Aspect-Oriented Runtime Monitor Certification*

Kevin W. Hamlen, Micah M. Jones, and Meera Sridhar

University of Texas at Dallas
{hamlen,micah.jones1,meera.sridhar}@utdallas.edu

Abstract. In-lining runtime monitors into untrusted binary programs via aspect-weaving is an increasingly popular technique for efficiently and flexibly securing untrusted mobile code. However, the complexity of the monitor implementation and in-lining process in these frameworks can lead to vulnerabilities and low assurance for code-consumers. This paper presents a machine-verification technique for aspect-oriented in-lined reference monitors based on abstract interpretation and model-checking. Rather than relying upon trusted advice, the system verifies semantic properties expressed in a purely declarative policy specification language. Experiments on a variety of real-world policies and Java applications demonstrate that the approach is practical and effective.

Keywords: Abstract interpretation, in-lined reference monitors, model-checking, security.

1 Introduction

Software security systems that employ purely static analyses to detect and reject malicious code are limited to enforcing decidable security properties. Unfortunately, most useful program properties, such as safety and liveness properties, are not generally decidable and can therefore only be approximated by purely static analyses. For example, signature-based antivirus products accept or reject programs based on their syntax rather than their runtime behavior, and therefore suffer from dangerous false negatives, inconvenient false positives, or both (cf., [16]). This has shifted software security research increasingly toward more powerful dynamic analyses, but these dynamic systems are often far more difficult to formally verify than provably sound static analyses.

An increasingly important family of such dynamic analyses are those that modify untrusted binary code prior to its execution. *In-lined reference monitors* (IRMs) instrument untrusted code with new operations that perform runtime security checks before potentially dangerous operations [27]. The approach is motivated by improved efficiency (since IRMs require fewer context switches than external monitors), deployment flexibility (since in-lining avoids modifying the VM or OS), and precision (since IRMs can monitor internal program operations

* Supported by AFOSR award FA9550-08-1-0044 and NSF award NSF-1065216. Any views expressed do not necessarily reflect those of the NSF or AFOSR.

C. Flanagan and B. König (Eds.): TACAS 2012, LNCS 7214, pp. 126–140, 2012.

not readily visible to an external monitor). Most modern IRM systems are implemented using some form of *aspect-oriented programming* (AOP) [32,28,7,8,14]. Such IRMs are implemented as *pointcut-advice* pairs: pointcuts identify security-relevant program operations and *advice* prescribes local code transformations sufficient to guard such operations. This suffices to enforce safety policies [27,18] and some liveness policies [26].

To provide exceptionally high assurance guarantees, recent work has sought to reduce the (potentially large) trusted computing bases (TCBs) of IRM frameworks by separately machine-verifying the *self-monitoring* code they produce [17,1,30,31]. For example, the S^{3}MS project uses a contract-based verifier [1] to avoid trusting the the much larger in-liner (over 900K lines of Java code if one includes the underlying AspectJ system [22]) that generates the IRMs.

However, TCB-minimization of large IRM systems has been frustrated by the inevitable inclusion of significant, trusted code within the AOP-style policy specifications themselves. Verifiers for these systems can prove that the IRM system has correctly in-lined the policy-prescribed advice code but not that this advice actually enforces the desired policy. Past case studies have demonstrated that such advice is extremely difficult to write correctly, especially when the policy is intended to apply to large classes of untrusted programs rather than individual applications [21]. Moreover, in many domains, such as web ad security, policy specifications change rapidly as new attacks and vulnerabilities are discovered (cf., [23,29,30]). Thus, the considerable effort that might be devoted to formally verifying one particular aspect implementation quickly becomes obsolete when the aspect is revised in response to a new threat.

To address this open challenge, we present Chekov✓: the first IRM-certification framework that verifies full, AOP-style IRMs against purely declarative policy specifications without trusting the code that implements the IRM. Chekov✓ uses light-weight model-checking and abstract interpretation to verify untrusted (but verifiably type-safe) Java bytecode binaries against trusted policy specifications that lack advice. Policies declaratively specify how security-relevant program operations affect an abstract system security state. Unlike contracts, which denote code transformations, policies in our system therefore denote pure code properties. Such properties can be enforced by untrusted aspects that dynamically detect impending policy violations and take corrective action. The woven aspects are verified (along with the rest of the self-monitoring code) against the trusted policy specification prior to its execution.

Chekov✓ was inspired by our prior work on model-checking IRMs [30,29,9], but includes numerous substantial theoretic and pragmatic leaps beyond those earlier works. These include:

- support for a full-scale Java IRM framework (the SPoX IRM system [14,20]) that includes *stateful* (history-based) policies, event detection by pointcut-matching, and IRM implementations that combine (untrusted) before- and after-advice insertions;
- a novel approach to dynamic pointcut verification using Constraint Logic Programming (CLP) [19]; and

– proofs of correctness based on Cousot's abstract interpretation framework [5] that link the denotational semantics of SPoX policies to the operational semantics of the abstract interpreter.

Section 2 begins with related work, followed by an overview of the SPoX policy language and the rewriter in §3. Section 4 presents a high-level description of the verification algorithm. (A more detailed treatment with proofs is available in the companion technical report [15].) Section 5 presents in-depth case-studies of several security policy classes that we enforced on numerous real-world applications, and discusses challenges faced in implementing and verifying these policies. Finally, §6 concludes with recommendations for future work.

2 Related Work

IRMs were first formalized in the PoET/PSLang/SASI systems [11,27,10], which implement IRMs for Java bytecode and GNU assembly code. IRM systems have subsequently been developed for many architectures (cf., [24,4]). Most of these express security policies in an AOP or AOP-like language with pointcut expressions for identifying security-relevant binary program operations, and code fragments (advice) that specify actions for detecting and prohibiting impending policy violations. A hallmark of these systems is their ability to enforce history-based, stateful policies that permit or prohibit each event based on the history of past events exhibited by the program. This is typically achieved by expressing the security policy as an automaton [27,25] whose state is *reified* into the untrusted program as a protected global variable. The IRM tracks the current security state at runtime by consulting and updating the variable as events occur.

Machine-certification of IRMs was first proposed as type-checking [33]—an idea that was later extended and implemented in the Mobile system [17]. Mobile transforms Microsoft .NET bytecode binaries into safe binaries with typing annotations in an effect-based type system. The annotations constitute a proof of safety that a type-checker can separately verify to prove that the transformed code is safe. Type-based IRM certification is efficient and elegant but does not currently support dynamic pointcut matching. It has therefore not been applied to AOP-style IRMs to our knowledge.

ConSpec [2,1] adopts a security-by-contract approach to AOP IRM certification. Its certifier performs a static analysis that verifies that contract-specified guard code appears at each security-relevant code point. While certification-via-contract facilitates natural expression of policies as AOP programs, it has the disadvantage of including the potentially complex advice code in the TCB.

Our prior work [30] is the first to adopt a model-checking approach to verify such IRMs without trusted guard code. The prototype IRM certifier in [30] supports reified security state, but it does not support dynamic pointcuts and its support for advice is limited to a very constrained form of before-advice. It therefore does not support real-world IRM systems or their policies, which regularly employ dynamic pointcuts and after-advice.

In contrast, the verifier presented in this work targets SPoX [14,20], a fully featured, purely declarative AOP IRM system for Java bytecode. SPoX policies are advice-free; any advice that implements the IRM remains untrusted and must therefore undergo verification. Policy specifications consist of pointcuts and declarative specifications of how pointcut-matching events affect the security state. The abstract security state-changes specified by SPoX policies are significantly higher-level and simpler than the arbitrary advice code admitted by non-declarative AOP languages. Thus, SPoX policies are a significant TCB reduction over AOP contracts that implement them.

3 Policy Language and Rewriter

As an example of how software security policies are specified in SPoX, Fig. 1 specifies a policy that permits applications to send at most 10 email messages per run. The policy says that `Mail.send` API calls increment security state `s` up to 10, but an 11th call triggers a policy violation. Such a policy could be useful for preventing spam.

```
1 (state name="s")
2 (forall "i" from 0 to 9
3   (edge name="count"
4     (call "Mail.send")
5     (nodes "s" i,i + 1)))
6 (edge name="10emails"
7   (call "Mail.send")
8   (nodes "s" 10,#))
```

Fig. 1. A policy permitting at most 10 email-send events

More formally, SPoX policies denote *security automata* [3]—finite- or infinite-state machines that accept languages of permissible *event sequences*. Sets of edges in the security automaton are described by `edge` structures, each of which consists of a pointcut expression (Lines 4 and 7) and at least one `nodes` declaration (Lines 5 and 8). The pointcut expression defines a common label for the edges in the set, while each `nodes` declaration imposes a transition pre-condition and post-condition for a particular state variable. The pre-condition constrains the set of source states to which the edge applies, and the post-condition describes how the state changes when an event satisfying the pointcut expression and all pre-conditions is exhibited. Events that satisfy none of the outgoing edge labels of the current security state leave the security state unchanged. Policy-violations are explicitly identified with the reserved post-condition "`#`".

SPoX derives its pointcut language from AspectJ, allowing policy writers to develop policies that regard static and dynamic method calls and their arguments, object pointers, and lexical contexts, among other properties. In order to remain fully declarative, SPoX omits explicit, imperative advice. Instead, policies declaratively specify how security-relevant events change the current security

automaton state. Rewriters then synthesize their own advice in order to enforce the prescribed policy. The use of declarative state-transitions instead of imperative advice facilitates formal, automated reasoning about policies without the need to reason about arbitrary code [21].

The SPoX rewriter takes as input a Java binary archive (JAR) and a SPoX policy, and outputs a new application in-lined with an IRM that enforces the policy. The high-level in-lining approach is essentially the same as the other IRM systems discussed in §2. Each method body is scanned for potentially security-relevant instructions, and sequences of guard instructions are in-lined around those to detect and preclude policy-violations at runtime.

In-lined guard code must track event histories if the policy is stateful. To do so, the rewriter reifies abstract security state variables into the untrusted code as static, private class fields. The guard code then tracks the abstract security state by consulting and updating the corresponding fields. The runtime guards must also evaluate any statically undecidable portions of pointcut expressions to decide whether impending events are actually security-relevant. For example, to evaluate pointcut `(argval 1 (intgt 2))`, it might dynamically test whether $x > 2$, where x is the impending operation's first argument.

	$(A{=}S \wedge A{=}T)$ **0.1**
`1 if (Policy.s >= 0 && Policy.s <= 9)`	
	$(A{=}S \wedge A{=}T \wedge S{\geq}0 \wedge S{\leq}9)$ **1.1**
`2  Policy.temp_s := Policy.s+1;`	
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1)$ **2.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9))$ **2.2**
`3 if (Policy.s == 10)`	
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1 \wedge S{=}10)$ **3.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9) \wedge S{=}10)$ **3.2**
`4  call System.exit(1);`	
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1 \wedge S{\neq}10)$ **4.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9) \wedge S{\neq}10)$ **4.2**
`5 Policy.s := Policy.temp_s;`	
	$(A{=}S' \wedge A{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T)$ **5.1**
	$(A{=}S' \wedge A{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T)$ **5.2**
`6 call Mail.send();`	
$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}I \wedge I{\geq}0 \wedge I{\leq}9 \wedge A{=}I{+}1)$	**6.1**
$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}10 \wedge A{=}\#)$	**6.2**
$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge (A'{<}0 \vee A'{>}9) \wedge A'{\neq}10 \wedge A{=}A')$	**6.3**
$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}I \wedge I{\geq}0 \wedge I{\leq}9 \wedge A{=}I{+}1)$	**6.4**
$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}10 \wedge A{=}\#)$	**6.5**
$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge (A'{<}0 \vee A'{>}9) \wedge A'{\neq}10 \wedge A{=}A')$	**6.6**

Fig. 2. An abstract interpretation of instrumented pseudocode

The left column of Fig. 2 gives pseudocode for an IRM that enforces the policy in Fig. 1. For each call to method `Mail.send`, the IRM tests two possible preconditions: (1) $0 \leq \mathtt{s} \leq 9$ and (2) $\mathtt{s} = 10$. In the first case, it increments `s`; in the second, it aborts the process. Observe that in this example security state `s` has been reified as two separate fields of class `Policy` (`s` and `temp_s`) in order to prevent join point conflicts. This reflects a reality that any given policy has a variety of IRM implementations, many of which contain unexpected quirks that address non-obvious, low-level enforcement details.

4 Verifier

Our verifier takes as input (1) a SPoX security policy, (2) an instrumented, type-safe Java bytecode program, and (3) some optional, untrusted hints from the rewriter (detailed shortly). It either accepts the program as provably policy-satisfying or rejects it as potentially policy-violating. Type-safety is checked by the JVM, allowing our verifier to safely assume that all bytecode operations obey standard Java memory-safety and well-formedness. This keeps tractable the task of reliably identifying security relevant operations and field accesses.

The main verifier engine uses abstract interpretation to non-deterministically explore all control-flow paths of the untrusted code, inferring an abstract program state at each code point. A model-checker then proves that each abstract state is policy-adherent, thereby verifying that no execution of the code enters a policy-violating program state. Policy-violations are modeled as *stuck states* in the operational semantics of the verifier—that is, abstract interpretation cannot continue when the current abstract state fails the model-checking step. This results in conservative rejection of the untrusted code. The verifier is expressed as a bisimulation of the program and the security automaton. Abstract states in the analysis conservatively approximate not only the possible contents of memory (e.g., stack and heap contents) but also the possible security states of the system at each code point.

The heart of the verification algorithm involves inferring and verifying relationships between the abstract program state and the abstract security state. When policies are stateful, this involves verifying relationships between the abstract security state and the corresponding reified security state(s). These relationships are complicated by the fact that although the reified state often precisely encodes the actual security state, there are also extended periods during which the reified and abstract security states are not synchronized at runtime. For example, guard code may preemptively update the reified state to reflect a future security state that will only be reached after subsequent security-relevant events, or it may retroactively update the reified state only after numerous operations that change the security state have occurred. These two scenarios correspond to the insertion of before- and after-advice in AOP IRM implementations. The verification algorithm must be powerful enough to automatically track these relationships and verify that guard code implemented by the IRM suffices to prevent policy violations.

To aid the verifier in this task, we modified the SPoX rewriter to export two forms of untrusted hints along with the rewritten code: (1) a relation $\sim$ that associates policy-specified security state variables s with their reifications r, and (2) marks that identify code regions where related abstract and reified states might not be *synchronized* according to the following definition:

Definition 1 (Synchronization Point). *A* synchronization point *(SYNC) is an abstract program state with constraints ζ such that proposition $\zeta \wedge \left(\bigvee_{r \sim s} (r \neq s) \right)$ is unsatisfiable.*

Chekov✓ uses these hints (without trusting them) to guide the verification process and to avoid state-space explosions that might lead to conservative rejection of safe code. In particular, it verifies that all non-marked instructions are *SYNC*-preserving, and each outgoing control-flow from a marked region is *SYNC*-restoring. This modularizes the verification task by allowing separate verification of marked regions, and controls state-space explosions by reducing the abstract state to *SYNC* throughout the majority of binary code which is not security-relevant. Providing incorrect hints causes Chekov✓ to reject (e.g., when it discovers that an unmarked code point is potentially security-relevant) or converge more slowly (e.g., when security-irrelevant regions are marked and therefore undergo unnecessary extra analysis), but it never leads to unsound certification of unsafe code.

A Verification Example. Figure 2 demonstrates a verification example step-by-step. The pseudocode constitutes a marked region in the target program, and the verifier requires that the abstract interpreter is in the *SYNC* state immediately before and after. At each code point, the verifier infers an abstract program state that includes one or more conjunctions of constraints on the abstract and reified security state variables. These constraints track the relationships between the reified and abstract security state. Here, variable A represents the abstract state variable `s` from the policy in Fig. 1. Reifications `Policy.s` and `Policy.temp_s` are written as S and T, respectively, with $S \sim A$ and $T \sim A$. Thus, state *SYNC* is given by constraint expression $(A = S \wedge A = T)$ in this example.

The analysis begins in the *SYNC* state, as shown in constraint list 0.1. Line 1 is a conditional, and thus spawns two new constraint lists, one for each branch. The positive branch (1.1) incorporates the conditional expression $(S \geq 0 \wedge S \leq 9)$ in Line 2, whereas the negative branch (2.2) incorporates the negation of the same conditional. The assignment in Line 2 is modeled by alpha-converting T to T' and conjoining constraint $S = T + 1$; this yields constraint list 2.1.[1]

Unsatisfiable constraint lists are opportunistically pruned to reduce the state space. For example, list 3.1 shows the result of applying the conditional of Line 3 to 2.1. Conditionals 1 and 3 are mutually exclusive, resulting in contradictory expressions $S \leq 9$ and $S = 10$; therefore, 3.1 is dropped. Similarly, 3.2 is dropped because no control-flows exit Line 4.

To interpret a security-relevant event such as the one in Line 6, the verifier simulates the traversal of all edges in the security automaton. In typical policies, any given instruction fails to match a majority of the pointcut labels in the policy, so most are immediately dropped. The remaining edges are simulated by conjoining each edge's pre-conditions to the current constraint list and modeling the edge's post-condition as a direct assignment to A. For example, edge `count` in Fig. 1 imposes pre-condition $(0 \leq I \leq 9) \wedge (A = I)$, and its post-condition can be modeled as assignment $A := I + 1$. Applying these to list 5.1 yields list 6.1. Likewise, 6.2 is the result of applying edge `10emails` to 5.1, and 6.4 and 6.5 are the results of applying the two edges (respectively) to 5.2.

[1] The + operation here denotes modular addition to model arithmetic overflows.

Constraints 6.3 and 6.6 model the possibility that no explicit edge matches, and therefore the security state remains unchanged. They are obtained by conjoining the negations of all of the edge pre-conditions to states 5.1 and 5.2, respectively. Thus, security-relevant events have a multiplicative effect on the state space, expanding n abstract states into at worst $n(m+1)$ states, where m is the number of potential pointcut matches.

If any constraint list is satisfiable and contains the expression $A = \texttt{\#}$, the verifier cannot disprove the possibility of a policy violation and therefore conservatively rejects. Constraints 6.2 and 6.5 both contain this expression, but they are unsatisfiable, proving that a violation cannot occur. Observe that the IRM guard at Line 3 is critical for proving the safety of this code because it introduces constraint $S' \neq 10$ that makes these two lists unsatisfiable. If Lines 3–4 were not included, the verifier would reject at this point because constraints 6.2 and 6.5 are satisfiable with $A = \texttt{\#}$ without clause $S' \neq 10$.

At all control-flows from marked to unmarked regions, the verifier requires a constraint list that implies *SYNC*. In this example, constraints 6.1 and 6.6 are the only remaining lists that are satisfiable, and conjoining them with the negation of *SYNC* expression $(A = S) \wedge (A = T)$ yields an unsatisfiable list. Thus, this code is accepted as policy-adherent.

Dynamically Decided Pointcuts. Verification of events corresponding to statically undecidable pointcuts (such as `argval`) requires analysis of dynamic checks inserted by the rewriter, which consider the contents of the stack and local variables at runtime. Numeric comparisons are translated directly into constraint expressions; for example, the instruction `if(x>2)` introduces clause $X > 2$ for the positive branch and $X \leq 2$ for the negative branch. Non-numeric dynamic pointcuts (e.g., `streq` pointcut expressions) are modeled by reducing them to equivalent integer encodings. For example, to support dynamic string regexp-matching, Cheko✓ introduces a boolean-valued variable X_{re} for each string-typed program variable `x` and policy regexp *re*. Program operations that test `x` against *re* introduce constraint $X_{re} = 1$ in their positive branches and $X_{re} = 0$ in their negative branches. An in-depth verification example involving dynamically decidable pointcuts is provided in the companion technical report [15].

Limitations. Our verifier supports most forms of Java reflection, but in order to safely track write-accesses to reified security state fields, the verifier requires such fields to be static, private class members, and it conservatively rejects programs that contain reflective field-write operations within classes that contain reified state. Thus, in order to pass verification, rewriters must implement reified state fields within classes that do not perform write-reflection. This is standard practice for most IRM systems including SPoX, so did not limit any of our tests. Instrumented programs may detect and respond to the presence of the IRM through read-reflection, but not in a way that violates the policy.

Our system supports IRMs that maintain a global invariant whose preservation across the majority of the rewritten code suffices to prove safety for small sections of security-relevant code, followed by restoration of the invariant. Our

experience with existing IRM systems indicates that most IRMs do maintain such an invariant (*SYNC*) as a way to avoid reasoning about large portions of security-irrelevant code in the original binary. However, IRMs that maintain no such invariant, or that maintain an invariant inexpressible in our constraint language, cannot be verified by our system. For example, an IRM that stores object security states in a hash table cannot be certified because our constraint language is not sufficiently powerful to express collision properties of hash functions and prove that a correct mapping from security-relevant objects to their security states is maintained by the IRM.

To keep the rewriter's annotation burden small, our certifier also uses this same invariant as a loop-invariant for all cycles in the control-flow graph. This includes recursive cycles in the call graph as well as control-flow cycles within method bodies. Most IRM frameworks do not introduce such loops to non-synchronized regions. However, this limitation could become problematic for frameworks wishing to implement code-motion optimizations that separate security-relevant operations from their guards by an intervening loop boundary. Allowing the rewriter to suggest different invariants for different loops would lift the limitation, but taking advantage of this capability would require the development of rewriters that infer and express suitable loop invariants for the IRMs they produce. To our knowledge, no existing IRM systems yet do this.

While our certifier is provably convergent (since it arrives at a fixpoint for every loop through enforcing *SYNC* on loop back-edges), it can experience state-space explosions that are exponential in the size of each contiguous, unsynchronized code region. Typical IRMs limit such regions to relatively small, separate code blocks scattered throughout the rewritten code; therefore, we have not observed this to be a significant limitation in practice. However, such state-space explosions could be controlled without conservative rejection by applying the same solution above. That is, rewriters could suggest state abstractions for arbitrary code points, allowing the certifier to forget information that is unnecessary for proving safety and that leads to a state-space explosion. Again, the challenge here is developing rewriters that can actually generate such abstractions.

Our current implementation and theoretical analysis are for purely serial programs; concurrency support is reserved for future work. Analysis, enforcement, and certification of multithreaded IRMs is an ongoing subject of current research with several interesting open problems (cf., [6]).

Soundness. Our certifier forms the centerpiece of the TCB of the system, allowing the monitor and monitor-producing tools to remain untrusted. An unsound certifier (i.e., one that fails to reject some policy-violating programs) can lead to system compromise and potential damage. It is therefore important to establish exceptionally high assurance for the certification algorithm. We proved the soundness of our approach using Cousot's abstract interpretation framework [5].

The proof models the verification algorithm as the small-step operational semantics of an abstract machine. A corresponding concrete operational semantics models the Java VM's interpretation of bytecode instructions. For brevity, the concrete and abstract operational semantics concern a small, relevant core

subset of Java bytecode instructions rather than the full bytecode language. The core language is semantically connected to full Java bytecode through Classic-Java [13,14]. Bisimulation of the abstract and concrete machines provably satisfies a soundness property that relates abstract states to the concrete states they abstract. This is proved via the following progress and preservation lemmas.

Lemma 1 (Progress). *If abstract machine state $\hat{\chi}$ is a sound abstraction of concrete machine state χ, and $\hat{\chi}$ takes a step (i.e., the verifier does not reject), then χ takes a step (i.e., the concrete machine does not exhibit a policy violation).*

Lemma 2 (Preservation). *If abstract machine state $\hat{\chi}$ soundly abstracts concrete machine state χ, and χ steps to χ', then $\hat{\chi}$ steps to some state $\hat{\chi}'$ that is a sound abstraction of χ'.*

The preservation lemma proves that a bisimulation of the abstract and concrete machines preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the abstract machine anticipates all policy violations of the concrete machine. Both proofs are standard (but lengthy) structural inductions over the respective operational semantic derivations. Together, these two lemmas dovetail to form an induction over arbitrary length execution sequences, proving that programs accepted by the verifier will not violate the policy. Detailed operational semantics and proofs can be found in the companion technical report [15].

5 Case Studies

Our prototype verifier implementation consists of 5200 lines of Prolog and 9100 lines of Java. The Prolog code runs under 32-bit SWI-Prolog 5.10.4, which communicates with Java via the JPL interface. The Java side parses SPoX policies and Java bytecode, and compares bytecode instructions to the policy to recognize security-relevant events. The Prolog code forms the core of the verifier, and handles control-flow analysis, model-checking, and linear constraint analysis using CLP. Model-checking is only applied to code that the rewriter has marked as security-relevant. Unmarked code is subjected to a linear scan that ensures that it lacks security-relevant instructions and reified security state modifications.

We have used our prototype implementation to rewrite and then successfully verify several Java applications, discussed throughout the remainder of the section. Statistics are summarized in Table 1. All tests were performed on a Dell Studio XPS notebook computer running Windows 7 64-bit with an Intel i7-Q720M quad core processor, a Samsung PM800 solid state drive, and 4 GB of memory. A more detailed description of each application can be found in [15].

In Table 1, file sizes are expressed in three parts: the original size of the main program before rewriting, the size after rewriting, and the size of system libraries that needed to be verified (but not rewritten). Verification of system library code is required to verify the safety of control-flows that pass through them. Likewise, each cell in the classes column has two parts: the number of classes in the main program and the number of classes in the libraries.

Table 1. Experimental Results

Program	Policy	File Sizes (KB) old / new / libs	# Classes old / libs	Rewrite Time (s)	# Evts.	Total Verif. Time (s)	Model Check Time (s)
EJE	NoExecSaves	439/ 439/ 0	147/ 0	6.1	1	202.8	16.3
RText		1264/1266/ 835	448/ 680	52.1	7	2797.5	54.5
JSesh		1923/1924/20878	863/ 1849	57.8	1	5488.1	196.0
vrenamer	NoExecRename	924/ 927/ 0	583/ 0	50.1	9	1956.8	41.0
jconsole	NoUnsafeDel	35/ 36/ 0	33/ 0	0.6	2	115.7	15.1
jWeather	NoSndsAftrRds	288/ 294/ 0	186/ 0	12.3	46	308.2	156.7
YTDownload		279/ 281/ 0	148/ 0	17.8	20	219.0	53.6
jfilecrypt	NoGui	303/ 303/ 0	164/ 0	9.7	1	642.2	2.8
jknight	OnlySSH	166/ 166/ 4753	146/ 2675	4.5	1	650.1	3.0
Multivalent	EncrpytPDF	1115/1116/ 0	559/ 0	129.9	7	3567.0	26.9
tn5250j	PortRestrict	646/ 646/ 0	416/ 0	85.4	2	2598.2	23.6
jrdesktop	SafePort	343/ 343/ 0	163/ 0	8.3	5	483.0	17.8
JVMail	TenMails	24/ 25/ 0	21/ 0	1.6	2	35.1	8.0
JackMail		165/ 166/ 369	30/ 269	2.5	1	626.7	8.9
Jeti	CapLgnAttmpts	484/ 484/ 0	422/ 0	15.3	1	524.3	8.8
ChangeDB	CapMembers	82/ 83/ 404	63/ 286	4.3	2	995.3	12.0
projtimer	CapFileCreat	34/ 34/ 0	25/ 0	15.3	1	56.2	6.1
xnap	NoFreeRide	1250/1251/ 0	878/ 0	24.8	4	1496.2	56.4
Phex		4586/4586/ 3799	1353/ 830	69.4	2	5947.0	172.7
Webgoat	NoSqlXss	429/ 431/ 6338	159/ 3579	16.7	2	10876.0	120.0
OpenMRS	NoSQLInject	1781/1783/24279	932/17185	78.7	6	2897.0	37.3
SQuirreL	SafeSQL	1788/1789/ 1003	1328/ 626	140.2	1	3352.1	37.3
JVMail	LogEncrypt	25/ 26/ 0	22/ 0	1.8	6	71.3	43.2
jvs-vfs	CheckDeletion	277/ 277/ 0	127/ 0	4.4	2	193.9	6.3
sshwebproxy	EncryptPayload	36/ 37/ 389	19/ 16	1.1	5	66.7	7.0

Six of the rewritten applications listed in Table 1 (`vrenamer`, `jWeather`, `jrdesktop`, `Phex`, `Webgoat`, and `SQuirreL`) were initially rejected by our verifier due to a subtle security flaw that our verifier uncovered in the SPoX rewriter. For each of those cases, a bytecode analysis revealed that the original code contained a form of generic exception handler that can potentially hijack control-flows within IRM guard code. This could cause the abstract and reified security state to become desynchronized, breaking soundness. We corrected this by manually editing the rewritten bytecode to exclude guard code from the scope of the outer exception handler. This resulted in successful verification. Our fix could be automated by in-lining inner exception handlers for guard code to protect them from interception by an outer handler.

The following discussion groups the case-studies into four policy classes. SPoX policies are provided in a generalized form representative of the various instantiations of the policy that we used for specific applications. The real policies substitute the simple pointcut expressions in each sample with more complex, application-specific pointcuts that are here omitted for space reasons.

Filename Guards. Our `NoExecSaves` policy (generalized below) prevents file-creation operations from specifying a file name with an executable extension. Such a policy could be used to prevent malware propagation.

```
1 (edge name="saveToExe"
2   (nodes "s" 0,#)
3   (and (call "java.io.FileWriter.new")
4     (argval 1 (streq ".*\.(exe|bat|...)"))
5     (withincode "FileSystem.saveFile")))
```

The regular expression in Line 4 matches any string that ends in an executable file extension. There are many file extensions that are considered to be executable on Windows; we included all listed at [12]. This policy was enforced on three applications: `EJE`, a Java code editor; `RText`, a text editor; and `JSesh`, a heiroglyphics editor for use by archaeologists. After rewriting, each program halted when we tried to save a file with a prohibited extension.

Another policy that prevents deletion of policy-specified file directories (not shown) was enforced on `jconsole`. The policy monitors directory-removal system API calls for arguments that match a regular expression specifying names of protected directories. For `vrenamer`, a mass file-renaming application, we prohibited files being renamed to include executable extensions.

Event ordering. A canonical information flow policy in the IRM literature prohibits all network-send operations after a secret file has been read. The following `NoSndsAftrRds` policy prevents calls to `Socket.getOutputStream` after any call to `java.io.File` where the first argument refers to the `Windows` directory.

```
1 (edge name="FileRead"
2   (nodes "s" 0,1)
3   (and (call "java.io.File.*")
4     (argval 1 (streq "[A-Za-z]*:\\Windows\\.*"))))
5 (edge name="NetworkSend"
6   (nodes "s" 1,#)
7   (call "java.net.Socket.getOutputStream"))
```

We enforced this policy on `jWeather`, a weather widget application, and `YouTube Downloader` (`YTDownload` in the table), which downloads videos from `YouTube`. Neither program violated the policy, so no change in behavior occurred. However, both programs access many files and sockets, so SPoX instrumented both programs with a large number of security checks.

For `multivalent`, a document browsing utility, we enforced a policy that disallows saving a PDF document until a call has first been made to its built-in encryption method. The two-state policy is similar to the one shown above.

Malicious SQL and XSS protection. SPoX's use of string regular expressions facilitates natural specifications of policies that protect against SQL injection and cross-site scripting attacks. One such policy is `NoSqlXss`, a policy that uses whitelisting to exclude potentially dangerous input characters. We enforced `NoSqlXss` on `Webgoat`.

One edge definition in the policy contained a large number of dynamic `argval` pointcuts (twelve); nevertheless, verification time remained roughly linear in the size of the rewritten code because the verifier was able to significantly prune

the search space by combining redundant constraints and control-flows during model-checking and abstract interpretation.

A similar policy was used to prevent SQL injection attacks on a search function in `OpenMRS`. The library portion of this application is extremely large but contains no security-relevant events; thus, our non-stateful verification approach for unmarked code regions was crucial for avoiding state-space explosions.

We also enforced a blacklisting policy (not shown) on the database access client `SQuirreL`, preventing SQL commands which drop, alter, or rename tables or databases. The policy used a regular expression guard to disallow all SQL commands that implement these operations.

Ensuring advice execution. Most aspectual policy languages (e.g., [4,2,10,26]) allow policies to include explicit advice code that implements IRM guards and interventions. Such systems can be applied to create custom implementations of SPoX policies, such as those that perform custom actions when impending violations are detected. Cheko✓ can then take the SPoX policy as input and verify that the implementation correctly enforces the policy.

To simulate this, we manually added encryption and logging calls immediately prior to email-send events in `JVMail`. Each email is therefore encrypted, then logged, then sent. The SPoX policy `LogEncrypt` requires these events occur in that order. After inserting the advice, we used the verifier to prove that the rewritten `JVMail` application satisfies the policy. A similar policy was applied to the Java Virtual File System (`jvs-vfs`), only allowing file deletion after execution of advice code that consults the user. Finally, we enforced a policy on `sshwebproxy` that requires the proxy to encrypt messages before sending.

6 Conclusion and Future Work

IRMs provide a more powerful alternative to purely static analysis, allowing precise enforcement of a much larger and sophisticated class of security policies. Combining this power with a purely static analysis that independently checks the instrumented, self-monitoring code results in an effective, provably sound, and flexible hybrid enforcement framework. Additionally, an independent certifier allows for the removal of the larger and less general rewriter from the TCB.

We developed Cheko✓—the first automated, model-checking-based certifier for an aspect-oriented, real-world IRM system [14]. Cheko✓ uses a flexible and semantic static code analysis, and supports difficult features such as reified security state, event detection by pointcut-matching, combinations of untrusted before- and after-advice, and pointcuts that are not statically decidable. Strong formal guarantees are provided through proofs of soundness and convergence based on Cousot's abstract interpretation framework. Since Cheko✓ performs independent certification of instrumented binaries, it is flexible enough to accommodate a variety of IRM instrumentation systems, as long as they provide (untrusted) hints about reified state variables and locations of security-relevant events. Such hints are easy for typical rewriter implementations to provide, since they typically correspond to in-lined state variables and guard code, respectively.

Our focus was on presenting main design features of the verification algorithm, and an extensive practical study using a prototype implementation of the tool. Experiments revealed at least one security vulnerability in the SPoX IRM system, indicating that automated verification is important and necessary for high assurance in these frameworks.

In future work we intend to turn our development toward improving efficiency and memory management of the tool. Much of the overhead we observed in experiments was traceable to engineering details, such as expensive context-switches between the separate parser, abstract interpreter, and model-checking modules. These tended to eclipse more interesting overheads related to the abstract interpretation and model-checking algorithms. We also intend to examine more powerful rewriter-supplied hints that express richer invariants. Such advances will provide greater flexibility for alternative IRM implementations of stateful policies.

References

1. Aktug, I., Dam, M., Gurov, D.: Provably Correct Runtime Monitoring. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 262–277. Springer, Heidelberg (2008)
2. Aktug, I., Naliuka, K.: ConSpec - a formal language for policy specification. Science of Comput. Prog. 74, 2–12 (2008)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2, 117–126 (1986)
4. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. Sym. on Principles of Prog. Lang., pp. 234–252 (1977)
6. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Security Monitor Inlining for Multithreaded Java. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 546–569. Springer, Heidelberg (2009)
7. Dantas, D.S., Walker, D.: Harmless advice. In: Proc. ACM Sym. on Principles of Prog. Lang. (POPL), pp. 383–396 (2006)
8. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. ACM Trans. Prog. Lang. and Systems 30(3) (2008)
9. DeVries, B.W., Gupta, G., Hamlen, K.W., Moore, S., Sridhar, M.: ActionScript bytecode verification with co-logic programming. In: Proc. ACM Workshop on Prog. Lang. and Analysis for Security (PLAS), pp. 9–15 (2009)
10. Erlingsson, Ú.: The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Cornell University, Ithaca, New York (2004)
11. Erlingsson, Ú., Schneider, F.B.: SASI enforcement of security policies: A retrospective. In: Proc. New Security Paradigms Workshop (NSPW), pp. 87–95 (1999)
12. FileInfo.com: Executable file types (2011), `http://www.fileinfo.com/filetypes/executable`

13. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and mixins. In: Proc. ACM Sym. on Principles of Prog. Lang. (POPL), pp. 171–183 (1998)
14. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: Proc. ACM Workshop on Prog. Lang. and Analysis for Security (PLAS), pp. 11–20 (2008)
15. Hamlen, K.W., Jones, M.M., Sridhar, M.: Chekov: Aspect-oriented runtime monitor certification via model-checking (extended version). Tech. rep., Dept. of Comput. Science, U. Texas at Dallas (May 2011)
16. Hamlen, K.W., Mohan, V., Masud, M.M., Khan, L., Thuraisingham, B.: Exploiting an antivirus interface. Comput. Standards & Interfaces J. 31(6), 1182–1189 (2009)
17. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified in-lined reference monitoring on. NET. In: Proc. ACM Workshop on Prog. Lang. and Analysis for Security (PLAS), pp. 7–16 (2006)
18. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Prog. Lang. and Systems 28(1), 175–205 (2006)
19. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program., 503–581 (1994)
20. Jones, M., Hamlen, K.W.: Enforcing IRM security policies: Two case studies. In: Proc. IEEE Intelligence and Security Informatics (ISI) Conf., pp. 214–216 (2009)
21. Jones, M., Hamlen, K.W.: Disambiguating aspect-oriented policies. In: Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD), pp. 193–204 (2010)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. Li, Z., Wang, X.: FIRM: Capability-based inline mediation of Flash behaviors. In: Proc. Annual Comput. Security Applications Conf. (ACSAC), pp. 181–190 (2010)
24. Ligatti, J.A.: Policy Enforcement via Program Monitoring. Ph.D. thesis, Princeton University, Princeton, New Jersey (2006)
25. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Int. J. Information Security 4(1-2), 2–16 (2005)
26. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Trans. Information and Systems Security 12(3) (2009)
27. Schneider, F.B.: Enforceable security policies. ACM Trans. Information and Systems Security 3(1), 30–50 (2000)
28. Shah, V., Hill, F.: An aspect-oriented security framework. In: Proc. DARPA Information Survivability Conf. and Exposition, vol. 2 (2003)
29. Sridhar, M., Hamlen, K.W.: ActionScript In-Lined Reference Monitoring in Prolog. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 149–151. Springer, Heidelberg (2010)
30. Sridhar, M., Hamlen, K.W.: Model-Checking In-Lined Reference Monitors. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 312–327. Springer, Heidelberg (2010)
31. Sridhar, M., Hamlen, K.W.: Flexible in-lined reference monitor certification: Challenges and future directions. In: Proc. ACM Workshop on Prog. Lang. meets Program Verification (PLPV), pp. 55–60 (2011)
32. Viega, J., Bloch, J.T., Chandra, P.: Applying aspect-oriented programming to security. Cutter IT J. 14(2) (2001)
33. Walker, D.: A type system for expressive security policies. In: Proc. of ACM Sym. on Principles of Prog. Lang. (POPL) (2000)