A Security Policy Oracle: Detecting Security Holes Using Multiple API Implementations

Varun Srivastava
Yahoo!
varun_iiit@yahoo.co.in

Michael D. Bond
The Ohio State University
mikebond@cse.ohio-state.edu

Kathryn S. McKinley Vitaly Shmatikov
The University of Texas at Austin
{mckinley,shmat}@cs.utexas.edu

Abstract

Even experienced developers struggle to implement security policies correctly. For example, despite 15 years of development, standard Java libraries still suffer from missing and incorrectly applied permission checks, which enable untrusted applications to execute native calls or modify private class variables without authorization. Previous techniques for static verification of authorization enforcement rely on manually specified policies or attempt to infer the policy by code-mining. Neither approach guarantees that the policy used for verification is correct.

In this paper, we exploit the fact that many modern APIs have *multiple, independent* implementations. Our flow- and context-sensitive analysis takes as input an API, multiple implementations thereof, and the definitions of security checks and security-sensitive events. For each API entry point, the analysis computes the security policies enforced by the checks before security-sensitive events such as native method calls and API returns, compares these policies across implementations, and reports the differences. Unlike code-mining, this technique finds missing checks even if they are part of a rare pattern. Security-policy differencing has no intrinsic false positives: implementations of the same API *must* enforce the same policy, or at least one of them is wrong!

Our analysis finds 20 new, confirmed security vulnerabilities and 11 interoperability bugs in the Sun, Harmony, and Classpath implementations of the Java Class Library, many of which were missed by prior analyses. These problems manifest in 499 entry points in these mature, well-studied libraries. Multiple API implementations are proliferating due to cloud-based software services and standardization of library interfaces. Comparing software implementations for consistency is a new approach to discovering "deep" bugs in them.

Categories and Subject Descriptors D. Software [*D.2 Software Engineering*]: D.2.4 Software/Program Verification; K. Computing Milieux [*K.6 Management of Computing and Information Systems*]: K.6.5 Security and Protection

General Terms Languages, Security, Verification

Keywords Security, Authorization, Access Control, Static Analysis, Java Class Libraries

PLDI'11, June 4–8, 2011, San Jose, California, USA.
Copyright © 2011 ACM 978-1-4503-0663-8/11/06...$10.00

1. Introduction

Demand for secure software is increasing, but ensuring that software is secure remains a challenge. Developers are choosing memory-safe systems [9, 39] and languages such as Java and C# in part because they improve security by reducing memory-corruption attacks. Even memory-safe systems, however, rely on the *access-rights model* to ensure that the program has the appropriate permissions before performing sensitive actions. Unfortunately, even experienced developers find it difficult to specify and implement access-rights policies correctly. Consequently, semantic mistakes—such as missing permission checks which enable malicious code to bypass protection—have become a significant cause of security vulnerabilities [26].

This paper presents a new approach to finding security vulnerabilities in software libraries, also known as Application Programming Interfaces (APIs). We leverage the increasing availability of multiple, independent implementations of the same API. By design, these implementations must be interoperable and must implement semantically consistent security policies. A security policy in the access-rights model consists of a mapping between *security checks*, such as verifying permissions of the calling code, and *security-sensitive events*, such as writing to a file or the network. We exploit this consistency requirement to build a *security policy oracle,* which accurately (i) derives security policies realized by each implementation and (ii) finds incorrect policies.

Previous static analysis techniques for finding security vulnerabilities either rely on manual policy specification and/or verification, or infer frequently occurring policies by code-mining. For example, Koved et al. derive a policy and then require programmers to check by hand that permissions held at all sensitive points in the program are sufficient [22, 28]. Sistla et al. take as input manually specified check-event pairs and verify that all occurrences of the event are dominated by the corresponding check [30]. These approaches are error-prone and limited in expressiveness. There are hundreds of potential security-sensitive events and 31 potential checks in the Java Class Library; some policies involve multiple checks and some checks do not always dominate the event. It is easy to overlook a security-sensitive event or omit a rare check-event pair in a manual policy. Koved et al. and Sistla et al. reported no bugs for the Sun Java Development Kit (JDK) and Apache Harmony implementations of the Java Class Library. By contrast, our analysis uncovered multiple, exploitable security vulnerabilities in the same code.

Another approach is code-mining, which infers policies from frequently occurring patterns and flags deviations as potential bugs [10, 14, 35]. These techniques fundamentally assume that the same pattern of security checks and security-sensitive operations occurs in many places in the code, and they are thus likely to miss vulnerabilities that violate a unique policy—such as the Harmony bug described in Section 2. Frequent patterns may or may not represent actual policies, resulting in false positives and negatives. Code-mining techniques face an inherent tradeoff between coverage and the number of false positives. As the statistical threshold is lowered to include more patterns, they may find more bugs, but the number of false positives increases since deviations from rare patterns can be mistakenly flagged as bugs.

Cross-implementation consistency as a "security policy oracle." Our key idea is to use any inconsistency in security semantics between implementations of the same API as an oracle for detecting incorrect policies. Unlike approaches that rely on explicit policies or frequent patterns, this technique completely avoids the need to determine whether the policy is *correct*, replacing it by a much easier task of determining whether two policies are *consistent*.

Our use of consistency checking is very different from the code-mining approaches. In our case, *there are no benign inconsistencies*! Any semantic inconsistency between two implementations of the same API is, at the very least, an interoperability bug or, in the worst case, a security hole. In theory, policy differencing has no intrinsic false positives. In practice, imprecision in conservative static analysis results in a very small number of false positives. Furthermore, our technique can discover a missing permissions check *even if this type of check occurs in a single place in the code*. It may produce false negatives if exactly the same semantic bug occurs in all implementations of a given library routine. This case is unlikely if the implementations are developed independently, as is the case for our main test subject, the Java Class Library.

Finding semantic inconsistencies with static analysis. Our analysis takes as input two or more Java API implementations, security-sensitive events, and security checks. Our definitions of security-sensitive events and checks directly reflect the Java security model. Security-sensitive events include calls to the Java Native Interface (JNI), which is the only mechanism by which Java code communicates to the network, file system, or other external devices. Since user code may change the internal state of the Java Virtual Machine (JVM) using the Java Class Library, we include API returns in the set of security-sensitive events. Application code should not be able to reach security-sensitive events without proper permissions. In the Java security model, security checks verify permissions held by user code by calling methods in the `SecurityManager` class.

For every API entry point, we take all implementations and, for each one, produce context-sensitive policies using flow- and context-sensitive interprocedural analysis together with interprocedural constant propagation. The policies specify which checks *may* and *must* precede each security-sensitive event. We compare the policies for all events relevant to this entry point and report any differences.

Case study: The Java Class Library. We demonstrate the effectiveness of our approach by applying it to the Java Class Library. This prominent, large, widely used API has multiple, independent implementations: Sun JDK, GNU Classpath, Apache Harmony, and several others. For example, JDK was announced in 1996 and is very broadly deployed. It is the product of hundreds of programmers and its source code has been actively studied since Sun published it in 2007. Different implementations of the Java Class Library API are intended to be fully interoperable.

Together, these implementations comprise over 1.7 million lines of code. Any context-sensitive analysis must be concerned with efficiency. Ours uses context memoization [38]. Analyzing each library takes 20 minutes or less, which is likely to be acceptable as part of testing. The three pairings share approximately 4,100 entry points, and over 230 of these methods perform security checks. Our analysis derives between 8,000 and 16,000 *may* and *must* security policies per API.

Our analysis is very precise. It produces only 3 false positives, while finding 20 new security vulnerabilities (6 in JDK, 6 in Harmony, 8 in Classpath) and 11 interoperability bugs that manifest in 499 API entry points. Example vulnerabilities include opening network connections and loading libraries without properly checking permissions of the calling code. We reported all vulnerabilities found by our analysis to the developers of the respective libraries, who accepted and immediately fixed many of them [31–33].

In summary, our analysis uncovered multiple security vulnerabilities in three mature implementations of the Java Class Library, with a minimal number of false positives. This paper demonstrates that our approach (i) scales to industrial implementations of real-world software libraries and (ii) is capable of discovering "deep" bugs even in well-studied, tested, widely used code. Although we describe our design and implementation with respect to the Java programming language, our approach is applicable to other languages, such as C#, that use a similar access-rights model.

Current software engineering trends are likely to increase the applicability of comparative API analysis. For example, the proliferation of distributed, multi-layer software services encourages separation of APIs from their implementations, and thus multiple implementations of the same API. Many platforms are adopting the Software-As-A-Service (SaaS) model [1, 16, 20, 29], which encourages multiple implementations of the same functionality and thus provides the input material for our "security policy oracle." Widely used APIs sometimes have an open-source version, which programmers can use as a reference policy oracle even when their own code is proprietary. Even if all implementations of the same API are proprietary, developers may be willing to share security policies with each other without sharing the actual code.

2. Motivating Example

Figure 1 motivates our approach with an example from the JDK and Harmony implementations of the Java Class Library. In Figure 1(a), the JDK implementation of `DatagramSocket.connect` calls either `checkMulticast`, or `checkConnect` and `checkAccept` before connecting to the network with a JNI call inside the method invocation at line 16. For clarity of presentation, this and all other examples elide unrelated code and the test for a non-null security manager required by all security checks. In Figure 1(b), the Harmony implementation calls either `checkMulticast`, or `checkConnect` before the equivalent method invocation at line 11.

Figure 2 illustrates some of the security policies inferred for this code. Each policy includes a security-sensitive event and a (possibly empty) set of security checks. Consider the API-return event for JDK. There is a *must* and a *may* policy for this event. The *must* check is the empty set and the *may* checks are $\{\{$`checkMulticast`$\}$ or $\{$`checkConnect, checkAccept`$\}\}$. The call to `checkAccept` before the API return and, more importantly, before using JNI to make a network connection as a result of invoking `impl.connect`, are missing in the Harmony *may* policy, introducing a vulnerability.

The security-checking pattern in `connect` is unique. The `checkMulticast` call is very rare and there is no other place in the JDK library where it occurs together with `checkAccept` and `checkConnect`. The "bugs as inconsistencies" methods [14, 35] are not only likely to miss this bug, but may even wrongly flag the JDK implementation because the pattern incorrectly used by Harmony is more common than the correct pattern used by JDK. The rarity of this pattern and the fact that it involves multiple checks mean that a manual policy is likely to miss it, too.

Both Harmony and JDK implement a *may* policy: there is no single check that dominates all paths to the security-sensitive event. Even if the manual policy happens to include all the right event-check pairs, analyses that require the same check(s) on every

```
// JDK
public void connect(InetAddress address,
                    int port) {
  ... connectInternal(address, port); ...
}
private synchronized void connectInternal(
    InetAddress address, int port) {
  ...
  if (address.isMulticastAddress()) {
    securityManager.checkMulticast(address);
  } else {
    securityManager.checkConnect(address.
        getHostAddress(), port);
    securityManager.checkAccept(address.
        getHostAddress(), port);
  }
  if (oldImpl) {
    connectState = ST_CONNECTED_NO_IMPL;
  } else {
    ... getImpl().connect(address, port); ...
  }
  connectedAddress = address;
  connectedPort = port;
  ...
}
```

(a) JDK implementation of `DatagramSocket.connect`

```
// Harmony
public void connect(InetAddress anAddr,
                    int aPort) {
  synchronized (lock) {
    ...
    if (anAddr.isMulticastAddress()) {
      securityManager.checkMulticast(anAddr);
    } else {
      securityManager.checkConnect(anAddr.
          getHostName(), aPort);
    }
    ...
    impl.connect(anAddr, aPort);
    ...
    address = anAddr;
    port = aPort;
    ...
  }
}
```

(b) Harmony implementation of `DatagramSocket.connect`

Figure 1: Security vulnerability in Harmony: `checkAccept` is missing. The correct security policy is unique to this method.

```
MUST check:  {}
Event: API return from DatagramSocket.connect

MAY check: {{checkMulticast},{checkConnect, checkAccept}}
Event: API return from DatagramSocket.connect
```

(b) JDK `DatagramSocket.connect` security policies

```
MUST check:  {}
Event: API return from DatagramSocket.connect

MAY check: {{checkMulticast},{checkConnect}}
Event: API return from DatagramSocket.connect
```

(b) Harmony `DatagramSocket.connect` security policies

Figure 2: Example security policies

path [7, 22, 28, 30] will produce a warning for both implementations, which is a false positive for the correct JDK implementation.

Our analysis finds this vulnerability using (1) precise, flow- and context-sensitive interprocedural analysis that computes both *may* and *must* policies, and (2) differencing of policies from multiple implementations.

3. Security Policies

This section explains security policies in the access-rights model in more detail. Section 4 explains how we compute the policies implemented by a given API entry point and Section 5 describes our algorithm for comparing policies.

A *security policy* in the access-rights model is a mapping from *security-sensitive events* to one or more *security checks* for each event. Our analysis takes events and checks as input. A security-sensitive event is a program event, such as a write to a file, that the application should not be able to execute unless it holds certain rights or privileges. A security check verifies that the application holds a particular right. The access-rights model is the cornerstone of Java security. For example, unchecked native method calls can give user applications unauthorized network access, while unchecked returns of internal JVM state, *i.e.*, the value of a private variable, can leak data. The definitions of security checks and events that we use in this paper directly reflect Java's security model.

A security policy "maps" an event to a check if the check occurs before the event. Our analysis computes both *must* and *may* policies. A *must* policy means that the check in question must occur on every execution path leading to the event. A *may* policy means that the check is predicated on some condition. At first glance, *may* policies may appear insecure, but they are often needed to implement the correct security logic. For example, Figure 2(a) shows a policy that always performs one or more checks, but the particular check differs depending on control flow.

Security checks. The `SecurityManager` class in Java provides 31 methods that perform security checks for user code and libraries. For example, `checkPermission`() verifies that the calling context holds a particular permission and throws an exception otherwise. We restrict our analysis to these methods, although programmers can define their own, additional security checks. Our analysis keeps track of *which* of the 31 security checks is invoked at any given point. For example, it differentiates between `checkPermission`() and `checkConnect`().

Our analysis does not ensure that the parameters to security checks are the same as the ones used by the security event. This imprecision could be a source of false negatives.

Security-sensitive events. The Java Native Interface (JNI) defines all interactions with the outside environment for Java programs, *e.g.*, opening files, connecting to the network, and writing to the console. We therefore define all calls to native methods as security-sensitive events.

In addition, we consider all API returns to be security-sensitive events. If a method performs security checks but does not use the JNI, these checks are thus included in its security policy. Such checks are used in API implementations that give user code access to internal JVM state or private variables, or store parameter values for later use. These accesses should be secured consistently in all implementations of an API because they reveal internal JVM or library state to untrusted applications and/or enable applications to modify this private state. By including API returns, we broaden the definition of security-sensitive events as compared to prior work [22, 28, 30]. This broader definition helps us find more vulnerabilities. For example, Figure 6 shows and Section 6.2

```
// Implementation 1
public Obj A(Obj obj) {
  if (condition) {
    checkRead();
    obj.add(data1);
    return obj;
  } else {
    return null;
  }
  checkRead();
  obj.add(data2);
  // Private data1 and data2 returned in obj
  return obj;
}

// Implementation 2
public Obj A(Obj obj, Data data1, Data data2) {
  if (condition) {
    obj.add(data1);
    return obj;
  } else {
    return null;
  }
  checkRead();
  obj.add(data2);
  // Private data1 and data2 returned in obj
  return obj;
}
```

Figure 3: Hypothetical bug showing the need for a broad definition of security-sensitive events.

describes a vulnerability that we detect because our definition of security-sensitive events is not limited to JNI calls only.

Broader definition of security-sensitive events. Our analysis can further broaden the definition of security-sensitive events. In addition to JNI calls and API returns, we experimented with including individual accesses to private variables, JVM state, and API parameters. In particular, security-sensitive events could include all reads, writes, and method invocations on API parameters and private variables, as well as reads, writes, and method invocations on variables that are data-dependent on API parameters and private variables.

For this definition of security-sensitive events, we computed data dependencies using a simple interprocedural dataflow analysis that propagated an event tag. It marked all statements in the definition-use chains involving private variables and API parameters, and propagated the mark interprocedurally through parameter binding in method invocations. If there were multiple instances of the same event type, *e.g.*, two returns or two accesses to the same parameter, we combined the corresponding policies. This definition of security-sensitive events is very liberal and marks many more events as sensitive. It generates over 90,000 security policies for each Java Library implementation, whereas restricting security-sensitive events to JNI calls and API returns results in 16,700 or fewer policies for each implementation.

This broad definition of security-sensitive events did not result in finding more bugs, nor generating more false positives during our analysis of the Java Class Library, but it helped us diagnose the cause of differences in the implementations' respective security policies. For other APIs, this broad definition may more accurately capture the semantics of library implementations and thus find errors missed with a narrower definition of security-sensitive events.

For example, this broad definition is needed to detect the inconsistency between implementations in the hypothetical example shown in Figure 3. Restricting security-sensitive events to JNI calls and API returns results in a {`checkRead`} *may* policy for the API-return event in both implementations. If we instead consider the reads of private variables `data1` and `data2` as distinct security-sensitive events, the analysis infers (1) a {`checkRead`} *must* policy for the read of `data1` in the first implementation and (2) an empty *must* policy for the same event in the second implementation. The analysis will thus report an inconsistency.

4. Computing Security Policies

Given an API and multiple implementations thereof, our approach extracts and compares *security policies* realized by the implementations. The analysis computes the security checks that each API always performs (*must* analysis) or may perform (*may* analysis) before each security-sensitive event. The analysis uses a flow- and context-sensitive interprocedural algorithm enhanced with flow- and context-sensitive forward constant propagation [17, 36]. Any context-sensitive analysis must be concerned with scalability. To reduce analysis time and guarantee convergence, we do not iterate over recursive call paths. We further eliminate useless re-analysis with a form of memoization [38] that records security policies and relevant parameters at each method invocation.

We implement the analysis in the Soot static analysis framework[1] [25] and use Soot's method resolution analysis, alias analysis, and intraprocedural constant propagation.

Call graph. The analysis starts by building call graphs rooted at all public and protected API entry points, and derives separate policies for each API entry point. Protected methods are included because API clients can potentially call them by overriding the corresponding class. These methods thus represent unintended paths into the API and are important to analyze. In theory, any static analysis is incomplete in the presence of dynamic class loading since the code is not available until run time. In practice, the Java Class Library and many other APIs are closed-world and do not depend on dynamic class loading.

Soot computes application call graphs, but its analysis is engineered for applications with a single entry point. Because APIs have many entry points, we build the call graph on the fly, using Soot's method resolution analysis which resolves 97% of method calls in the Java libraries. If Soot does not resolve a method invocation, our implementation does not analyze it. This inaccuracy could be a source of false negatives or positives, but did not produce false positives in our evaluation. As shown in prior work, type-resolving events, such as allocation, make simple type hierarchy analysis very effective at resolving method invocations [11, 34]. The Java libraries' coding conventions—for example, the use of final methods—further improve the precision of method resolution.

Analysis overview. Computing security policies is essentially a reaching definitions analysis where the *definitions* are security checks and the *uses* are security-sensitive events. The dataflow lattice is the power set of the 31 security-checking methods. The analysis propagates checks (definitions) to events (uses). After converging, the resulting security policy is a mapping from event statements to the checks that reach them.

We perform *interprocedural* and *context-sensitive* analysis, propagate constants across method calls, and eliminate the resulting dead code, for the following reasons. It is typical for an API method to first call a method that performs a security check and then call another method that contains a security-sensitive event. It is also common to perform the security check(s) conditionally, based on the value of a parameter (see Section 4.2 for an example).

[1] `http://www.sable.mcgill.ca/soot/`

Algorithm 1 Intraprocedural Security Policy Dataflow Analysis

```
procedure $SPDA$
$\{$Initialize dataflow values$\}$
for all $v \epsilon AllStatements$ do
    if MUST then
        $OUT(v) \leftarrow \bot$
    else $\{$MAY$\}$
        $OUT(v) \leftarrow \top$
    end if
    $NOTVISITED \leftarrow NOTVISITED \cup v$
end for
$\{$Propagate dataflow values$\}$
$worklist \leftarrow EntryPoint$
while $worklist \neq \phi$ do
    $v \leftarrow getNode(worklist)$
    if MUST then
        $IN(v) \leftarrow \bigcap_{p \in PRED(v)} OUT(p)$
    else $\{$MAY$\}$
        $IN(v) \leftarrow \bigcup_{p \in PRED(v)} OUT(p)$
    end if
    $OUT(v) \leftarrow IN(v) \cup SP(v)$
    if $OUT(v)$ $changed \wedge v \in NOTVISITED$ then
        $worklist \leftarrow worklist \cup SUCC(v)$
        $NOTVISITED \leftarrow NOTVISITED - v$
    end if
end while
```

Algorithm 2 Interprocedural Security Policy Analysis

```
procedure $ISPA(m, SP, paramConsts)$
if $hashTable.hasKey(\langle m, SP, paramConsts \rangle)$ then
    $\{$Same analysis state, so return hashed result$\}$
    return $hashTable.lookup(\langle m, SP, paramConsts \rangle)$
else
    $\{$Compute result for this analysis state$\}$
    $newSP \leftarrow SPDA(m, SP, paramConsts)$
    $hashTable.put(\langle m, SP, paramConsts \rangle, newSP)$
    return $newSP$
end if
```

4.1 Intraprocedural analysis

For ease of presentation, we first describe the intraprocedural component of the analysis. Algorithm 1 shows our intraprocedural security policy dataflow analysis (SPDA), which propagates security checks. The MUST and MAY analyses differ in (i) the meet function, performing *intersection* for MUST and *union* for MAY, and (ii) the initial flow values, which are $\bot$ for MUST and $\top$ for MAY. SPDA assigns a unique identifier to each of the 31 security-checking methods. It initializes $OUT(v)$ of each statement that performs a check to the corresponding identifier, *i.e.*, $OUT(v) = SP = \{check\}$. For all other statements, it initializes $SP = \emptyset$.

Because SPDA is a reaching definition analysis and uses a lattice that is a powerset, it is *rapid*, converging in two passes with structured control flow [27].

4.2 Interprocedural, context-sensitive analysis

This section explains how we extend the intraprocedural dataflow analysis to make it interprocedural and context sensitive. ISPA, our interprocedural security policy analysis, is shown in Algorithm 2. When ISPA encounters a statement v that contains a method invocation, it invokes the analysis recursively on the target method. As mentioned above, our implementation ignores method invocations that Soot cannot resolve to a unique target, which may lead to false negatives and positives. ISPA performs interprocedural constant propagation together with the security analysis, tracking constant parameter values (described below). The analysis binds any current constant dataflow values to the parameters and uses them together with the current policy as the initial state to analyze the target method.

This analysis does not scale because it analyzes every statically possible calling context separately. In practice, however, many of the method's calling contexts have the same dataflow values. We eliminate redundant work via the following memoization scheme. The first time our analysis encounters a call to some method, it analyzes the method, passing in the initial policy and any constant parameters. If analyzing the method changes the policy, we store the policy and the constant parameters. Otherwise, we simply store the initial policy. When the analysis encounters the same method in a different calling context but with the same policy flow values, it reuses the stored policy and avoids re-analyzing the method.

We use ISPA to analyze each API entry point. The algorithm takes as input a method m and the current analysis state, which consists of the current security policy *SP* and any known constant parameters *paramConsts* to m. The algorithm is mutually recursive with SPDA (Algorithm 1), which invokes ISPA at method invocations with the current values of *SP* and *paramConsts*. Thus, the computation of *OUT(v)* in the while loop requires a change—a recursive call to gather *SP(v)*, if v is a resolved method invocation. SPDA uses *IN(v)* as the initial flow value of the method's entry basic block and seeds constant propagation with the incoming constant parameters (both not shown).

Constant propagation. For better precision, our analysis propagates constants intraprocedurally and interprocedurally. The security policy from Harmony in Figure 4 motivates this analysis. It shows a common pattern in which a parameter determines whether a security check is performed. This code uses two `URL` constructors. The first constructor always passes `null` as the `handler` parameter to the second constructor, which—correctly—performs no security checks. In other contexts, however, the second constructor performs one security check (line 7). Interprocedural constant propagation is required to accurately differentiate these two contexts and prevent false positives.

Soot supports Wegman-Zadeck intraprocedural constant propagation [36]. Soot's algorithm propagates integer and boolean constants and null reference assignments into conditional statements and eliminates unexecutable statements. We extend the analysis at method invocations to pass constant parameters to the target method. Because constant parameters are an essential part of the analysis state, our memoization scheme includes them as dataflow values: both the policy *SP* and the constant parameters *paramConsts* must match in order to reuse previous results.

Convergence. SPDA and ISPA are guaranteed to converge because they are monotone and distributive. Constant propagation is a monotone forward analysis. Intraprocedural constant propagation is guaranteed to converge on the structured control-flow graphs in Java, but because the call graph is recursive and not structured, context-sensitive interprocedural constant propagation is not guaranteed to converge. For example, if a call passes in a constant parameter that the callee increments until it reaches some limit, each context would be considered unique. In a context-sensitive analysis, constant propagation can produce an unbounded number of unique constant parameters due to method recursion. We prevent this case by terminating early on recursive methods. In our implementation, if the analysis encounters a recursive call, *i.e.*, a call to a method that is already on the call stack, it does not re-analyze the method. An alternative implementation could instead bound the number of traversals of a recursive subgraph.

```
// Harmony
public URL(String spec) {
  this((URL) null, spec,
       (URLStreamHandler) null);
}
public URL(URL context, String spec
           URLStreamHandler handler) {
  if (handler != null) {
    securityManager.checkPermission(
        specifyStreamHandlerPermission);
    strmHandler = handler;
  }
  ... protocol = newProtocol; ...
}
```

Figure 4: **A context-sensitive *may* policy.** Note that deriving the precise policy associated with the first entry point requires propagating the `null` constant into the `handler` parameter.

Scalability. Making context-sensitive analysis efficient is difficult in general [18]. Our memoization is similar to Wilson and Lam [38], who record context-sensitive alias sets to avoid redundant analysis. As described above, we memoize context-sensitive security policies and any relevant constant parameters. Unlike context-sensitive, whole-program pointer analyses that must store large sets, our analysis deals with a relatively small number of security checks which must or may precede security-sensitive events.

Even though our analysis essentially explores every call path, it performs well in practice. Section 6 shows that reusing dataflow analysis values for identical incoming flow values improves performance significantly. Other properties of our analysis, such as not analyzing recursive calls and ignoring method invocations that Soot cannot resolve to a unique target, improve scalability as well. We find that calls to a method in the same or other contexts often have the same incoming flow values. Nonetheless, context sensitivity, which differentiates contexts that have different flow values, is key to the precision of our approach.

5. Comparing Security Policies

Given two implementations of the same API entry point, we compute the policies realized by each implementation using the analysis described in the previous section. Our analysis combines distinct occurrences of calls to the same JNI routine and API returns. For example, if a method calls the same JNI routine three times with different policies, we combine all three *must* and *may* policies. We perform intersection to combine *must* policies and union to combine *may* policies. Although this step causes a loss of precision for *must* policies in particular and thus may lead to false positives, we did not find it to be a problem in practice. This step does not introduce false negatives when distinct instances of the same JNI call use different *must* policies, but may introduce false negatives when combining *may* policies. A more precise analysis could try to order and align each instance of a security-sensitive event, but we did not find this necessary.

After combining security-sensitive events in each API, the analysis then compares two policies as follows:

1. If neither implementation has any security policies, or both implementations have identical security policies, the comparison analysis reports no error.
2. If one implementation has no security policy, but the other implementation has one or more security policies, the comparison analysis reports an error.
3. Otherwise, the comparison analysis matches events that occur in both implementations. We ignore events unique to one implementation. Matched events are compared as follows:

   (a) If the two implementations have different sets of security checks for the same event, the comparison analysis reports an error.

   (b) If the two implementations have the same security checks, but at least one check is *may* in one implementation and *must* in the other, the comparison analysis reports an error.

Case 2 is responsible for most of the security vulnerabilities and interoperability bugs we found. Case 3(a) produced one vulnerability, one interoperability bug, and one false positive. Case 3(b) produced one interoperability bug.

6. Evaluation

We evaluate our approach by applying it to recent versions of three mature implementations of the standard Java Class Library. The libraries perform two critical functions in Java. First, they make Java portable by providing an abstract interface to I/O, user interfaces, file systems, the network, operating system, and all other machine-dependent functionality. Second, they provide standard, optimized implementations of widely used data structures and abstractions, such as sets and queues. All Java programs rely on the security and correctness of these libraries.

We analyze three implementations of the Java Class Library, listed below. For each implementation, we analyze the main packages: java.io, java.lang, java.nio, java.security, java.text, java.util, javax.crypto, javax.net, javax.security, and java.net.

1. *JDK:* Sun JDK, version 1.6.0_07, first released in 1996.
2. *Classpath:* GNU Classpath, version 0.97.2, started in 1998.
3. *Harmony:* Apache Harmony libraries, version 1.5.0, svn revision r761593, started in 2005.

Note that two of the implementations are well over 10 years old, yet our approach still found security errors in them. Together, these libraries total about 2.5M lines of code.

Table 1 summarizes the non-comment lines of code, API entry points, and characteristics of security policies for each library. We analyze all public and protected methods because applications can invoke them either directly or via a derived class. Because of this, the number of API entry points varies from implementation to implementation.

We analyze the entire call graph rooted at each of these API entry points. The third row in Table 1 shows that only a small subset of methods performs security checks. The analysis computes between 4,208 and 9,580 *must* and *may* security policies for each implementation. We only compare policies for the API entry points that are identical in two implementations—over 4,100 entry points for each pair—shown in the first row of Table 3. The sheer volume of policies demonstrates that any approach that relies on developers to manually examine inferred policies to detect errors is unlikely to succeed.

Analysis time. Table 2 shows the time in minutes to compute *must* and *may* policies for each of the three libraries. For both *may* and *must* policies, the first row shows the analysis time without memoization of method summaries (Section 4). The second and third rows show time with memoization. In the second row, method summaries are reused only within the same API entry point. In the third row, they are reused across the entire library.

Reuse of summaries within the same entry point yields a factor of 1.5 to 13 improvement and reuse across the entire library yields an additional factor of 3 to 18, resulting in the overall factor of 15 to

	JDK	Harmony	Classpath
Non-comment lines of code	632K	572K	563K
Entry points	6,008	5,835	4,563
Entry points w/ security checks	239	262	250
may security policies	9,580	7,126	4,652
must security policies	7,181	6,757	4,208

Table 1: Library characteristics

		JDK	Harmony	Classpath
MAY	No summaries	300	190	340
	Summaries (per entry point)	180	130	190
	Summaries (global)	10	13	20
MUST	No summaries	560	290	650
	Summaries (per entry point)	50	40	50
	Summaries (global)	10	12	10

Table 2: Analysis time in minutes

	Classpath v Harmony		**JDK v Harmony**		**JDK v Classpath**	
Matching APIs		4,161		4,449		4,758
False positives eliminated by ICP		4 (63)		4 (35)		4 (74)
False positives		3 (3)		3 (3)		0 (0)
Root cause of policy difference						
Intraprocedural		1 (1)		5 (6)		2 (3)
Interprocedural		14 (140)		13 (43)		16 (300)
MUST/MAY difference		0 (0)		1 (5)		0 (0)
Total differences		15 (142)		19 (54)		18 (303)
Total interoperability bugs		3 (115)		9 (39)		5 (222)
	Classpath	Harmony	JDK	Harmony	JDK	Classpath
Security vulnerabilities in	5 (12)	4 (11)	1 (2)	6 (10)	5 (21)	8 (60)

	JDK	**Harmony**	**Classpath**
Total security vulnerabilities	6 (23)	6 (11)	8 (61)

The table reports distinct errors with manifestations in parentheses: *distinct (manifestations)*.

Table 3: Security vulnerabilities and interoperability errors detected by security policy differencing analysis

65 improvement in performance due to memoization. Although our analysis is still not blazingly fast, it is intended to be used relatively infrequently, as part of correctness and interoperability testing.

6.1 Analysis results

We categorize the results of our analysis as follows:

Vulnerability: A semantic difference that can be exploited to perform some security-sensitive action without permission.

Interoperability bug: A semantic difference that causes interoperability problems. These differences do not, on the surface, enable applications to perform security-sensitive actions without permission, but could be part of a multi-stage attack.

False positive: Policies are identical, but a difference is mistakenly reported due to imprecision of our analysis.

False negative: A security vulnerability *not* reported by our analysis. False negatives may arise if identical *may* policies apply under different conditions, or if two policies are identical but incorrect. Although these cases seem unlikely, their frequency is hard to quantify.

The remainder of this section surveys the results of security-policy differencing and how different algorithmic features of our analysis contribute to the results. Sections 6.2 and 6.3 discuss the security vulnerabilities and interoperability errors in detail and include examples of each. Section 6.4 explains in more detail why our analysis can have false positives and false negatives.

Table 3 shows the results of our analysis. We compare each implementation to the other two. The analysis finds most errors in both comparisons, but because some entry points differ between implementations, each pairwise comparison finds a few unique errors.

To reduce the number of reports the developer must read, our analysis automatically combines reports when the error stems from the same *root cause*, *i.e.*, when the method containing the error is called from multiple API entry points. The number of entry points (*manifestations*) that can exploit the error is shown in parentheses. We manually examined all root causes to determine the responsible library implementation and classify the bug.

The *ICP* row shows that interprocedural constant propagation eliminates 4 false positives that have over 70 manifestations, resulting in the overall false-positive rate of less than 1%: 3 of 499 manifestations. The *Intraprocedural*, *Interprocedural*, and *MUST/-MAY* rows show that every component of our analysis contributes to finding errors. Intraprocedural analysis, which only computes policies local to a method, would miss the majority of the errors. Differences due to a *must* policy in one implementation and a *may* policy in the other revealed one bug. Not shown in a dedicated row in the table is the small number of vulnerabilities revealed by differences between two *may* policies, including the one in Figure 1. Finding most errors requires context-sensitive interprocedural analysis.

Overall, our analysis found 20 security vulnerabilities and 11 interoperability bugs across *all three library implementations*, not just in the least mature one. These errors witness how difficult it is for programmers to get access-rights policies correct. For example, even though more than a hundred developers worked on the JDK implementation for over 15 years, it is still not error-free. We reported all vulnerabilities to the respective implementors, who recognized all of them as bugs and fixed some of them [31–33].

6.2 Security vulnerabilities

This section explains a few of the security vulnerabilities uncovered by our analysis. To demonstrate the power of implementation

```
// JDK
class Runtime {
  public void loadLibrary(String libname) {
    loadLibrary0(System.getCallerClass(),
                 libname);
    return;
  }
  synchronized void loadLibrary0(
      Class fromClass, String libname) {
    ... securityManager.checkLink(libname); ...
    ClassLoader.loadLibrary(fromClass, libname,
                            false);
  }
}

class ClassLoader {
  static void loadLibrary(Class fromClass,
      String name, boolean isAbsolute) {
    ... loadLibrary0(fromClass, libfile); ...
  }
  private static boolean loadLibrary0(
      Class fromClass, final File file) {
    ...
    NativeLibrary lib =
      new NativeLibrary(fromClass, name);
    lib.load(name);
    ...
  }
}
```

(a) JDK implementation of `Runtime.loadLibrary`

```
// Classpath
class Runtime {
  public void loadLibrary(String libname) {
    loadLibrary(libname, VMStackWalker.
        getCallingClassLoader());
    return;
  }
  void loadLibrary(String libname, ClassLoader
      loader) {
    ... securityManager.checkLink(libname); ...
    ... loadLib(filename, loader); ...
  }
  private static int loadLib(String filename,
      ClassLoader loader) {
    ...
    securityManager.checkRead(filename);
    ...
    return VMRuntime.nativeLoad(filename,
        loader);
  }
}
```

(b) Classpath implementation of `Runtime.loadLibrary`

Figure 5: **Security vulnerability:** JDK is missing `checkRead` that Classpath performs before loading a library at run time.

```
// Harmony
public URLConnection openConnection(
    Proxy proxy) throws IOException {
  ...
  return
    strmHandler.openConnection(this, proxy);
}
```

(a) Harmony implementation of `URLConnection.openConnection`

```
// JDK
public URLConnection openConnection(
    Proxy proxy) {
  ...
  if (proxy.type() != Proxy.Type.DIRECT) {
    InetSocketAddress epoint =
      (InetSocketAddress) proxy.address();
    if (epoint.isUnresolved()) {
      securityManager.checkConnect(epoint.
          getHostName(), epoint.getPort());
    } else {
      securityManager.checkConnect(
          epoint.getAddress().getHostAddress(),
          epoint.getPort());
    }
  }
  return handler.openConnection(this, proxy);
}
```

(b) JDK implementation of `URLConnection.openConnection`

Figure 6: **Security vulnerability:** Harmony is missing `checkConnect` that JDK performs before opening a network connection.

differencing, each vulnerability is accompanied by a correct implementation from another library.

Figure 5 shows one of the six vulnerabilities in JDK. The JDK code returns from `Runtime.loadLibrary` having called only `checkLink` on a dynamically loaded library. By contrast, the Classpath implementation calls both `checkLink` and `checkRead`. Detecting this vulnerability requires interprocedural analysis.

The other five JDK vulnerabilities are detected when our analysis compares JDK with Classpath. They arise because JDK performs some security checks inside a privileged block. Security checks inside a privileged block always succeed. Therefore, they are semantic no-ops and our analysis correctly ignores them. It seems especially difficult for developers to detect this kind of error through manual inspection because a call to the security-checking method is actually present in the source code.

Figure 6 shows one of the six vulnerabilities in Harmony, detected when our analysis compared it to JDK. Finding the vulnerability in `OpenConnection` requires using API returns as security-sensitive events because `OpenConnection` does not actually perform network reads and writes with JNI calls. The user must subsequently call `getInputStream()` or `getOutputStream()` to read or write to the network. In Harmony, `OpenConnection` calls a method on the private `strmHandler` variable and returns internal API state to the application without any checks. By contrast, the JDK implementation has a *may* policy that calls `checkConnect` before returning internal API state to the application.

Figure 7 shows one of the eight vulnerabilities in Classpath. Classpath omits all security checks in the `Socket.connect` method, whereas JDK always calls `checkConnect`. This error seems simple to spot, but the method is called in many contexts, some of which do perform checks. Since this method is directly accessible by an application, this vulnerability is easy to exploit and has now been fixed.

6.3 Interoperability bugs

Some interoperability problems arise because Classpath performs much more dynamic class loading than the other implementations. For example, it dynamically loads the `CharsetProvider` class, whereas JDK statically loads it at boot time. The reason may be that Classpath is trying to reduce the size of its JVM when running on an embedded platform. Because of this difference, Classpath contains code that performs `checkPermission(new RuntimePermission("charsetProvider"))`, whereas JDK and Harmony do not.

```java
// JDK
class Socket {
  public void connect(SocketAddress, int) {
    ...
    securityManager.checkConnect(...);
    ...
    impl.connect(...);
  }
}
```

(a) JDK implementation of Socket.connect

```java
// Classpath
class Socket {
  public void connect(SocketAddress, int) {
    ...
    getImpl().connect(endpoint, timeout);
  }
}
```

(b) Classpath implementation of Socket.connect

Figure 7: **Security vulnerability:** Classpath is missing checkConnect that JDK performs before opening a network connection.

```java
// JDK
class String {
 public byte[] getBytes() {
   return
     StringCoding.encode(value, offset, count);
 }
}

class StringCoding {
 static byte[] encode(...) {
   try {
     return encode("ISO-8859-1", ca, off, len);
   } catch (UnsupportedEncodingException x) {
     System.exit(1);
     return null;
   }
 }
}
```

(a) JDK implementation of String.getBytes

```java
//Harmony
class String {
 public byte[] getBytes() {
   ByteBuffer buffer =
     defaultCharset().encode(...);
   ...
 }
 private Charset defaultCharset() {
   if (DefaultCharset == null) {
     DefaultCharset =
       Charset.forName("ISO-8859-1");
   }
   ...
 }
}
```

(b) Harmony implementation of String.getBytes

Figure 8: **Interoperability bug:** JDK requires checkExit permission to call System.exit(), whereas Harmony throws an exception.

Interoperability problems also arise due to additional functionality in one of the implementations. Figure 8 shows an example from String.getBytes. If the default "ISO-8859-1" character set decoder is not present, JDK terminates the application by calling System.exit(), whereas forName in Harmony throws UnsupportedEncodingException. To perform System.exit(), the application needs checkExit permission which is not needed in the Harmony implementation.

6.4 False positives and negatives

The main reason for false positives is inherent to any static analysis—it is conservative because it includes all possible program paths and not just the actual paths taken during code execution. In practice, our analysis produced only three false positives when analyzing the three implementations of the Java Class Library, all of them due to questionable coding practices in the Harmony implementation. For example, in java.security.Security.getProperty(String), JDK uses checkPermission(), whereas Harmony uses checkSecurityAccess(). There is a mismatch between required permissions, but both checks achieve the same goal. In this example, both implementations should have used checkPropertyAccess(). Similarly, Harmony unnecessarily uses checkConnect() to check address reachability inside the getInetAddresses() method, whereas JDK simply returns the result of InetAddrImpl.isReachable().

There are two causes of false negatives. First, two libraries may both implement the security policy incorrectly and in the same way. The second cause is imprecision of our analysis. For example, our analysis is not field or variable sensitive. If the library sends the wrong parameter to the security-checking method, or if it checks one variable and then makes a JNI call on another, our analysis will not report an error. Furthermore, our analysis does not determine the potential targets of unresolved method invocations for incomplete call graphs. Since 97% of method invocations were resolved, we did not perform additional analysis, and this inaccuracy may result in false negatives. Finally, our comparison of *may* policies does not consider the conditions under which the checks are executed. Changing our analysis to report these conditions is easy, but would result in an overwhelming number of reports. Verifying whether the conditions for two *may* policies are equivalent is difficult. We examined some of the reports by hand and did not find any false negatives. A quantitative evaluation of false negatives would require an enormous amount of time and expertise from the developers.

7. Related Work

This section describes the closest related work on static analysis for finding security errors and other bugs, as well as program differencing. None of the prior work exploits multiple API implementations to automatically derive correctness criteria for security policies.

7.1 Static analysis and model checking

The closest related static analysis techniques for verifying security mediation in Java code are by Koved et al. [22, 28] and Sistla et al. [30]. Koved et al. take security checks as inputs and use a flow- and context-sensitive interprocedural *must* analysis to compute, for each point in the program, the set of checks performed prior to reaching it. The programmer must manually identify all security-sensitive operations and verify whether the computed checks are sufficient. Our *must* polices are similar, but our policy differencing eliminates the error-prone, manual verification.

Sistla et al. take as input a manual policy specified as pairs of security checks (calls to the security manager) and security events (only JNI calls in their model). They use flow- and context-sensitive interprocedural *must* analysis to find "bad" paths that reach a security event without performing the corresponding check. The MOPS project [7] applied a similar, but flow-insensitive, approach to C programs.

As our analysis shows, correct security enforcement sometimes requires multiple checks. Furthermore, the check(s) may not dominate the event (*e.g.*, see Figure 1), necessitating *may* analysis. In addition to native calls, the set of security events should also include at least API returns. The final and most significant deficiency of these prior approaches is that they provide no mechanism for determining whether the policy used for verification is correct and complete.

These deficiencies are significant in practice. Sistla et al. analyzed the JDK and Harmony libraries, while Koved et al. analyzed JDK—but *neither paper reports any actual bugs.* By using differences between API implementations as an oracle for detecting incorrect policies, we found multiple, confirmed security vulnerabilities in JDK, Harmony, and Classpath, many of which were missed by prior analyses. Furthermore, our analysis produces precise security policies specific to concrete security-sensitive events.

There is an enormous amount of research on model checking that aims to prove programs correct using precise specifications [4–6, 8, 13]. In general, model checking does not yet scale to interprocedural security analysis of large programs, and rigorous specification of security policies and vulnerability patterns has proven elusive so far.

Whaley et al. extract finite-state models of the interface of a class and its accesses to common data structures [37]. Security policies do not always follow their model. For instance, consider the following simple example:

```
    securityManager.checkPermission();
    doSensitiveOperation();
```

Their analysis cannot detect that `doSensitiveOperation()` should be dominated by `checkPermission()` because there is no data structure shared between the two methods. Furthermore, their analysis is not context sensitive. Our approach could be configured to use arbitrary data structure accesses as security-sensitive events, but we did not find it necessary.

Ganapathy et al. use concept analysis to extract security-sensitive events [15]. Their approach is complementary to ours, since our analysis could take as input the security-sensitive events they generate.

7.2 "Bugs as inconsistencies"

Mining programs for static and dynamic models of correct behavior has become a popular approach for finding bugs [2, 10, 14, 35]. In general, this approach finds bugs that reveal themselves as anomalies in common patterns of correct behavior. The fundamental assumption is that correct patterns occur many times in the program and are thus easy to recognize. Mining algorithms must be tuned to ignore unique or rarely observed behavior, or else they produce an overwhelming number of false positives. Because security policies are often unique to a particular API, the "bugs as inconsistencies" approach is likely to miss many vulnerabilities (*e.g.*, see Section 2).

AutoISES statically mines security patterns in C code and flags deviations from frequent patterns as vulnerabilities [35]. Its analysis targets security checks performed prior to security-sensitive operations in SELinux kernel routines. The AutoISES paper claims that the analysis is context sensitive and flow *insensitive*, but does not describe the actual algorithm or convergence criteria. It does mention that the analysis partitions the call graph into modules for scalability, thus missing security patterns and bugs on cross-module call paths, although the authors argue that maintainable security policies should not cross modules—at least in C programs. In our experience with the Java libraries, both flow and context sensitivity are essential for precision.

Engler et al. statically extract *intraprocedural* patterns of correct behavior, including some security policies, and use them to detect anomalies [14]. Their policy descriptions include "a call to A always or sometimes dominates a call to B" and are thus similar to ours. However, these local patterns are not *interprocedural*—they do not include flow- and context-sensitive policies in which the security check occurs in a different method than the operation it protects. Furthermore, the approach of Engler et al. ignores rare patterns. Dillig et al. find bugs by detecting static inconsistencies in the program [10]. For example, they detect if the program contains both "`if (x==NULL) foo.m();`" and "`*x=y;`" which indicate both that x may be null and that x is definitely non-null. This analysis finds local contradictions rather than semantic differences. It captures neither rare events, nor interprocedural patterns. Both approaches will miss many vulnerabilities in Java code, where security-sensitive events and the corresponding checks often occur in different methods.

Another approach to mining specification is to observe policies at run time. For example, Ammons et al. observe dynamic traces of method call sequences and use machine learning to create API usage specifications in the form of finite-state machines [2]. Dynamic mining approaches are at a disadvantage when policies are unique and/or rarely or never exercised by the test inputs.

Kim et al. use a database of previously fixed bugs to find their recurrences in new code [21]. Hovemeyer and Pugh use programmer-defined bug patterns to find their occurrences [19]. To avoid false positives, these tools require well-defined bug patterns that apply to all contexts. These types of patterns are useful for describing coding conventions, but generally do not capture context-specific semantics. Because many security policies are context dependent and unique, these techniques cannot classify them as correct or incorrect.

Our work is also distantly related to program differencing. We find inconsistencies between implementations of the same API using identical API entry points, but do not assume that the implementations are identical or even very similar. In our experience, different implementations often choose different algorithms and data structures. Techniques for finding *method clones*—methods that implement semantically similar functionality within the same program [3, 12, 23, 24]—are thus unlikely to classify two methods that implement the same API as clones. However, it might be possible to leverage clone detection by first finding clones, then extracting and comparing their security policies. We leave this exploration to future work.

In summary, our interprocedural, flow-sensitive and context-sensitive analysis is more precise than prior approaches. It does not require multiple instances of correct behavior within the same program and can thus find bugs in very rare patterns, such as the missing `checkAccept` in the example of Section 2. It does, however, require at least two independent implementations of the same API.

8. Discussion and Conclusions

This paper shows how to use precise, flow- and context-sensitive security policy analysis to infer thousands of relationships between security checks and security-sensitive events in Java code, and how to use this information to compare implementations of the same library API. The number of check-event relationships in these implementations is so large that it is clearly impractical for developers to analyze them manually. In fact, prior techniques that produced similar policies for each implementation in isolation did not find any errors. By comparing precise policies from multiple implementations, we create a *security policy oracle*: any policy difference between two implementations of the same functionality indicates an error! Our approach uncovered many security vulnerabilities and interoperability bugs in three large, mature, widely used implementations of the Java Class Library.

Of course, security-policy differencing requires multiple, independent implementations and thus limits the applicability of our approach. However, many critical APIs, such as the Java, C#, and C libraries, have multiple implementations. Furthermore, they are an essential component of virtually every substantial application written in these languages. At least one open-source version of widely used APIs is often available and can be used as a reference. In this use case, programmers of proprietary versions may not even need to read the open-source code, but only study any reported differences in security policies. If all API implementations are proprietary, developers could use this approach to make their respective implementations more secure if one or more of them are willing to share extracted policies.

Security vulnerabilities and other semantic errors in popular libraries open the door to attacks that can compromise many systems, not just a single application. With the advent of cloud computing and increasing demand for portability and architecture-specific optimizations, the prevalence of multiple implementations of the same API is likely to grow. This work shows for the first time *how to leverage multiple implementations to improve the interoperability and security of each.*

Our approach may be applicable to other types of interoperability bugs. As Figure 8 shows, semantic differences sometimes accidentally show up as security-policy differences. A generalization of our analysis that extracts and compares more general semantics of API implementations seems promising. For example, differences in how implementations access parameters, private variables, return values, and flow values from parameters to private variables and return values may reveal interoperability bugs. Similar analysis could detect differences in exceptions that may get thrown by each implementation and in the semantic information carried by return values.

Acknowledgments

Thanks to Andrew John Hughes for his generous help with verifying Classpath bugs and for feedback on the paper text; Sam Guyer for useful discussions about static analysis; and the anonymous reviewers for helpful feedback on the text.

The research described in this paper was partially supported by the NSF grants CNS-0746888, CCF-0811523, CNS-0905602, SHF-0910818, and CCF-1018721, Google research award, and the MURI program under AFOSR Grant No. FA9550-08-1-0352.

References

[1] Amazon. Amazon Web Services. `http://aws.amazon.com/`.

[2] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *ACM Symposium on the Principles of Programming Languages*, pages 4–16, 2002.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *IEEE Working Conference on Reverse Engineering*, pages 86–95, 1995.

[4] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *ACM Symposium on the Principles of Programming Languages*, pages 1–3, 2002.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM European Conference on Computer Systems*, pages 73–85, 2006.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker BLAST. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, 2007.

[7] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2): 244–263, 1986.

[9] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition*, pages 119–129, 2000.

[10] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *ACM Conference on Programming Language Design and Implementation*, pages 435–445, 2007.

[11] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, 2001.

[12] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *IEEE International Conference on Software Maintenance*, pages 109–118, 1999.

[13] E. A. Emerson and E. M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Colloquium on Automata, Languages and Programming*, pages 169–181, 1980.

[14] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *ACM Symposium on Operating Systems Principles*, pages 57–72, 2001.

[15] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In *ACM International Conference on Software Engineering*, pages 458–467, 2007.

[16] Google. Google Apps. `http://www.google.com/apps/`.

[17] D. Grove and L. Torczon. Interprocedural constant propagation: A study of jump function implementations. In *ACM Conference on Programming Language Design and Implementation*, pages 90–99, 1993.

[18] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

[19] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *ACM OOPSLA Onward!*, pages 92–106, 2004.

[20] IBM. Cloud Computing. `http://ibm.com/developerworks/cloud/`.

[21] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *ACM Symposium on the Foundations of Software Engineering*, pages 35–45, 2006.

[22] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 359–372, 2002.

[23] J. Krinke. Identifying similar code with program dependence graphs. In *IEEE Working Conference on Reverse Engineering*, pages 301–309, 2001.

[24] A. M. Leitao. Detection of redundant code using R^2D^2. *Software Quality Control*, 12(4):361–382, 2004.

[25] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: Is it worth it? In *International Conference on Compiler Construction*, pages 47–64, 2006.

[26] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, pages 25–33, 2006.

[27] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatics (ACTA)*, 28(2):121–163, 1990.

[28] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural analysis for privileged code placement and tainted variable detection. In *European Conference on Object-Oriented Programming*, pages 362–386, 2005.

[29] Salesforce. Salesforce Platform. `http://www.salesforce.com/platform/`.

[30] A. P. Sistla, V. N. Venkatakrishnan, M. Zhou, and H. Branske. CMV: Automatic verification of complete mediation for Java Virtual Machines. In *ACM Symposium on Information, Computer and Communications Security*, pages 100–111, 2008.

[31] V. Srivastava. Vulnerabilities submitted to Classpath, Dec 2009–Jan 2010. `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=42390`.

[32] V. Srivastava. Vulnerabilities submitted to Harmony, Nov 2009. `https://issues.apache.org/jira/browse/HARMONY-6367`.

[33] V. Srivastava. Vulnerabilities submitted to Sun JDK, Jan–Oct 2010. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6914460`.

[34] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *ACM Conference on Object–Oriented Programming, Systems, Languages, and Applications*, pages 264–280, 2000.

[35] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou. AutoISES: Automatically inferring security specifications and detecting violations. In *USENIX Security Symposium*, pages 379–394, 2008.

[36] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.

[37] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ACM International Symposium on Software Testing and Analysis*, pages 218–228, July 2002.

[38] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *ACM Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[39] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.