code in Figure 10 adds some text to an SQL statement following the `SELECT` keyword. The length of the `SELECT` keyword is hard-coded.

```
StringBuffer sb = new StringBuffer(...);
int offset = sb.indexOf("SELECT") + 6;
sb.insert(offset, " field1, field2");
```

Figure 10: Example program that manipulates an SQL query using indices

We implemented a solution to correct these issues, making `indexOf`, `charAt`, `length`, and `substring` consistent with respect to un-mutated positions. All four receive and return position values as if they were indexed into the original string. To implement `indexOf`, `charAt`, and `length`, it is sufficient to de-mutate the string before passing it to the real API method. However, the `substring` method requires some extra work.

For `substring`, we would like to preserve the randomization of a keyword if and only if it is fully contained within the requested substring. For example, `substring(7, 15)` should return `"* FROM123 u"`, whereas `substring(7, 11)` should only return `"* FR"`, and `substring(11, 15)` should only return `"OM u"`.

The first step is to build a table describing where the keywords exist in both the un-mutated and mutated string. In our example string, this looks as in Figure 11. We build this table using the same regular expression we use when de-randomizing strings. Each time it matches a randomized keyword, we add its position in the mutated string to the table. The offsets in the original string can be calculated by subtracting the number of randomized keywords we have seen so far, multiplied by the length of the random key string (3 in this example). Keyword extents can never overlap, because only one keyword precedes each instance of the key.

Keyword	Original Start	Original End	Mutated Start	Mutated End
SELECT	0	6	0	9
FROM	9	13	12	19

Figure 11: Keyword randomization table

Once we have the table, we use it to adjust the beginning and ending index of the substring we are to retrieve. The substring's beginning position in the mutated string must be offset by the same amount as the end of the closest preceding keyword. So, we find the keyword with greatest "original end" position smaller than the substring beginning position, and adjust the substring beginning position by the difference between that keyword's "original end" and "mutated end" positions.

The end position shifts similarly. However, if the beginning of the substring falls in the middle of a keyword, the logic is not quite the same. In this case, if the substring end position coincides with that keyword's end position then we shift by the amount for the preceding keyword (as we would if the position were in the middle of this keyword). This prevents us from including the random key when a substring does not pick up the whole keyword. If the substring continues past the end of the keyword, we have to break the substring operation up into two operations, one ending at the end of the keyword, and the other beginning there. We then concatenate their results to produce the correct substring.

The call `substring(7, 15)` does not begin in the middle of a keyword, so it does not have to be broken into two operations. To adjust the beginning position, we find that `SELECT` is the closest preceding keyword. `SELECT`'s end position shifts by 3 positions, so the beginning position in the mutated string starts at 10. The end position does not coincide with the end of a keyword, so we shift it by 6, the same amount as the end position of `FROM`; producing the ending position 21. If we examine Figure 9remembering that `substring` is not inclusive of the end position, we can see that the characters from 10 to 20 represent the substring we wanted to generate, `"* FROM123 u"`.

The call `substring(7, 11)` proceeds similarly. The beginning position 7 shifts to 10 as before, but the closest preceding keyword for ending position 11 is `SELECT`, so we shift 11 by 3, to 14. The characters from 10 to 13 in the mutated string are, again, the correct substring, `"* FR"`.

The call `substring(11, 15)` does have to be broken into two operations, because its beginning position is in the middle of the keyword `FROM` (which extends from positions 9 to 13), and its end position is after the end of the keyword. The first sub-operation `substring(11, 13)` begins in the middle of `FROM` and ends at its end, so the end positions shifts according to `SELECT`, by 3 to 16. The beginning position also shifts by 3, to 14. Characters 14 and 15 of the mutated string produce the string `"OM"`. The second sub-operation `substring(13, 15)` shifts both positions by 6 using the offset from `FROM`. Positions 19 and 20 of the mutated string produce `" u"`, so the final result is `"OM u"`, as expected.

One consequence of making the operations consistent in this way is that random keys may be lost in certain cases. For example, consider a program that copies from a string to a character array by looping through it, character by character. Each time it calls `charAt`, it retrieves a single character from the de-randomized string; at the end, the character array contains only the de-randomized string. Loss of random keys may lead to false positives—our system would detect an injection where none existed. However, it is our conjecture that applications are unlikely to perform this kind of manipulations on string constants. This conjecture has been validated by all the test applications that we have tried throughout the project.

4.4.3 Output Envelope

Besides affecting program correctness, improper handling of the mutated strings can pose a security risk. If a keyword has been properly mutated, then at query verification we assume it is safe, because it must have originated in one of the program's string constants. However, if the attacker can figure out how to mutate his own keywords correctly, he can make them appear to also be safe. If the program outputs `"Select123 whether you want guacamole or123 not123"` to an attacker's display, then he has learned how to mutate the keywords `SELECT` and `OR`, and can then use them as part of future injection attacks.

By examining the Java API, we can identify method calls that can carry string data back to a user. For example, `PrintWriter.print(String)` is usually used to display output to a user. The constructor `File(String)` is used to create a reference to a file; a user may be able to access the file system and see the names of created files. It is unlikely that string data that passes through the output envelope will ever be used as query text.

4.5 Optimization via Static Analysis

If an SQL keyword in a string constant can never end up in an SQL API call, that keyword does not need to be randomized. The static analysis component of VIBRANCE finds (a sound under-approximation of) string constants that can never end up in an SQL API calls. This information is passed to the randomization component of VIBRANCE, which it uses to reduce the set of string constants that must be randomized.

4.6 Overhead

VIBRANCE has command line options that allow to run tracking/confinement only, randomization only, or both. We measured the overhead of VIBRANCE's randomization (only) on a sample of the Phase 2 T&E SQL tests, and also on OpenCMS.

We found the average overhead to be less than 5%. This means that VIBRANCE's randomization can protect applications against SQL injection (perhaps the most prevalent vulnerability in Java applications) with minimal overhead.

4.7 Related Work

Keyword randomization applies the same basic concept as ASLR (Bhatkar, Duvarney, & Sekar, 2003) to external textual communications. In both cases the attacker requires knowledge of the program internals (e.g. SQL syntax) in order to create an effective attack. ASLR makes it difficult for an attacker to call existing code by randomizing addresses. VIBRANCE randomizes SQL keywords so that injected syntax will not parse correctly.

SQL keyword randomization has been previously implemented in SQLRand (Boyd & Keromytis, 2004) and PachyRand (Locasto & Keromytis, 2005). The metadata required by SQLRand is embedded in the query strings themselves. When a programmer creates a new query template, he must modify the keywords in the template, appending a random string of digits to the end of each one. SQLRand provides a tool that applies the modification to a single query, but the programmer is responsible for making sure it is applied to each query. In Figure 12, we show how the randomized query template appears in the program's source code.

```
SELECT123 student id, class, assignment, grade
FROM123 grades
WHERE123 student id = '%s' AND123 class = '%s'
```

Figure 12: Randomized retrieval query

SQLRand uses a proxy as its transparent software layer. The programmer modifies the program so that instead of connecting to the database directly, it connects to SQLRand's proxy, which implements the database network protocol. The proxy maintains its own connection to the real database. When the program sends a query to the proxy, it parses it using a grammar that expects randomized keywords instead of standard ones. If the proxy can parse the grammar successfully, then it produces an un-randomized version and forwards it on to the database. If it cannot parse the query, then it is possible that it contains an injection attack, so the proxy simply returns an error to the program without forwarding anything to the database. In our example, when the curious student uses the attack string "`xx' OR class = 'CSCI 100`", the parser rejects it because the token `OR` without the key appended to it, doesn't have any meaning.

VIBRANCE extends this approach significantly. Most importantly, no manual changes are required to the application. Keywords are randomized automatically and de-randomized

automatically when they are visible. Also, a proxy is not necessary. VIBRANCE adds its checks to all SQL calls by modifying the application's byte code. The check on the SQL query is made before passing the query on to the database.

VIBRANCE is also database independent. SQLRand is specific to MySQL and PachyRand is specific to PostgreSQL. SQL grammars vary significantly between different implementations. There are also significant implementation-specific extensions. Rather than using a parser to validate queries, VIBRANCE performs its checks in a grammar independent fashion on the tokens of the query allowing it to support a wide range of databases and extensions.

There are several reasons to desire a more automated solution than those proposed previously. Some amount of programmer effort must be spent to make the required modifications to the program. In addition, the modified source code may be harder to read and understand, which also contributes to development costs.

Another advantage we gain from fully automating this process is assurance that all code paths leading from user input to the database are checked. With manual approaches any mistake in the program, such as un-sanitized string data or a missed data type check, leads to vulnerabilities.

Most importantly though, a major disadvantage of any solution that requires source code modification is that the application of the solution requires access to the source code. If a person or organization wishes to deploy a web application developed by a third party, they may not be able to modify query templates or ensure that the program always sanitizes string data before inserting them into a query. VIBRANCE provides a system that can automatically secure a program even without access to its source code.

5 TESTING

We have performed a significant amount of internal testing on VIBRANCE. The tool has also been tested by the STONESOUP T&E teams (MITRE in Phases 1 and 2, and TASC in Phase 3).

5.1 Juliet Test Suite

The Juliet Test Suite contains tests for numerous CWEs. These tests consist of programs without test data (e.g. input/output pairs). As such, they are suited for testing static analyzers. Thus, in order to test VIBRANCE as a whole, we had to create test data for some of the programs in the Juliet Test Suite.

5.1.1 CWE-89 (SQL Injection)

We created both benign and attack test inputs for the CWE-89 programs in the Juliet Test Suite. We ran a total of 1,316 tests. VIBRANCE detected all attacks with no false alarms.

5.1.2 CWE-78 (OS Command Injection)

We created both benign and attack test inputs for the CWE-78 programs in the Juliet Test Suite. We ran a total of 329 tests. VIBRANCE detected all attacks with no false alarms.

5.1.3 CWE-606 (Tainted Loop Bounds)

We used the CWE-606 programs in the Juliet Test Suite only to test the static analysis component of VIBRANCE. Thus, we did not create any benign or attack test inputs for these tests.

There are a total of 2,492 loops in these programs. Of those:

- 1,040 loops are not reachable (dead code).
- 417 loops are unsafe, in the sense that their bound is controlled by an external input and the program does not enforce any limit on this external input.
- 1,035 loops are safe, because either their bound is controlled by an internal value or it is controlled by an external value but the program enforces an (internal) limit on that external value.

CodeHawk generated the following results:

- The 1,040 unreachable loops are correctly recognized as unreachable.
- All 417 unsafe loops are recognized as unsafe. This means no false negatives.
- 1,002 of the 1,035 safe loop are recognized as safe. The remaining 33 are flagged as potentially unsafe. This means that the false positive rate is impressively low: 1.3% (over the total number of loops).

5.1.4 CWE-190/191 (Integer Overflow and Underflow)

We used the CWE-190 and CWE-191 programs in the Juliet Test Suite only to test the static analysis component of VIBRANCE. Thus, we did not create any benign or attack test inputs for these tests.

There are a total of 10,400 integer operations in these programs. Of those:

- 2,860 are not reachable (dead code).
- 1,950 are unsafe, in the sense that their operands are controlled by external inputs and the program does not enforce any guards.
- 5,590 are safe, because either the operands are always benign or the program enforces suitable guards.

CodeHawk generated the following results:

- The 2,860 unreachable operations are correctly recognized as unreachable.
- All 1,950 unsafe operations are recognized as unsafe. This means no false negatives.
- 5,460 of the 5,590 safe operations are recognized as safe. The remaining 130 are flagged as potentially unsafe. This means that the false positive rate is impressively low: 1.5% (over the total number of integer operations).

5.2 Daikon and Tomcat

We have tested VIBRANCE on:

- Daikon, a system developed at MIT.
- Apache Tomcat, a well-known web server and servlet container.

Both applications are about 200K of source code. The main reasons for testing these relatively large applications were:

- Ensure that VIBRANCE can process the applications, without hitting scalability problems.
- Test the efficacy and the efficiency of the dynamic tracking component of VIBRANCE.

Testing how well VIBRANCE protects these two applications from attacks was not a goal of these tests.

We found that VIBRANCE can process these two applications without hitting scalability problems. We also found that the dynamic tracking component of VIBRANCE never loses any taint status information, in the sense that no character or byte has ever an unknown taint status, but its status is always either trusted or untrusted. This was tested by examining the taint status of the output of Daikon and the variables made available to servlets in Tomcat.

5.3 DOLL Testing

MIT subcontracted Dynamic Object Language Labs (DOLL) to perform stress testing of VIBRANCE. This included unit tests, injection tests and standalone applications (e.g., OpenCMS). DOLL also verified the system on a variety of different SQL servers (MySQL,

PostgreSQL, Hibernate, etc). They created 4800 Hibernate tests, 8 command injection tests, 20,000 XQuery tests, 3 reflection tests and 6 server tests (covering both *java.io* and *java.nio*).

All these tests were successful.

5.4 STONESOUP T&E

In all of the three Phases' T&E testing, VIBRANCE stopped most attacks while altering benign functionality in only a few cases.

We believe that the VIBRANCE approach has not yet hit the limits of what can be achieved. In particular, based on our analysis of the Phase 3 T&E results, we believe that the VIBRANCE implementation could be improved to pass even more tests.

The T&E tests have also been useful for our own internal testing. For example, we used the Phase 2 T&E tests for SQL injection to measure the overhead of VIBRANCE's randomization (without tracking and confinement).

6 FUTURE WORK

6.1 Reduction of Run-Time Overhead

VIBRANCE's metadata tracking is very comprehensive and precise, and incurs acceptable overhead. We believe that we could significantly reduce the overhead by making more extensive use of static analysis information to optimize both tracking and confinement. We were not able to implement these optimizations within the available resources, but we believe that we could significantly reduce overhead with a relatively modest effort.

6.2 Protection against Cross-Site Scripting

VIBRANCE's tracking and confinement could be extended to protect applications against cross-site scripting, which is one of the most prevalent and dangerous attacks on Java applications. Since the taint tracking infrastructure is already in place, only an extension of the confinement mechanism is needed. We believe that this could be achieved with a relatively modest effort.

6.3 Extending Randomization beyond SQL

VIBRANCE's randomization approach could be extended, with a relatively modest effort, to other injectable languages. This can be achieved by (1) extending the set of keywords (that are randomized) with the ones of other languages and (2) adding a verification layer to the API calls that process the injectable language (to check that every keyword is properly randomized).

REFERENCES

ASM. (n.d.). (OW2 Consortium) Retrieved from asm.ow2.org

Bacon, D. F., & Sweeney, P. F. (1996). Fast Static Analysis of C++ Virtual Function Calls. In L. Anderson, & J. Coplien (Ed.), *Proc. SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA)* (pp. 324-341). ACM.

Bhatkar, E., Duvarney, D. C., & Sekar, R. (2003). Address obfuscation: an efficient approach to combat a broad range of memory error exploits. *12th USENIX Security Symposium.*

Bodden, E., Sewe, A., Sinchek, J., Oueslati, H., & Mezini, M. (2011). Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In R. N. Taylor, H. Gall, & N. Medvidovic (Ed.), *Proc. 33rd Int.Conf. on Software Engineering (ICSE)* (pp. 241-250). ACM.

Boyd, S. W., & Keromytis, A. D. (2004). Sqlrand: Preventing SQL injection attacks. *2nd Applied Cryptography and Network Security (ACNS) Conference.*

Chess, B., & West, J. (2008). *Dynamic Taint Propagation: Finding Vulnerabilities without Attacking.* Information Security Technical Report 13.

Chin, E., & Wagner, D. (2009). Efficient Character-level Taint Tracking for Java. *ACM Workshop on Secure Web Services (SWS).*

Cousot, P., & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In R. M. Graham, M. A. Harrison, & R. Sethi (Ed.), *Fourth ACM Symposium on Principles of Programming Languages (POPL)* (pp. 238-252). ACM.

Cousot, P., & Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In A. V. Aho, S. N. Zilles, & B. K. Rosen (Ed.), *Sixth ACM Symposium on Principles of Programming Languages (POPL)* (pp. 269-282). ACM.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Mine, A., Monniaux, D., et al. (2005). The ASTREE Analyzer. In S. Sagiv (Ed.), *14th European Symposium on Programming (ESOP). 3444*, pp. 21-30. Springer.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., & Zadeck, K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. (TOPLAS) , 13* (4), 451-490.

Dean, J., Grove, D., & Chambers, C. (1995). Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In W. G. Olthoff (Ed.), *ECOOP. LNCS 952*, pp. 77-101. Springer.

DeFouw, G., Grove, D., & Chambers, C. (1998). Fast interprocedural class analysis. In D. B. MacQueen, & L. Cardelli (Ed.), *Proc. 25th ACM Symposium on Priniples of Programming Languages (POPL)* (pp. 222-236). ACM.

Demange, D., Jensen, T. P., & Pichardie, D. (2010). A provably correct stackless intermediate representation for Java byte code. In K. Ueda (Ed.), *8th Asian Symposium on Programming Languages and Systems (APLAS). 6461*, pp. 97-113. Springer.

Grove, D., & Chambers, C. (2001). A framework for call graph construction algorithms. *ACM Transactions Program. Lang. Syst. (TOPLAS) , 23* (6), 685-746.

Halder, V., Chandra, D., & Franz, M. (2005). Dynamic Taint Propagation in Java. *Annual Computer Security Applications Conference (ACSAC).*

Halfond, W., Orso, A., & Manolios, P. (2008). WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering (TSE)* .
Lhotak, O. (2006). *Program Analysis Using Binary Decision Diagrams.* Phd Thesis, McGill University, School of Computer Science.
Lhotak, O., & Hendren, L. J. (2003). Scaling Java Points-to Analysis Using SPARK. In G. Hedin (Ed.), *12th Int. Conf. on Compiler Construction (CC). 2622*, pp. 153-169. Springer.
Livshits, B., & Lam, M. (2005). Finding Security Vulnerabilities in Java Applications with Static Analysis. *Usenix Security Symposium.* Baltimore.
Livshits, B., Whaley, J., & Lam, M. (2005). Reflection analysis for Java. In K. Yi (Ed.), *Third Asian Symposium on Programming Languages and Systems* (pp. 139-160). Springer.
Locasto, M., & Keromytis, A. D. (2005). *PachyRand: Randomization for the PostgreSQL JDBC Driver (Extended Abstract).* Columbia University.
Logozzo, F., & Fahndrich, M. F. (2008). On the relative completeness of bytecode analysis versus source code analysis. In L. J. Hendren (Ed.), *17th Int. Conf. on Compiler Construction (CC). 4959*, pp. 197-212. Springer.
Sidiroglou, S., Ladan, O., & Keromytis, A. D. (2007). Using rescue points to navigate software recovery. *IEEE Symposium on Security and Privacy.*
Sundaresan, V., Hendren, L. J., Razafimahefa, C., Vallee-Rai, R., Lam, P., Gagnon, E., et al. (2000). Practical virtual method call resolution for Java. In M. Rosson, & D. Lea (Ed.), *Proc. SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA)* (pp. 264-280). ACM.
Tip, F., & Palsberg, J. (2000). Scalable propagation-based call graph construction algorithms. In M. Rosson, & D. Lea (Ed.), *Proc. ACM SIGPLAN Conf. on Object-oriented Programming Systems, Languages and Applications (OOPSLA).* ACM.
Venet, A. (2007). A practical approach to formal software verification by static analysis. In P. Black (Ed.), *Static Analysis Workshop.*
Xu, W., Bhatkar, S., & Sekar, R. (2006). Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. *15th USENIX Security Symposium.*

APPENDIX A: TAINTED DATA / INPUT VALIDATION

The following sections describe the CWEs listed under the weakness class 'Tainted Data / Input Validation'. CWEs that are children of the name CWEs are placed in sections under their parents.

A.1 CWE-15: External Control of System or Configuration Setting

The CWE says:
One or more system settings or configuration elements can be externally controlled by a user. Allowing external control of system settings can disrupt service or cause an application to behave in unexpected, and potentially malicious ways.

There are two examples. One sets the hostid and the second sets the catalog of a database (`setCatalog()`). There are no references or child CWEs.

This refers to calling powerful system configuration methods using untrusted data. The problem here is identifying the methods that might do dangerous things. We address this in two ways. First, we identify a list of possibly dangerous methods by hand and add appropriate checks. Second, we perform conservative checking on otherwise unchecked native calls and socket communications (see backstop confinement in Section 4).

Some of the calls we address are:

- `System.getProperty(String key)`,
  `System.getProperty(String key, String default)`
  There are no checks here, but the property value returned for an untrusted key is set to be untrusted.
- `System.MapLIbraryName (String libname)`
  This propagates taint to its returned value.
- `Connection.setCatalog (String catalog)`
- `Connection.setClientInfo (Properties properties)`
- `Connection.setClientInfo (String name, String value)`
- `Connection.setSavePoint (String name)`
- `Connection.setSchema (String schema)`
- `Connection.setTypeMap (Map<String,Class<? » map)`
- `Class.forName (String className)`, `Class.forName (String name, boolean init, ClassLoader loader)`
- `Class.getDeclaredField (String name)`,
  `getField (String name)`,
  `getDeclaredMethod (String name, Class<? >... parameters)`,
  `getMethod (String name, Class<? >... parameters)`
- `Class.getResource (String name)`,
  `getResourceAsStream (String name)`
- `ClassLoader.defineClass()`
- `ClassLoader.definePackage()`
- `ClassLoader.findClass()`
- `ClassLoader.findLibrary()`

- `ClassLoader.findLoadedClass()`
- `ClassLoader.getResource (String name)`,
  `getResourceAsStream (String name)`,
  `getSystemResource (String name)`,
  `getSystemResourceAsStream (String name)`,
  `getSystemResources (String name)`
- `Classloader.getPackage()`
- `Classloader.loadClass()`
- `Classloader.resolveClass()`
- `Classloader.setSigners()`
- `setClassAssertionStatus (String classname, boolean enabled)`
  These are disallowed if any of the *String* or byte array parameters contain any tainted data.

A.2 CWE-23: Relative Path Traversal

The CWE says:
The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize sequences such as ".." that can resolve to a location that is outside of that directory.
This allows attackers to traverse the file system to access files or directories that are outside of the restricted directory.

There are two examples. Each is a relatively straightforward use of "..” to reference a parent directory. There is one reference that is actually a reference to a list of references by OWASP. There are 12 child CWEs. Each of the child CWEs provides a variant on the original case.

Checking for this is very straightforward. References to parent directories in untrusted data should not be allowed. This is specified in the regular expression check in the configuration file. There are a couple of options for continued execution:

- Remove the references to parent directories.
- Throw an exception. This works well with our server checks—the current operation will not succeed, but all others should work correctly.

As currently configured, VIBRANCE throws an exception.

This check is performed on the constructor for `File` and on all other calls that accept filenames of type `String`.

A.3 CWE-36: Absolute Path Traversal

The CWE says:
The software uses external input to construct a pathname that should be within a restricted directory, but it does not properly neutralize absolute path sequences such as "/abs/path" that can resolve to a location that is outside of that directory.
This allows attackers to traverse the file system to access files or directories that are outside of the restricted directory.

There is one example. It is an untrusted filename with no checks for validity. There are no references. There are 4 child CWEs that refer to various ways that absolute pathnames can be made. All of them are Windows-specific except for CWE-37 which describes files that start with forward slash.

This is also easily handled by checks for filenames that begin with an untrusted slash. This can easily be specified in the filename check in the configuration file.

The same methods as listed in the CWE-23 (Relative Path Traversal) are applicable here. The options for continued execution are also the same as for CWE-23.

A.4 CWE-41: Improper Resolution of Path Equivalence

The CWE says:
The system or application is vulnerable to file system contents disclosure through path equivalence. Path equivalence involves the use of special characters in file and directory names. The associated manipulations are intended to generate multiple names for the same object. Path equivalence is usually employed in order to circumvent access controls expressed using an incomplete set of file name or file path representations. This is different from path traversal, wherein the manipulations are performed to generate a name for a different object.

There are no examples or references. There are 15 child CWEs with a variety of specific filename issues. The child CWEs have a number of examples and references.

This problem seems to arise when code has a blacklist of checks. If a pathname can be specified that evades the blacklist (perhaps by adding a trailing blank) but still works in the file system, files on the blacklist can still be accessed. It is also possible that the program can do later cleansing modifications that will remove the odd characters (validate-before-cleanse). See CWE-180.

VIBRANCE checks filenames for a normal form when *File* objects are created or when strings are used for filename operations (e.g., *open*). VIBRANCE can either terminate the application, modify the offending characters, or take any of the other standard error actions (based on config file settings). Note that filename checks are also performed against filenames used as command arguments. The same checks will be used in both places.

Filenames are checked against regular expressions in the config file for validity. There is a white list check (for a valid filename) and a number of black list checks for specific problems. If repairing the filename is chosen, the characters in any matching black list regular expressions can be modified to a character specified in the configuration file (probably underscore). If there are not black list matches, but the filename doesn't pass the white list check, a repair other than fixing the filename is used.

Some of the black list checks are:

- Trailing characters (CWE-42/43) `[.+ \0/\\]+$`
- Multiple Internal Dots (CWE-45) `[.][.]+`

- Multiple internal slash (CWE 51) `//+`
- Multiple Leading slash (CWE 50) `\^//+`
- Leading space (CWE 47) `\^ +}`
- Internal Whitespace (CWE 48) `+`
- Multiple internal backslash (CWE 53) `\\\\+`
- Single dot directory (CWE 55) `./`
- Wildcards (CWE 56) `[*]`

In addition to the black list checks, we have implemented white list checks that limit filenames to normal filename characters (where normal is defined as the characters that are commonly used on a Unix system). This will prevent filenames from including characters that are meaningful to a shell such as semicolon or vertical bar (pipe).

The same methods as listed in the CWE-23 (Relative Path Traversal) are applicable here. The options for continued execution are also the same as for CWE-23.

A.5 CWE-73: External Control of File Name of Path

The CWE says:
The software allows user input to control or influence paths or file names that are used in filesystem operations.
This could allow an attacker to access or modify system files or other files that are critical to the application. Path manipulation errors occur when the following two conditions are met:
1. *An attacker can specify a path used in an operation on the filesystem.*
2. *By specifying the resource, the attacker gains a capability that would not otherwise be permitted.*

For example, the program may give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

There are two examples and two references. Each seems to rely on one of the specific vulnerabilities described under CWEs 23, 36, and 41. Most issues seem covered there.

Since servers have a specific security policy for files that they can manipulate, the security manager for servers covers all manipulations (regardless of whether or not they contain untrusted characters).

A.6 CWE-99: Improper Control of Resource Identifiers

The CWE says:
The software receives input from an upstream component, but it does not restrict or incorrectly restricts the input before it is used as an identifier for a resource that may be outside the intended sphere of control.
This may enable an attacker to access or modify otherwise protected system resources.

The weakness class definition says:

It should be noted that CWE-73 and CWE-99 are related. CWE-73 focuses on file names and paths. CWE-99 is included in the list to cover other possible resource types. For example: externally-provided process IDs.

There are two examples and no references. Both of the examples concern filenames. The weakness class definition mentions process IDs, but Java doesn't provide any APIs that manipulate process IDs.

URLs and URIs are both similar to filenames. VIBRANCE checks these in the same basic way that filenames are checked (no absolute names, no relative names, no obscure characters, etc.). Also, untrusted JavaScript URLs will not be allowed.

Any additional resources will be covered by our conservative approach to untrusted data in native/sockets (see CWE-15).

A.7 CWE-228: Improper Handling of Syntactically Invalid Structure

The CWE says:
The product does not handle or incorrectly handles input that is not syntactically well-formed with respect to the associated specification.
If the specification requires that an input value should be delimited with the "[" and "]" square brackets, then any input that does not follow this specification would be syntactically invalid. If the input between the brackets is expected to be a number, but the letters "aaa" are provided, then the input is syntactically invalid.

There are *no* examples or other information. However, it is the parent of several other CWEs (229, 233, 237, and 241). These are discussed below. Note that CWE-241 is explicitly listed in the weakness class document.

A.7.1 CWE-229: Improper Handling of Values

The CWE says:
Weaknesses in this category are related to missing or incorrect handling of values that are associated with parameters, fields, or arguments.

There is no further information here, but is the parent of CWEs 230, 231, and 232.

CWE-230: Improper Handling of Missing Values

The CWE says:
The software does not handle or incorrectly handles when a parameter, field, or argument name is specified, but the associated value is missing, i.e. it is empty, blank, or null.

This contains 4 examples. Each example is server-related. Three of these refer to error messages that occur when certain parameters are blank. Those error messages leak certain possibly sensitive information such as the IP address of a host or the installation path of a program. At

this point VIBRANCE does not check error messages. This will be addressed in phase 2 under error handling.

The other example is a crash. This is handled by our overall server approach. The exception handling blocks should ensure that processing of other commands is not perturbed.

CWE-231 Improper Handling of Extra Values

The CWE says:
The software does not handle or incorrectly handles when more values are specified than expected.
This typically occurs in situations where only one value is expected.

There are no children, examples, or references.

CWE-232 Improper Handling of Undefined Values

The CWE says:
The software does not handle or incorrectly handles when a value is not defined or supported for the associated parameter, field, or argument name.

There is an example here of a Java servlet that crashes if a requested parameter is not present. There is also a reference to a crash in Windows 95/98 when a file sharing service returns an unknown driver type.

Both of these are handled by our server approach.

A.7.2 CWE-233: Parameter Problems

The CWE says:
Weaknesses in this category are related to improper handling of parameters, fields, or arguments.

There is no other information, but there are 3 other child CWEs (234, 235, and 236).

CWE-234: Failure to Handle Missing Parameter

The CWE says:
If too few arguments are sent to a function, the function will still pop the expected number of arguments from the stack. Potentially, a variable number of arguments could be exhausted in a function as well.

The examples (and description) all have to do with C/C++ varargs methods. However, there are a number of references to cases where arguments are missing from data communicated from one program to another (e.g., content-length fields in HTTP). There are 15 references. Of these 15, 11 (2004-0276, 2002-1488, 2002-1169, 2003-0239, 2002-1023, 2002-1236, 2003-0422, 2002-

1531, 2002-1077, 2002-1358, 2003-0477) result in a crash because a parameter is missing. These should be caught be our server approach. 3 (2000-0521, 2001-0590, 2002-0107) allow source code to be read when a url request doesn't include an HTTP protocol specification. This is an odd problem that does not feel that general. 1 (2002-0596) leaks path information. This is similar to CWE-230.

CWE-235: Improper Handling of Exra Parameters

The CWE says:
The software does not handle or incorrectly handles when a particular parameter, field, or argument name is specified two or more times.
This typically occurs in situations when only one element is expected to be specified.

This seems very similar to CWE-230 (missing values rather than missing parameters). There are no examples, but there is one reference (2003-1014). This refers to MIME messages where parameters are repeated multiple times. This is allowed in the spec, but not defined and the result is not consistent. This reference doesn't seem generalizable to any solvable problem.

CWE-236: Improper Handling of Undefined Parameters

The CWE says:
The software does not handle or incorrectly handles when a particular parameter, field, or argument name is not defined or supported by the product.

There are two references (2002-1488 and 2001-0650). Both result in crashes. VIBRANCE's overall server approach should address these types of issues.

A.7.3 CWE-237: Improper Handling of Structural Elements

The CWE says:
The software does not handle or incorrectly handles inputs that are related to complex structures.

There is no additional information, references, or examples. There are three child CWEs (238, 239, and 240). Note that 239 and 240 are called out specifically in the weakness class document.

CWE-238: Improper Handling of Incomplete Structural Elements

The CWE says:
The software does not handle or incorrectly handles when a particular structural element is not completely specified.

There is no additional information, examples, references, or child CWEs.

CWE-239: Failure to handle incomplete element

This is called out directly in the weakness class document.

CWE-240: Improper handling of inconsistent structural elements

This is called out directly in the weakness class document.

A.8 CWE-239: Failure to Handle Incomplete Element

The CWE says:
The software does not properly handle when a particular element is not completely specified.

There is no further information, examples or child CWEs.

There are four references. Each of the problems occurs when an incomplete request is sent through a socket. The program either hangs indefinitely (blocking other users) or consumes CPU time.

VIBRANCE modifies the application to ensure that all socket requests time out in a reasonable amount of time (specified in the configuration file). VIBRANCE's server error handling ensures that such timeouts are handled so that the server continues correctly.

A.9 CWE-240: Improper Handling of Inconsistent Structural Elements

The CWE says:
The software does not handle or incorrectly handles when two or more structural elements should be consistent, but are not.

There is no further information, examples, or references. There is one child CWE (CWE-130).

A.9.1 CWE-130: Improper Handling of Length Parameter Inconsistency

The CWE says:
The software parses a formatted message or structure, but it does not handle or incorrectly handles a length field that is inconsistent with the actual length of the associated data.
If an attacker can manipulate the length parameter associated with an input such that it is inconsistent with the actual length of the input, this can be leveraged to cause the target application to behave in unexpected, and possibly, malicious ways. One of the possible motives for doing so is to pass in arbitrarily large input to the application. Another possible motivation is the modification of application state by including invalid data for subsequent properties of the application. Such weaknesses commonly lead to attacks such as buffer overflows and execution of arbitrary code.

In the examples, data is read from a socket and lengths do not match the actual data read. Usually this causes more data to be transferred than actually exists. There are also 24 references. These

have not been examined in detail, but most seem to fall into the same category. Note that many result in buffer overflows (which in Java should be converted into exceptions).

VIBRANCE's server protections ensure that any exceptions that are generated are handled.

A.10 CWE-241: Improper Handling of Unexpected Data Type

The CWE says:
The software does not handle or incorrectly handles when a particular element is not the expected type, e.g. it expects a digit (0-9) but is provided with a letter (A-Z).

There are no examples or child CWEs. There are two references. Both are crashes that occur when a non-numeric string is provided where a numeric string is expected. If we presume that the corresponding Java code would raise an exception then VIBRANCE's server protections should handle these cases. All of the `Integer` methods that parse strings throw `NumberFormatException` if any character is not a valid digit.

A.11 CWE-426: Untrusted Search Path

The CWE says:
The application searches for critical resources using an externally-supplied search path that can point to resources that are not under the application's direct control.
This might allow attackers to execute their own programs, access unauthorized data files, or modify configuration in unexpected ways. If the application uses a search path to locate critical resources such as programs, then an attacker could modify that search path to point to a malicious program, which the targeted application would then execute. The problem extends to any type of critical resource that the application trusts.
Some of the most common variants of untrusted search path are:

- *In various UNIX and Linux-based systems, the* `PATH` *environment variable may be consulted to locate executable programs, and* `LD_PRELOAD` *may be used to locate a separate library.*
- *In various Microsoft-based systems, the* `PATH` *environment variable is consulted to locate a DLL, if the DLL is not found in other paths that appear earlier in the search order.*

There are two examples. Both of these examples use the PATH variable.

There are also 6 references. 3 (1999-1120, 2008-1810, 2008-3485) also are based on `PATH`. VIBRANCE handles these by checking each executable that is not specified with an absolute pathname to ensure that it is found in its normal location. If it is not, it can either be repaired (by putting in an absolute path) or an exception can be thrown. 1 (2008-2613) is based on `LD_LIBRARY_PATH`. This is not applicable to Java source. 1 (2007-2027) is based on an untrusted home directory. The program uses `../dir` as a pathname and presumes that will point to a trusted location. This could be a problem for privileged programs that treat the contents of the file as trusted. We can resolve this problem by treating any file that is not owned by the running process or is writable by others as untrusted. 1 (2008-1319) is not a path issue at

all. The program allows untrusted input to specify the path to use to find an executable. Our command execution checks would not allow this.

Even though it is not mentioned, this CWE probably should cover classpaths as well. VIBRANCE checks to ensure that all application classes loaded match the classes checked during static analysis. If a non-matching class is loaded, VIBRANCE will throw an exception.

There are no child CWEs, but there is one peer CWE (427).

A.11.1 CWE-427: Uncontrolled Search Path Element

The CWE says:
The product uses a fixed or controlled search path to find resources, but one or more locations in that path can be under the control of unintended actors.
Although this weakness can occur with any type of resource, it is frequently introduced when a product uses a directory search path to find executables or code libraries, but the path contains a directory that can be modified by an attacker, such as "/tmp" or the current working directory.
In Windows-based systems, when the LoadLibrary or LoadLibraryEx function is called with a DLL name that does not contain a fully qualified path, the function follows a search order that includes two path elements that might be uncontrolled:

- *the directory from which the program has been loaded*
- *the current working directory.*

In some cases, the attack can be conducted remotely, such as when SMB or WebDAV network shares are used.
In some Unix-based systems, a PATH might be created that contains an empty element, e.g. by splicing an empty variable into the PATH. This empty element can be interpreted as equivalent to the current working directory, which might be an untrusted search element.

To the extent that this refers to the execution of commands, VIBRANCE's command execution checks handle this. We check to ensure that any command that is not fully qualified is found in a standard location (which should not be writable by the user). We could extend this to check a signature of the file before executing it to be completely safe.

However, some of the references are to programs that load configuration files from the current directory. As with CWE-426, we can resolve this by treating any file not owned by the current process or writable as untrusted.

A.12 CWE-434: Unrestricted Upload of file with Dangerous Type

The CWE says:
The software allows the attacker to upload or transfer files of dangerous types that can be automatically processed within the product's environment.

There are two examples. The first is specific to php. The second is a java servlet. The servlet does not restrict the file being uploaded. The example says The CWE says: *This could allow an attacker to upload any executable file or other file with malicious code*. This doesn't help us

determine very well what a *dangerous type* is. However, one consistent approach seems to be uploading a file that can then later be addressed (via a URL) from the web. That would make extensions that are executed by a web server be the ones that should be avoided (e.g., `php`, `shtml`, `cgi`, `.php .asp` etc). For safety VIBRANCE defines a white list of legal extensions and disallow extensions other than those.

There are ten references. 2001-0901 stores .shtml file on a web server (from mail attachments). 5 (2002-1841, 2005-1868, 2005-1881, 2004-2262, 2006-4558) are php servers that allows .php files to be uploaded (and thus executed by an attacker). 3 (2006-6994, 2005-3288, and 2006-2428) are similar to the php servers except with asp. 2005-0254 only intends to upload .ps or .pdf files, but allows any file to be uploaded. Users expecting ps or pdf files could possibly be tricked. These checks are included as part of the filename checks for servers only (a user program should be allowed to create any file). VIBRANCE also checks to ensure that the extension matches the actual contents of the file. There are a number of problems that can occur when these do not match. This check will be performed when the file is closed. If the contents to not match, VIBRANCE will rename the file and move it to a safe location.

Checks are performed when closing a file output stream. On files that have been classified as untrusted, the content itself is determined and checked against the extension of the file. In the configuration file you are able to specify which file extensions to allow and their respective MIME types. If the extension is either not listed as allowable or does not match the actual content of the file then an appropriate action is taken. This action can include throwing an exception, terminating the program, or jailing the file to some user-specified safe location.

A.13 CWE-470: Use of Externally Controlled Input to Select Classes or Code

The CWE says:
The application uses external input with reflection to select which classes or code to use, but it does not sufficiently prevent the input from selecting improper classes or code.
If the application uses external inputs to determine which class to instantiate or which method to invoke, then an attacker could supply values to select unexpected classes or methods. If this occurs, then the attacker could create control flow paths that were not intended by the developer. These paths could bypass authentication or access control checks, or otherwise cause the application to behave in an unexpected manner. This situation becomes a doomsday scenario if the attacker can upload files into a location that appears on the application's classpath (CWE-427) or add new entries to the application's classpath (CWE-426). Under either of these conditions, the attacker can use reflection to introduce new, malicious behavior into the application.

The weakness class definition says: *It has been argued that this in scope for this class when any external input directly taints a string that is used in the selection of a class/method/etc.*

There isn't an example per se, but the CWE basically says that you shouldn't allow untrusted input to specify a class and/or method name.

There is one reference that seems to refer to plugin code which is not applicable in our situation.

VIBRANCE disallows any untrusted input in reflection methods. From `Class` this includes `forName`, `getDeclaredField`, `getField`, `getDeclaredMethod`, and `getMethod`.

A.14 CWE-601: URL Redirection to Untrusted Site

The CWE says:
A web application accepts a user-controlled input that specifies a link to an external site, and uses that link in a Redirect. This simplifies phishing attacks.
An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance.

The weakness class definition says:
For CWE-601, it is understood that not all open-redirects are bad. For Phase 1, one may assume that all redirects using common API's are bad if their input is tainted in any way. For example, API functions that do redirects directly, and/or API functions that manipulate headers where it's clear that the "Location" header is being manipulated - e.g. response.setHeader("Location"...)
Use of HTML META tag redirects is out of scope for this weakness class (those are achievable using XSS).

Both of the examples show HTTP input (from get variables) that are used to set redirects. The Java example uses `HttpServletResponse.sendRedirect()`. The comment from the weakness class definition also refers to `HttpServletResponse.setHeader()`. These appear to be the only applicable APIs.

VIBRANCE will check to ensure that there is no untrusted data in the parameters to either of those methods.

A.15 CWE-606: Unchecked Input for Loop Condition

The CWE says:
The product does not properly check inputs that are used for loop conditions, potentially leading to a denial of service because of excessive looping.

The weakness class definition says:
Interpretation may be logical-level, but at the very least there should be a test for complete lack of validation of a user-controlled loop variable. It should be enforced that no externally supplied value can directly modify a loop index without clear checks for the max/min values of the index, plus these max/min values cannot be specified by untrusted input.

There is one simple unrealistic example of a number read from standard input that is used as a loop upper bound. There are no references or child CWEs.

VIBRANCE uses static analysis to identify loops whose iterations are controlled by untrusted input. The static analysis will indicate whether the loop is definitely controlled by untrusted input or may be controlled by untrusted input. The configuration file will specify the maximum loop bounds for each situation and the response. In future phases, VIBRANCE will have a more precise combined static/dynamic analysis for primitives so that these checks will only be invoked when the loop count is known to be untrusted.

We determined the maximums for untrusted loops by experimenting with existing programs. An effective attack will almost certainly require a very high loop count so it is relatively easy to specify a value that keeps false positives low while still detecting attacks.

A.16 CWE-626: Null Byte Interaction Error

The CWE says:
The product does not properly handle null bytes or NUL characters when passing data between different representations or components.
A null byte (NUL character) can have different meanings across representations or languages. For example, it is a string terminator in standard C libraries, but Perl and PHP strings do not treat it as a terminator. When two representations are crossed - such as when Perl or PHP invokes underlying C functionality - this can produce an interaction error with unexpected results. Similar issues have been reported for ASP. Other interpreters written in C might also be affected.
The poison null byte is frequently useful in path traversal attacks by terminating hard-coded extensions that are added to a filename. It can play a role in regular expression processing in PHP.
There are not many CVE examples, because the poison NULL byte is
1. a design limitation, which typically is not included in CVE by itself; and
2. it is typically used as a facilitator manipulation to widen the scope of potential attacks against other vulnerabilities.
Current (2007) usage of "poison null byte" is typically related to this C/Perl/PHP interaction error, but the original term in 1998 was applied to an off-by-one buffer overflow involving a null byte.

Seemingly similar things could happen in the interaction between Java and native calls. There are two references. Both refer to situations where an embedded null causes an incorrect result from a regular expression check in PHP.

VIBRANCE disallows untrusted nulls in Strings. Any nulls are removed.

APPENDIX B: SQL/COMMAND INJECTION

The following sections describe the CWEs listed under the weakness class 'SQL/Command Injection'.

B.1 CWE-78: O/S Command Injection

This CWE refers to injection into operating system command strings. There is no java equivalent to the C library `system` routine. But an application can easily create such a command by using argument to shell (e.g., `/bin/bash -c 'ps aux – wc -l'`. The java exec commands also take each argument in a separate array element so that the injection is somewhat different than it is in C.

There are several cases to consider:

- Passing an arbitrary command as an argument to a shell

  In this case user input should be restricted as follows. This discussion is specific to `bash`. Other supported shells are `sh`, `csh`, `tcsh`, `ksh`, `dash`, and `zsh`. Similar restrictions are imposed on each of them. The list of supported shells and the command line option(s) that introduce arbitrary command strings are specified in the configuration file.

  Commands - This is the first token in the command string and specifies the command to be executed. VIBRANCE checks this token to ensure that it does not contain any untrusted data.

  Additional Commands - Shell syntax allows multiple commands to be specified in a single string. For example, pipes, semicolons, command substitution (back ticks), etc. VIBRANCE will check each additional command for any untrusted characters.

  Quoted Strings – A quoted string is a contiguous string of untrusted characters surrounded by application (trusted) quotes. VIBRANCE first checks the content of the quoted string against a white list (must match) and a blacklist regular expression (must not match). VIBRANCE will also escape any untrusted embedded quote within a quoted string.

  Other tokens - VIBRANCE will check the content of any contiguous string of untrusted data that is not surrounded by application (safe) quotes against white list and blacklist regular expressions. As configured, these checks are the same as the filename checks XREF

- Executing a standard command (not a shell) directly

  Most standard commands (e.g., ls, ps, cat) do not provide the ability to execute arbitrary commands via their command line options. These commands are considered *safe*.

Filename arguments to safe commands are subject to the normal filename checks. Other arguments (such as regular expressions) are not checked. In phase 1, the filename check is applied to all arguments.

Some standard commands do provide the ability to specify arbitrary commands and/or code on their command lines. For example, `perl`, `awk`, `find`, `ssh`, etc. However, the restriction that all arguments pass the filename check removes the danger of these commands.

- Executing a non-standard command

  A non-standard command is one that is not present in the standard debian distribution. The command line options to these commands are not known. Conservative rules will thus be enforced on each argument. Each argument is presumed to be a filename and is checked against the basic filename rules.

- Commands within a shell argument

  The above restrictions on standard/non-standard commands will be enforced on those that appear as part of a command string passed to a shell.

B.2 CWE-89: SQL Injection

The CWE says:
The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

VIBRANCE enforces a number of SQL specific rules on untrusted data within an SQL query. First, it checks for all untrusted characters that appear between trusted (application) quotes. Any quote characters are properly escaped to ensure that they do not circumvent the application quotes. Unquoted untrusted characters are checked to ensure that they are valid in their location. In all cases, they can only form a single alphanumeric token. Following an equals sign, they can only specify a valid SQL number.
In most cases VIBRANCE is able to correct the SQL string so that the resulting query acts as expected by the application. In some cases, where this is not possible, VIBRANCE will throw an exception.

B.3 CWE-90: LDAP Injection

The CWE says:
The software constructs all or part of an LDAP query using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended LDAP query when it is sent to a downstream component.

LDAP search queries follow the basic lisp style expression syntax (i.e. `(&(A=string1)(B=string2))`). VIBRANCE treats all untrusted inputs as string values and as it tokenizes the expression, all tainted special characters with their respective character code.

The following is a list of special characters that we perform replacements on:

- Wildcard Character: Tainted `*` are converted to `\2a`.
- Parenthesis: Tainted `(`, `)` are converted to `\28`, `\29` (resp.)
- Escape Character: Tainted `\` are converted to `\5c`.

B.4 CWE-643: XPath Injection

The CWE says:
The software uses external input to dynamically construct an XPath expression used to retrieve data from an XML database, but it does not neutralize or incorrectly neutralizes that input. This allows an attacker to control the structure of the query.

XPath queries syntactically differ from SQL queries, however for the purpose of injection attacks they are similar enough that they can be treated in a similar fashion. For example, the following is one such XPath query:

```
//book/title/text()='Some Random Book'
```

The main difference is that XPath doesn't support the escaping of characters such as double quotes and single quotes. Instead, in order to escape a double-quote a single-quoted string must be used (and vice versa).

As with SQL queries, VIBRANCE tokenizes the input and checks are made based on whether the previous token to the tainted token is an '='. We ensure that the immediate token following the '=' is numeric, alpha-numeric, or String (either single or double quotes). For quoted strings, if the string contains tainted double quotes, the double-quoted string is converted to a single-quoted string and vice versa. If both types of tainted quotes are encountered, a repair is not possible and an exception is thrown.

B.5 CWE-652 XQuery Injection

The CWE says:
The software uses external input to dynamically construct an XQuery expression used to retrieve data from an XML database, but it does not neutralize or incorrectly neutralizes that input. This allows an attacker to control the structure of the query.

XQuery is very similar to XPath and a similar approach is used.

APPENDIX C: NUMBER HANDLING

The weakness class definition says: *For C/C++ programs, this weakness class is often related to buffer errors, since incorrect calculations can lead to buffer overflows. This weakness class will focus on the actual number handling issues, while the Buffer Overflow weakness class will focus on the actual buffer overflow issues. However, test cases developed for one of these weakness classes may in fact incorporate issues from the other weakness class.*

Buffer errors are not normally an issue in Java. For Java applications, this weakness class usually results in denial of service attacks or unintended control flow. This is especially true when an input or calculation is used as a loop counter.

For Phase 1, any conversion or calculation between integer data types that can produce a value that is not equivalent to the pure-math result may be considered in scope, whether the issue has a CWE entry assigned or not. Performers may assume that all such potentially-incorrect calculations/conversions are invalid.

All number handling weakness types related to (1) memory management, (2) allocation of system/application resources, or (3) number of iterations MUST be treated as invalid, and appropriately confined. Any other type of number handling weakness MAY be treated as invalid. For example, a cryptographic function may use integer wraps as part of modular arithmetic, or a financial application may perform price calculations. If the confinement or diversification step changes this functionality, it will not be regarded as an error for the purpose of STONESOUP.

Future Direction:

The STONESOUP Test and Evaluation team is unaware of any formal investigation into the security implications of floating point operations, and there are very few real-world examples as documented in CVE. (two known examples are CVE-2010-0131 and CVE-2009-2195) This is likely a research gap in the vulnerability research community. Potential errors might be inferred from the safety, correctness, and scientific computing communities. In some cases, a known type of floating-point error might not have any clear security implications. For example, a floating point comparison using "==" may be a common error, but even though incorrect comparison could have security consequences, it might be mostly applicable in security decisions such as authentication, privilege level, or prioritization, which are probably implemented using integers instead of floating points. Floating point errors might be more relevant at a logical/design level (e.g. where precision problems might generate a financial advantage to an attacker), but this is arguably outside the scope of STONESOUP. In other cases related to reliability, the application to STONESOUP is clearer, such as divide-by-zero and not-a-number operations that can cause a denial of service. Other operations may involve floating points as an incidental player, such when a format string does not have a size limitation in floating-point output, which could play a role in buffer overflows, but is not directly related to the modification or calculation of floating-point values.

Some possible additions to the weakness types included in this class are listed below.

- *floating point overflow and underflow*
- *double precision floating point overflow and underflow*
- *floating point comparison using ==*
- *floating point not-a-number operations*
- *incorrect floating point conversion*
- *missing enforcement of positive value in signed context - In many complex memory-corruption vulnerabilities reported for major software in CVE, a chain occurs in which the code checks for a maximum value, but does not check a minimum, and assumes that the value is always positive. This is not necessarily the case when the variable is signed, and the signed/unsigned conversion can produce a large positive number. This type of problem is likely to get its own CWE identifier, or otherwise be combined into CWE-195.*
- *incorrect mathematical shift or other bitwise operation - Performing a shift or similar operation that produces a value that is not the same as the pure-math value. Example: shifting an 8-bit value more than 7 times, causing successive values of 0 to be produced, instead of monotonically-increasing or monotonically-decreasing values.*

Our basic approach is to detect overflows on untrusted primitive values and mark those values as overflowed. When security critical operations occur, primitives are checked to see if an overflow has occurred. If it has occurred, we throw an exception which will normally terminate the program/request. The overflow tag can be removed on operations that clear the high order bits of a primitive (eg. masking). This should, for example, allow hash-codes to be used in hash tables (the hashed value is normally masked before being used as an index).

This requires primitive taint tracking. Both for trusted/untrusted values and for overflows/underflows.

Security critical operations include:

- Array indexing
- Loop control. Probably the use of an overflowed value in any conditional indicates an error.
- Memory allocation.
- Array and string copies/moves
- Seeking in a stream
- Other JDK calls

We considered checking for bounds checks on primitive values. Arguably a value that is checked against an upper and lower bound can be thought of as trusted. In an operation over two untrusted values, the result has a min/max check only if both operands had min/max checks. It is not clear if certain operations should be checked if their operands do not have min/max checks. We need some way to infer valid values for operations over untrusted values. We did this in phase 1 for loop bounds. We could possibly do something similar for memory allocations. Array indices have a defined min/max that is already checked.

Testing in practice indicated that checking for upper/lower bounds led to too many false positives in practice. We thus did not implement this check. Instead VIBRANCE checks for tainted values used in sensitive operations (such as array indices) and checks for out-of-bound accesses directly.

The following actions should be included in the implementation.

- **ok-overflow**. This is the basic check outlined above. The check includes both underflow and overflow on integer values.
- **ok-infinity** Check divides and possibly some other math operations for infinite results. The result should be the max value possible and should be tagged as infinity. Infinity propagates through any other operations (including masks). The normal check are applied when the value is used in a critical operation.
- **ok-java**. Some CWEs simply don't apply to Java. In particular, since Java doesn't have unsigned types, none of the sign conversions CWEs would appear to apply.

C.1 CWE-190 Integer Overflow or Wraparound

The CWE says: *The software performs a calculation that can produce an integer overflow or wraparound, when the logic assumes that the resulting value will always be larger than the original value. This can introduce other weaknesses when the calculation is used for resource management or execution control.*

An integer overflow or wraparound occurs when an integer value is incremented to a value that is too large to store in the associated representation. When this occurs, the value may wrap to become a very small or negative number. While this may be intended behavior in circumstances that rely on wrapping, it can have security consequences if the wrap is unexpected. This is especially the case if the integer overflow can be triggered using user-supplied inputs. This becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors such as memory allocation, copying, concatenation, etc.

There are 4 examples (all in C). Two end up in malloc, 1 as a loop bound, and 1 as a calculation in an accounting application.

There are 6 references. 2 have a length value of -1 that leads to zero length buffer allocation (and heap buffer overflow in C). 3 others appear to be buffer overflows and 1 is a use after free. While all of these overflows can occur in Java, none would lead to vulnerabilities (other than DOS).

This is handled by our general overflow approach.

C.2 CWE-191 Integer Underflow (Wrap or Wraparound)

The CWE says: *The product subtracts one value from another, such that the result is less than the minimum allowable integer value, which produces a value that is not equal to the correct result.*

There is one uninteresting example and 4 references.

Note that this refers to underflow from -214748348 to 2174483647 not underflow from 0 to -1.

This is handled by our general overflow approach.

C.3 CWE-194 Unexpected Sign Extension

The CWE says: *The software performs an operation on a number that causes it to be sign extended when it is transformed into a larger data type. When the original number is negative, this can produce unexpected values that lead to resultant weaknesses.*

This is not as relevant to Java as it only has one unsigned type (char) that is not typically used for integer calculations. This weakness can also occur when casting between two signed types. For example, a byte read from a file that contained 0xFF would not be equal to 255 in an integer, but would be sign extended instead.

VIBRANCE checks for casts on untrusted data where the resulting value is arithmetically different than the original value. For example, if an integer is cast to a byte, an exception will be thrown if the value is not within the range of a byte (i.e., -128 to 127).

In some cases this can lead to false positives if the application is performing low level bit manipulations on the value (and thus expects the truncation). VIBRANCE keeps track of whether or not the value is *bitwise*. A bitwise value is one that has been most recently manipulated with a bit operation (and, or, not). That allows programs that are reading values from binary stream one byte at a time and forming them into larger values to be able to manipulate those values as expected. For example:

```
byte b1 = stream.read();
byte b2 = stream.read();
short s = (short) b1 << 8 | b2;
```

The calculation is done in 32 bit integers and then the result is cast to a short. As an integer a negative short value would appear to be a positive number too large for a short. But this is expected behavior. VIBRANCE does not throw an exception on this cast because the value being cast is bitwise rather than numeric. Conversely, in the following example:

```
short count = (short) Integer.parseInt(stream.readLine());
```

If a value greater than 32K were read from the stream, an exception would be thrown, because the default type is numeric.

C.4 CWE-195 Signed to Unsigned Conversion Error

The CWE says: *A signed-to-unsigned conversion error takes place when a signed primitive is used as an unsigned value, usually as a size variable.*

There are 4 examples and one reference.

Java has only one unsigned type (char) that is not typically used for integer calculations. This is thus not a common problem for Java.

In the same fashion as for other casts, VIBRANCE checks to ensure that the resulting value is the same as the original value. Thus if an untrusted negative value numeric value is cast to a character, VIBRANCE will throw an exception. See CWE-194 for more details.

C.5 CWE-196 Unsigned to Signed Conversion Error

The CWE says: *An unsigned-to-signed conversion error takes place when a large unsigned primitive is used as a signed value.*

Often, functions will return negative values to indicate a failure. In the case of functions that return values which are meant to be used as sizes, negative return values can have unexpected results. If these values are passed to the standard memory copy or allocation functions, they will implicitly cast the negative error-indicating value to a large unsigned value. In the case of allocation, this may not be an issue; however, in the case of memory and string copy functions, this can lead to a buffer overflow condition which may be exploitable. Also, if the variables in question are used as indexes into a buffer, it may result in a buffer underflow condition.

Although less frequent an issue than signed-to-unsigned casting, unsigned-to-signed casting can be the perfect precursor to dangerous buffer underwrite conditions that allow attackers to move down the stack where they otherwise might not have access in a normal buffer overflow condition. Buffer underwrites occur frequently when large unsigned values are cast to signed values, and then used as indexes into a buffer or for pointer arithmetic.

There is one example and no references.

As with CWE-194 and CWE-195, VIBRANCE checks for casts on untrusted data where the resulting value is arithmetically different than the original value. For example, if a character is cast to a short, an exception will be thrown if the value is not within the range of a short (-32K to 32K).

C.6 CWE-197 Numeric Truncation Error

The CWE says: *Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.*

When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion, potentially resulting in an unexpected value that is not equal to the original value. This value may be required as an index into a buffer, a loop iterator, or simply necessary state data. In any case, the value cannot be trusted and the system will be in an undefined state. While this method may be employed viably to isolate the low bits of a value, this usage is rare, and truncation usually implies that an implementation error has occurred.

It is interesting that the CWE asserts that this is usually an error and not a purposeful masking operation. It is not clear how to ascertain that. We could look for examples.

As previously noted, VIBRANCE distinguishes between a purposeful truncation and an incorrect truncation by distinguishing between bitwise and numeric values. Truncation of a bitwise value is considered to be intentional. Truncation of a numeric value will result in the value being marked as overflowed. In practice this distinction seems to work very well.

C.7 CWE-198 Use of Incorrect Byte Ordering

The CWE says: *The software receives input from an upstream component, but it does not account for byte ordering (e.g. big-endian and little-endian) when processing the input, causing an incorrect number or value to be used.*

Because byte ordering bugs are usually very noticeable even with normal inputs, this bug is more likely to occur in rarely triggered error conditions, making them difficult to detect using black box methods.

There are no examples, references, or child CWEs.

Seemingly this is largely irrelevant. Whether or not the byte ordering is correct, a possible attacker always has control over the bytes and can generate any desired values. The key is that the resulting value is untrusted and should be checked in the normal ways.

C.8 CWE-369 Divide By Zero

The CWE says: *The product divides a value by zero.*

This weakness typically occurs when an unexpected value is provided to the product, or if an error occurs that is not properly detected. It frequently occurs in calculations involving physical dimensions such as size, length, width, and height.

There are three simple examples and three references.

We treat this very similarly to an overflow. The result is marked as infinity and action is taken if the value is used in a sensitive operation. If the value is printed it is printed as infinity.

C.9 CWE-682 Incorrect Calculation

The CWE says: *The software performs a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management.*

When software performs a security-critical calculation incorrectly, it might lead to incorrect resource allocations, incorrect privilege assignments, or failed comparisons among other things. Many of the direct results of an incorrect calculation can lead to even larger problems such as failed protection mechanisms or even arbitrary code execution.

There are three examples. One is an integer overflow, one is a divide by zero error, and one is a calculation error because of an incorrectly placed cast in C.

There doesn't seem to be any way to address an incorrect calculation in general. There is no information to determine what is correct and what is not. The only seeming response is to catch the results (such as an overflow or divide by zero)

C.10 CWE-839 Numeric Range Comparison Without Minimum Check

The CWE says: *The program checks a value to ensure that it does not exceed a maximum, but it does not verify that the value exceeds the minimum.*

Some programs use signed integers or floats even when their values are only expected to be positive or 0. An input validation check might assume that the value is positive, and only check for the maximum value. If the value is negative, but the code assumes that the value is positive, this can produce an error. The error may have security consequences if the negative value is used for memory allocation, array access, buffer access, etc. Ultimately, the error could lead to a buffer overflow or other type of memory corruption.

The use of a negative number in a positive-only context could have security implications for other types of resources. For example, a shopping cart might check that the user is not requesting more than 10 items, but a request for -3 items could cause the application to calculate a negative price and credit the attacker's account.

In languages with unsigned values, this is obviously related to the signed to unsigned conversion issues. In Java this is less of a problem, but it can certainly still lead to DoS.

There are 4 examples and 8 references. The security issues in all of the references are not applicable to java (they include buffer overflow/underflows, out of range array indices, etc).

Testing in practice indicated that checking for upper/lower bounds led to too many false positives in practice. We thus did not implement this check. Instead VIBRANCE checks for tainted values used in sensitive operations (such as array indices) and checks for out-of-bound accesses and other sensitive uses directly.

APPENDIX D: ERROR HANDLING

The following sections describe the CWEs listed under the weakness class 'Error Handling'. CWEs that are children of the name CWEs are placed in sections under their parents.

Some open issues/todo items are:

- Are there functions that *drop privilege* in java? Can these functions fail in some way that would leave the program with a higher privilege?

  In general, we don't believe this is done in the same fashion as can be done in C. The related system calls are not directly available to Java. The closest parallel is running code under a security manager.
- How can we find functions that we need to check for unchecked return values (or other error indications)? The only example we know of is `read`. There are probably others. We could check for a particular application, but how do we check in general? Perhaps by analyzing some open source programs for calls with return values that are not checked?

  CodeHawk analyzed the JDK looking for constants that are returned from functions. These were examined to see if they were error codes and information concerning the error code values was added to the summary for the function. Additionally the parameter value that influenced whether or not the error code was returned was identified. This allows error checks for return codes to only be implemented when they are triggered by untrusted data.
- When an unexpected error or exception occurs, what action should we take? If there is no exception handler, then seemingly the obvious choice is to terminate the program or request. If there is an exception handler, are there cases where we don't want to use it? For example, if an unexpected null return value occurs should we terminate the program/request if there is not an explicit check?
- How can we determine what exceptions a handler is *intended* to handle? Should we only allow handlers for checked exceptions (those declared by the code). Or only those that occur in the same method?
- Do we need to be concerned about a security manager? Seemingly all of the classes are known, so a security manager would be unnecessary, but a test case could be made where some of the classes had more privilege than others. In particular, do we need to be concerned about `AccessController.doPrivileged()`? That call allows code with privilege to execute calls that its caller does not have privilege for (which is normally not allowed).

  This would only normally make sense if the application included some code for which was intended to have lesser privilege. In that case we would want to somehow taint values that came from the less privileged classes and make sure that they didn't get passed to privileged operations. I tend to think this violates the basic rules of the program (that the code is not malicious). There is a set of interesting bugs, however, in dealing with code that may be malicious. But this seems out of scope.
- Do we need to consider cleanup beyond what is obvious from known APIs? Obvious cleanup would include closing files, closing database connections, releases locks, etc (and

is proposed below as *close-res*. Other necessary cleanup would be dependent on program semantics. For example, removing a partially populated database record or object.

There are at least two possible problems with more complex cleanup. First, it is difficult to determine what a transaction is. Consider an FTP server, a thread handles all of the requests from a single client. Each client may send multiple requests (eg, ls, cd, upload file, etc). How do we determine when these requests begin and end? Second, what sort of ACID transactions can we create? External changes (such as file-system changes and database changes) would probably be doable. For databases we could use the transactions built into the database. For file-systems we could probably keep track of changes and unroll them (though changes to the content of a file might be difficult). Keeping track of changes to objects within the program would seem both difficult and expensive. There are some systems that at least try to do this. For now, we should probably not attempt transaction cleanup beyond what can be handled in close-res*. It is interesting to note that none of the CWEs specifically indicate this as a problem. It is certainly related to CWE-460, but there are no examples there (or elsewhere) that would require cleanup more complex than that of* close-res*.*

We should consider implementing the following. In many cases, we will want to throw an exception that will normally terminate the program/request. For non-servers this means terminating the program. For servers this means terminating the current request using our server error handling techniques.

- **ok-leak**. Remove all trusted text from writes that take place in exception handling blocks. See CWE-209 for more information.
- **ok-short-read**. Add checks for read calls that return the number of bytes read. In all cases where the number of bytes read is less than the requested number of bytes, the unset values in the buffer are zeroed out. We might want to consider if there is a similar fix to the `Reader.skip(n)` method
- **ok-ret-final**. Remove returns (and break/continue) from finally clauses when an exception is the reason the finally clause is being executed.See CWE-584 for more information.
- **ok-ret-check**. Add checks for all JDK calls that return error codes. Some sort of static analysis would be required to determine if the call returns an error code. The check of such calls would work in the same manner as the null-check. We can look for calls that return constant integers, constant strings or enums in an effort to find these.
- **ok-empty-catch**. Empty catch blocks should be modified to terminate the program/request. Its not entirely clear how we should define ’empty’. Simply printing a message is probably not sufficient in most cases.
- **ok-close-res**. When errors are not handled correctly or sufficiently, objects can be created that are not correctly closed (eg, file descriptors, locks, SQL connections, etc). VIBRANCE will track the creation of these objects. such object created during a request processing is tracked. When the request completes (either correctly or erroneously), VIBRANCE performs the correct cleanup for the object.

We could attempt to determine statically which of these objects might be used outside the lifetime of the request (its certainly possible that an object created during one request could be used by subsequent requests, though it seems rare). For non-servers, we could ensure that all objects were properly cleaned up when the program exits.

We could augment this to keep track of objects created in `try/catch` blocks. When a `catch` block is executed, cleanup could be performed for any objects created in the try block. The same static analysis as above could be used to determine which objects may have escaped the `try` block and should thus not be cleaned up at this point.
Note that java 7 supports AutoClosable. The classes that implement AutoClosable are a reasonable list of the classes we should support.
Another thought is that we should look for items (files, locks, etc) that are 'opened' and would be 'closed' except that there is an intervening exception. If we could statically identify such a situation (eg, a matching open/close), then we could perhaps add or correct proper exception handling for it.

- **ok-unchecked**. Generic exception handlers will be modified to check for unchecked exceptions (Error or RuntimeException). If those exceptions occur, they will be rethrown normally resulting in program/request termination, unless the program has a higher exception handling block that explicitly checks for the error in question. We could also throw one of our own exceptions, guaranteeing that we would end up at our exception handler.
- **ok-servlet-exc**. Wrap servlets in exception handlers and only print interesting information to the log file. Ensures that servlet containers do not leak sensitive information when errors occur. See CWE-600
- **ok-null-check**. Add checks for all JDK calls that can return nulls that are not checked by the code. We should be able to statically analyze the JDK to determine the list of calls (or use annotations added for Javari). For each call of such a routine, static analysis should determine if a null check is already present and if the resulting value is dereferenced. If there is no existing null check and the value is dereferenced, a null check should be added. The program/request should terminate if a null is encountered.

  This check could be expanded to handle application routines that may return null as well. This shouldn't change the program semantics significantly (as the null check will occur anyway and throw an exception), but it allows us to indicate the problem and where it occurred.

The following are best accomplished by static analysis (CodeHawk).

- **Identify catch blocks**. Identify the code that is in an catch block. Special checks are placed in catch blocks (such as checking writes for sensitive data). We also need to identify an empty catch block.
- **Identify finally blocks**. Identify the code that is in a finally block. Special checks are required in finally blocks (such as removing returns and break/continue)
- **Identify servlets** Servlets need to be wrapped in special exception blocks.

- **Identify objects to cleanup**. See *ok-close-res* above for details. Static analysis needs to identify objects in a try/catch block that need cleanup and don't escape the block. Similarly for request processing. This can be augmented by a dynamic check.
- **Identify JDK calls that return error codes** We will look for calls that return an enumeration or a constant. And calls that propagate such a value. If multiple constants can be returned they might form a likely set of values that should be checked for. It seems that such a set of calls could be identified and we could check by hand to see which (if any) are returning error codes.

  When we are analyzing a user program we could likewise look at calls (both to the JDK and not) that don't use a return value.
- **Identify JDK calls that return null**. See *ok-null-check* above. This is similar to error codes. We also might want to extend this to the user program.

D.1 CWE-209 Information Exposure through an Error Message

The CWE says: *The sensitive information may be valuable information on its own (such as a password), or it may be useful for launching other, more deadly attacks. If an attack fails, an attacker may use error information provided by the server to launch another more focused attack. For example, an attempt to exploit a path traversal weakness (CWE-22) might yield the full pathname of the installed application. In turn, this could be used to select the proper number of ".." sequences to navigate to the targeted file. An attack using SQL injection (CWE-89) might not initially succeed, but an error message could reveal the malformed query, which would expose query logic and possibly even passwords or other sensitive information used within the query.*

There are four examples in the CWE. Each essentially simply prints an exception out (eg, println(exception)). The last also prints the query that led to the failure. The query would reveal table/column names that would help in injection attacks.

There are 7 references. Revealed information includes passwords, file contents, user names, and pathnames.

The weakness class definition says: *It may be necessary to define which type of information is regarded as exposed. However, for Phase 1, treat it as "any information that does NOT come from a tainted source".*

One approach here would be to add a check to all write commands that occur within exception handling blocks and remove any trusted data (with the exception of the exception name itself (eg, `java.sql.SQLException`) That would be relatively straightforward. It is not entirely clear what constitutes a 'write' however. Nor is it necessarily easy to tell whether or not the destination of the write is dangerous. For non-servers it would seem that any write would be dangerous (possibly viewable by an attacker). So would any GUI writes. For servers, we could possibly limit it to network connections.

A more difficult problem might be determining what actions are taken in response to an exception. Everything might not happen in an exception block. Flags could be set and then looked at later. Methods could be called from within an exception block and outside of a block. We could probably use a thread-local flag to differentiate those. It would be more difficult to determine actions taken outside of the block that are triggered by the block (eg, setting a boolean). Perhaps the static analysis could look for these (or at least some subset of them).

D.2 CWE-248 Uncaught Exception

The CWE says: *An exception is thrown from a function, but it is not caught. When an exception is not caught, it may cause the program to crash or expose sensitive information.*

There are three examples. Most are of unexpected exceptions that the programmer did not bother to catch.

There are no references.

There is one child (CWE-600 Uncaught Exception in Servlet), which is called out independently (below) in the weakness class document.

Seemingly this is handled by our existing server mechanisms. A crash is the most likely result from an uncaught exception. Our server checks will ensure that only the current request is terminated. In non-servers, exiting the program seems like the best choice. We can add code to make sure that if a program exits due to an uncaught exception that no useful information is printed as a result of the crash.

D.3 CWE-252 Unchecked Return Value

The CWE says: *The software does not check the return value from a method or function, which can prevent it from detecting unexpected states and conditions. Two common programmer assumptions are "this function call can never fail" and "it doesn't matter if this function call fails". If an attacker can force the function to fail or otherwise return a value that is not expected, then the subsequent program logic could lead to a vulnerability, because the software is not in a state that the programmer assumes. For example, if the program calls a function to drop privileges but does not check the return code to ensure that privileges were successfully dropped, then the program will continue to operate with the higher privileges.*

There are 7 examples that are applicable to Java (3 through 9). Three are related to I/O commands (such as Reader.read()) that do not necessarily read the number of bytes requested (they can return with a smaller number). Three are null pointer related (program does not checked for a returned value of null) and one is related to mutexes (program doesn't check for an error in getting the lock).

There are 4 references related to Java. One is an unchecked return value that leads to integer overflow, two don't check the return value of function that drops privileges, and one is a null

pointer. Note that none of these are actually of Java code, but the idea could possibly apply to Java.

VIBRANCE checks all function calls that return error codes or null to ensure that the error code is checked by the application. If it is not checked for by the application and the unexpected result is triggered by untrusted data, VIBRANCE will throw an exception when the error occurs. Details are in the ret-check and null-check sections above.

The exception will normally terminate the program/request (unless the program explicitly handles it). VIBRANCE's server protections ensure that only the request with the problem is terminated and that the server itself will continue to run correctly.

VIBRANCE also ensures that any `read()` call that returns less than the full size of the buffer, zeroes out any unread values in the buffer (see short-read above).

D.4 CWE-253 Incorrect check of Function Return Value

The CWE says: *The software incorrectly checks a return value from a function, which prevents the software from detecting errors or exceptional conditions.*

There is one example and no references. The example is a call to malloc that checks for a negative return value, but not a return value of null.

VIBRANCE handles this in exactly the same manner as CWE-252. VIBRANCE takes action on any error code return value that is not checked for. See the ret-check section for details.

D.5 CWE-273 Improper Check for Dropped Privileges

The CWE says: *The software attempts to drop privileges but does not check or incorrectly checks to see if the drop succeeded. If the drop fails, the software will continue to run with the raised privileges, which might provide additional access to unprivileged users.*

There is one example and two references. None, however, are applicable to Java (they are all setuid). Since setuid() is not available to Java, this seems not to apply.

D.6 CWE-274 Improper Handling of Insufficient Privileges

The CWE says: *The software does not handle or incorrectly handles when it has insufficient privileges to perform an operation, leading to resultant weaknesses.*

There are no examples and three references. None of the references is specific to Java.

This seems very generic. Two of the references are to crashes that occur when the program encounters an error. VIBRANCE's server protections should be sufficient here (the server should not terminate).

D.7 CWE-280 Improper Handling of Insufficient Permissions/Privileges

The CWE says: *The application does not handle or incorrectly handles when it has insufficient privileges to access resources or functionality as specified by their permissions. This may cause it to follow unexpected code paths that may leave the application in an invalid state.*

This is very similar to CWE-274.

This is also very generic. Whether the unexpected error is caused by insufficient privileges or something else (like a null dereference), doesn't seem that relevant (unless there were specific APIs to check for and we don't know of any).

There are no examples and two references. One of the reference is that an FTP server places a user in the root directory if the users permissions don't allow access to his/her own home directory. This is a good example of the program ending up in a bad state because of an unexpected error. As long as VIBRANCE ensures that *unexpected errors* result in termination of the program/request, this should resolve the problem.

D.8 CWE-390 Detection of Error Condition Without Action

The CWE says: *The software detects a specific error, but takes no actions to handle the error*.

There are no references and three examples (2 in C and 1 in Java). Each shows a `try/catch` with an empty catch. In each case there is some obvious cleanup to do (eg, closing a file). Interestingly, they show a 'correct' version that doesn't close the file, but just rethrows the exception.

Its easy to detect these situations statically, but less clear what we should do. The best approach would seem to be to terminate the program/request when this happens. We could try to add cleanup for simple cases (such as files). Its not clear, though that this is worth the trouble.

Perhaps we could statically find open files (and other similar things) that are only stored in locals (perhaps parameters/return values). We could add a local `try/catch` around them and close them when an error occurs. Or we could store them in a global table and close them as part of the server loop cleanup. We could presume that anything opened while processing a request should be closed when the request is finished. This would require us to set a flag of some sort indicating we were processing a request and adding all objects that need to be closed to the request's list. Then when the request is finished, we would close all of objects on the list (and empty the list).

The later approach of tracking open objects at run-time and closing them during server loop error processing seems pretty easy to do. The only possible problem is if the program expects some things opened during one request to be available for subsequent requests.

We could also close all such objects when a non-server exits.

Upon investigation we found that taking an action on empty catch blocks was too likely to cause a change in the program's behavior. There are too many cases where a program does this purposefully. Even though it is almost certainly not good practice, there is no reasonable remediation when the program makes this choice.

VIBRANCE thus does not take an action on empty catch blocks. However, it does resolve resources that were not closed during a request. See the File Handler description in the resource consumption section.

D.9 CWE-391 Unchecked Error Condition

The CWE says: *Ignoring exceptions and other error conditions may allow an attacker to induce unexpected behavior unnoticed.*

There is one example and no references. The example is exactly the same as in CWE-390. In Java, we don't need to worry about checked exceptions not being handled. The compiler forces them to be handled. Unchecked return values are described under CWE-252.

D.10 CWE-394 Unexpected Status Code or Return Value

The CWE says: *The software does not properly check when a function or operation returns a value that is legitimate for the function, but is not expected by the software.*

The weakness class definition says: *This may require design-level interpretation, but not always. For Phase 1 focus will be only on knowledge of error codes for established APIs. Example: a socket call can return 5 different "errno" values that indicate a fatal problem, but the code only checks 3 of them*

There are no examples and 8 references. Two references are unexpected error codes returned from recv(). One is an unexpected return of null. One is a read command that returns fewer bytes than expected (leaving unchanged whatever was previously in the buffer). Two improperly handle a return value from a reverse DNS lookup (when the lookup fails). One is a C program that doesn't properly check return values from code that creates strings from remote input and one is a C program that doesn't handle an error return value of -1 in some text processing.

This is essentially the same as CWE-252.

D.11 CWE-395 Use of NullPointerException Catch to Detect NULL Pointer Dereference

The CWE says: *Catching NullPointerException should not be used as an alternative to programmatic checks to prevent dereferencing a null pointer.*

There is one example and no references. The example is not illuminating. The CWE does not make clear what the problem is, though it is implied that adding `try/catch` is more expensive than explicit null checks.

The null-check approach should catch these that occur as a result of making an API call. But it wouldn't catch null dereferences of fields (though it might catch where the field was set).

D.12 CWE-396 Declaration of Catch for Generic Exception

The CWE says: *Catching overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.*

Multiple catch blocks can get ugly and repetitive, but "condensing" catch blocks by catching a high-level class like Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention.

The biggest problem here would seem to be that unexpected run-time exceptions (such as null pointers, array indexing, class cast, etc) may trigger exception handling intended for something else.

It does not seem possible to determine if a generic exception handler intended to handle all of the possible specified exceptions. I think we will have to presume that it did. We could augment logging to indicate exactly what type of exception occurred and we could also indicate that this occurred to the test harness.

If an *unchecked* exception occurs (Error or RuntimeException), it seems reasonable to presume that this was not expected. From an Oracle Java Tutorial on exceptions:

> Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

Unfortunately, investigation of real-world applications showed that programs do expect to be able to recover from these errors. Taking actions when unchecked exceptions occur leads to modifications to program behavior. VIBRANCE thus does not take any direct action here.

D.13 CWE-397 Declaration of Throws for Generic Exception

The CWE says: *Throwing overly broad exceptions promotes complex error handling code that is more likely to contain security vulnerabilities.*

Declaring a method to throw Exception or Throwable makes it difficult for callers to perform proper error handling and error recovery. Java's exception mechanism, for example, is set up to

make it easy for callers to anticipate what can go wrong and write code to handle each specific exceptional circumstance. Declaring that a method throws a generic form of exception defeats this system.

There is one example and no references. The example simply shows a method that throws Exception rather than something more specific.

There doesn't seem to be anything specific to do here. We can detect such methods. We can even detect the explicit list of exceptions that they might throw. But its not clear what could be done with that information. It doesn't make sense to change the program semantics because it throws an exception. Again, perhaps all that makes sense is to log the situation.

D.14 CWE-460 Improper Cleanup on Thrown Exception

The CWE says: *The product does not clean up its state or incorrectly cleans up its state when an exception is thrown, leading to unexpected state or control flow.*

There is one simple example and no references. The example shows a lock that is obtained before an exception and not released in the cleanup handler.

This is similar in some fashion to the close-res item that we are considering. Locks are simply another sort of object that requires cleanup. One question is whether or not the cleanup should be done immediately in the `catch` block or can be safely delayed until the server handler. Optimally it would take place in the local `catch` block. This would complicate the work of keeping track of objects that might need cleanup (we would need a stack of `try/catch` blocks).

VIBRANCE closes locks and file handlers at the end of each request as part of its server protections.

D.15 CWE-584 Return Inside Finally Block

The CWE says: *The code has a return statement inside a finally block, which will cause any thrown exception in the try block to be discarded.*

There is one simple example and no references.

The problem with return statements in finally blocks is that their semantics is not intuitive. They will immediately return and if there was an exception that triggered the finally block, it will be discarded. Similar things occur with breaks and continues in a finally block.

The return statement should either be inside of the `try` block or outside of the entire `try/catch/finally` block. However given that it is inside of the finally block, its not entirely clear what should happen.

Seemingly the best solution would be for the code to execute as written whenever an exception is *not* thrown. When an exception is thrown, however, we want to ensure that exception still gets thrown from the block. Essentially, we want to add something like:

```
if (!exception)
  return xyz
```

It turns out that finally blocks are not directly supported by the JVM. Exceptions are directly supported. Each method indicates the place to jump when exceptions occur within a range of code. Since finally blocks must be executed when exceptions occur, they appear in this list. For example:

```
Exception table:
 from   to  target type
  11    29    48   Class java/lang/Exception
  11    29    71   any
  48    73    71   any
```

The type of *any* seems to occur only for `finally` blocks (a catch block of `Throwable` shows up as `Throwable` and not `any`).

Note that the code within the `finally` block is actually compiled into multiple locations within the method: the end of the `try` block, the end of each `catch` block, and in the separate `finally` block.

The separate `finally` block begins with a store of the exception (because its really a catch block) and ends with a throw of the original stored exception. The throw is removed if the block ends with its own transfer of control command (eg, return, throw, etc). Within this `finally` block, any flow of control instructions (return, goto, throw) should probably be removed. The throw is somewhat problematical. Seemingly the original exception makes more sense, but an explicit throw may be what the programmer intended. And the CWE only refers to `return` statements. Another question is what to do with unchecked exceptions that happen in the finally block. These normally just get thrown (losing the original exception). Probably best for now to simply ignore these problems and just handle return.

Another question is whether or not we should remove returns that occur with a finally block inserted into the end of the catch block? This would seem to depend on how the catch block itself terminated. If it terminates with return or throw of its own, then that should take precedence. If it does not, then the one in the finally block should execute. If the return inside the `finally` block is conditional, then this is pretty easy to figure out. The value to be returned is stored in a local, the `finally` block is executed and when it is complete, the value to be returned is pushed from the local and then returned. If the conditional return is removed, everything will happen as expected.

If however, the return in the `finally` block is not conditional, then the return from the `capture` block is simply not emitted (it would be redundant). The save of the value to be

returned is still performed though (though this is clearly redundant as well). It should be possible to infer all of this and perform the correct return. But its not entirely clear that is worth the extra effort. In each case, a return and not a throw terminates the routine. It is certainly unexpected though that the value returned from the `catch` block is not actually returned though.

Note that in the case where the `catch` block rethrows an exception, the original `finally` block is triggered from the exception. If we correctly remove the returns from there, all should work as expected (ie, the rethrown exception will get thrown after the cleanup work is complete).

D.16 CWE-600 Failure to Catch All Exceptions in Servlet

The CWE says: *The Servlet does not catch all exceptions, which may reveal sensitive debugging information. When a Servlet throws an exception, the default error response the Servlet container sends back to the user typically includes debugging information. This information is of great value to an attacker. For example, a stack trace might show the attacker a malformed SQL query string, the type of database being used, and the version of the application container. This information enables the attacker to target known vulnerabilities in these components.*

There is one simple example and no references. The example simply shows an uncaught exception.

A straightforward way to address this would be to just add a `try/catch` block around the entry points to servlets. Those should be easy to find (since servlets have to implement `javax.servlet.Servlet`). That is probably a safe way to ensure that we both catch all of these and can correctly identify them to the test harness.

VIBRANCE takes this approach to catching servlet exceptions.

D.17 CWE-617 Reachable Assertion

The CWE says: *The product contains an assert() or similar statement that can be triggered by an attacker, which leads to an application exit or other behavior that is more severe than necessary.*

For example, if a server handles multiple simultaneous connections, and an assert() occurs in one single connection that causes all other connections to be dropped, this is a reachable assertion that leads to a denial of service.

There is one simple example (with an assertion) and 5 references

All of the references are to server assertions that can be triggered by certain erroneous input. Seemingly this is handled by VIBRANCE's server protections that ensure that servers don't exit when this happens. We could instrument assertions so that we know its an assertion and indicate that to the test harness, but other than that no other information is necessary. We could also do that check in the server exception handling code.

APPENDIX E: RESOURCE DRAINS

The following sections describe the CWEs listed under the weakness class 'Resource Drains'. We are considering the following approaches.

- **Virtual Close** - Some resources (primarily files) can be closed and re-opened without impacting the underlying application. This can ensure that the application never runs out of file handles regardless of the use pattern of the application.

  This would require that we instrument file opens and closes as well as reads and writes. When we near the maximum number of open file handles, we close the least recently used file (keeping track of the filename and current offset within the file). If the file is used again, we reopen it and continue operation. Hopefully this can be implemented on the lower level file streams without impacting all of the various users of the stream.
- **GC-close** - We can add code to close any resource when it is garbage collected (by using weak references and reference queues). It is not clear however that this is any better than the existing finalizers for the resource. We could perhaps improve the performance by running garbage collection after each request. It is certainly possible that frequently running garbage collection will have significant performance impacts though.
- **Cleanup Bad Request** - We instrument all opens and close and keep track of how many have been opened by each thread (request). If we cross the limit for maximum number of open handles, we terminate the request with the most open handles. We also augment the server protections to close any open handles on any request that terminates with an exception. It might be possible to extend this to close all open handles on all requests when they complete (if handles are not used in multiple requests).

  This could be combined with the *virtual close*. If any handles are used between requests they would then be automatically re-opened allowing us to be more aggressive about closing them.
  We could also possibly track how many file handles were left open by a request. If the number is small, they could be left alone. Or we could only start to close them when they get close to the maximum number of handles.
- **Static Close** - We can statically look for opens that do not have a close (or have a close that can be skipped) before all of the references to the handle are lost. We can modify the code to follow the `try-with` style introduced in Java 7. This has the advantage of solving problems directly, but will not work if a reference to the handle is kept or the code is actually trying to work with a large number of handles.
- **File Handles** - The proposed plan for file handles is to initially implement *cleanup-bad-request*. This will work for all of the CWEs and is relatively easy to implement. We can test this on real world applications and see if handles are used between requests. If handles do persist across requests, we could add *virtual-close* as well to ensure that our closes do not cause a problem.
- **Memory Consumption Daemon** - The problem occurs when a request uses up too much memory and thus brings the application as a whole past its limit (proposed by the T&E team as 75% of available memory). The basic concept is to track memory usage for each

thread. When the program as a whole exceeds the limit, the thread with the most memory usage is interrupted. This requires the following:

- Tracking memory usage by thread. This can be rough as a thread that is not constrained is likely to be very obvious. As a first approximation, we can simply track each object as it is allocated (estimating the size of the allocation by the fields in the object). This will not account for objects that get garbage collected, but in most cases, this is probably ok (we presume that most requests don't use a vast amount of memory that is quickly reclaimed). This could be augmented by static analysis to skip allocations that have a very short lifetime. The memory usage can be stored in a map indexed by thread id, or more efficiently in a data structure associated with the dummy variable passed to each method.
- Daemon that checks for excessive memory consumption. The JVM keeps track of total memory usage making it relatively easy to determine when the application as a whole is using too much memory. When that happens the daemon can determine which thread has used the most memory and interrupt it. Note that the daemon will have to check relatively often as memory allocation in a loop will blow up memory usage very quickly. Alternatively, we could check at each memory allocation itself.
- Add interrupt checks. Threads must check for interrupts manually (though some system calls will throw an InterruptedException). We will need to add such checks into the loop checks we already perform on tainted loops. If the thread is interrupted, we throw an InterruptedException (or possibly a StoneSoup exception of some sort). Checking in tainted loops should be sufficient as it will be difficult to make too many allocations outside of such a loop.

- **CPU Consumption Daemon** - Similar to memory consumption, the daemon can check for excessive CPU consumption and interrupt any requests (threads) that are taking up too much CPU time. The JVM tracks CPU usage by thread so we do not need to add extra instrumentation for this. And the checks for interrupts that we are already adding into tainted loops should be sufficient to interrupt a request efficiently.

E.1 CWE-400 Uncontrolled Resource Consumption (ok-Memory/CPU daemon)

The wiki says: *Applications may slow down or crash when resources such as network connections, CPU cycles, threads, memory, etc are not readily available. Resource consumption is dynamic, depending on the execution details of the application. The use of resources may depend on the inputs an application receives, or other external factors; software that assumes without protection that such conditions are within a reasonable range may be vulnerable to excessive resource consumption via malicious inputs or other manipulation.*

This is not to say a program may not use all the resources that are legitimately available to it. Systems may be dedicated to a single application in which it is desirable that usage be 100%. Rather, the application should not enter a fail state (denial of service) upon tainted input. It is acceptable to deny all or part of any processing meant to be triggered by tainted input, even those parts which may be legitimate.

That said, we note that for testing purposes the memory and thread usage are constrained by ulimits. This is so that the testing framework may continue to run even when the exploit condition is reached.

There are two detailed Java vulnerabilities: memory and CPU. See the entries for Memory Consumption Daemon and CPU Consumption Daemon in Section .

The wiki says: *Exploit Process: Malicious input causes are large number of memory allocations to be made.*

Required Preconditions: Software does not have adequate checks on resource consumption.

Expected Results / Exploit Condition: At some point, a resource cannot be provided, and the program will crash or exit early. A second exploit condition results if the program slows such that it's run-time is increased such that it does not complete in the time allotted. Though not explicitly tested for, this by inference includes the case where the program cannot handle legitimate inputs after having received a tainted input.

Areas of Concern: The JVM is configured with limits on memory that are lower than ulimit. The default memory allocation is 1/4 of the total system memory, but this can be configured by the user. What is a successful response by the PT team to an attempt to use too much memory? On its own, the JVM will raise an OutOfMemoryError and exit. This result in a DoS of the program, but preserves the performance of the overall system. If the program simply requests too much memory, how should the PT code recover from that without closing the program? T&E can write code that catches the OutOfMemory exception and does something graceful (based on knowledge of the meaning of the program). Is the exploit considered to have succeeded if the OutOfMemory exception is thrown? However, if the T&E code catches the OutOfMemory exception, how would the PT code know that that is invalid/inappropriate behavior?

Code Description: For example, allocate memory for each member of an input list. If the list is made long enough, memory will be exhausted.

CPU: The wiki says: *Exploit Process: Malicious input causes a large number of process threads to be created.*

Required Preconditions: Software does not have adequate checks on resource consumption. In the testing context, ulimits are set. This may change the point at which the application fails compared to an unrestricted system, but should otherwise have no effect.

Expected Results / Exploit Condition: At some point, a resource cannot be provided, and the program will crash or exit early. A second exploit condition results if the program slows such that its run-time is increased such that it does not complete in the time allotted. Though not explicitly tested for, this by inference includes the case where the program cannot handle legitimate inputs after having received a tainted input.

In addition to the basic CPU consumption protections outlined in the introduction, VIBRANCE also monitors thread creation and limits the number of threads that can be created by a single request. If the limit is exceeded, an exception is thrown that will normally terminate the request.

E.2 CWE-401 Improper Release of Memory Before Removing Last Reference (ok-n/a)

This is not applicable to Java.

E.3 CWE-440 Improper Resource Shutdown or Release (ok-n/a)

This is covered by other more specific CWEs.

E.4 CWE-459 Incomplete Cleanup (ok-misc)

The wiki says: *The code to return a resource to the system may depend on execution paths that are not always realized. When the assumed conditions for release are not met, the resource remains unavailable for reuse, causing a resource drain.*

The Java description is:

The wiki says: *Exploit Process: The program expects a certain number of temporary files to be created, and ultimately cleans up only that many. If the input is such that more files than that are created, some files remain after execution is ended.*

Required Preconditions: Logic of the program is flawed, such that cleanup sections are not invoked in every case. In the test environment, the ability of the program to write temporary files to disk.

Expected Results / Exploit Condition: Files remain in the target directory after execution of the program ends. These could represent sensitive information becoming available to a threat actor. These files may also exhaust disk space on the system.

Areas of Concern: It is not necessary to attempt to determine whether the user intended the files to be saved in some cases. The files in this example are created through methods that clearly indicate they are temporary, and hence should be cleaned up before program end.

Code Description: The number of files used in the file delete function is fixed, but the number of files created is variable, depending on user input.

We can address this by the following.

1. Change all instances of File.CreateTempFile to add the file to a list of temp files. If the program is a user application any files on this list are deleted when the program exits. If the program is a server, the file is deleted when the request is complete.

2. (Optional) Create a weak reference to each temp file. When the reference is garbage collected, delete the corresponding file. This may (in some circumstances) remove the files more quickly than waiting for the request to complete. If we implement this, we could also not remove items when a request finishes unless there are a large number of files on the list (reducing the chance that we break the program in some odd way).

E.5 CWE-674 Uncontrolled Recursion (ok-server protections)

The wiki says: *Recursion routines that make assumptions about the inputs that they act on may be vulnerable to exploitation by the use of inputs which cause the recursion to be much larger than anticipated. The program under these conditions may exhaust the resources available to it.*

Uncontrolled recursion will quickly use up the thread stack and throw an exception. VIBRANCE's server protections will catch this error and allow concurrent and future requests to execute normally. The amount of memory given to a thread for its stack is relatively small, so there should be no side effects on other activities.

We could add instrumentation to catch this directly, but there seems no benefit to duplicating the JVM's existing checks.

E.6 CWE-770 Allocation of Resources without Limits or Throttling (ok-n/a)

The wiki says: *Possible errors are entirely covered by CWE-400 (processes), CWE-834 and CWE-835 (CPU), CWE-774 (file handles), and CWE-789 (memory). T&E investigated possibility of resource exhaustion by using all sockets. However, this behavior was different for different operating systems and different machines, and could not be standardized for purposes of evaluation.*

It is good to note that there seems to be a complete list of resources to consider.

E.7 CWE-771 Missing Reference to Active Allocated Resource (ok-n/a)

This CWE is not applicable to Java.

E.8 CWE-773 Missing Reference to Active File Descriptor or Handle (ok-file handles)

The wiki says: *The software does not properly maintain references to file descriptors or handles, which prevents those file descriptors or handles from being reclaimed. If this happens often enough, the software can retain control of all (or the vast majority of) file handles, thereby causing DoS for the software or other programs on the same machine. This is distinct from CWE-774 and CWE-775 in that in this CWE, the program loses the references to its file descriptors, so it could not close them if it wanted to. The software should close any file descriptors it does have handles for, to avoid triggering CWE-775.*

How can the performer team know how many is "too many file descriptors"? There must be some contract between the program and the operating system regarding how many file descriptors are in use at once.

The exploit must not trigger additional CWEs, such as CWE-252 (Unchecked Return Value). To prevent this, the exploit must check file handles it gets to make sure they are valid before use.

Proposed contract: software must not open 75% or more of the available file handles on the system. But the PT software must intervene if the system tries to open one file handle beyond that.

What is success for the PT code? Closing file handles once they go out of scope and can no longer be referenced. All file handles used by the exploit code should be opened, and the PT code should silently close them once they are no longer reachable. Is it not acceptable to shut down the exploit code, because this would be result in denial of service.

See file handler approach.

E.9 CWE-774 Allocation of File Descriptors Without Limits (ok-file handles)

The wiki says: *The software allocates file descriptors based on user input, without imposing any restrictions on how many descriptors can be allocated. In this CWE, all file handles are ultimately closed. It is just that too many are in use at one time. In CWE-773, by contrast, the software loses references to file handles, so it cannot close them even if it wants to. In CWE-775, by contrast, the file descriptors are not closed after their effective lifetime*

Areas of Concern:

How can the performer team know how many is "too many file descriptors"? There must be some contract between the program and the operating system regarding how many file descriptors are in use at once.

The exploit must not trigger additional CWEs, such as CWE-252 (Unchecked Return Value). To prevent this, the exploit must check file handles it gets to make sure they are valid before use.

Proposed contract: software must not open 75% or more of the available file handles on the system. It is acceptable for the software to open ceiling(75% of file handles) - 1. But the PT software must intervene if the system tries to open one file handle beyond that.

What is success for the PT code? The exploit code is requesting a large number of file handles that it then goes on to use. Is it OK for the PT code to shut down the exploit code when it request too many file handles? How is this different from a denial of service attack? Should the PT code manage file handles behind the scenes, opening and closing them transparently so that the exploit code only has a small number that are actually open, even though it has requested a large number?

See file handler approach.

E.10 CWE-775 Missing Release of File Handle after Effective Lifetime (ok-file handles)

The wiki says: *The software does not release a file descriptor after its effective lifetime is over, i.e., after it is no longer needed. If this happens often enough, the software can retain control of all (or the vast majority of) file handles, thereby causing DoS for the software or other programs on the same machine.*

In CWE-774, by contrast, file descriptors are eventually released; it is just that too many file descriptors are requested at one time. In CWE-773, by contrast, the program loses the reference to an active file descriptor, meaning that the code cannot close them even if it wanted to. In this CWE, the program retains references to the file descriptors, but does not close them all, resulting in a resource drain.

Failing to close a single file descriptor is an error, but does not cause significant system impacts. In order to trigger this CWE, a substantial number of file descriptors must be held by the program. How can the performer teams know what is a "substantial number"? There must be some contract between the software and the operating system that specifies the allowed number of file descriptors. If the software violates that contract, then the operating system should respond.

The exploit must not trigger additional CWEs, such as CWE-252 (Unchecked Return Value). To prevent this, the exploit must check file handles it gets to make sure they are valid before use.

Proposed contract: software must not open more than ceiling(75% of the file handles).

Since these are file handles that are being retained after their effective lifetime, it is plausible that the PT code could determine that the file handles are no longer needed and shut them automatically.

See file handler approach.

E.11 CWE-789 Uncontrolled Memory Allocation (ok-memory daemon)

The wiki says: *The program allocates memory based on input provide by the user. It does not validate, or improperly validates, the amount of memory requested, so it allows arbitrary amounts of memory to be allocated. This may either cause a crash in the current process or cause problems in other processes on the same machine, resulting in a DoS.*

Areas of Concern:

What counts as "too much memory"? On a dedicated machine, it is acceptable to use a far higher percentage of available memory than on a shared machine or server. There must be some

kind of contract between the program and the operating system about how much memory it is allowed to use. If the program violates that limit, the performer team should shut it down.

If the program does not check for that error or null pointer, then the code will result in a different CWE, such as CWE-248 (Uncaught Exception) or CWE-252 (Unchecked Return Value). T&E must be careful to not trigger additional CWEs.

Proposed contract: the process will use no more than 75% of total available memory. This limit is set so that the test software can continue to run even during the exploit condition. The standard test configuration includes 8G of memory, so the process must use no more than 6G of memory. To prevent triggering other CWEs for uncaught exceptions or unchecked return value, the exploit will check memory allocations to see if they are valid, and if the memory allocation is not valid, they will further check the errno. If it finds that the errno is ENOMEM, the test program will log an error. To successfully defeat this exploit, the PT code must not trigger this logging case.

What is acceptable behavior for the PT code in this CWE? Is it acceptable for the PT code to simply close the application? This prevents the logging, but also results in a denial of service by shutting down the program. If the PT code can determine that the uncontrolled memory allocation is never actually used, then it could simply avoid making the allocation. However, what if the allocation is used? What is acceptable behavior for the PT code in this case?

See Memory Consumption Daemon (Section).

E.12 CWE-834 Excessive Iteration (ok-CPU Daemon)

The wiki says: *If the user inputs data that controls the number of iterations a loop performs, and the system performs no checks on that input, the program can be compelled to use excessive CPU or memory resources. The extreme example is an infinite loop, but a loop does not have to be infinite in order to consume so much resources that it impacts the software or underlying system.*

Areas of Concern:

How can the performer teams tell whether the code is doing a useful (if time-intensive) calculation, or if it is doing "excessive" iterations? This seems to require some contract between the software and the operating system about how much resources it will use, and a way of detecting that the software has violated that contract.

Proposed contract: the program will take less than 10 minutes of clock time to execute to completion. If the program is in an excessively long loop, the PT software should recognize that and cause the loop to terminate, allowing program execution to continue after the loop. It is not enough to halt the program in the middle of the loop; the code should continue after the loop.

Possible concern: Is this a valid agreement? Perhaps the programmer wants that excessively long loo to complete. Also, at what point during an infinite loop should the loop be terminated? Is the end of a single loop pass the correct place to terminate? Will this leave the program data in a consistent state?

See CPU Consumption Daemon.

E.13 CWE-835 Infinite Loop (ok-CPU Daemon)

This is essentially the same as CWE-834 for our purposes. We don't need to do anything different here.

APPENDIX F: CONCURRENCY HANDLING

The following sections describe the CWEs listed under the weakness class ‘Concurrency Handling’. CWEs that are children of the named CWEs are placed in sections under their parents.

We considered the following approaches. Note that the term ’Data Race’ is used to refer to any situation where a lock is needed and not simply data races.

- **ok-TOUTOC-files** - We can probably catch most errors on files by keeping track of checks done on files. If a file that is checked for some attribute is later used, we can ensure that the attribute is still true at the time of open. If it is not, we can throw an exception. This is discussed in some detail in Section .
- **Data Race - Add Synchronization** - One approach to data races is to add synchronization code where appropriate. To do this we would want to identify code that could possibly be accessed from multiple threads that might have race conditions. The easiest fix would be to add synchronized annotations to methods. For other situations, we could add individual locks. See section for more information.
- **Data Race - Dynamic Check** An alternative approach would be to try and detect data races dynamically. This item discusses that possibility. Our plan, however, is to add synchronization.

  We could try to add a dynamic check to all updates to see if there was an out of order access. An out of order access occurs if either of the following sequences occurs:

  1. T1-read T2-read T1-write T2-write
  2. T1-read T2-read T2-write T1-write

  A read can always be considered safe. A write (that is an update) is unsafe if it occurs after a write from another thread. The key problem is determining if a write is an update. If the write occurs after a read, an update seems like a reasonable assumption. However, it is not clear how to even semi-efficiently keep track of when a thread has read a particular field. It would be easy to keep track of the last thread to read a field, but not all of the threads that have read the field.

  Consider a field that is set from an external source (file, database, network etc). It may be written from multiple threads and read from multiple threads but not ’updated’ by any thread. Without trying this approach it is hard to tell if this would be a common problem or if there are ways (such as some level of static analysis) to avoid it. Seemingly a points-to based analysis could detect field that are updated by a thread and the check could be restricted to those fields.

  Another possibility might be to use a bit field to keep track of reads. Each thread is assigned a bit. A long could support up to 64 threads at a time (which seems sufficient). Optimally, though, a thread’s status would get reset on each request (a read in one request followed by a write in another is not an update). But there isn’t any obvious way to clear the read bit set on a previous request. It would be possible to provide a new unique thread id for each request, but it is not clear how to store them efficiently. Keep a list of reads at each field seems expensive. One could create an LRU of the most recent reads, but this

would be relatively expensive to maintain and would need to have a relatively small size (2, 5, 10?)

It is not clear, however, whether or not resetting the read bits is necessary for each request. We will have to try this on real applications and see how it works.

One way to try and detect this would be to add three pieces information to each field for which this could occur:

1. **last-write** - The thread that last wrote the field
2. **last-read** - the thread that last read the field
3. **rw** - Whether or not the last access was a read or write

When a write occurs, it is illegal if *last−write*!=*thread*

```
(last-write != thread) &&
```

- **Signal Handlers** - There are a number of CWEs that involve signal handlers. These are explicitly called out because a writer of a signal handler may not realize that they can occur concurrently and should be reentrant. In Java, signal handles are not officially supported and the T&E suggests that shutdown handlers will be used instead.

  In both cases it seems reasonable to simply add locks that ensure that the signal/shutdown handlers do not interleave. There should be no conditions under which interleaving is required for correct operation.

  We will thus add a single lock around all shutdown hooks and signal handlers. Shutdown hooks are threads. The `run()` method is called when the JVM starts to shutdown. All application classes that extend thread will be instrumented to add an instance field (`is_shutdown_handler`) and code to the run method that implements a lock around the run method if `is_shutdown_handler` is true:

```
public void run() {
  if (is_shutdown_handler) {
    StoneSoup.shutdown_handler_lock.lock();
  try {
    ...
  } finally {
    if (is_shutdown_handler)
      StoneSoup.shutdown_handler_lock.unlock();
  }
}
```

  Note that this adds the overhead of the try/finally to every run method. There are other approaches we could use if this is an issue. For example, we could create a `user\s\do5(r)un()` method that contains the contents of `run()`

```
public void run() {
  if (is_shutdown_handler) {
    synchronized (StoneSoup.shutdown_handler_lock) {
      user_run();
    }
  } else {
    user_run();
  }
}
```

This is somewhat simpler to understand, but adds another level of call to the application stack. We could also just duplicate the entire contents of `run()` rather than creating `user_run()`.

- **Deadlocks** - Deadlocks occur when threads obtain locks in a cycle or a thread does not release a lock when it is finished using it. The second case does not happen with the synchronized keyword which guarantees a lexical scope for the lock, but can certainly happen for direct uses of locks (which can end up being unbalanced if there is a coding error).

  We have two basic options. First, we can change locks that might potentially deadlock so that they can timeout or are interruptable. If a deadlock occurs (or sufficient time passes), we can either simply proceed without the lock or throw an exception. Throwing an exception seems like the right choice given our server protections.

  We can use a deadlock daemon to detect deadlocks and resolve them. The JVM has methods to check for deadlocks and return which threads are blocked. Our daemon checks periodically and then interrupts one of the deadlocked threads. Since all locks have been changed to be interruptable, this will cause one thread to break out of its locks and relieve the deadlock. The interrupted thread will be handled by the server protections.

  We also want to ensure that a request doesn't end without releasing all of its locks. We will keep a weak identity hash map on all of the locks acquired per thread. The map will be cleared at the beginning of each request and checked at the end. At the end of the request, any locks that are still held will be released. This requires us to instrument any lock acquisitions to add them to the map.

  Alternatively, we could instrument both lock and unlock operations. That would allow us to remove from the map any lock that was completely unlocked. But this seems like it would both be more work to implement (more calls to instrument) and possibly slower (more map manipulations)

  Note that `ThreadInfo` contains all of the locks currently owned by a thread. But unfortunately, it only returns the `identityHashCode` of the lock and not the lock itself. This does not allow the lock to be released. Presuming, however, that a single request does not obtain a large number of locks, the overhead of this check should be relatively small. It is possible that we might be able to identify locks that are lexically scoped (by the proper use of try/catch/finally blocks) and avoid adding them to the map. Or we could possibly change the implementation of whatever is keeping track of the

information in ThreadInfo to keep the object as well. Looking at the 1.6 implementation, it looks like this would be very easy. ThreadInfo already contains each of the lock objects, it just chooses not to share them. Perhaps there is some non-obvious reason for this. Note that the overhead of getting the ThreadInfo for a thread may be significant.

Each CWE is also marked with a shorthand indicating the number of examples, references, and child CWEs (examples-references-childCWEs).

F.1 CWE-362 Race Condition (2-12-5) (ok-n/a)

The CWE says: *The program contains a code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.*

This can have security implications when the expected synchronization is in security-critical code, such as recording whether a user is authenticated or modifying important state information that should not be influenced by an outsider.

A race condition occurs within concurrent environments, and is effectively a property of a code sequence. Depending on the context, a code sequence may be in the form of a function call, a small number of instructions, a series of program invocations, etc.

A race condition violates these properties, which are closely related:

Exclusivity - the code sequence is given exclusive access to the shared resource, i.e., no other code sequence can modify properties of the shared resource before the original sequence has completed execution.

Atomicity - the code sequence is behaviorally atomic, i.e., no other thread or process can concurrently execute the same sequence of instructions (or a subset) against the same resource.

A race condition exists when an "interfering code sequence" can still access the shared resource, violating exclusivity. Programmers may assume that certain code sequences execute too quickly to be affected by an interfering code sequence; when they are not, this violates atomicity. For example, the single "x++" statement may appear atomic at the code layer, but it is actually non-atomic at the instruction layer, since it involves a read (the original value of x), followed by a computation (x+1), followed by a write (save the result to x).

The interfering code sequence could be "trusted" or "untrusted." A trusted interfering code sequence occurs within the program; it cannot be modified by the attacker, and it can only be invoked indirectly. An untrusted interfering code sequence can be authored directly by the attacker, and typically it is external to the vulnerable program.

The wiki says: *This CWE is too general to be tested. Specific race conditions are contained in other CWEs*

The CWE further claims that race conditions can lead to resource consumption issues, denial of service, and accessing confidential data. Its not entirely clear how resource consumption and confidential data work.

There are two examples. The first is a banking application that doesn't lock the account balance. If there are multiple simultaneous updates, the resulting balance could be incorrect. The second example has a lock request whose result is not checked for errors.

There are 12 references. Most lead to crashes. Two are TOCTOU race conditions that could allow a user to trick a privileged program into accessing a file the user does not have access to. A number of others are related to privileged programs (2008-5044)

F.2 CWE-363 Race Condition Enabling Link Following (ok-TOCTOU)

The wiki says: *The software checks the status of a file or directory before accessing it, which produces a race condition in which the file can be replaced with a link before the access is performed, causing the software to access the wrong file.*

Although a developer may believe that there is a small amount of time between the check and the access of a file, a race condition still exists. An attacker may be able to cause the software to slow by consuming memory or resources resulting in a larger time window. As a testament to the difficulty of solving this problem, Dean and Hu thought the problem was solved in their paper Fixing Races for Fun and Profit: How to use access(2). This solution was later shown not only to not work, but also that there is no situation where the use of access in C is ever advised as shown by Fixing Races for Fun and Profit: How to abuse atime. This attack can easily be extrapolated to any computer language where a check of the file is made before accessing it.

In `363-Java-1`, they describe the exploit process as: The software provides some mechanism to allow notification that the software has accessed the file, but has not tried to open, read from, or write to it. Once this notification is received, replace file with a link.. A more sophisticated exploit would be to implement a file maze. There is an example of a possible attack there.

This would appear to be solved by our TOCTOU approach. In particular, checking to see if a file is a link.

F.3 CWE-364 Signal Handler Race Condition (ok-signal handlers, n/a)

The wiki says: *Signal handlers support asynchronous actions which creates an environment for race conditions. If shared state or non-reentrant functions are used in a signal handler there is a possibility of a race condition occurring. Also, the use of the same signal handler for multiple signals can cause a race condition.*

Unfortunately, the example code for this (`364-Java-1`, using signal handlers) does not seem to define a race condition. The first signal handler simply removes a static variable that the second signal handler requires.

Note that they acknowledge that Java doesn't really support signals, but indicate that shutdown hooks could be used instead.

This is resolved by the signal-handlers approach detailed in the introduction.

Note that T&E team decided *not* to include this and similar CWEs for java since signal handlers are not supported by Java.

F.4 CWE-365 Race Condition in Switch (ok-n/a)

The wiki says: *Code contains a switch statement where the switched variable can be modified while the switch statement is executing, resulting in unexpected behavior. T&E does not believe this CWE is possible in modern compilers. See language-specific discussion below.*

F.5 CWE-366 Race Condition within a Thread (ok-data race)

The wiki says: *Two threads which share a resource or code sequence run the risk of using that resource or code segment while it is in an unintended state. For example, consider code where two threads access and modify the same variable in shared memory. If the code is executed such that the variable is modified by thread A and is then modified by thread B before thread A retrieves the value for use, the variable will contain a value that is not expected by thread A. If there is more than one thread accessing an unprotected (unsynchronized) shared location, there is a potential for a race condition.*

Seemingly this is just a standard race condition over a variable.

F.6 CWE-367 Time of Check Time of Use Race Condition (ok-data race)

The wiki says: *The software checks the state of a shared resource before using it. During the time between the check and use, another thread could modify the shared resource.*

As with all race conditions, timing is critical and the condition will not always present itself, and will require multiple threads and runs (within a loop). To exploit the TOCTOU condition, we would check a resource (file, variable, etc.) and then use it. Although not necessary, putting a sleep or some code between the check and the use will assist in creating a window of opportunity to modify the tested resource before it is used.

The example code is over a variable that is checked for null and then operated on. The variable is shared and can be set to null before it is accessed.

Essentially this is a race condition without an update. The check serves as the read and subsequent use as the write. Our proposed data race approach should handle this.

F.7 CWE-412 Unrestricted Externally Accessible Lock (ok-trylock)

The CWE says: *The software properly checks for the existence of a lock, but the lock can be externally controlled or influenced by an actor that is outside of the intended sphere of control*

The examples and references all deal with file locks that can possibly be obtained by an attacker running on the same system. From the CWE description this is not an issue for locks within the program.

The wiki, however, presumes that an internal lock is used. They are concerned about this, but the example is all internal.

For external locks a reasonable fix would seem to be to stop and throw an exception after a certain amount of wait time. File locks are implemented in `java.nio.channels.FileLock`. The FileChannel class has both a `lock()` and `trylock()` methods. Seemingly we could convert all calls to `lock()` to a function of our own that uses `trylock()` to create a timeout.

F.8 CWE-414 Missing Lock Check (ok-data race)

The wiki says: *Lock on a shared resource is not checked prior to use of the resource.*

The wiki notes that there may be no real difference between this and CWE-366. Not adding a lock is essentially a data race.

F.9 CWE-479 Signal Handler use of a Non-reentrant Function (ok-signal handlers, n/a)

The wiki says: *The program defines a signal handler that calls a non-reentrant function. Register 2 or more shutdown hooks that both make calls to a function that is non-reentrant. That is, if the function is halted in the middle, and then starts in another thread, the behavior is incorrect or damaging. When given the shutdown signal, the JVM will start all registered shutdown hooks at the same time and allow them to run concurrently. Since both shutdown hooks calls the same non-reentrant function, this will cause a race condition that may result in bad behavior.*

The easiest solution to this problem would seem to make all shutdown hooks execute in some order (by having each obtain a lock before proceeding). If the shutdown hooks contain large delays or if they communicate externally in some way in an effort to force a deadlock, this could cause a problem). There would seem to be very little need for multiple shutdown hooks (or signal handlers) to be running concurrently (except because of T&E additions). See the signal-handling plan in the introduction for details.

Note that T&E team decided *not* to include this and similar CWEs for java since signal handlers are not supported by Java.

F.10 CWE-543 Singleton Pattern (ok-data race)

The CWE says: *The software uses the singleton pattern when creating a resource within a multi-threaded environment.*

The example is:

```
private static NumberConverter singleton;
public static NumberConverter get_singleton() {
  if (singleton == null) {
    singleton = new NumberConverter();
  }
  return singleton;
}
```

This is essentially another TOCTOU race condition (see CWE-367). The difference here is that rather than reading the checked value, the checked value is set.

This could be fixed by applying our base data race check. Note that while it seems that `Double-Checked locking` would resolve this problem it does not in practice unless the singleton is declared as volatile. The easiest solution would seem to just mark the function as synchronized.

F.11 CWE-558 Use of getlogin() (ok-n/a)

This vulnerability is specific to C.

F.12 CWE-567 Unsynchronized access to Shared Data (ok-data race)

The wiki says: *Unsynchronized Access to Shared Data in a Multi-threaded Context. The product does not properly synchronize shared data, such as static variables across threads, which can lead to undefined behavior and unpredictable data changes. The shared data should not be a singleton pattern, because singletons are covered by CWE-543.*

This seems the same as CWE-366. The shared data may be subject to a data race. Some of the comments in the wiki seem to indicate that they think this is specific to static data.

F.13 CWE-572 Call to Thread run() instead of start() (ok-thread run)

The wiki says: *Program calls the Thread's run() method instead of start() which does not cause a new thread to be created. In most cases a programmer creates a new thread to run in its own*

thread. The run() method does not create a new thread, but instead runs the run() method in the current thread.

If we presume that each call to `run()` really meant `start()` this is easy to fix. Are there circumstances under which the programmer really means to call `run()`? How could we identify those?

After investigating real-world applications, we concluded that programs do *not* call the run method purposefully. VIBRANCE thus changes these to call start() instead.

F.14 CWE-609 Double Check Locking (ok-data race)

The wiki says: *The program uses double-checked locking to access a resource without the overhead of synchronization, but the locking is insufficient. The programmer checks to see if a resource has been initialized. The programmer then grabs a lock, checks again to see if the resource has been initialized and performs initialization if it has not yet occurred.*

Though it seems like it should work, this construct does not work under all circumstances:

```
class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
    }
  }
```

This is essentially a more subtle TOCTOU issue where the returned helper can be improperly initialized when returned. See `f`or more information. This can be fixed by marking helper as `volatile` or by simply synchronizing the method.

The trick here is identifying this pattern. The basic pattern to look for would seem to be:

```
if (helper != null)
  return helper
```

Seemingly any method that contains an if-check on a variable and later sets that variable should be synchronized. This should be an easy decision to make statically and our data-race approach should thus resolve this.

F.15 CWE-663 Use of a Non-reentrant Function (ok-data race)

The wiki says: *The software calls a non-reentrant function in a concurrent context in which a competing code sequence may have an opportunity to call the same function or otherwise influence its state.*

This is similar to a data race except that the lock needs to cover all uses of the shared variable and not simply its updates. A non-reentrant function can be identified as one that both reads and writes a static variable. There are two possible fixes. One is to synchronize the function itself. The second is to replace the static variables with local ones. The second approach would be more efficient, but only works if the static variable in question is only used within the function.

We will implement the locking approach first and consider using local variables if there appear to be efficiency issues with locking.

F.16 CWE-764 Multiple Locks of a Critical Resource (ok-deadlock)

The wiki says: *A thread locks the resource more times than expected, resulting in a failure to unlock the resource and a denial of service.*

Using an instance of a Lock implementation, thread A locks a number of times that depends on user input. With tainted input thread A locks the resource more times than it unlocks it. Then thread B attempts to acquire the lock, and cannot do so.

A `ReentrantLock` keeps track of how many times it has been acquired. the lock will still be held if it is released fewer times than it was acquired.

The basic deadlock approach solves this problem. The deadlocked code will timeout and throw an exception. If any request ends with additional locks held, those locks will be deallocated. This ensures that the problem will not be perpetuated into future requests (and that any locks not released as part of the exception handling will be released).

F.17 CWE-765 Multiple Unlocks of a Critical Resource (ok-misc)

The wiki says: *The software unlocks a critical resource more times than intended, leading to an unexpected state in the system. If the lock is a semaphore, the extra calls to unlock will increase the count for the number of available resources, likely resulting in a crash or unpredictable behavior when the system nears capacity.*

The wiki says: *Use of the lock/tryLock and unlock methods of the Lock interface. Create a lock,and then attempt to unlock it a number of times that depends on user input. From the Java docs for ReentrantLock, "If the current thread is not the holder of this lock then IllegalMonitorStateException is thrown."*

For Lock, the only bad result of too many unlocks is a thrown exception. We could simply swallow that exception (probably a reasonable choice), or we could just let our normal error handling code handle it. Swallowing the exception seems to make the most sense. There is no obvious downside to continuing if the lock is unlocked more times than necessary.

F.18 CWE-820 Missing Synchronization (ok-data race)

The wiki says: *The software uses a shared resource in a concurrent manner, but does not attempt to synchronize access to the resource. T&E must avoid child CWE-543 (Use of Singleton Pattern without Synchronization in Multi-threaded Context) and CWE-567 (Unsynchronized Access to Shared Data in a Multi-threaded Context). This mean that the shared resource must not be data.*

This seems to be the same as a data race.

F.19 CWE-821 Incorrect Synchronization (ok-data race)

The wiki says: *Two separate threads want to used a shared resource. Rather than using standard thread synchronization primitives, they use home-grown, incorrect ones, thereby not obtaining the synchronization they need. Alternately, the threads use standard synchronization primitives, but fail to use the correct locks to synchronize on. When the threads run, their code may be interleaved, leading to incorrect results, denial of service, thrown exceptions, or exposure of sensitive data.*

Code Description: Have a technique such as setting a shared variable 'have_resource'. One thread polls on that while the other has the resource. The method is subject to race conditions. As long as our data race approach is not confused by extraneous locks, the standard approach should work here as well. Incorrect synchronization should look just like missing synchronization to our analysis.

F.20 CWE-828 Signal Handler not Asynch Safe (ok-data race)

The wiki says: *The software defines a signal handler that contains code sequences that are not asynchronous-safe.*

T&E must be careful to not overlap with either parent CWE-364 (Signal Handler Race Condition) or child CWE-479 (Signal Handler Use of Non-Reentrant Function). This means the exploit cannot make use of non-reentrant functions, and instead must develop exploit-specific asynchronous-unsafe data structures. In addition T&E should take care not to overlap with CWE-831 (Signal Handler Function Associated with Multiple Signals), so this CWE must not use the same signal handler for multiple signals.

Exploit Process: Software uses global data structures that are also used by signal handler. Signal handler expects global data structures to be in standard state. Since the software is

altering the global data structures, the signal handler may be called while the global data structures are not in the standard state.

Code Description: Implement a signal handler. After the signal, in the main thread modify a data structure used by the handler. Depending on the timing of the modification (user input dependent), the change of state has different effects.

The T&E team decided *not* to include this and similar CWEs for java since signal handlers are not supported by Java.

F.21 CWE-831 Signal Handler Function Associated with Multiple Signals(ok-signal handlers)

Seemingly this is the same as CWE-364

The T&E team decided *not* to include this and similar CWEs for java since signal handlers are not supported by Java.

F.22 CWE-832 Unlock of a Resource that is not Locked (ok-misc)

The wiki says: *Exploit Process: A Lock object is created. Depending upon user input the object is locked. The object is unlocked.*

Required Preconditions: The code must have access to a locking mechanism, and the locking mechanism must have an adverse reaction to unlocking an already unlocked mutex.

Expected Results / Exploit Condition: Some lock implementations do not have an adverse reaction to unlocking an already unlocked mutex. However, some (such as Reentrant Lock) will throw a RuntimeException, such as IllegalMonitorStateException if the user attempts to unlock a lock that is not currently locked, resulting in a DoS.

Essentially this seems the same as CWE-765. We can just ignore exceptions on such unlocks (or let them get handled naturally).

F.23 CWE-833 Deadlock (ok-deadlock)

The wiki says: *The software contains multiple threads or executable segments that are waiting for each other to release a necessary lock, resulting in deadlock. An exploit could manipulate a deadlock, causing a denial-of-service.*

The example is a classic deadlock from an Oracle Java Tutorial using `synchronized`.

This is handled by our deadlock approach.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	DESCRIPTION
API	Application Programming Interface
ASLR	Address Space Layout Randomization
BCEL	Bytecode Engineering Library
CHA	Class Hierarchy Analysis
CHIF	CodeHawk Internal Form
CWE	Common Weakness Enumeration
DBMS	Database Management System
DTA	Declared Type Analysis
HTTP	Hypertext Transfer Protocol
JDBC	Java Database Connectivity
JDK	Java Development Kit
OS	Operating System
RTA	Rapid Type Analysis
SQL	Structured Query Language
SSA	Static Single Assignment
STONESOUP	Security Taking On New Executable Software Of Uncertain Provenance
T&E	Test & Evaluation
TOCTOU	Time-Of-Check-to-Time-Of-Use
VIBRANCE	Vulnerabilities in Bytecode Removed by Analysis, Nuanced Confinement, and Diversification
VTA	Variable Type Analysis
XML	Extensible Markup Language