Efficient, Context-Sensitive Detection of Semantic Attacks

UT Austin Computer Sciences technical report TR-09-14. Under submission to CCS 2009.

Michael D. Bond Varun Srivastava Kathryn S. McKinley Vitaly Shmatikov
The University of Texas at Austin
{mikebond, varun, mckinley, shmat}@cs.utexas.edu

ABSTRACT

Software developers are increasingly choosing memory-safe languages such as Java because they help deploy higher-quality software faster. As a result, semantic vulnerabilities—omitted security checks, misconfigured security policies, and other software design errors—are supplanting memory-corruption exploits as the primary cause of security violations.

We present PECAN, a precise, efficient defense against semantic attacks based on dynamic anomaly detection. We show that detection of semantic exploits requires both context and history sensitivity. PECAN supports very efficient run-time tracking of calling contexts and histories, and thus enables accurate detection of unusual behaviors associated with security violations.

We evaluate our approach on several real-world semantic exploits that target subtle bugs in real Java applications and libraries. Our sample attacks are representative of common types of semantic vulnerabilities. All were successfully detected by PECAN. The run-time overhead of our approach on standard benchmarks is 5% on average and at most 9%. The efficiency of PECAN is a qualitative advance in the state of the art: unlike many existing methods, PECAN can be deployed in a production system with a minimal performance penalty. Furthermore, we investigate the tradeoff between sensitivity and accuracy, and empirically demonstrate that PECAN achieves high sensitivity with few false positives.

1. INTRODUCTION

With the increasing popularity of memory-safe languages such as Java, C#, JavaScript, and Ruby, semantic vulnerabilities have overtaken memory corruption bugs as the primary cause of security violations in software applications [27]. Exploitable semantic bugs include accidental omission of access-control checks, unintentional exposure of security-sensitive methods to untrusted code, misconfiguration of security policies, and other security-logic errors.

Detecting attacks that target semantic vulnerabilities is a difficult task. Unlike memory-corruption exploits, semantic attacks do not rely on a violation of the underlying programming language semantics, nor, typically, do they involve code injection. Static analysis-based intrusion detection methods, which ensure that the program's dynamic behavior conforms to the statically inferred control-flow graph and memory-access patterns [1, 2, 9, 14, 40], are ineffective against semantic attacks because the code paths and memory accesses executed during such an attack are already present in the original code. Static and dynamic taint-tracking methods, which can detect injection attacks on Web applications [5, 25, 29, 42], do not help against other semantic attacks, *e.g.*, a malicious Java applet that exploits a sensitive method mistakenly left accessible by a misconfigured security policy.

Dynamic anomaly detection is potentially capable of recognizing that an unusual code path is being executed, and is thus attractive as a defense against semantic attacks. Practical implementations, however, face two major challenges. The first is *precision*. As we show in this paper, detection of many semantic attacks requires context sensitivity, *i.e.*, the defense must consider a method's calling context before making the security decision. Furthermore, some attacks can only be detected by history-sensitive defenses that take into account whether certain calls—for example, those associated with mandatory security checks—have occurred or not. Unfortunately, simplistic approaches such as blindly increasing the size of the context and/or history considered by the detection algorithm can lead to a dramatic increase in the number of false positives when monitoring legitimate executions.

The second challenge is *efficiency*. Naive solutions for context sensitivity, such as direct stack inspection, can result in prohibitive performance overheads, precluding their deployment in production applications.

Efficient context sensitivity is an especially important challenge for modern, object-oriented languages. Programs written in these languages typically have many small methods and use virtual methods, making dynamic calling context critical to understanding program behavior [7]. These features also tend to lead to deeper call stacks, increasing the cost of maintaining calling context in dynamic analyses.

This paper aims to develop a general anomaly detection method that provides both the precision and the efficiency required to detect modern semantic attacks. We emphasize that the objective is not merely a quantitative improvement in performance. In the case of dynamic anomaly detection, there is a *qualitative* difference between an efficient defense that can be feasibly deployed in a production system and techniques that may be of academic interest, but impose prohibitive run-time overheads and/or result in an overwhelming number of false positives when monitoring complex applications and libraries.

We argue that testing on *real-world exploits* should be an essential part of evaluation for any anomaly detection system. None of the previous work on context-sensitive anomaly detection demon-

strated real-world semantic attacks that require non-trivial context sensitivity. For example, Feng *et al.* construct artificial *memory-corruption* exploits to motivate context sensitivity [11], while our focus is on semantic attacks that only use code paths already present in the program. As we show in this paper, real-world exploits often exhibit fairly complex behavior that can expose limitations of naive detection methods. We use representative real-world attacks to calibrate the tradeoff between sensitivity and accuracy in our system.

Our contributions. We design and implement a new approach to detecting semantic attacks on Java applications based on dynamic anomaly detection. We chose Java because of its popularity, but the principles behind our approach apply to other memory-safe languages.

Our attack detection system is called PECAN (Precise, Efficient, Context-sensitive ANomaly detection). PECAN keeps track of the application's execution context and history by dynamically computing a compact, probabilistically unique 32-bit value. When monitoring program execution, PECAN checks this value at each security-sensitive call and verifies whether it was observed during training.

Since higher sensitivity can lead to false positives, the context must be *depth-limited*. This presents a technical challenge: how to compute a depth-limited representation of the calling stack efficiently and *continuously*, as the context changes. To the best of our knowledge, PECAN is the first anomaly detection system that provides context sensitivity of arbitrary depth and history sensitivity for memory-safe languages in a practically efficient manner.

We show the effectiveness of our approach by systematically evaluating it on standard benchmarks and several real vulnerabilities in Java (listed below), which are representative of important categories of semantic security violations. We view the demonstration that context sensitivity is essential for detecting real-world semantic attacks—in particular, attacks on Java applications that do not involve memory corruption and execution of infeasible code paths—as an important contribution of this paper.

The average performance overhead of our detection method is 5%, whereas other context-sensitive techniques such as stack-walking add 100% overhead for some programs.

Four representative vulnerabilities. The first vulnerability is a bug in Sun Java Virtual Machine 1.3 [34]. It permits a malicious applet to circumvent the security manager by providing a class path with "/" instead of ".". This exploit is representative of the "mistakenly omitted security check" category. We emphasize that a conventional history-based anomaly detector would *not* be able to detect this attack because there are legitimate call sequences—for example, when the class is loaded from the root package—in which the security check is not performed. Therefore, it is impossible to differentiate between legitimate and malicious behavior simply by asking whether the execution history contains a `checkPermission` call. Furthermore, detection must take place inside the Java API libraries and not just at the boundary between the applet and the libraries.

The second vulnerability is a bug in the Java 2 Runtime Environment 1.4 [28]. It allows a malicious applet to load unsafe classes and execute arbitrary code via the reflection API. This exploit is representative of the "sensitive methods mistakenly exposed to untrusted code" category.

The third vulnerability is a bug in the Sun Java System Portal Server 7.0 [45], which allows untrusted code contained in a malicious XSLT stylesheet to bypass the standard security checks. This exploit is representative of the "untrusted code executed in the wrong security context" category. It is fundamentally context-dependent, requiring a non-trivial context-sensitive defense. This exploit is also interesting because the vulnerable application is a recursive XML parser, and the depth of recursion depends on the structure of the input. This structure makes it infeasible to enumerate all valid contexts during training, presenting a challenge to any context-sensitive attack detection method (to the best of our knowledge, none of the context-sensitive anomaly detection systems in the literature have been evaluated on recursive applications of this kind). We solve the problem by using *depth-limited* context sensitivity, which is still precise enough to detect the attack.

The fourth vulnerability is a bug in the custom security manager of the Opera Web browser version 7.24, which allows untrusted applets to elevate their privileges [35]. This exploit is representative of the "misconfigured security policy" category.

PECAN detects all of these exploits, demonstrating its effectiveness against many types of semantic attacks. Very precise intrusion detection methods run the risk of "overfitting" the model of normal behavior to the training runs, resulting in a large number of false positives during benign production runs. Our analysis explores the tradeoff between accuracy and sensitivity. We show that by calibrating and bounding the system's context and history sensitivity, dynamic detection can precisely recognize malicious activity without a prohibitive increase in the number of false positives.

2. RELATED WORK

Much of previous research on run-time detection of maliciously behaving applications focuses on defenses against memory-corruption exploits. These attacks affect programs implemented in unsafe languages such as C. In contrast, this paper focuses on attacks that exploit application semantics and thus affect even programs implemented in memory-safe languages such as Java.

Our approach is based on anomaly detection. It constructs a model of correct application behavior in the training phase and then detects deviations from this model during production runs. Even outside the realm of security, anomalies are broadly useful for finding semantic errors. For example, DIDUCE establishes and records dynamic invariants such as variable values, variable inequalities, and branch directions and finds that anomalous paths and values often reveal bugs in programs [16]. Unfortunately, high run-time overheads are a bane of anomaly detection systems. For example, DIDUCE increases execution times of monitored applications by a factor of 2 to 10.

Forrest *et al.* observed that system-call histories associated with security violations often appear anomalous viz. correct executions, and developed a *context-insensitive* anomaly detection system operating at the level of system calls [12, 18]. Sekar *et al.* show how to learn a finite state automaton (FSA), which provides a more compact representation of system-call histories [37]. Because these approaches target memory-unsafe languages, monitoring must be performed beyond the reach of injected malicious code and requires system-call interposition. The latter imposes overheads between 100% and 250% [32, 37]. Inoue and Forrest extend context-insensitive call monitoring to application methods in Java programs [21, 22]. A violation is reported if the program executes a method that has never been seen in a training run. For several vulnerabilities analyzed in this paper (*e.g.*, SlashPath), this simple method would result in false negatives because malicious and benign behavior cannot be differentiated on the basis of system-call history alone. Context-insensitive detection methods can be evaded by "mimicry attacks" [41], in which the malicious code executes the same sequence of system calls as the correct application.

Much work has been devoted to combining static analysis with anomaly detection in order to accurately detect execution of infea-

sible control paths [40]. Because the statically computed control-flow graph over-approximates the set of feasible paths, this approach suffers from false negatives, which can be reduced by adding context sensitivity—for example, using the Dyck model [14], or the Dyck model combined with stack-walking [10]. Infeasible control transfers can also be found by a combination of static and dynamic analysis [48]. On the other hand, Kruegel *et al.* present a mimicry attack against context-sensitive anomaly detection that uses adversarial static analysis of the vulnerable application to construct a fake call stack and then return control back to the injected code [26]. This entire line of research focuses on memory-unsafe languages. By contrast, semantic attacks on applications implemented in Java do not involve execution of any control paths that are not already present in the original code. Therefore, techniques for detecting execution of infeasible paths do not offer any additional power.

It has been observed that context sensitivity is essential for improving precision of anomaly detection [11, 20, 46, 47]. Feng *et al.* record the calling context at each system call and compare successive histories of length two. They prune any stack context that the two successive system calls have in common and store the context differences, which results in more precise detection than using just the history [11]. Performance overhead is only reported in milliseconds per system call, making it difficult to determine the penalty on realistic benchmarks, but the mechanism fundamentally depends on system-call interposition and walking the stack, both of which would be prohibitively expensive in production systems: as we mention above, the overhead of system call interposition can exceed 100% [32, 37], and as we show in Figure 1, the overhead of walking the stack depends on the application and can be very high as well. It appears very challenging to extend the techniques of [11] to object-oriented languages like Java, with many small methods and virtual methods, while achieving practical efficiency. Xu *et al.* introduce waypoints, which make the stack more visible to the monitor, but reduce efficiency [46]. Zhuang *et al.* efficiently find anomalous interprocedural paths, which include calling context, but their approach requires new hardware [47].

In contrast to the prior work on context-sensitive anomaly detection, our approach (1) targets an entirely different class of attacks (we focus on semantic attacks that exploit existing code paths as opposed to memory-corruption attacks that involve execution of infeasible code paths); (2) provides an especially efficient implementation of continuously available context of arbitrary depth that requires neither walking the stack, nor special hardware, and has a demonstrated overhead of 2-9% on realistic applications; and (3) is evaluated on real-world semantic attacks against deployed Java applications that have been reported in the National Vulnerability Database, as opposed to artificially constructed memory-corruption exploits.

3. SEMANTIC VULNERABILITIES

We present several concrete attacks that will serve as our running examples throughout this paper. These exploits are representative of different classes of semantic vulnerabilities and will allow us to explore the tradeoffs between sensitivity and accuracy in dynamic intrusion detection. We use the term *semantic vulnerability* to refer to logic bugs and design errors in the code. Technically, attacks against these vulnerabilities are characterized by control-flow behavior that exploits code paths already present in the code, but not observed during training (in Section 4, we explain why it is realistic to expect that modern techniques for application testing will have exercised all legitimate paths, including error handling). Because attacks of this type have received relatively little attention in the literature, we start by surveying their essential features and explain why they require development of conceptually new defenses.

First, unlike memory-corruption exploits, these attacks do not violate the semantics of the underlying programming language and do not involve execution of statically infeasible code sequences. Therefore, standard methods for ensuring code integrity and memory safety do not help.

Second, unlike cross-site scripting and SQL injection, these attacks do not generate executable statements from untrusted network inputs. Therefore, taint-tracking and similar methods do not help.

Third, all of our sample attacks work "in the wild" and have been reported in the National Vulnerability Database [31]. For the purposes of this paper, we reproduced them in a controlled virtual environment. Previous context-sensitive intrusion detection systems have been evaluated primarily on artificial examples. By contrast, the attacks analyzed in this paper exploit design errors in deployed Java applications and libraries, demonstrating the variety and subtlety of real-world semantic vulnerabilities. Semantic vulnerabilities in memory-safe languages are still poorly understood. We argue that analyzing real-world attacks is a critical component in developing principled defenses, and view our work as a step in this direction.

Two of the four exploits require context sensitivity to be detected (note that there are no examples of real-world exploits whose detection requires context-sensitivity in prior literature—see Section 2). This illustrates the importance of context sensitivity in real-world intrusion detection, even for programs written in memory-safe languages like Java. The other two exploits are detectable with a context-insensitive detector, but we present them here for completeness.

SlashPath. This vulnerability is representative of a common mistake in the implementation of security managers and reference monitors, where the enforcement mechanism forgets to perform a mandatory security check.

In the Java security model, Java libraries rely on the security manager to check access permissions for all security-sensitive actions. A security check in the Sun Java Virtual Machine (JVM) version 1.3 and earlier cannot detect class paths expressed using "/" instead of "." [34]. Therefore, even if the security manager disallows the loading of some class, a malicious applet can circumvent the restriction by specifying the path with "/". After loading, the class executes within the existing applet environment.

This attack is somewhat subtle and cannot be detected by a pure history-based detector. The security check is *correctly* omitted for class paths that do not contain any "/" or "." characters. Therefore, a history-based detector would accept histories that do not contain security checks as legitimate. Similarly, this vulnerability—and "omitted-check" vulnerabilities in general—cannot be characterized as a simple source-sink property, because there exist valid paths on which the check need not be performed.

Our approach, on the other hand, combines context sensitivity with history sensitivity and is thus capable of telling the difference between the class-loading context in which the check should be omitted and those in which it must be performed.

XSLT. This vulnerability is representative of a common class of bugs, where untrusted code is mistakenly executed in the wrong security context (*e.g.*, outside the normal Java sandbox).

"The Java XML Digital Signature implementation in Sun JDK and JRE 6 before Update 2 does not properly process XSLT stylesheets in XSLT transforms in XML signatures, which allows context-dependent attackers to execute arbitrary code via a crafted stylesheet" [45]. Similar vulnerabilities occur in the Sun Java System

Application Server and Web Server 7.0 through 9.0 [43] and the Sun Java System Portal Server 7.0 [44].

Hill provides a detailed explanation of this vulnerability [17]. XSLT (Extensible Stylesheet Language Transform), which is used for XML document processing, permits platform-specific extension mechanisms, including embedded code. The `KeyInfo` field of a digital signature used to authenticate XML and other content can be used to trick the victim system into executing this embedded code outside the normal security context for untrusted code.

We emphasize that this vulnerability is characterized *not* by the fact that untrusted code is executed (which is not a bug, but a feature of XSLT), but by the fact that the code executes in the wrong context. This exploit highlights the importance of context sensitivity in detecting this and similar attacks. Furthermore, because XML parsing is recursive, it is essential that only contexts of a certain depth be considered. We are not aware of any prior work that focused on the importance of bounded context sensitivity.

LiveConnect. This vulnerability involves gaining access to security-sensitive methods that are not normally available to untrusted code.

"The Sun Java Plugin capability in Java 2 Runtime Environment (JRE) 1.4.2_01, 1.4.2_04, and possibly earlier versions, does not properly restrict access between JavaScript and Java applets during data transfer, which allows remote attackers to load unsafe classes and execute arbitrary code by using the reflection API to access private Java packages" [28]. This vulnerability is caused by a design error in a Web browser feature called LiveConnect, which allows Java and JavaScript code to communicate with one another on a Web page, *i.e.*, a Java applet can access JavaScript objects, and JavaScript code can access Java runtime libraries.

The bug in LiveConnect allows a malicious applet to invoke a `netscape.javascript.JSObject` JavaScript object and use it to determine the user's browser and to obtain a reference to `sun.plugin.liveconnect.SecureInvocation`. The applet then disables the security manager with a call to the `setSecurityManager()` method, executed via the `SecureInvocation` proxy. Once outside the sandbox, the applet can download the executable payload and run it on the victim's machine. It can also use the reflection API to get access to desirable methods. In our virtual environment, we reproduced the exploit using code provided by Dino Dai Zovi.

OperaPolicy. This vulnerability is representative of logic errors and misconfigurations in custom security policies.

The Opera 7.54 Web browser uses the JRE directly with a proprietary adapter, in contrast to other major browsers, which use the Java Plugin. Opera also introduces its own default policy, allowing unprivileged applets access to internal Sun-packages by specifying in `Opera.policy`:

```
grant {
  permission java.lang.RuntimePermission
    "accessClassInPackage.sun.*";
};
```

"This [*feature*] opens the gate to some undocumented functionality and violates Sun's guidelines for secure Java programming" [36]. We reproduced this vulnerability with the exploit code provided by Marc Schonefeld on `securityfocus.com`.

4. DETECTING SEMANTIC ATTACKS WITH CONTEXT AND HISTORY SENSITIVITY

We now describe the design and implementation of our PECAN system for detecting semantic attacks on Java code. As explained in Section 3, semantic attacks exploit unintended behaviors of implementations that have not been considered by the programmer. Our approach is based on the standard two-stage paradigm for dynamic anomaly detection. First, PECAN is trained to learn normal behaviors; then, during deployment, the code is monitored to detect execution of unusual behaviors, *i.e.*, those not observed during training.

We assume that **training** is sufficiently thorough to exercise all normal behaviors and thus reduce the false positive rate. Training can be easily "piggybacked" onto standard quality-assurance testing, in which the program is executed on comprehensive test inputs. The tools for systematic, exhaustive testing of all legitimate code paths are now widely available [15, 33, 23]. These approaches are driven by the intended functionality of the software being tested and thus guarantee that the training phase is free of semantic attacks (which, by definition, exploit unintended functionality). Furthermore, these tools specifically exercise error-handling functionality, thus reducing the danger that an infrequently executed code path will be flagged as anomalous after deployment. Our experimental evaluation, too, shows that false positives can be eliminated and/or greatly reduced with sufficient training.

PECAN's **deployment** phase observes program behavior and reports history/context combinations that it did not observe during training. PECAN may be optionally configured to terminate the application when anomalous behavior is detected, or to write a warning to a log. Developers can examine the report to decide whether the anomaly represents an attack or a false positive, *i.e.*, legitimate, but previously unobserved, behavior. The latter can be added to the training set to avoid future false positives.

The critical design issue for any anomaly-detection system is the four-way tradeoff between (i) granularity of monitoring (which method invocations to monitor and how often), (ii) efficiency and scalability, (iii) false positive rate, and (iv) comprehensiveness of training. Frequent monitoring of a large number of methods, with context and history sensitivity, enables more precise and timely detection of anomalies. At the same time, it imposes a larger performance overhead and makes it more likely that the system will generate a false positive due to valid behavior that was not observed during training. Therefore, finer granularity requires much more comprehensive training in order to exercise all possible contexts of every method call in the application. If taken to the extreme of exhaustive testing, the defense would apply only to relatively small applications. The investigation of this tradeoff is one of the contributions of this paper.

4.1 Security Calls

To limit the number of false positives and restrict the amount of information that needs to be maintained by the system, PECAN only tracks calls to methods that can potentially throw a `java.lang.SecurityException`, which we refer to as *security calls*. Security calls are important because they can affect the system outside the JVM, *e.g.*, I/O and system calls. Similarly, prior work on anomaly-based intrusion detection typically tracks behavior at the level of system calls [18]. To identify the methods that can potentially throw a `SecurityException`, our implementation parses the Java API documentation.

The `SecurityManager` class provides methods that enable applications to implement their own security policies. This flexible model leaves applications that need special security policies open to bugs of omission and misconfiguration. For example, the devel-

oper can forget a corner case and omit a needed check in a new policy. This bug can go undetected at testing time and will leave the application vulnerable. Dynamic anomaly detection is the last line of defense in this case.

In our experience, limiting monitoring to security calls provides a good balance between efficiency and precision, while ensuring a low false positive rate. Because these methods are an inherent part of the application's security policy, the context and history of their behavior are indicative of security violations, as confirmed by our experiments.

4.2 Context Sensitivity

Dynamic calling context is the sequence of active call sites that lead to a program location. It is an important component of program behavior because the same call may be malicious or benign, depending on its context. Prior work recognized the importance of context sensitivity for precise anomaly detection, but focused on memory-corruption attacks involving invalid code paths [11, 46, 47]. Inoue considered context sensitivity in intrusion detection, but did not build or evaluate an actual intrusion detector [20]. Context is a critical element of program behavior for programs written in modern, object-oriented languages, which typically have small methods and use virtual methods [7].

Obtaining context is expensive if done frequently. A typical approach is to walk the stack to obtain the list of active call sites [30], but as our experiments show, the overhead of stack-walking is high for some applications. An alternative approach is to build a calling context tree (CCT) in which each node represents a distinct calling context [4, 38]. Application-level instrumentation constructs the CCT and maintains each thread's position in the CCT. When the intrusion detector needs to record the current context, it simply records a pointer to the current node in the CCT. Unfortunately, CCT-based approaches add high time and space overhead.

Our efficient solution is to keep track of context continuously and *probabilistically* by continuously computing a probabilistically unique value (e.g., a 32-bit integer) that represents the current calling context. PECAN uses this value to represent context-sensitive calls; an anomalous value indicates an anomalous context-sensitive call. Prior dynamic analyses have computed a hash value for calling contexts, e.g., computing a hash value is essential for virtually any context-sensitive analysis, in order to look up contexts in a hash table. However, unlike most prior approaches, (1) we want to compute context incrementally, i.e., compute a new hash value at each call site using only the current hash value and a call site identifier, and (2) we need a function that produces ideally very few conflicts, i.e., two distinct contexts that map to the same hash value.

Prior work introduces *probabilistic calling context* (PCC), which computes a probabilistically unique value that naturally represents every call site in the current calling context [7]. However, we have found that full context sensitivity provides too much sensitivity, resulting in many false positives on real programs, especially highly object-oriented and recursive programs. For example, XML processing performed by XSLT often executes mutually recursive contexts, resulting in many contexts flagged as anomalous even after thorough training. This example crisply illustrates the tradeoff between precision and accuracy, and motivates the need for *k-limited context sensitivity*, which limits context to the top k methods on the stack.

While the function from prior work represents a infinite-depth calling context [7], it is challenging to design a function that produces values that represent only a fraction of context. particularly so that each call site in the depth-limited context affects many bits of the value (to reduce the potential for conflicts between similar contexts). The difficulty arises because, at each call site, the function needs to "eliminate" the kth call site from the calling context value so that the value represents only the top k call sites on the stack. We propose an approach called *k-limited probabilistic calling context (k-PCC)*, and we introduce the *k-PCC function* as follows:

$$f(V, cs) \equiv 2^{\lceil bits/k \rceil} \times V + cs$$

The function takes two inputs: the k-PCC value, V, and an identifier for the call site at which the function is computed, *cs*. In our implementation, both of these inputs are 32-bit values. On the right-hand side, *bits* is the size of the k-PCC value (32 in our implementation), and k is context depth. For example, if $k = 3$,

$$f(V, cs) \equiv 2^{11} \times V + cs$$

This function is equivalent to shifting the current k-PCC value 11 bits to the left, then adding the call site value. Bits affected by call sites lower on the stack are pushed off the end of the value, so that only the top k call sites affect the PCC value.

We modify the dynamic, just-in-time (JIT) compiler in the JVM to insert instrumentation at each call site that computes the next k-PCC value from the current k-PCC value and the current call site ID. The following example shows the instrumentation the compiler adds to a method:

```
  method() {
      int tmp = V; // save current k-PCC value
      ...
      V = f(tmp, cs_1); // compute k-PCC value
cs_1: foo();
      ...
      V = f(tmp, cs_2); // compute k-PCC value
cs_2: bar();
      ...
      V = f(tmp, cs_3); // compute k-PCC value
cs_3: bar();
      ...
  }
```

Note that different k-PCC values are computed at different call sites to the same method (`cs_2` and `cs_3`).

This instrumentation continuously maintains the k-PCC value, but this value only needs to be *examined* at call sites of interest, which in the case of PECAN are security calls. Consider the example below. The system adds instrumentation to compute the k-PCC value at all call sites, but only checks the k-PCC value for calls that may throw a `SecurityException`.

```
  method() {
      int tmp = V; // save current k-PCC value
      ...
      V = f(tmp, cs_1); // compute k-PCC value
cs_1: foo();
      ...
      V = f(tmp, cs_2); // compute k-PCC value
      check(V);         // check k-PCC value
cs_2: SecurityManager.checkPermission(...);
      ...
      V = f(tmp, cs_3); // compute k-PCC value
      check(V);         // check k-PCC value
cs_3: readFile(...);
      ...
  }
```

The `check()` method looks up the k-PCC value in a global hash table:

```
  check(V) {
    if (!table.contains(V)) {
```

```
    table.add(V);
    if (deployed) {
      walkStack();
      reportAnomaly();
    }
  }
}
```

If the value is anomalous, then the resulting context is guaranteed to be anomalous. If PECAN is executing in training mode, it simply adds the anomalous k-PCC value to the table. In deployed mode, it also reports the anomalous context, which it obtains by walking the stack.

A disadvantage of the k-PCC function is that only the top call site affects all bits in the k-PCC value. For $k = 3$, the top call site affects all 32 bits of the PCC value, the second call site affects 21 bits, and the third call site affects only 10 bits. Thus the chance of a conflict may be increased if another call site shares the third call site's lowest 10 bits. However, a mediating factor is that for a conflict to occur, this call site must be capable of calling the second call site and causing it to invoke the top call site. In practice, we find that k-limited context sensitivity is sufficient for accurately recognizing anomalies associated with real attacks.

4.3 History Sensitivity

Context sensitivity alone is not sufficient for detecting real-world semantic exploits. Program *history* is an essential ingredient of accurate anomaly detection, as we show in Section 5 and others have shown [12, 18, 37]. For example, Java API methods often call a security check method, such as `SecurityManager.checkPermission()`, prior to a security call that performs some potentially dangerous action, *e.g.*, reading a file. If the file read occurs without a prior `SecurityManager` check, this anomaly represents a possible attack. Note that there are two types of security calls: (1) calls to `SecurityManager` methods that check whether an action is permitted and (2) calls that actually perform some potentially dangerous task. To reduce the number of histories that need to be tracked and to mitigate false positives, PECAN only considers the program's history of calls to `SecurityManager`, because correctly executing these checks is critical to enforcing security policies.

PECAN naturally combines history and context sensitivity by combining prior k-PCC values for `SecurityManager` calls with the current k-PCC value. The following modified `check()` method incorporates history using a hash function $h(H, V)$:

```
check(value) {
  if (useHistory) {
    H = h(thread.history, V);
    if (isSecurityManagerCall) {
      thread.history = V;
    }
  }
  if (!table.contains(H)) {
    table.add(H);
    if (production) {
      walkStack();
      reportAnomaly();
    }
  }
}
```

Each thread uses a variable `thread.history` that maintains the `SecurityManager` call history, if any. The `check()` method hashes this value together with the current k-PCC value V to obtain a new value H. Whenever `check()` is called by a `SecurityManager` call (which occurs at some, but not all security calls), it updates H to include this latest `SecurityManager` call. We have found that, as with context sensitivity, using unlimited history provides too much sensitivity, resulting in many false positives. Thus, PECAN uses only the k-PCC value from the most recent `SecurityManager` call, and combines it with the current k-PCC value.

PECAN uses the following function $h(H, V)$ for hashing together history and k-PCC values (this is the same function as used in [7]):

$$f(H, V) \equiv 3 \times H + V$$

In the rest of this paper, we will refer to the k-limited probabilistic context/history value tracked by our system as the *k-PCH value*.

4.4 Component Granularity

Modern software is usually assembled from independently developed components. PECAN training and monitoring may be applied only to some of the components. For example, an application developer may configure PECAN to instrument only the application that she is implementing, and in the deployment stage only monitor for anomalies in that application (as opposed to the Java libraries). On the other hand, an implementer of a library routine may only be interested in anomalous executions inside the code he is responsible for, as opposed to the entire context from the application to the library. When PECAN is applied to the Java libraries, it resets history before each application $\rightarrow$ library call. This helps avoid mimicry attacks, *e.g.*, a malicious applet might call a security check prior to calling a buggy library method in a such a way that a security check is skipped.

5. EVALUATION

This section evaluates PECAN's performance and ability to detect attacks. We first describe our implementation of PECAN in a Java Virtual Machine. Then we compare the overhead of k-PCH to stack-walking. Next we evaluate PECAN's ability to detect real-world semantic exploits. Finally, we perform leave-one-out cross-validation on non-vulnerable programs to evaluate PECAN's false positive rate.

5.1 Implementation

We implemented PECAN in Jikes RVM 2.9.2, a research Java Virtual Machine [3, 24]. Jikes is a research tool, but its performance compares well with commercial VMs: same, on average, as Sun Hotspot 1.5, and 15–20% worse than Sun 1.6, JRockit, and J9 1.9, as of August 2008 [8]. Our performance measurements are thus relative to an excellent baseline.

Like other VMs, Jikes RVM uses just-in-time compilation to produce machine code for each method at run time. When a method executes for the first time, a baseline compiler quickly generates machine code directly from bytecode. If a method executes many times and becomes *hot*, the VM recompiles it with an optimizing compiler at successively higher optimization levels. We modify both compilers to insert instrumentation that (1) maintains the k-PCH value and (2) records (training) or checks (deployment) the k-PCH value at method calls that can potentially throw a `SecurityException`.

5.2 Performance

PECAN adds overhead to applications because it inserts instrumentation to track the k-PCH value and check it at security-critical method calls. In this section, we show that using k-PCH for context sensitivity is superior to techniques such as stack-walking. Walking the stack only when needed, *i.e.*, at security-critical points, sometimes has low overhead, but in some programs increases overhead by factors of two or more, for unlimited context sensitivity, and up

to 31% for k-limited context sensitivity when $k = 3$. By contrast, the overhead of k-PCH is consistently low across multiple benchmarks.

Figure 1 shows the normalized execution time of our approach for the DaCapo benchmarks and fixed-workload version of SPECjbb2000 called pseudojbb [6, 39]. Each bar is the median of three trials. We use an execution methodology called *replay compilation* to eliminate nondeterminism due to timer-based sampling [13, 19]. We exclude the bloat benchmark since its performance is erratic even with replay compilation.

Each bar is the overhead compared with the execution time on unmodified Jikes RVM.[1] *Pecan* is the overhead of continuously maintaining k-PCH values and checking them at security calls. We use $k = 3$, but the overhead is the same for any value of k. Pecan adds 5% on average and at most 9%. We have found that almost all of this overhead comes from maintaining the k-PCH value; less than a tenth of the overhead comes from checking the k-PCH value at security calls. The *Walk stack* configurations show the estimated overhead of alternate approaches that walk the stack when context is needed, rather than keeping track of context continuously. These configurations walk the top three call sites and all call sites of the stack, respectively. They have low overhead for most programs, but for two programs (antlr and pseudojbb) they add very high overhead. Walking the entire stack adds very high overhead for these programs, but admittedly full context sensitivity produces too many false positives in practice, as we show in Section 5.4. Even for context sensitivity with depth 3, overheads are lower, but as high as 18 and 31% for antlr and pseudojbb, which have relatively frequent security calls. Programs with more frequent security calls will incur higher overheads. In short, stack-walking does not scale well to higher levels of context sensitivity, nor to more frequent security checks. In contrast, k-PCH's overhead is minimally affected by higher values of k and programs with more frequent security checks.

5.3 Detecting Real Attacks

This section evaluates PECAN's ability to detect semantic attacks from Section 3. For each of the exploits, we train the system using a benign input that leads to functionality that is similar to the exploit (but not malicious). For example, the training runs for SlashPath load a class named using the conventional dot syntax and a class that has no package (and thus uses no dots). We then execute the exploits in the trained system and observe whether PECAN reports anomalous behavior. Our experiments explore several combinations of granularity, history, and context sensitivity.

The *granularity* of PECAN's checks is important for avoiding both false positives and false negatives. As discussed previously, PECAN checks the k-PCH value only at call sites to methods that can throw a `SecurityException`, since these security calls may be harmful, but are not so numerous as to cause many false positives. Similarly, developers may choose to further restrict the granularity of these checks to reduce false positives.

The first three exploits, SlashPath, LiveConnect, and OperaPolicy, are applets that take advantage of misconfigured security policies or bugs in the Java libraries. To detect this class of vulnerabilities (*i.e.*, errors in libraries), monitoring can be restricted to the libraries because exploits will trigger anomalies in the library code. Furthermore, it does not make sense to check for anomalies in the applets because each applet has different code, which may not even be known in advance, and will generate many false positives. The fourth exploit, XSLT, takes advantage of a semantic bug in a specific application. To detect this class of vulnerabilities, monitoring must be performed inside that application.

We experiment with two levels of history sensitivity: none and one. Without history, PECAN records and checks k-PCH values that represent the current (context-sensitive) program location. With history of level one, PECAN combines the current location with the previous context-sensitive call to `SecurityManager` methods (Section 4.3). One level of history is sufficient to detect all of our attacks (often context sensitivity is also needed). We have experimented with infinite history as well, but we do not report results because we found this configuration reports too many false positives.

Finally, we experiment with different amounts of *context sensitivity*. By default, PECAN uses three levels of context sensitivity. We compare it to lower levels of context sensitivity: 0, which uses the callee method as the program location, and 1, which uses the caller method as the program location. We have collected results for infinite context sensitivity, but we do not present them here because these configurations produce many false positives for the programs in Section 5.4. Deeper context sensitivity leads to a richer set of behaviors and thus more false positives, but some attacks (*e.g.*, SlashPath) require context sensitivity to be detected.

The rest of this section refers to configurations of PECAN with the notation $C_k H_h$, where k is the context depth and h is the amount of history used. Our recommended configuration, bolded in the tables that follow, is $C_3 H_1$, which can detect all exploits but also produces few false positives.

SlashPath. The SlashPath vulnerability exploits the fact that Sun JVM 1.3 does not correctly check whether it is okay to load a class if that class's package is delimited with slashes (*e.g.*, `sun/applet/AppletClassLoader`) instead of dots (*e.g.*, `sun.applet.AppletClassLoader`). We found that this vulnerability is present in the system class loader in Jikes RVM. This loader calls the security manager to check if the application can load the package, but it assumes that package names are delimited with dots. The following code shows a simplified version of the vulnerabile class loader:

```
protected Class loadClass(String name,
                          boolean resolve) {
  SecurityManager sm =
    SecurityManager.current;
  if (sm != null) {
    int lastDot = name.lastIndexOf('.');
    if (lastDot != -1)
      // [paper authors' note:
      //  won't execute if no dots in name]
      String pkg =
        name.substring(0, lastDot);
      sm.checkPackageAccess(pkg);
  }
  return super.loadClass(name, resolve);
}
```

Our exploit code (based on an available sample exploit [34]) is an applet that loads a class in a package that applets should not be able to access. We execute this applet with a custom-defined security manager that allows all operations. This setup makes sense because one important application of PECAN is to detect malicious behavior allowed by a faulty security manager or security policy.

Table 1 shows results for executing the SlashPath attack in a system monitored by PECAN. Each row is a configuration with varying levels of context sensitivity and history. The cells show the number of anomalous behaviors associated with the attack and, in

[1] Overheads are negative in a few cases because of architectural effects, *e.g.*, instrumentation perturbs code layout, which can affect caching performance for better or worse.

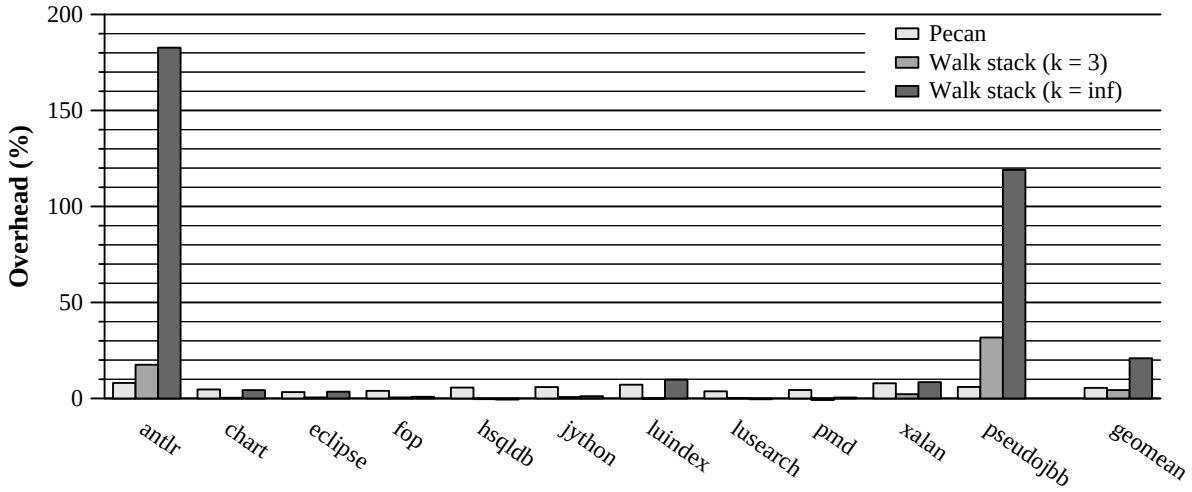

Figure 1: Application execution time overhead of maintaining the k-PCH value and querying it at calls to `SecurityException` methods, compared with walking the stack.

	No history			1-level history		
k	Config	Anom	(All)	Config	Anom	(All)
0	C_0H_0	0	(35)	C_0H_1	0	(59)
1	C_1H_0	0	(54)	C_1H_1	1	(90)
3	C_3H_0	0	(110)	$\boldsymbol{C_3H_1}$	**2**	**(145)**

Table 1: Intrusion detection results for SlashPath. Detecting this exploit requires both context sensitivity and history.

	Config	Anom	(All)
CS = 0	C_0H_1	0	(20)
CS = 1	C_1H_1	0	(40)
CS = 3	$\boldsymbol{C_3H_1}$	**2**	**(42)**

Table 2: Intrusion detection results for XSLT. Detecting this exploit requires context sensitivity. `SecurityManager` history is not relevant since PECAN profiles only the application, which does not make `SecurityManager` calls.

parentheses, the total number of behaviors observed during training. We only check for anomalies inside the Java libraries, because the objective is to detect exploitation of faults in the security logic of library code.

Table 1 shows that context and history sensitivity are required for PECAN to detect the SlashPath attack. Sometimes the number of anomalies is greater than 1 because an attack triggers multiple anomalous k-PCH values. In general, the tables can be interpreted as follows: if *Anom* is 0, the PECAN configuration cannot detect the attack; otherwise, the PECAN configuration detects the attack, although detection could fail with a more thorough training set because the call used in the attack would no longer appear anomalous.

We also implemented a *mimicry* attack that calls `SecurityManager.checkPackageAccess()` immediately prior to attempting to load a class name delimited with slashes. This attack would defeat naive history-based detection, but is successfully detected by PECAN because PECAN clears history on each applet → library call.

XSLT. To reproduce this exploit in our Jikes-based experimental setup, we used the Xalan XML parsing library, which comes with the Sun JVM. We wrote an XSL file with embedded code in the "select" attribute of the `xsl:variable` tag. If a user gives this XSL file as an input for parsing an XML command, the JVM executes the embedded code on the client machine. The code invokes a test script, to which it should not have access, on the client machine.

Table 2 shows that context sensitivity is essential for detecting the attack. We do not show various levels of history for this attack because the XSLT application does not directly call any `SecurityManager` methods (PECAN only instruments the application for this exploit, since it is a standalone application and not an applet). Thus, results are equivalent regardless of the amount of history sensitivity.

Context is essential because executing arbitrary security-sensitive methods from the context of parsing XSL files is semantically incorrect, but it is reasonable for the XSLT application to call security-sensitive methods in some other context (*e.g.*, to load local configuration files). Our training set calls security-sensitive methods outside the context of XSL parsing in order to demonstrate the need for context sensitivity.

LiveConnect. As with the SlashPath exploit, we execute LiveConnect with a security manager that allows all operations, so PECAN can record all behavior during training and report anomalous behavior during deployment. LiveConnect uses Sun's browser plugin, so we track calls only in the plugin, not in the other Java libraries.

Table 3 shows the anomalies reported by PECAN for different amounts of context sensitivity and history. For this exploit, PECAN detects anomalous behavior regardless of the amounts of context sensitivity and history. The reason is that the exploit relies on calling a method that should not be accessible to applets, so a benign applet will not call it. Thus, calling this method always triggers an anomaly, even without context or history sensitivity. However, a more thorough training set or a potential mimicry attack could further constrain the precision required to detect this exploit.

OperaPolicy. The OperaPolicy attack exploits the security policy of the Opera 7.54 browser. Our test exploit uses the `getBootstrapClassPath()` method of the `sun.misc.Launcher` class to get the URLs and access core JVM library classes of the `sun.*` package.

We reproduce the exploit in Jikes RVM with a security manager that allows all behaviors. Table 4 shows that PECAN detects

k	No history Config	Anom	(All)	1-level history Config	Anom	(All)
0	C_0H_0	6	(6)	C_0H_1	6	(6)
1	C_1H_0	6	(6)	C_1H_1	6	(6)
3	C_3H_0	6	(6)	$\boldsymbol{C_3H_1}$	**6**	**(6)**

Table 3: Intrusion detection results for LiveConnect. Detecting this exploit does not require context or history sensitivity.

k	No history Config	Anom	(All)	1-level history Config	Anom	(All)
0	C_0H_0	3	(3)	C_0H_1	3	(3)
1	C_1H_0	4	(4)	C_1H_1	4	(4)
3	C_3H_0	5	(5)	$\boldsymbol{C_3H_1}$	**5**	**(5)**

Table 4: Intrusion detection results for OperaPolicy. Detecting this exploit does not require context or history sensitivity.

the attack at all levels of history and context sensitivity. Similar to LiveConnect, the OperaPolicy calls a method that should not be accessible. When the attack code calls this method, it appear anomalous regardless of the amount of context or history sensitivity.

5.4 Evaluating False Positives with Regular Programs

The prior results showed how well PECAN detects semantic exploits, *i.e.*, how well it avoids false negatives. Now we estimate PECAN's false positive rate by evaluating it on *non-vulnerable* applications, since any anomalies must be false positives. We use two classes of programs: applets, which are similar to the first three vulnerabilities, and XSL inputs, which are similar to the XSLT exploit.

We use *leave-one-out cross-validation* to measure false positives fairly. For each of n programs, PECAN trains on the other $n-1$ programs.

Table 5 shows the number of false positives (anomalous k-PCH values) using leave-one-out cross-validation for 12 sample applets. The methodology of training on one set of applets and deploying on a different applet is reasonable because PECAN only profiles the libraries called by the applets, not the applets themselves (Section 4.4). The number in parentheses is the total number of distinct k-PCH values. For higher levels of context and history sensitivity, there are many more false positives. We do not show these configurations because the number of false positives makes them impractical. This highlights the advantage of using depth-limited (rather than infinite) context sensitivity.

For our recommended configuration, C_3H_1, the number of false positives is always less than 10 and often equal to 0. For the four applets with more than one false positive, the number of anomalous *behaviors* is fewer than the number of false positives because a single anomalous execution path often executes several security calls. AtomViewer, DitherTest, Euler, and ReflFrame execute just 1, 3, 3, and 2 distinct anomalous behaviors. Even if the false-positive rate shown for C_3H_1 is too high for production use, standard industrial testing will be much more comprehensive than the limited set of programs we use here, further reducing the number of false positives (Section 4).

Table 6 shows false positives using leave-one-out cross-validation running XSLT on eight XSL inputs we obtained by searching with Google. History sensitivity is omitted since XSLT does not call `SecurityManager` methods directly, so results are not affected by history sensitivity (Section 5.3). The number of false positives is low: 0 in most cases and 2 at most. The false positive rate could be even lower with a more comprehensive test suite.

6. LIMITATIONS AND TRADEOFFS

Like most anomaly detection methods, PECAN can suffer from false positives (valid calls mistakenly flagged as attacks) and false negatives (calls associated with attacks permitted to go through). Because PECAN considers a call to a security-sensitive method anomalous if it occurs in a context and/or history other than those observed during the training phase, good training is crucial for accurate detection.

To minimize false negatives, the training runs should be attack-free. This assumption is common to anomaly detection systems, and we do not view it as a significant limitation. As we argue in Section 4, modern approaches to comprehensive software testing are driven by the software's intended functionality, and it is highly unlikely that automatically generated test inputs contain a semantic attack. As we show in Section 5, PECAN has been able to successfully detect a broad range of semantic attacks. While mimicry attacks remain possible, they are significantly more difficult to stage because they have to match not just the history, but also the context in which a legitimate call takes place. In one of our experiments, PECAN successfully detected an (artificially constructed) mimicry attack targeting the SlashPath vulnerability.

To minimize false positives, it is important that the training phase exercise all legitimate contexts of the application. This can be achieved during normal testing, as long as the latter tests all of the application's intended functionality. Developing comprehensive test suites is a topic of active research, and there exist several tools for systematic generation of test inputs with exhaustive coverage [15, 33, 23]. As shown by our experiments in Section 5, PECAN generates few, if any, false positives on real-world Java applications. That said, there is a tradeoff between precision and the number of false positives. The more precise the contexts (*i.e.*, the deeper the context and the longer the history), the higher the chance that a legitimate context, which varies only slightly from a context observed in training, will be flagged as anomalous.

Recursive applications such as XML parsers, which is one of our case studies, present an interesting challenge to context-sensitive anomaly detection. Because the depth of recursion depends on the structure of the input and a typical input may result in dozens or even hundreds of recursive calls, it is not feasible to enumerate all possible valid calling contexts during training. A typical run involves thousands of distinct contexts, which vary only in the number of invocations of some recursive function. It is worth noting that none of the existing context-sensitive intrusion detection methods (see Section 2) have been evaluated on applications that exhibit this behavior.

Our solution is to consider only contexts of depth k and history of length h during both training and detection, where k and h are parameters to the system. In our experience, this provides sufficient precision to detect attacks, yet does not generate hundreds of false positives due to new recursive contexts which have not been observed during training. The recommended configuration C_3H_1 detects all the real-world exploits and incurs few false positives on benign programs.

7. SUMMARY

Semantic attacks are hard to detect because they violate rules that typically exist only in the programmer's head. Anomaly detection can help recognize attacks, but many existing methods suffer from false positives and poor performance. PECAN is a novel anomaly detection system for Java with probabilistic, depth-limited context

	False positives (total distinct behaviors)											
	ArcTest		AtomViewer		CardTest		DiffEq		DitherTest		DrawTest	
C_0H_0	0	(31)	0	(31)	0	(31)	0	(33)	0	(31)	0	(31)
C_0H_1	0	(53)	0	(53)	0	(52)	0	(60)	0	(54)	0	(52)
C_1H_0	0	(45)	0	(45)	0	(45)	0	(56)	0	(45)	0	(45)
C_1H_1	0	(75)	0	(74)	0	(71)	0	(100)	0	(75)	0	(71)
C_3H_0	0	(96)	0	(93)	0	(93)	0	(125)	4	(100)	0	(93)
$\boldsymbol{C_3H_1}$	**1**	**(127)**	**9**	**(125)**	**0**	**(111)**	**1**	**(184)**	**7**	**(133)**	**0**	**(123)**
	Euler		Gas		Matrix		Puzzle		ReflFrame		StringWave	
C_0H_0	0	(33)	0	(31)	0	(33)	0	(31)	0	(31)	0	(31)
C_0H_1	0	(60)	0	(54)	0	(54)	0	(52)	0	(43)	0	(42)
C_1H_0	0	(56)	0	(45)	0	(56)	0	(45)	0	(44)	2	(47)
C_1H_1	0	(100)	0	(75)	0	(100)	0	(74)	0	(62)	0	(55)
C_3H_0	2	(127)	0	(99)	0	(121)	0	(93)	4	(89)	0	(65)
$\boldsymbol{C_3H_1}$	**6**	**(189)**	**1**	**(130)**	**0**	**(173)**	**0**	**(123)**	**6**	**(114)**	**0**	**(84)**

Table 5: Leave-one-out cross-validation for 12 non-vulnerable applets. Even though this experiment represents relatively little training compared with expected industrial efforts, false positive rates are low for our recommended configuration, C_3H_1. For the applets with more than one false positive k-PCH value, the number of anomalous *behaviors* is smaller the number of false positives because one anomalous path often results in several security calls.

	False positives (total distinct behaviors)													
	ui		resume		testcase		testcase2		testcase3		testcase4		testcase5		testcase6	
C_0H_1	0	(5)	0	(5)	0	(6)	0	(5)	0	(5)	0	(5)	0	(6)	0	(5)
C_1H_1	0	(21)	0	(21)	0	(23)	2	(22)	0	(21)	1	(22)	0	(23)	0	(21)
$\boldsymbol{C_3H_1}$	**0**	**(22)**	**0**	**(22)**	**1**	**(25)**	**2**	**(23)**	**0**	**(22)**	**1**	**(22)**	**0**	**(23)**	**0**	**(21)**

Table 6: Leave-one-out cross-validation for eight non-vulnerable XSLT inputs. The recommended configuration of PECAN, C_3H_1, generates few false positives. `SecurityManager` history is not relevant since PECAN profiles only the application, which does not make `SecurityManager` calls.

and history sensitivity and low overhead. We evaluate PECAN on four real-world exploits and with various levels of context and history sensitivity. Context and history sensitivity are both important, but limiting them is key to keeping false positives low. PECAN's demonstrated ability to detect attacks precisely, accurately, and efficiently on real-world programs makes it compelling for all-the-time use in deployed systems.

Acknowledgments

We would like to thank Elton Pinto for help with finding and reproducing semantic exploits; Brad Hill, Marc Schonefeld, and Dino Dai Zovi for providing exploit code; Graham Baker for porting PCC to Jikes 2.9.2; Chris Ryder for PCC bug fixes; Sam Guyer for helpful discussions; and Wei Le and the anonymous reviewers for valuable feedback on the text.

8. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM Conference on Computer and Communications Security*, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory errors with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. Shepherd, S. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[4] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *ACM Conference on Programming Language Design and Implementation*, pages 85–96, Las Vegas, NV, 1997.

[5] P. Bisht, P. Madhusudan, and V. Venkatakrishnan. CANDID: Preventing SQL injection attacks using dynamic candidate evaluations. *TISSEC*, 2008.

[6] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 169–190, 2006.

[7] M. D. Bond and K. S. McKinley. Probabilistic Calling Context. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–112, 2007.

[8] DaCapo Benchmark Regression Tests. `http://jikesrvm.anu.edu.au/~dacapo/`.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In *OSDI*, 2006.

[10] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *IEEE Symposium on Security and Privacy*, pages 194–208, 2004.

[11] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *IEEE Symposium on Security and Privacy*, page 62. IEEE Computer Society, 2003.

[12] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *IEEE Symposium on*

Security and Privacy, pages 120–128, 1996.

[13] A. Georges, L. Eeckhout, and D. Buytaert. Java Performance Evaluation through Rigorous Replay Compilation. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 367–384, 2008.

[14] J. Giffin, S. Jha, and B. Miller. Efficient Context-Sensitive Intrusion Detection. In *Network and Distributed Systems Security Symposium*, 2004.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[16] S. Hangal and M. S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *ACM International Conference on Software Engineering*, pages 291–301, 2002.

[17] B. Hill. Command injection in XML signatures and encryption. `http://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf`, 2007.

[18] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[19] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The Garbage Collection Advantage: Improving Program Locality. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.

[20] H. Inoue. *Anomaly Detection in Dynamic Execution Environments*. PhD thesis, University of New Mexico, 2005.

[21] H. Inoue and S. Forrest. Anomaly Intrusion Detection in Dynamic Execution Environments. In *Workshop on New Security Paradigms*, pages 52–60, 2002.

[22] H. Inoue and S. Forrest. Inferring Java Security Policies Through Dynamic Sandboxing. In *International Conference on Programming Languages and Compilers*, pages 151–157, 2005.

[23] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. A concolic whitebox fuzzer for Java. In *NASA Formal Methods Workshop*, 2009.

[24] Jikes RVM. `http://www.jikesrvm.org`.

[25] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *S&P*, 2006.

[26] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating Mimicry Attacks Using Static Binary Analysis. In *USENIX Security Symposium*, pages 11–11, 2005.

[27] 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 *ASID*, pages 25–33, 2006.

[28] CVE-2004-1029. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-1029`, 2004.

[29] B. Livshits and M. Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security*, 2005.

[30] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM Conference on Programming Language Design and Implementation*, pages 89–100, 2007.

[31] National Vulnerabilities Database. `http://nvd.nist.gov/`.

[32] N. Provos. Improving host security with system call policies. In *USENIX Security*, 2003.

[33] V. Roubtsov. EMMA: a free Java code coverage tool. `http://emma.sourceforge.net/`, 2005.

[34] SecurityFocus. Sun Java Virtual Machine slash path security model circumvention vulnerability. `http://www.securityfocus.com/bid/8879/info`, 2003.

[35] SecurityFocus. Java vulnerabilities in Opera 7.54. `http://www.securityfocus.com/archive/1/381634`, 2004.

[36] SecurityTracker. Opera Java sandbox flaws let malicious applets access system information and crash the browser. `http://securitytracker.com/alerts/2004/Nov/1012279.html`, 2004.

[37] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[38] J. M. Spivey. Fast, Accurate Call Graph Profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[39] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[40] D. Wagner and D. Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[41] D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[42] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *ACM Conference on Programming Language Design and Implementation*, 2007.

[43] CVE-2007-3715. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3715`, 2007.

[44] CVE-2007-4289. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-4289`, 2007.

[45] CVE-2007-3716. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3716`, 2007.

[46] H. Xu, W. Du, and S. J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *International Symposium on Recent Advances in Intrusion Detection*, pages 21–38, 2004.

[47] T. Zhang, X. Zhuang, S. Pande, and W. Lee. Anomalous Path Detection with Hardware Support. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 43–54, 2005.

[48] X. Zhuang, T. Zhang, and S. Pande. Using Branch Correlation to Identify Infeasible Paths for Anomaly Detection. In *IEEE/ACM International Symposium on Microarchitecture*, pages 113–122, 2006.