Efficient, Context-Sensitive Detection of Real-World Semantic Attacks*

Michael D. Bond Varun Srivastava Kathryn S. McKinley Vitaly Shmatikov

Department of Computer Science
The University of Texas at Austin
{mikebond,varun,mckinley,shmat}@cs.utexas.edu

ABSTRACT

Software developers are increasingly choosing memory-safe languages. 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. Semantic attacks are difficult to detect because they violate *program* semantics, rather than *language* semantics. This paper presents PECAN, a new dynamic anomaly detector. PECAN identifies unusual program behavior using history sensitivity and depth-limited context sensitivity. Prior work on context-sensitive anomaly detection relied on stack-walking, which incurs overheads of 50% to over 200%. By contrast, the average overhead of PECAN is 5%, which is low enough for practical deployment. We evaluate PECAN on four representative *real-world attacks* from security vulnerability reports. These attacks exploit subtle bugs in Java applications and libraries, using legal program executions that nevertheless violate programmers' expectations. Anomaly detection must balance precision and sensitivity: high sensitivity leads to many benign behaviors appearing anomalous (false positives), while low sensitivity may miss attacks. With application-specific tuning, PECAN efficiently tracks depth-limited context and history and reports few false positives.

1. INTRODUCTION

With the increasing popularity of memory-safe languages such as Java, C#, JavaScript, and Ruby, semantic vulnerabilities are overtaking memory corruption bugs as the primary cause of security violations in software applications [25]. 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 inject malicious code.

To understand and characterize semantic security exploits, we examined hundreds of reports in the National Vulnerability Database and reproduced four representative attacks. Detection of these and similar attacks requires both context sensitivity and history sensitivity. 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, 3, 8, 14, 33], are ineffective against semantic attacks because the malicious code paths and memory accesses are *already present in the original code*. Static and dynamic taint-tracking can detect injection attacks on Web applications [5, 23, 27, 35], but cannot detect semantic attacks, e.g., when a malicious Java applet exploits a sensitive method that is mistakenly left accessible by a misconfigured security policy. Prior dynamic techniques for detecting anomalous contexts are either too imprecise, or too inefficient, or both. (A detailed comparison with related work can be found in Section 3 is and summarized in Table 1.)

Dynamic anomaly detection recognizes *unusual* rather than illegal code paths, thus practical implementations face two major challenges: *precision* and *efficiency*. Detailed analysis of real-world exploits shows that detecting semantic attacks requires context sensitivity, i.e., the defense must consider a method's calling context when making the security decision. Furthermore, some attacks require history sensitivity in order to detect whether or not certain calls, such as mandatory security checks, have occurred. Simplistic approaches such as blindly increasing the size of the context and/or history can dramatically increase the number of false positives (reporting anomalies when there is no actual attack) when monitoring legitimate executions.

The second challenge is *efficiency*. Naïve solutions for context sensitivity, such as direct stack inspection, result in prohibitive performance overheads of 100% or more, precluding their deployment in production applications. Efficient context sensitivity is especially challenging for modern, object-oriented languages, which exercise millions of distinct calling contexts [7]. Prior work on *probabilistic calling context* (PCC) tracks calling context efficiently for the purpose of detecting anomalous context-sensitive behavior [7]. However, PCC tracks only *full* (infinite depth) context sensitivity, which, as we show, incurs many false positives.

Inspired by real-world semantic attacks, this paper presents the design and implementation of a dynamic anomaly detector for Java called PECAN (Precise, Efficient, Context-sensitive ANomaly detection). PECAN uses training runs to compile the list of typical contexts and histories, and then monitors execution in production runs to find anomalies. We chose Java because of its popularity,

*This work is supported by NSF SHF-0910818, NSF CSR-0917191, NSF CCF-0811524, NSF CNS-0719966, NSF CNS-0746888, NSF CNS-0905602, and the "Collaborative Policies and Assured Information Sharing" MURI. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

PLAS '10 Toronto, Canada

but our approach applies to other memory-safe languages. To efficient track *depth-limited* context and history at the sites of security-sensitive calls, PECAN extends prior work on PCC [7].

We evaluate PECAN's performance and effectiveness on standard benchmarks, as well as on four real-world exploits that have been fixed, but are representative of several important categories of semantic security vulnerabilities. We show that context sensitivity is essential for detecting semantic attacks. Although the tension between false positives and negatives remains a challenge, we show that there exists a PECAN configuration (with the context depth of 3 and history length of 1) that detects all of our sample exploits while reporting few false positives. Further exploration and application-specific tuning would be necessary for real deployments. The average performance overhead of PECAN is 5%.

To the best of our knowledge, PECAN is the first system that provides context sensitivity of arbitrary depth and history in a practically efficient manner. We emphasize that there is a *qualitative* difference between a precise and efficient dynamic 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. In contrast to previous techniques for context-sensitive anomaly detection, which were evaluated only on artificial attacks, we evaluate PECAN on real-world exploits and use this evaluation to guide our design decisions.

2. REPRESENTATIVE EXPLOITS

To better understand which types of semantic vulnerabilities actually occur in managed code, we combed through hundreds of reports in the National Vulnerability Database,[1] tried to reproduce around twenty exploits, and successfully reproduced four in our experimental environment. This section surveys these four exploits, which illustrate common categories of semantic vulnerabilities and motivate our approach. Section 5 explains each exploit in more detail.

Three of the four exploits, SlashPath, LiveConnect, and OperaPolicy, are applets that take advantage of misconfigured security policies or bugs in the Java libraries. The fourth exploit, called XSLT, represents insufficient checking of untrusted code in which XML executable code is not sandboxed.

SlashPath. The first vulnerability is a bug in the Sun Java Virtual Machine 1.3.[2] It permits a malicious applet to circumvent the security manager by providing a class path with "/" instead of ".". We identify the general category that each exploit represents. This exploit is representative of the "mistakenly omitted security check" category. A conventional history-based anomaly detector would *not* 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.

LiveConnect. This vulnerability represents a common bug class, where untrusted code executes in the wrong security context (e.g., outside the normal Java sandbox). The particular bug we reproduce is in the Java 2 Runtime Environment 1.4, which permits a malicious applet to load unsafe classes and execute arbitrary code via the reflection API.[3] This exploit is representative of the "sensitive methods mistakenly exposed to untrusted code" category.

XSLT. This vulnerability is a bug in the Sun Java System Portal Server 7.0, which allows untrusted code contained in a malicious XSLT stylesheet to bypass the standard security checks.[4] 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 for 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 such as XSLT. While model-based techniques can handle recursive calls [14, 33], these techniques cannot detect any of the exploits we consider in this paper because, by design, they are only capable of detecting statically infeasible calls (e.g., those caused by code injection). We solve the problem by using *depth-limited* context sensitivity, which is still precise enough to detect the attack.

OperaPolicy. This vulnerability is a bug in the custom security manager of the Opera Web browser version 7.24, which permits untrusted applets to elevate their privileges.[5] This exploit is representative of the "misconfigured security policy" category.

Summary. While OperaPolicy and LiveConnect do not require context or history sensitivity, SlashPath and XSLT need context sensitivity to be detected, and SlashPath also needs history. Although these exact exploits are now fixed, they represent a general class of security errors. As we show below, prior approaches do not detect these attacks or are too inefficient for practical deployment, thus motivating our approach.

3. RELATED WORK

Much of the prior work on run-time detection of maliciously behaving applications focused on memory-corruption exploits. These attacks subvert *language semantics* for programs implemented in memory-unsafe languages such as C and C++. In contrast, this paper focuses on attacks that exploit *program semantics* and thus subvert programs implemented in memory-safe languages such as Java.

Table 1 summarizes the differences between PECAN and the related work on detecting security anomales. Column 2 (*Only Language Semantics*) shows that much of the related work only focuses on illegal program paths. While some prior work combines context and history (columns 3–4), the performance overheads are prohibitive (columns 5–6). No related work except the approaches by Inoue and Forrest [20, 21] has been evaluated on real-world semantic attacks.

Our approach is based on anomaly detection. The training phase constructs a model of correct application behavior and then during production executions the system detects deviations from this model. Anomalies have proven broadly useful for finding semantic

[1] `http://nvd.nist.gov/`

[2] `http://www.securityfocus.com/bid/8879/info`

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

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

[5] `http://www.securityfocus.com/archive/1/381634`

Attack Detection	Only Language Semantics	Program Semantics		Overhead		Real-World Exploits
		History	Context Sensitive	$\geq 100\%$	$< 10\%$	
Statically infeasible paths [1, 3, 8, 14, 33]	✓				✓*	
System call history [12, 17, 31]		✓		✓		
Memory-safe anomaly detection [20, 21]			✓†		✓	✓
Dynamic context sensitivity [11, 19, 28, 36]		✓	✓	✓		
PECAN		✓	✓		✓	✓

Table 1: Comparison of PECAN with previous anomaly detectors. *Some but not all static analysis-based approaches incur less than 10% run-time overhead. †Inoue and Forrest infer Java security policies, which are inherently context sensitive because the Java security model uses stack inspection [21].

errors [9, 16]. For example, DIDUCE establishes and records dynamic invariants such as variable values, variable inequalities, and branch directions [16]. This work shows that anomalous paths and values often reveal security violations in programs.

Memory-unsafe languages. Forrest et al. observed that system-call histories associated with security violations often appear anomalous with respect to correct executions. They developed a history-sensitive, *context-insensitive* anomaly detector that operates at the level of system calls [12, 17]. Sekar et al. next showed how to learn a finite state automaton (FSA), which provides a compact representation of system-call histories [31]. 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% [29, 31].

For some vulnerabilities analyzed here (e.g., SlashPath, Section 5.2), this simple method would miss the attack 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" [34], in which the malicious code executes the same sequence of calls as the correct application.

Much work combines static analysis with anomaly detection in order to accurately detect execution of infeasible control paths [1, 3, 8, 33]. 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, e.g., using the Dyck model [14], or Dyck with stack-walking [10]. A combination of static and dynamic analysis can find infeasible code paths [38]. 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 [24]. This entire line of research focuses on memory-unsafe languages. By contrast, semantic attacks on Java code do not involve execution of any paths that are not already present in the original code, so techniques for detecting execution of infeasible paths are not useful.

Dynamic context sensitivity. Feng et al. and others observed that context sensitivity is essential for improving precision of anomaly detection [11, 19, 28, 36, 37]. 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 history [11]. Performance overhead is 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 are prohibitively expensive in production systems. The overhead of system call interposition can exceed 100% [29, 31], and as we show in Figure 1, the overhead of walking the stack depends on the application and can be very high, too. Xu et al. introduce *waypoints*, which make the stack more visible to the monitor, but reduce efficiency [36]. Zhuang et al. efficiently find anomalous interprocedural paths, which include calling context, but their approach requires new hardware [37]. It appears very challenging to extend these techniques to object-oriented languages like Java, with many small methods and virtual methods, while achieving practical efficiency.

The Java security model is context sensitive: it inspects the execution stack to determine whether an operation is permissible [26]. Inoue and Forrest automatically infer Java security policies based on training [20, 21]. Abadi and Fournet propose access control based on *history* rather than on stack inspection [2]. PECAN is also sensitive to context and history, but instead of relying on security checks inserted by programmers, it infers security policies from observing dynamic program behavior.

Our approach. Our work differs substantially from prior work on context-sensitive anomaly detection. (1) We target an entirely different class of attacks than most of the prior work, focusing on semantic attacks that exploit existing code paths as opposed to memory-corruption attacks that introduce new code paths. (2) Our approach provides an efficient implementation of continuously available context of configurable depth that requires neither walking the stack, nor special hardware, and has a demonstrated overhead of 2-9% on realistic applications. (3) We evaluate our approach on real-world semantic attacks taken from the National Vulnerability Database, as opposed to artificially constructed memory-corruption exploits.

4. CONTEXT- AND HISTORY-SENSITIVE DETECTION OF SEMANTIC ATTACKS

We now describe the design and implementation of PECAN. Our approach uses the standard two-stage paradigm for dynamic anomaly detection. First, we train PECAN to learn normal behaviors. Then during deployment, PECAN monitors the code 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, which executes the program with comprehensive test inputs. The tools for systematic, exhaustive testing of legitimate code paths are now widely available [15, 22, 30].

PECAN's **deployment** phase observes program behavior and reports history/context combinations that it did not observe during training. PECAN may be 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. Developers can add the latter 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, i.e., which method invocations to monitor and how often, (ii) efficiency and scalability, (iii) false positive rate, and (iv) comprehensiveness of training. Frequent monitoring of many methods, with context and history sensitivity, yields more precise and timely detection of anomalies, but it imposes larger performance overheads, making it more likely that the system will generate a false positive due to valid behavior that was not observed during training. Finer granularity requires much more comprehensive training.

4.1 Security Calls

To limit the number of false positives and restrict the amount of information that the system must maintain, PECAN restricts its attention 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 for memory-unsafe languages typically tracks behavior at the level of system calls [17].

We consider two types of security calls: (1) calls to methods that perform security-sensitive actions such as I/O and system calls, and (2) calls to methods in Java's `SecurityManager` that check if the application is allowed to perform some security-sensitive action. Programs typically make the call of the second type prior to the first. Applications use the `SecurityManager` class 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 developer can forget a corner case and omit a needed check in a new policy. This bug can go undetected during testing, leaving 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. Depending on context, the same call may be malicious or benign. Context is a critical element of program behavior for programs written in modern, object-oriented languages, which tend to have small methods and use virtual method dispatch [7]. Obtaining context is expensive if it is needed more than rarely [11,28,36,37]. In prior work, Bond and McKinley introduce *probabilistic calling context* (PCC), which continuously computes a *PCC value* that represents the current calling context [7]. It uses the *PCC function* to compute the next PCC value at a call site, from the current value *V* and the current call site ID *cs*:

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

To detect anomalous context-sensitive behavior, clients check, at program locations of interest, whether a PCC value observed in deployment was also observed during training. For PECAN, the program locations of interest are security calls. We use PCC and extend it in two ways. First, we apply PCC to a real client (anomaly-based intrusion detection) and evaluate its performance and effectiveness. Second, we introduce a variant of PCC called *k-PCC* that supports depth-limited context sensitivity.

4.3 Depth-Limited Context Sensitivity

While the PCC function from prior work represents an 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 *k*th 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), and k is context depth. For example, if $k = 3$,

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

This function shifts the current k-PCC value 11 bits to the left and then adds the call site value. Bits affected by call sites lower on the stack are pushed off the end of the value, thus only the top k call sites affect the PCC value. Note that the function is only suitable for $k \leq bits$.

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. At security calls only, PECAN also checks whether the k-PCC value is anomalous. The following example shows the instrumentation added by the compiler to a method that, prior to instrumentation, makes calls at the three labeled call sites.

```
            void applicationMethod() {
              int tmp = V; // save current k-PCC

              ...

              V = f(tmp, cs_1); // compute k-PCC
/* cs_1: */   foo();

              ...

              V = f(tmp, cs_2); // compute k-PCC
              check(V);         // check k-PCC
/* cs_2: */   SecurityManager.checkPermission(...);

              ...

              V = f(tmp, cs_3); // compute k-PCC
              check(V);         // check k-PCC
/* cs_3: */   readFile(...);

              ...
            }
```

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

```
void check(int V) {
  if (!table.contains(V)) {
    table.add(V);
    if (deployed) {
      context = walkStack();
      reportAnomaly(context);
} } }
```

If the value is anomalous, then the resulting context is guaranteed to be anomalous. In training mode, PECAN simply adds the new k-PCC value to the table. In deployed mode, it reports the new context, which it obtains (after detection) by walking the stack, as anomalous.

A disadvantage of the k-PCC function is that only the top call site affects all bits in the k-PCC value. For example, with $k = 3$, 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 increase because another call site only needs to share the third call site's 10 lowest bits for a conflict to occur. This pitfall is mitigated by the fact 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-PCC is sufficient for accurately recognizing anomalies associated with real attacks. Future work could evaluate k-PCC's conflict rate for all program calling contexts, as in the original PCC work [7].

4.4 History Sensitivity

Program *history* is also an essential ingredient of accurate anomaly detection [12, 17, 31]. 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. Remember 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 and 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. Each thread uses a variable 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 security call and combines it with the current k-PCC value (using a variant of the original PCC function, $5 \times V_{prev} + V_{current}$), which we refer to as history of level 1.

In the rest of the paper, we refer to k-limited probabilistic context with history as the *k-PCH value*.

4.5 Component Granularity

Modern software is usually assembled from independently developed components. PECAN training and monitoring may be applied to a subset 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. Similarly, an implementor 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. This decision makes sense especially for general-purpose libraries such as the Java API. Otherwise, each call to the library by a new application would appear anomalous.

When PECAN is applied to libraries only, it resets history before each application → library call. Note that PECAN does not instrument these calls because the call sites are inside the application, not the libraries. This approach helps avoid mimicry attacks. For example, consider a library method that omits a necessary security check. A malicious applet may try to avoid detection by calling this check itself, prior to exploiting the vulnerable method.

4.6 Implementation

We implemented PECAN in Jikes RVM 2.9.2, a research Java Virtual Machine [4].[6] Jikes RVM is a research tool, but its performance compares well with commercial VMs.[7] Our performance measurements are thus relative to an excellent baseline. We have made our implementation of PECAN publicly available on the Jikes RVM Research Archive.[8]

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 (in training) or checks (in deployment) the k-PCH value at security calls.

5. EVALUATION

This section analyzes PECAN's performance and ability to detect attacks. First, we measure PECAN's performance and compare the overhead of using k-PCH values to walking the stack. We then evaluate PECAN's ability to detect real-world semantic exploits. Finally, we perform leave-one-out cross-validation on benign programs to evaluate PECAN's false positive rate.

5.1 Performance

PECAN adds overhead to applications because it inserts instrumentation to track the k-PCH value and check it at security calls. This section shows that using k-PCH for context sensitivity has consistently low overhead across all benchmarks, whereas walking the stack at each security call sometimes adds very high overhead.

Figure 1 shows the normalized execution time of our approach for the DaCapo benchmarks and fixed-workload version of SPECjbb2000 called pseudojbb [6, 32]. Each bar is the median of three trials. We use an execution methodology called *replay compilation* to reduce nondeterminism due to timer-based sampling [13,18]. 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.[9] *Pecan* is the overhead of continuously maintaining k-PCH values and checking them at security calls. *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 experiments use $k = 3$, but the overhead is the same for larger values of k up to $k = \infty$.

The *Walk stack* configurations show the overhead of walking the stack when context is needed, rather than keeping track of context

[6] `http://www.jikesrvm.org`

[7] `http://jikesrvm.anu.edu.au/performance/2009-07/`

[8] `http://www.jikesrvm.org/Research+Archive`

[9] Overheads are negative (shown as zero overhead in Figure 1) in a few cases because of architectural effects, e.g., instrumentation perturbs code layout, which can affect caching performance for better or worse.

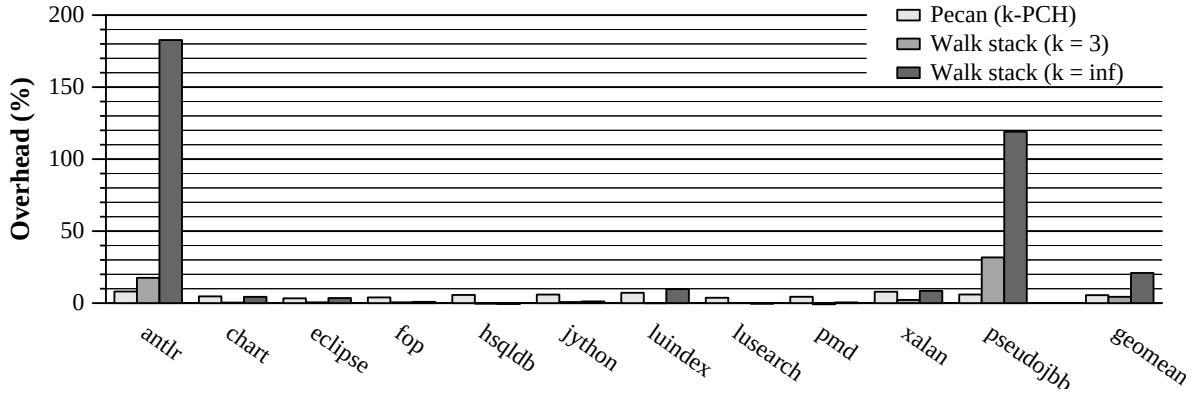

Figure 1: Application execution time overhead of maintaining the k-PCH value and querying it at calls to `SecurityException` methods, compared with walking the stack.

continuously with k-PCH. These configurations walk the top three call sites and all call sites on the stack, respectively. We note that walking the stack is semantically equivalent to k-PCH, but not performance equivalent! Stack-walking is deterministic, whereas k-PCH has a small probability of a numeric collision. In practice, the two techniques should report the same vulnerabilities. The stack-walking configurations have low overhead for most programs, but for two programs (antlr and pseudojbb) they add very high overhead. Walking the entire stack is very expensive for these programs, but, admittedly, full context sensitivity produces too many false positives in practice, as we show in Section 5.3. 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 scales well with these factors.

5.2 Detecting Real Attacks

This section evaluates PECAN's ability to detect semantic attacks. 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, SlashPath loads a class in a malicious way. To train PECAN, we load a class in a benign way. We then execute the exploits in the trained system and observe whether PECAN reports anomalous behavior. Our experiments explore several combinations of granularity, context, and history sensitivity. However, they are not comprehensive, and a production system would likely need application-specific training to appropriately configure the parameters of context sensitivity and history 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 security calls. 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 these classes of vulnerabilities, we restrict instrumentation to the libraries because exploits 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, we perform monitoring inside that application. Developers would need to make similar decisions.

Finally, we experiment with different levels of *context sensitivity*. By default, PECAN uses context sensitivity of depth 3. 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 also compare to full (infinite) context sensitivity. We experiment with two levels of history sensitivity: 0 (none) and 1 (includes the previous `SecurityManager` call). Deeper context sensitivity leads to a richer set of behaviors and thus more false positives, but is required for detection of some attacks (e.g., SlashPath). Additional history sensitivity beyond 1 is likely to result in many false positives, but this paper does not explore this direction because history level of 1 is sufficient to detect all of our sample attacks. We have, however, verified that the SlashPath vulnerability cannot be detected with history alone: even with infinite history, detection requires context sensitivity.

The rest of this section refers to configurations of PECAN using 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, is $C_3 H_1$, which detects all of our sample attacks while producing few false positives.

We now present the details of the four vulnerabilities and evaluate PECAN's ability to detect them. Two exploits, SlashPath and XSLT, need context sensitivity to be detected; SlashPath also needs history. The other two exploits do not need context or history sensitivity.

SlashPath. The SlashPath vulnerability is representative of a common mistake in the implementation of security managers, where the enforcement mechanism forgets to perform a mandatory security check. The vulnerability exploits the fact that Sun JVM 1.3 does not correctly check whether it is okay to load a class or not if that class's package name is delimited with slashes (e.g., `sun/applet/AppletClassLoader`) instead of dots (e.g., `sun.applet.AppletClassLoader`).[10]

The vulnerability occurs because of a mismatch between two pieces of code. The code that potentially performs a security check assumes that class names are delimited only with dots, while the code that actually loads the class allows names to be delimited with dots or slashes. We found that this vulnerability is also present in the system class loader in Jikes RVM. The code in Figure 2 shows a simplified version of the vulnerable class loader. Note that the call to `checkPackageAccess()` does not occur if the class name

[10] `http://www.securityfocus.com/bid/8879/info`

k	No history Config	Anoms.	(All)	1-level history Config	Anoms.	(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)**
∞	$C_\infty H_0$	0	(194)	$C_\infty H_1$	2	(222)

Table 2: Intrusion detection results for SlashPath. Detection requires both context sensitivity and history.

```
Class loadClass(String name,
                boolean resolve) {
  int lastDot = name.lastIndexOf('.');
  if (lastDot != -1)
    String pkg = name.substring(0, lastDot);
    SecurityManager.checkPackageAccess(pkg);
  }
  return super.loadClass(name, resolve);
}
```

Figure 2: Vulnerable `loadClass()` method from Jikes RVM's system class loader.

is delimited with slashes instead of dots.

Our exploit code 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. Our training code models benign behavior by loading a class whose package name is delimited with dots and also a class without a package name (no dots or slashes). Table 2 shows results for executing the SlashPath attack in a system monitored by PECAN. Each row is a configuration with varying levels of context and history sensitivity. The cells show the number of anomalous behaviors associated with the attack and, in parentheses, the total number of behaviors observed during training.

Table 2 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 the attack behavior results in anomalous k-PCH values at multiple security calls. In general, Tables 2–5 can be interpreted as follows. If *Anoms.* is 0, the PECAN configuration cannot detect the attack; otherwise, the PECAN configuration detects the attack. Detection is contingent on the training set and could potentially fail if the training set were such that the malicious call(s) 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 (not shown). This attack would defeat naïve history-based detection, but PECAN detects it because PECAN resets history on each applet → library call (Section 4.5).

XSLT. This vulnerability represents a common class of bugs, where untrusted code mistakenly executes in the wrong security context (e.g., outside the normal Java sandbox). EXtensible Stylesheet Language (XSL) documents specify XSL Transformations (XSLT) to perform on XML documents. XSLT processing may use XSL documents to generate Java code that the host JVM then executes. Typically this code is sandboxed, but in a specific context, XSLT may be "tricked" into executing arbitrary code without security checks. The following is an excerpt from the vulnerability report:[11]

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

k	Config	Anoms.	(All)
0	C_0H_1	0	(20)
1	C_1H_1	0	(40)
3	$\boldsymbol{C_3H_1}$	**2**	**(42)**
∞	$C_\infty H_1$	222	(1573)

Table 3: Intrusion detection results for XSLT. Detection requires context sensitivity. `SecurityManager` history is not relevant since PECAN profiles only the application, which does not make `SecurityManager` calls.

k	No history Config	Anoms.	(All)	1-level history Config	Anoms.	(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)**
∞	$C_\infty H_0$	6	(6)	$C_\infty H_1$	6	(6)

Table 4: Intrusion detection results for LiveConnect. Detection does not require context or history sensitivity.

> 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.

To reproduce this exploit in our Jikes RVM-based experimental setup, we use the Xalan XML parsing library, which comes with Sun JVM. Based on more details from the report quoted above, 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 embedded code can then perform arbitrary illegal actions, such as accessing local data on the machine.

Table 3 shows that context sensitivity is essential for detecting the attack. We do not show various levels of history 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. The results do not depend on 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, which demonstrates the need for context sensitivity.

LiveConnect. This vulnerability involves gaining access to security-sensitive methods that are not normally available to untrusted code. A design error in a web browser feature called LiveConnect mistakenly allows Java and JavaScript code to communicate with one another on a web page, i.e., a Java applet accesses JavaScript objects illegally, and JavaScript code illegally accesses Java runtime libraries.[12] A malicious applet can then use a JavaScript object to determine the user's browser and obtain a reference to a class `sun.plugin.liveconnect.SecureInvocation`. The applet can use this class to disable the current security manager.

To reproduce LiveConnect, we use exploit code provided by another security researcher [39]. The vulnerability occurs in Sun's browser plugin, so PECAN instruments only the plugin, not the

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

k	No history Config	Anoms.	(All)	1-level history Config	Anoms.	(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)**
∞	$C_\infty H_0$	6	(6)	$C_\infty H_1$	6	(6)

Table 5: Intrusion detection results for OperaPolicy. Detection does not require context or history sensitivity.

other Java libraries. We could configure PECAN to instrument all libraries, although this would result in additional anomalous behaviors.

Table 4 shows the anomalies reported by PECAN for different amounts of context sensitivity and history. PECAN detects anomalous behavior regardless of the amounts of context or 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. A more thorough training set or a potential mimicry attack might increase the precision needed to detect this exploit.

OperaPolicy. This vulnerability is representative of logic errors and misconfiguration in custom security policies. The Opera 7.54 web browser specifies its own Java security policy for applets. The default policy `Opera.policy` allows unprivileged applets access to internal packages:[13]

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

Our test exploit uses the `getBootstrapClassPath()` method of the `sun.misc.Launcher` class to obtain private 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 5 shows that PECAN detects the attack at all levels of context and history sensitivity. Similar to LiveConnect, this exploit calls a method that should not be accessible. This call looks anomalous regardless of the amount of context or history sensitivity, so any PECAN configuration detects it.

5.3 Evaluating False Positives with Benign 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 *benign* 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 6 shows the number of false positives (anomalous k-PCH values) using leave-one-out cross-validation for 12 sample applets obtained quickly via Google search. The methodology of training on one set of applets and deploying on a different applet is reasonable because PECAN profiles only the libraries called by the applets, not the applets themselves (Section 4.5). The number in parentheses is the total number of distinct k-PCH values. For higher

[13] `http://www.securityfocus.com/archive/1/381634`

	False positives (total distinct behaviors) ArcTest		AtomViewer		CardTest		DiffEq	
C_0H_0	0	(31)	0	(31)	0	(31)	0	(33)
C_0H_1	0	(53)	0	(53)	0	(52)	0	(60)
C_1H_0	0	(45)	0	(45)	0	(45)	0	(56)
C_1H_1	0	(75)	0	(74)	0	(71)	0	(100)
C_3H_0	0	(96)	0	(93)	0	(93)	0	(125)
$\boldsymbol{C_3H_1}$	**1**	**(127)**	**9**	**(125)**	**0**	**(111)**	**1**	**(184)**
$C_\infty H_0$	32	(196)	113	(318)	0	(146)	125	(287)
$C_\infty H_1$	40	(221)	61	(251)	10	(211)	131	(314)
	DitherTest		DrawTest		Euler		Gas	
C_0H_0	0	(31)	0	(31)	0	(33)	0	(31)
C_0H_1	0	(54)	0	(52)	0	(60)	0	(54)
C_1H_0	0	(45)	0	(45)	0	(56)	0	(45)
C_1H_1	0	(75)	0	(71)	0	(100)	0	(75)
C_3H_0	4	(100)	0	(93)	2	(127)	0	(99)
$\boldsymbol{C_3H_1}$	**7**	**(133)**	**0**	**(123)**	**6**	**(189)**	**1**	**(130)**
$C_\infty H_0$	77	(218)	10	(182)	46	(281)	14	(194)
$C_\infty H_1$	94	(253)	5	(207)	101	(328)	28	(219)
	Matrix		Puzzle		ReflFrame		StringWave	
C_0H_0	0	(33)	0	(31)	0	(31)	0	(31)
C_0H_1	0	(54)	0	(52)	0	(43)	0	(42)
C_1H_0	0	(56)	0	(45)	0	(44)	2	(47)
C_1H_1	0	(100)	0	(74)	0	(62)	0	(55)
C_3H_0	0	(121)	0	(93)	4	(89)	0	(65)
$\boldsymbol{C_3H_1}$	**0**	**(173)**	**0**	**(123)**	**6**	**(114)**	**0**	**(84)**
$C_\infty H_0$	56	(250)	10	(156)	74	(178)	9	(240)
$C_\infty H_1$	73	(285)	12	(209)	93	(170)	0	(124)

Table 6: Leave-one-out cross-validation for 12 non-vulnerable applets. Even though there is relatively little training compared with expected industrial efforts, false positive rates are low for the C_3H_1 PECAN configuration.

levels of context and history sensitivity, there are many more false positives. 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 test programs that we use here, further reducing the number of false positives (Section 4).

Table 7 shows false positives using leave-one-out cross-validation when XSLT is executed on eight XSL inputs found by a quick Google search. History sensitivity is omitted because XSLT does not call `SecurityManager` methods directly, so results are not affected by history sensitivity (Section 5.2). 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.

Additional evaluation is needed, but even these results indicate that developers may need to configure PECAN for a given application by setting the right levels of context sensitivity and history sensitivity. Because different applications have different security requirements and implementations, requiring developers to configure an anomaly detector in such a manner does not seem unreasonable.

	False positives (total distinct behaviors)							
	ui		resume		testcase		testcase2	
C_0H_1	0	(5)	0	(5)	0	(6)	0	(5)
C_1H_1	0	(21)	0	(21)	0	(23)	2	(22)
$\boldsymbol{C_3H_1}$	**0**	**(22)**	**0**	**(22)**	**1**	**(25)**	**2**	**(23)**
$C_\infty H_1$	15	(69)	3	(44)	63	(141)	1409	(1476)
	testcase3		testcase4		testcase5		testcase6	
C_0H_1	0	(5)	0	(5)	0	(6)	0	(5)
C_1H_1	0	(21)	1	(22)	0	(23)	0	(21)
$\boldsymbol{C_3H_1}$	**0**	**(22)**	**1**	**(22)**	**0**	**(23)**	**0**	**(21)**
$C_\infty H_1$	6	(54)	2	(43)	49	(82)	1	(42)

Table 7: Leave-one-out cross-validation for eight non-vulnerable XSLT inputs. The C_3H_1 PECAN configuration generates few false positives. `SecurityManager` history is not relevant since PECAN profiles only the application, which does not make `SecurityManager` calls.

6. SUMMARY

Semantic attacks on programs implemented in memory-safe languages are hard to detect because they violate informal coding rules rather than the semantics of the programming language. Anomaly detection can help, but many existing methods suffer from poor precision and performance. PECAN is a novel anomaly detection system for Java with probabilistic, depth-limited context sensitivity, history sensitivity, and low performance 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 the number of false positives low. While the tension between false negatives and false positives remains challenging, PECAN's demonstrated ability to detect attacks precisely, accurately, and efficiently on real-world programs makes a compelling case for its use in deployed systems. These initial results are promising, but further evaluation is needed. For production use, application-specific configuration would likely be required, but we believe developers would be willing to invest this effort in order to increase the security of critical applications.

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 RVM 2.9.2; Chris Ryder for PCC bug fixes; Sam Guyer for helpful discussions; and Bert Maher, Wei Le, and the anonymous reviewers for valuable feedback on the text.

7. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] M. Abadi and C. Fournet. Access Control based on Execution History. In *Network and Distributed Systems Security Symposium*, pages 107–121, 2003.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[4] 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.

[5] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan. CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks. *ACM Trans. Inf. Syst. Secur.*, 13(2):1–39, 2010.

[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] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software Guards for System Address Spaces. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, 2006.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[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*, pages 62–75, 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] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[18] 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.

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

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

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

[22] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun. A Concolic Whitebox Fuzzer for Java. In *NASA Formal Methods Workshop*, pages 121–125, 2009.

[23] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[24] 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.

[25] 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*, pages 25–33, 2006.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, 1999.

[27] B. Livshits and M. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, pages 271–286, 2005.

[28] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, 2007.

[29] N. Provos. Improving Host Security with System Call Policies. In *USENIX Security Symposium*, pages 257–271, 2003.

[30] V. Roubtsov. EMMA: A free Java code coverage tool.

[31] 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.

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

[33] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[34] 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.

[35] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[36] 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.

[37] 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.

[38] 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.

[39] D. D. Zovi, 2008. Personal communication.