Chekov✓: Aspect-Oriented Runtime Monitor Certification via Model-Checking (Extended Version)

Kevin W. Hamlen Micah M. Jones Meera Sridhar

The University of Texas at Dallas

Technical Report UTDCS-16-11

May 16, 2011

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.

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 a static analysis. 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 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 [30]. 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 not readily visible to an external monitor).

Recent work has observed that the resulting *self-monitoring* code can potentially be machine-verified by a purely static analysis prior to execution (e.g., [17, 1, 33]). This hybrid static-dynamic approach to software security effectively imbues a provably sound, static enforcement system with the power of a dynamic monitor. The resulting protection system can precisely enforce and verify policies that are not decidable by static code-inspection alone because in-lining obviates the proof of safety by adding dynamic security checks that ease the verification task.

Most modern IRM systems are implemented using some form of *aspect-oriented programming* (AOP) [36, 31, 9, 10, 15]; therefore, verification of AOP-based IRM systems is of particular importance. IRMs are implemented in AOP using *pointcut-advice* pairs. The pointcuts identify security-relevant program operations, while the *advice* prescribes a local code transformation sufficient to guard each such operation. This suffices to enforce safety policies [30, 18] and some liveness policies [25].

Independent certification of these AOP-based IRM systems has become increasingly critical due to rapid expansion of their trusted computing bases (TCBs) [34]. Expressing security policies as trusted aspects often has the disadvantage of placing a large portion of the IRM implementation within the TCB by expressing it as the advice of a trusted aspect. When the policy applies to a large class of untrusted binaries rather than just one particular application, these aspects are extremely difficult to write correctly, inviting specification errors [20]. Moreover, in many domains, such as web ad security IRM systems, policy specifications change rapidly as new attacks appear and new vulnerabilities are discovered (cf., [22, 32, 33]). 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.

Therefore, rather than proving that one particular IRM framework correctly modifies all untrusted code instances, we instead consider the challenge of machine-certifying a broad class of instrumented code instances with respect to purely declarative (i.e., advice-free) policy specifications. In contrast to proof-carrying code (PCC) [27], in which a compiler performs instrumentation using source code, instrumentation in an IRM framework typically occurs at a purely binary level where source code is unavailable. Verification therefore demands a binary-level code analysis that does not rely upon an explicit safety proof derived from source-level information. We therefore take the approach of abstract interpreting binary IRMs without appealing to such a proof.

The result of our efforts is Chekov✓, a light-weight Java bytecode abstract interpreter and model-checker capable of certifying a large class of realistic IRMs fully automatically, but without introducing the significant overheads typically

required for model-checking arbitrary code. Chekov✓ is the first IRM certification system that can verify AOP-style IRMs and IRM-policies without appealing to trusted advice. It extends our prior work on model-checking IRMs [33, 32, 11] to a full-scale Java IRM framework (the SPoX IRM system [15, 19]) with support for *stateful* (history-based) policies, event detection by pointcut-matching, and IRM implementations that combine (untrusted) before- and after-advice insertions. Formal proofs of soundness and convergence using Cousot's abstract interpretation framework [7] provide strong formal guarantees that the system successfully prohibits runtime security violations.

The rest of the paper is organized as follows. Section 2 begins with an overview of our framework, including the policy language, its enforcement by Java binary rewriting, and a high-level description of the verification algorithm. Section 3 presents Chekov✓'s formal mathematical model. A detailed treatment of Chekov✓'s soundness and associated proofs is provided in §4. Section 5 presents in-depth case-studies of eight classes of security policies that we enforced on numerous real-world applications, along with a discussion of challenges faced in implementing and verifying these policies. Finally, §6 and §7 discuss related work and recommendations for future work respectively.

2 System Overview

2.1 SPoX Background

SPoX (Security Policy XML) is a purely declarative, aspect-oriented policy specification language [15]. A SPoX specification denotes a *security automaton* [2]—a finite- or infinite-state machine that accepts all and only those *event sequences* that satisfy the security policy.

Security-relevant program *events* are specified in SPoX by pointcut expressions similar to those found in other aspect-oriented languages. In source-level AOP languages, pointcuts identify code *join points* at which advice code is to be inserted. 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 [20]. State-transitions can be specified in terms of information gleaned from the current join point, such as method argument values, the call stack, and the current lexical scope. This allows advice typically encoded imperatively in most other aspect-oriented security languages to be declaratively encoded in SPoX policies. Typically this results in a natural translation from these other languages to SPoX, making SPoX an ideal target for our analysis.

$n \in \mathbb{Z}$	**integers**
$c \in C$	**class names**
$sv \in SV$	**state variables**
$iv \in IV$	**iteration vars**
$en \in EN$	**edge names**
$pn \in PCN$	**pointcut names**
$pol ::= np^* sd^* e^*$	**policies**
$np ::=$ `(pointcut name="`pn`"` pcd`)`	**named pointcuts**
$sd ::=$ `(state name="`sv`")`	**state declarations**
$e ::=$	**edges**
`(edge name="`en`"` [`after`] pcd ep^*`)`	edgesets
| `(forall "`iv`" from` a_1 `to` a_2 e^*`)`	iteration
$ep ::=$	**edge endpoints**
| `(nodes "`sv`"` a_1`,`a_2`)`	state transitions
| `(nodes "`sv`"` a_1`,#)`	policy violations
$a ::= a_1$`+`a_2 | a_1`-`a_2 | b	**arithmetic**
$b ::= n$ | iv | b_1`*`b_2 | b_1`/`b_2 | `(`a`)`	

Figure 1: SPoX policy syntax

The remainder of this section outlines SPoX syntax as background for the case studies in §5. A formal denotational semantics can be found in §3. We here use a simplified Lisp-style syntax for readability; the implementation uses an XML-based syntax for easy parsing, portability, and extensibility.

A SPoX policy specification (*pol* in Fig. 1) is a list of security automaton edge declarations. Each edge declaration consists of three parts:

- Pointcut expressions (Fig. 2) identify sets of related security-relevant events that programs might exhibit at runtime. These label the edges of the security automaton.
- *Security-state variable* declarations (*sd* in Fig. 1) abstract the security state of an arbitrary program. The security state is defined by the set of all program state variables and their integer[1] values. These label the automaton nodes.
- *Security-state transitions* (*e* in Fig. 1) describe how events cause the security automaton's state to change at runtime. These define the transition relation for the automaton.

[1]Binary operator `/` in Fig. 1 denotes integer division.

$re \in RE$	**regular expressions**
$md \in MD$	**method names**
$fd \in FD$	**field names**
$pcd ::=$	**pointcuts**
`(call` mo^* rt $c.md$`)`	method calls
| `(execution` mo^* rt $c.md$`)`	callee executions
| `(get` mo^* $c.fd$`)`	field get
| `(set` mo^* $c.fd$`)`	field set
| `(argval` n vp`)`	stack args (values)
| `(argtyp` n c`)`	stack args (types)
| `(target` c`)`	object refs
| `(withincode` mo^* rt $c.md$`)`	lexical contexts
| `(pointcutid "`pn`")`	named pc refs
| `(cflow` pcd`)`	control flows
| `(and` pcd^*`)`	conjunction
| `(or` pcd^*`)`	disjunction
| `(not` pcd`)`	negation
$mo ::=$ `public` | `private` | $\cdots$	**modifiers**
$rt ::= c$ | `void` | $\cdots$	**return types**
$vp ::=$ `(true)`	**value predicates**
| `(isnull)`	object predicates
| `(inteq` n`)` | `(intne` n`)`	integer predicates
| `(intle` n`)` | `(intge` n`)`	
| `(intlt` n`)` | `(intgt` n`)`	
| `(streq` re`)`	string predicates

Figure 2: SPoX pointcut syntax

```
(state name="s")

(forall "i" from 0 to 9
  (edge name="count"
    (call "Mail.send")
    (nodes "s" i,i+1)))

(edge name="10emails"
  (call "Mail.send")
  (nodes "s" 10,#))
```

Figure 3: A policy permitting at most 10 email-send events

An example policy is given in Fig. 3. Edges are specified by `edge` structures, each of which defines a set of edges in the security automaton. Each `edge` structure consists of a pointcut expression (Lines 5 and 9) and at least one `nodes` declaration (Lines 6 and 10). 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 identified with the reserved post-condition "`#`".

Multiple, similar edges can be introduced with a single `edge` structure by enclosing them within `forall` structures, such as the one in Line 3. These introduce iteration variables (e.g., `i`) that range over the integer lattice points of closed intervals. Thus, Fig. 3 allows state variable `s` to range from 0 to 10, while an 11th send event triggers a policy violation. Such a policy could be useful for preventing spam.

A syntax for a subset of the SPoX pointcut language is given in Fig. 2. SPoX pointcut expressions consist of all pointcuts available in AspectJ [35] except for those that are specific to AspectJ's advice language.[2] This includes all regular expression operators available in AspectJ for specifying class and member names. Since SPoX policies are applied to Java bytecode binaries rather than to source code, the meaning of each pointcut expression is reflected down to the bytecode level. For example, the `target` pointcut matches any Java bytecode instruction whose *this* argument references an object of class *c*.

Instead of AspectJ's `if` pointcut (which evaluates an arbitrary, possibly effectful, Java boolean expression), SPoX provides a collection of effect-free *value predicates* that permit dynamic tests of argument values at join points. These are accessed via the `argval` predicate and include object nullity tests, integer equality and inequality tests, and string regular expression matching. Regular expression tests of non-string objects are evaluated by obtaining the

[2] For example, AspectJ's `adviceexecution()` pointcut is omitted because SPoX lacks advice.

`toString` representation of the object at runtime. (The call to the `toString` method itself is a potentially effectful operation and is treated as a matchable join point by the SPoX enforcement implementation. However, subsequent use of the returned string within injected security guard code is non-effectful.)

2.2 Rewriter

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 §6. Each method body is unpacked, parsed, and scanned for potentially security-relevant instructions—i.e., those that match the statically decidable portions of one or more pointcut expressions in the policy specification. Sequences of guard instructions are then in-lined around these potentially dangerous instructions to detect and preclude policy-violations at runtime. The runtime guards evaluate the statically undecidable portions of the pointcut expressions in order to decide whether the impending event is actually security-relevant. For example, to evaluate the pointcut `(argval 1 (intgt 2))`, the rewriter might guard method calls of the form $m(x)$ with the test $x > 2$.

In-lined guard code must also track event histories if the policy is stateful. To do so, the rewriter reifies abstract security state variables (e.g., `s` in Fig. 3) into the untrusted code as program variables. The guard code then tracks the abstract security state by consulting and updating the corresponding reified state. To protect reified state from tampering, the variables are typically added as private fields of new classes with safe accessor methods. This prevents the surrounding original bytecode from corrupting the reified state and thereby effecting a policy violation.

The left column of Fig. 4 gives pseudocode for an IRM that enforces the policy in Fig. 3. For each call to method `Mail.send`, the IRM tests two possible preconditions: one where $0 \leq \mathtt{s} \leq 9$ and another where $\mathtt{s} = 10$. 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`. 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. In this case the double reification is part of a mechanism for resolving potential join point conflicts in the source policy [20]. A certifier must tolerate such variations in order to be generally applicable to many IRMs and not just one rewriting strategy.

2.3 Verifier Overview

Our verifier takes the approach of [33], using abstract interpretation to non-deterministically explore all control-flow paths of untrusted code, and 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

$(A{=}S \wedge A{=}T)$ **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**

Figure 4: An abstract interpretation of instrumented pseudocode

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 $\zeta \wedge \left(\bigvee_{r \sim a} (r \neq a) \right)$ 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.

2.4 A Verification Example

Figure 4 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. 3. 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.

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. 3 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.

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 $A = \#$, 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.

At all control-flows from marked to unmarked regions, the verifier requires

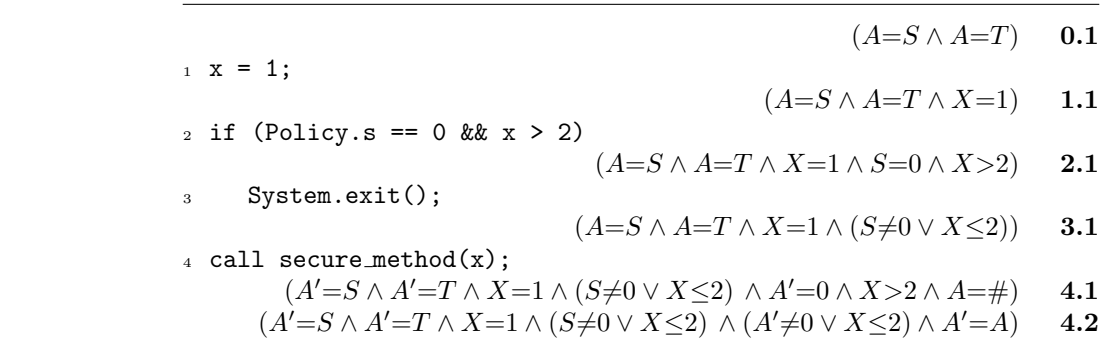

Figure 5: An example verification with dynamically decidable pointcuts

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.

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. An example is shown in Fig. 5, which enforces a policy that prohibits calls to method `secure_method` with arguments greater than 2. Verifying this IRM requires the inclusion of abstract state variable X in constraint lists to model the value of local program variable `x`. The abstract interpreter therefore tracks all numerically typed stack and local variables, and incorporates Java bytecode conditional expressions that test them into constraint lists.

Non-numeric dynamic pointcuts are modeled by reducing them to equivalent integer encodings. For example, to support dynamic string regexp-matching (`streq` pointcut expressions), Chekov✓ 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.

2.5 Limitations

Our system supports IRM's 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 Chekov✓ arrives at a fixpoint for every loop through enforcing *SYNC* at the loop back-edge), 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.

Another limitation of our current framework is that our implementation and theoretical analysis do not provide support for concurrency, which we plan to add in future work. Adding concurrency support requires a standard race detection analysis such as the one implemented by Racer [5].

3 System Formal Model

The certifier in our certifying IRM framework forms the centerpiece of the trusted computing base 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 and its implementation.

In this section we address the former requirement by formalizing the certification algorithm as the operational semantics of an abstract machine. For brevity, we here limit our attention to a core subset of Java bytecode that is representative of important features of the full language.[3] We additionally formalize the JVM as the operational semantics of a corresponding concrete

[3]The implementation supports the full Java bytecode language (see §5).

ifle L	conditional jump
getlocal ℓ	read field given in static operand
setlocal ℓ	write field given in static operand
jmp L	unconditional jump
event$_y$ n	security-relevant operation

Figure 6: Core subset of Java bytecode

$$
\begin{aligned}
\mathit{pol} &::= \mathit{edg}^* \\
\mathit{edg} &::= (\texttt{forall}\ \hat{v}{=}e_1..e_2\ \mathit{edg}) \mid (\texttt{edge}\ \mathit{pcd}\ \mathit{ep}^*) \\
\mathit{pcd} &::= (\texttt{or}\ \mathit{pcc}^*) \\
\mathit{pcc} &::= (\texttt{and}\ \mathit{pct}^*) \\
\mathit{pct} &::= \mathit{pca} \mid (\texttt{not}\ \mathit{pca}) \\
\mathit{pca} &::= (\texttt{event}_y\ n) \mid (\texttt{arg}\ n_1\ (\texttt{intleq}\ n_2)) \\
\mathit{ep} &::= (\texttt{nodes}\ a\ e_1\ e_2) \\
e &::= n \mid \ell \mid r \mid a \mid \hat{v} \mid e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1/e_2 \mid (e)
\end{aligned}
$$

Figure 7: Core subset of SPoX

machine over the same core subset. These two semantics together facilitate a proof of soundness in §4. The proof establishes that executing any program accepted by the certifier never results in a policy violation at runtime.

3.1 Java Bytecode Core Subset

Figure 6 lists the subset of Java bytecode that we consider. Instructions **ifle** L and **jmp** n implement conditional and unconditional jumps, respectively, and instructions **getlocal** n and **setlocal** n read and set local variable values, respectively. Instruction **event**$_y$ n models a security-relevant operation that exhibits event n and pops y arguments off the operand stack. While the real Java bytecode instruction set does not include **event**$_y$, in practice it is implemented as a fixed instruction sequence that performs a security-relevant operation (e.g., a system call).

Figure 7 defines a core subset of SPoX for the Java bytecode language in Figure 6. Without loss of generality, it assumes all pointcuts are expressed in disjunctive normal form.

$$
\begin{array}{ll}
\chi ::= \langle L : i, \rho, \sigma \rangle & \text{(configurations)} \\
L & \text{(code labels)} \\
i & \text{(Java bytecode instructions)} \\
\Sigma : (r \uplus a \uplus \ell) \rightharpoonup \mathbb{Z} & \text{(concrete store mappings)} \\
\sigma \in \Sigma & \text{(concrete stores)} \\
\rho ::= \cdot \mid x :: \rho & \text{(concrete stack)} \\
x \in \mathbb{Z} & \text{(concrete program values)} \\
\chi_0 & \text{(initial configurations)} \\
P ::= (L, p, s) & \text{(programs)} \\
p : L \longrightarrow i & \text{(instruction labels)} \\
s : L \longrightarrow L & \text{(label successors)}
\end{array}
$$

Figure 8: Concrete machine configurations and programs

3.2 Concrete Machine

We start out by formalizing the JVM as the operational semantics of a concrete machine over our core Java bytecode subset. Following the framework established in [33], Fig. 8 defines the concrete machine as a tuple $(\mathcal{C}, \chi_0, \mapsto)$, where $\mathcal{C}$ is the set of concrete configurations, χ_0 is the initial configuration, and $\mapsto$ is the transition relation in the concrete domain. A *concrete configuration* $\chi ::= \langle L{:}i, \rho, \sigma \rangle$ is a triple consisting of a labeled bytecode instruction $L{:}i$, a concrete operand stack ρ, and a concrete store σ. The store σ maps heap and the local variables ℓ, abstract security state variables a, and reified security state variables r to their integer values. A security automaton state is σ restricted to the abstract state variables, denoted $\sigma|_a$.

Figure 9 provides the small-step operational semantics of the concrete machine. Policy-violating events fail to satisfy the premise of Rule (CEvent); therefore the concrete semantics model policy-violations as stuck states. The concrete semantics have no explicit operation for normal program termination; we model termination as an infinite stutter state. The soundness proof in §4 shows that any program that is accepted by the abstract machine will never enter a stuck state during any concrete run; thus, verified programs do not exhibit policy violations when executed.

3.3 SPoX Concrete Denotational Semantics

A SPoX security policy denotes a security automaton whose alphabet is the universe *JP* of all *join points*. We refer to such an automaton as an *aspect-oriented security automaton*. A join point, defined in Fig. 10, is a recursive structure that abstracts the control stack [37]. Join point $\langle k, v^*, jp \rangle$ consists

$$\frac{x_2 \leq x_1}{\langle L_1 : \textbf{ifle}\ L_2, x_1 :: x_2 :: \rho, \sigma\rangle \mapsto \langle L_2 : p(L_2), \rho, \sigma\rangle}\,(\textsc{CIfleLePos})$$

$$\frac{x_2 > x_1}{\langle L_1 : \textbf{ifle}\ L_2, x_1 :: x_2 :: \rho, \sigma\rangle \mapsto \langle s(L_1) : p(s(L_1)), \rho, \sigma\rangle}\,(\textsc{CIfleNeg})$$

$$\frac{}{\langle L : \textbf{getlocal}\ \ell, \rho, \sigma\rangle \mapsto \langle s(L) : p(s(L)), \sigma(\ell) :: \rho, \sigma\rangle}\,(\textsc{CGetlocal})$$

$$\frac{}{\langle L : \textbf{setlocal}\ \ell, x :: \rho, \sigma\rangle \mapsto \langle s(L) : p(s(L)), \rho, \sigma[\ell := x]\rangle}\,(\textsc{CSetlocal})$$

$$\frac{}{\langle L_1 : \textbf{jmp}\ L_2, \rho, \sigma\rangle \mapsto \langle L_2 : p(L_2), \rho, \sigma\rangle}\,(\textsc{CJmp})$$

$$\frac{\sigma' \in \delta(\sigma|_a, \langle \textbf{event}_y\ n, x_1 :: x_2 :: \cdots :: x_y :: \cdot, \langle\rangle\rangle)}{\langle L : \textbf{event}_y\ n, x_1 :: x_2 :: \cdots :: x_y :: \rho_r, \sigma\rangle \mapsto \langle s(L) : p(s(L)), \rho_r, \sigma[a := \sigma'(a)]\rangle}\,(\textsc{CEvent})$$

Figure 9: Concrete small-step operational semantics

$o \in \mathit{Obj}$	objects
$v ::= o \mid \mathsf{null}$	values
$jp ::= \langle\rangle \mid \langle k, x^*, jp\rangle$	join points
$k ::= \texttt{call}\ c.md \mid \texttt{get}\ c.fd \mid \texttt{set}\ c.fd$	join kinds

Figure 10: Join points

of static information k found at the site of the current program instruction, dynamic information v^* including any arguments consumed by the instruction, and recursive join point jp modeling the rest of the control stack. The empty control stack is modeled by the empty join point $\langle\rangle$.

The denotational semantics in Fig. 11 transform a SPoX policy into an aspect-oriented security automaton, which accepts or rejects (possibly infinite) sequences of join points. We use $\uplus$ for disjoint union, Υ for the class of all countable sets, 2^A for the power set of A, $\sqsubseteq$ and $\sqcup$ for the partial order relation and join operation (respectively) over the lattice of partial functions, and $\bot$ for the partial function whose domain is empty. For partial functions f and g we write $f[g] = \{(x, f(x)) \mid x \in Dom(f) \backslash Dom(g)\} \sqcup g$ to denote the replacement of assignments in f with those in g.

Security automata are modeled in the literature [30] as tuples (Q, Q_0, E, δ) consisting of a set Q of states, a set $Q_0 \subseteq Q$ of start states, an alphabet E of events, and a transition function $\delta : (Q \times E) \to 2^Q$. Security automata are non-deterministic; the automaton accepts an event sequence if and only if there exists an accepting path for the sequence. In the case of aspect-oriented security automata, Q is the set of partial functions from security-state variables to values, $Q_0 = \{q_0\}$ is the initial state that assigns 0 to all security-state variables, $E = JP$ is the universe of join points, and δ is defined by the set of edge declarations in the policy (discussed below).

Each edge declaration in a SPoX policy defines a set of source states and the destination state to which each of these source states is mapped when a join point occurs that *matches* the edge's pointcut designator. The denotational semantics in Fig. 11 defines this matching process in terms of the *match-pcd* function from the operational semantics of AspectJ [37]. We adapt a subset of pointcut matching rules from this definition to SPoX syntax in Fig. 12.

3.4 Abstract Machine

In order to statically detect and prevent policy violations, we model the verifier as an abstract machine. The abstract machine is defined as a triple $(\mathcal{A}, \hat{\chi}_0, \rightsquigarrow)$, where $\mathcal{A}$ is the set of configurations of the abstract machine, $\hat{\chi}_0 \in \mathcal{A}$ is an initial configuration, and $\rightsquigarrow$ is the transition relation in the abstract domain. Figure 13 defines *abstract configurations* $\hat{\chi}$ to be either $\bot$ (denoting an unreachable state) or a tuple $\langle L{:}i, \zeta, \hat{\rho}, \hat{\sigma}\rangle$, where $L{:}i$ is a labeled instruction, ζ is a constraint list, and $\hat{\rho}$ and $\hat{\sigma}$ model the abstract operand stack and abstract store, respectively. The domains of $\hat{\rho}$ and $\hat{\sigma}$ consist of symbolic expressions instead of integer values.

The small-step operational semantics of the abstract machine are given in Fig. 14. Rules (AIflePos), (AIfleNeg), and (AEvent) are non-deterministic—the abstract machine non-deterministically explores both branches of conditional jumps and all possible security automaton transitions for security-relevant events.

Rule (AEvent) is the model-checking step. Its premise appeals to an *abstract denotational semantics* $\hat{\mathcal{P}}$ for SPoX, defined in Fig. 15, to infer possible security automaton transitions for policy-satisfying events. Policy-violating events (for

$$
\begin{array}{rll}
q \in Q = a \to \mathbb{Z} & & \text{security states} \\
S \in SM = SV \rightharpoonup \mathbb{Z} & & \text{state-variable maps} \\
\psi \in \Psi = \hat{v} \rightharpoonup \mathbb{Z} & & \text{meta-variable maps} \\
\mu \in \Psi \times \Sigma & & \text{abstract-concrete map pairs} \\
\mathcal{P} : pol \to \big(\Upsilon \times 2^Q \times \Upsilon \times & & \text{policy denotations} \\
\qquad ((Q \times JP) \to 2^Q)\big) & & \\
\mathcal{ES} : edg \to \Psi \to 2^{(JP \to \{Suceed, Fail\}) \times SM \times SM} & & \text{edgeset denotations} \\
\mathcal{PC} : pcd \to JP \to \{Succ, Fail\} & & \text{pointcut denotations} \\
\mathcal{EP} : s \to \Psi \to (SM \times SM) & & \text{endpoint constraints} \\
\mathcal{E} : e \longrightarrow (\Psi \times \Sigma) \longrightarrow \mathbb{Z} & & \text{expression denotations}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[\![edg_1 \ldots edg_n]\!] = (Q, \{q_0\}, JP, \delta) \\
\quad \text{where } q_0 = SV \times \{0\} \\
\quad \text{and } \delta(q, jp) = \{q[S'] \mid (f, S, S') \in \cup_{1 \le i \le n} \mathcal{ES}[\![edg_i]\!]\bot, S \sqsubseteq q, f(jp) = Succ\} \\
\mathcal{ES}[\![\texttt{(forall } \hat{v} \texttt{ from } a_1 \texttt{ to } a_2 \ edg\texttt{)}]\!]\psi = \\
\qquad \cup_{\mathcal{A}[\![a_1]\!]\psi \le j \le \mathcal{A}[\![a_2]\!]\psi} \mathcal{ES}[\![edg]\!](\psi[j/\hat{v}]) \\
\mathcal{ES}[\![\texttt{(edge } pcd \ ep_1 \ldots ep_n\texttt{)}]\!]\psi = \\
\qquad \big\{(\mathcal{PC}[\![pcd]\!],\ \sqcup_{1 \le j \le n} S_j,\ \sqcup_{1 \le j \le n} S'_j)\big\} \\
\quad \text{where } \forall j \in \mathbb{N} \,.\, (1 \le j \le n) \Rightarrow ((S_j, S'_j) = \mathcal{EP}[\![ep_j]\!]\psi) \\
\mathcal{PC}[\![pcd]\!]jp = \textit{match-pcd}(pcd)jp \\
\mathcal{EP}[\![\texttt{(nodes "}sv\texttt{" } a_1\texttt{,}a_2\texttt{)}]\!]\psi = \\
\qquad \big(\{(sv, \mathcal{E}[\![a_1]\!](\psi, \bot))\}, \{(sv, \mathcal{E}[\![a_2]\!](\psi, \bot))\}\big) \\
\mathcal{E}[\![n]\!]\mu = n \\
\mathcal{E}[\![x]\!](\psi, \sigma) = \sigma(x) \qquad\qquad (x \in r \uplus a \uplus \ell) \\
\mathcal{E}[\![\hat{v}]\!](\psi, \sigma) = \psi(\hat{v}) \\
\mathcal{E}[\![e_1 + e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu + \mathcal{E}[\![e_2]\!]\mu \\
\mathcal{E}[\![e_1 - e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu - \mathcal{E}[\![e_2]\!]\mu \\
\mathcal{E}[\![e_1 \cdot e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu \cdot \mathcal{E}[\![e_2]\!]\mu \\
\mathcal{E}[\![e_1 / e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu / \mathcal{E}[\![e_2]\!]\mu
\end{array}
$$

Figure 11: Denotational semantics for SPoX

$$
\begin{array}{l}
\mathit{match\text{-}pcd}(\texttt{(call}\ \ c.md\texttt{)})\langle\texttt{call}\ c.md, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(get}\ \ c.fd\texttt{)})\langle\texttt{get}\ c.fd, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(set}\ \ c.fd\texttt{)})\langle\texttt{set}\ c.fd, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(argval}\ \ n\ vp\texttt{)})\langle k, v_0 \cdots v_n \cdots, jp\rangle \\
\quad = \mathit{Succ} \text{ if } vp = \texttt{(true)} \text{ or } (vp = \texttt{(isnull)} \text{ and } v_n = \texttt{null}) \\
\mathit{match\text{-}pcd}(\texttt{(and}\ \ pcd_1\, pcd_2\texttt{)})jp = \\
\quad \mathit{match\text{-}pcd}(pcd_1)jp \wedge \mathit{match\text{-}pcd}(pcd_2)jp \\
\mathit{match\text{-}pcd}(\texttt{(or}\ \ pcd_1\, pcd_2\texttt{)})jp = \\
\quad \mathit{match\text{-}pcd}(pcd_1)jp \vee \mathit{match\text{-}pcd}(pcd_2)jp \\
\mathit{match\text{-}pcd}(\texttt{(not}\ \ pcd\texttt{)})jp = \neg \mathit{match\text{-}pcd}(pcd) \\
\mathit{match\text{-}pcd}(\texttt{(cflow}\ \ pcd\texttt{)})\langle k, v^*, jp\rangle = \\
\quad \mathit{match\text{-}pcd}(pcd)\langle k, v^*, jp\rangle \vee \mathit{match\text{-}pcd}(\texttt{(cflow}\ \ pcd\texttt{)})jp \\
\mathit{match\text{-}pcd}(pcd)jp = \mathit{Fail} \quad \text{otherwise}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{Succ} \vee \mathit{Succ} = \mathit{Succ} & \mathit{Succ} \wedge \mathit{Succ} = \mathit{Succ} & \neg \mathit{Succ} = \mathit{Fail} \\
\mathit{Fail} \vee \mathit{Fail} = \mathit{Fail} & \mathit{Fail} \wedge \mathit{Fail} = \mathit{Fail} & \neg \mathit{Fail} = \mathit{Succ} \\
\mathit{Succ} \vee \mathit{Fail} = \mathit{Succ} & \mathit{Succ} \wedge \mathit{Fail} = \mathit{Fail} & \\
\mathit{Fail} \vee \mathit{Succ} = \mathit{Succ} & \mathit{Fail} \wedge \mathit{Succ} = \mathit{Fail} &
\end{array}
$$

Figure 12: Matching pointcuts to join points

$$
\begin{array}{lr}
\hat{\chi} ::= \bot \mid \langle L{:}i, \zeta, \hat{\rho}, \hat{\sigma}\rangle & \text{(abstract configs)} \\
\zeta ::= \bigwedge_{i=1\ldots n} t_i \quad (n \geq 1) & \text{(constraints)} \\
t ::= T \mid F \mid e_1 \leq e_2 & \text{(predicates)} \\
\hat{\rho} ::= \cdot \mid e :: \hat{\rho} & \text{(abstract stack)} \\
\hat{\Sigma} : (r \uplus \ell) \longrightarrow e & \text{(abstract store mappings)} \\
\hat{\sigma} \in \hat{\Sigma} & \text{(abstract stores)} \\
\hat{\chi}_0 & \text{(initial abstract config)}
\end{array}
$$

Figure 13: Abstract machine configurations

$$\frac{}{\langle L_1 : \textbf{ifle}\ L_2, \zeta, e_1 :: e_2 :: \hat{\rho}, \hat{\sigma}\rangle \rightsquigarrow \langle L_2 : p(L_2), \zeta \wedge (e_2 \leq e_1), \hat{\rho}, \hat{\sigma}\rangle}\,(\textsc{AIflePos})$$

$$\frac{}{\langle L_1 : \textbf{ifle}\ L_2, \zeta, e_1 :: e_2 :: \hat{\rho}, \hat{\sigma}\rangle \rightsquigarrow \langle s(L_1) : p(s(L_1)), \zeta \wedge (e_2 > e_1), \hat{\rho}, \hat{\sigma}\rangle}\,(\textsc{AIfleNeg})$$

$$\frac{}{\langle L : \textbf{getlocal}\ \ell, \zeta, \hat{\rho}, \hat{\sigma}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \zeta, \hat{\sigma}(\ell) :: \hat{\rho}, \hat{\sigma}\rangle}\,(\textsc{AGetlocal})$$

$$\frac{\hat{v} \text{ is fresh}}{\langle L : \textbf{setlocal}\ \ell, \zeta, e :: \hat{\rho}, \hat{\sigma}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \zeta[\hat{v}/\ell], \hat{\rho}[\hat{v}/\ell], \hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]]\rangle}\,(\textsc{ASetlocal})$$

$$\frac{}{\langle L_1 : \textbf{jmp}\ L_2, \zeta, \hat{\rho}, \hat{\sigma}\rangle \rightsquigarrow \langle L_2 : p(L_2), \zeta, \hat{\rho}, \hat{\sigma}\rangle}\,(\textsc{AJmp})$$

$$\frac{\zeta_2 \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\langle \textbf{event}_y\ n, e_1 :: e_2 :: \cdots :: e_y :: \cdot, \langle\rangle\rangle}{\begin{array}{c}\langle L : \textbf{event}_y\ n, \zeta_1, e_1 :: e_2 :: \cdots :: e_y :: \hat{\rho}_r, \hat{\sigma}\rangle \rightsquigarrow \\ \langle s(L) : p(s(L)), \zeta_1[\theta(a)/a] \wedge \zeta_2[\theta(a)/a_0], \hat{\rho}_r, \hat{\sigma}\rangle\end{array}}\,(\textsc{AEvent})$$

Figure 14: Abstract small-step operational semantics

$$\begin{aligned}\hat{\mathcal{P}} &: pol \to \widehat{JP} \to 2^{\zeta}\\ \widehat{\mathcal{ES}} &: edg \to \widehat{JP} \to 2^{\zeta}\\ \widehat{\mathcal{PCD}} &: pcd \to \widehat{JP} \to 2^{\zeta}\\ \widehat{\mathcal{PCC}} &: pcc \to \widehat{JP} \rightharpoonup \zeta\\ \widehat{\mathcal{EP}} &: ep \to \zeta\end{aligned}$$

$$\hat{\mathcal{P}}[\![edg_1 \ldots edg_n]\!]\hat{jp} = \bigcup_{i=1}^{n} \widehat{\mathcal{ES}}[\![edg_i]\!]\hat{jp}$$

$$\widehat{\mathcal{ES}}[\![\texttt{forall}\ \hat{v}{=}e_1..e_2\ edg]\!]\hat{jp} = \{(\hat{v} \geq e_1) \wedge (\hat{v} \leq e_2) \wedge \zeta \mid \zeta \in \widehat{\mathcal{ES}}[\![edg]\!]\hat{jp}\}$$

$$\widehat{\mathcal{ES}}[\![\texttt{edge}\ pcd\ ep_1 \ldots ep_n]\!]\hat{jp} = \{\zeta \wedge (\wedge_{i=1}^{n} \widehat{\mathcal{EP}}[\![ep_i]\!]) \mid \zeta \in \widehat{\mathcal{PCD}}[\![pcd]\!]\hat{jp}\}$$

$$\widehat{\mathcal{PCD}}[\![\texttt{or}\ pcc_1 \ldots pcc_n]\!]\hat{jp} = \{\widehat{\mathcal{PCC}}[\![pcc_i]\!]\hat{jp} \mid 1 \leq i \leq n\}$$

$$\widehat{\mathcal{PCC}}[\![\texttt{and}\ pct_1 \ldots pct_n]\!]\hat{jp} = \bigwedge_{i=1}^{n} \widehat{\mathcal{PCC}}[\![pct_i]\!]jp$$

$$\widehat{\mathcal{PCC}}[\![\texttt{not}\ pca]\!]\hat{jp} = \neg(\widehat{\mathcal{PCC}}[\![pca]\!]\hat{jp})$$

$$\widehat{\mathcal{PCC}}[\![\texttt{event}_y\ n]\!]\langle \texttt{event}_z\ m, e^*, \hat{jp}\rangle = (n{=}m)$$

$$\widehat{\mathcal{PCC}}[\![\texttt{arg}\ n\ (\texttt{intleq}\ m)]\!]\langle k, e_1{::}\cdots{::}e_n{::}e^*, \hat{jp}\rangle = (e_n \leq m)$$

$$\widehat{\mathcal{EP}}[\![\texttt{nodes}\ a\ e_1\ e_2]\!] = (a_0 = e_1) \wedge (a = e_2)$$

Figure 15: Abstract Denotational Semantics

$$\mathcal{T} : e \longrightarrow 2^{\Psi \times \Sigma}$$

$$\mathcal{T}[\![T]\!] = \Psi \times \Sigma$$

$$\mathcal{T}[\![F]\!] = \emptyset$$

$$\mathcal{T}[\![e_1 \leq e_2]\!] = \{\mu \in \Psi \times \Sigma \mid \mathcal{E}[\![e_1]\!]\mu \leq \mathcal{E}[\![e_2]\!]\mu\}$$

$$\mathcal{C} : \zeta \longrightarrow 2^{\Psi \times \Sigma}$$

$$\mathcal{C}[\![\bigwedge_{i=1\ldots n} t_i]\!] = \bigcap_{i=1\ldots n} \mathcal{T}[\![t_i]\!]$$

$$\frac{\mathcal{E}[\![e_1]\!](\mathcal{C}[\![\zeta_1]\!]) \subseteq \mathcal{E}[\![e_2]\!](\mathcal{C}[\![\zeta_2]\!])}{(\zeta_1, e_1) \preceq_e (\zeta_2, e_2)}$$

$$\frac{\mathcal{C}[\![\zeta_1]\!] \subseteq \mathcal{C}[\![\zeta_2]\!]}{(\zeta_1, \cdot) \preceq_\rho (\zeta_2, \cdot)}$$

$$\frac{(\zeta_1, e_1) \preceq_e (\zeta_2, e_2) \qquad (\zeta_1, \hat{\rho}_1) \preceq_\rho (\zeta_2, \hat{\rho}_2)}{(\zeta_1, e_1 :: \hat{\rho}_1) \preceq_\rho (\zeta_2, e_2 :: \hat{\rho}_2)}$$

$$\frac{\hat{\sigma}_1^{\leftarrow} = \hat{\sigma}_2^{\leftarrow} \qquad \forall x \in \hat{\sigma}^{\leftarrow}.\mathcal{E}[\![\hat{\sigma}_1(x)]\!](\mathcal{C}[\![\zeta_1]\!]) \subseteq \mathcal{E}[\![\hat{\sigma}_2(x)]\!](\mathcal{C}[\![\zeta_2]\!])}{(\zeta_1, \hat{\sigma}_1) \preceq_\sigma (\zeta_2, \hat{\sigma}_2)}$$

$$\frac{\mathcal{C}[\![\zeta_1]\!] \subseteq \mathcal{C}[\![\zeta_2]\!] \qquad (\zeta_1, \hat{\rho}_1) \preceq_\rho (\zeta_2, \hat{\rho}_2) \qquad (\zeta_1, \hat{\sigma}_1) \preceq_\sigma (\zeta_2, \hat{\sigma}_2)}{\hat{\chi}_1 = \langle L : i, \zeta_1, \hat{\rho}_1, \hat{\sigma}_1 \rangle \preceq_{\hat{\chi}} \hat{\chi}_2 = \langle L : i, \zeta_2, \hat{\rho}_2, \hat{\sigma}_2 \rangle}$$

Figure 16: State-ordering relation $\preceq_{\hat{\chi}}$

which there is no transition in the automaton) therefore correspond to stuck states in the abstract semantics.

In Fig. 15, $\widehat{jp} \in \widehat{JP}$ denotes an *abstract join point*—a join point (see Fig. 10) whose stack consists of symbolic expressions instead of values. Valuation function $\hat{\mathcal{P}}$ accepts as input a policy and an abstract join point that models the current abstract program state. It returns a set of constraint lists, one list for each possible new abstract program state. If ζ_1 is the original constraint list, then the new set of constraint lists is

$$\{\zeta_1[\theta(a)/a] \wedge \zeta_2[\theta(a)/a_0] \mid \zeta_2 \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\widehat{jp}\}$$

Here, $\theta : (a \uplus \hat{v}) \to \hat{v}$ is an alpha-converter that assigns meta-variables fresh names. We lift θ to policies so that $\theta(pol)$ renames all iteration variables in the policy to fresh names. Meta-variable a_0 is a reserved name used by $\hat{\mathcal{P}}$ to denote the old value of a. Substitution $[\theta(a)/a_0]$ replaces it with a fresh name, and substitution $[\theta(a)/a]$ re-points all old references to a to the same name.

3.5 Abstract Interpretation

Abstract interpretation is implemented by applying the abstract machine to the untrusted, instrumented bytecode until a fixed point is reached. When

$$\frac{\mu, \hat{\rho} \models \rho \qquad \mu, jp \models \widehat{jp}}{\mu, \langle k, \hat{\rho}, \widehat{jp} \rangle \models \langle k, \rho, jp \rangle} (\textsc{Jp-Sound})$$

$$\frac{}{\mu, \langle\rangle \models \langle\rangle} (\textsc{EmpJp-Sound})$$

$$\frac{}{\mu, \cdot \models \cdot} (\textsc{EmpStk-Sound})$$

$$\frac{\mathcal{E}[\![e]\!]\mu = n \qquad \mu, \hat{\rho} \models \rho}{\mu, e::\hat{\rho} \models n::\rho} (\textsc{Stk-Sound})$$

$$\frac{\sigma^{\leftarrow} = \hat{\sigma}^{\leftarrow} \qquad \forall x \in \sigma^{\leftarrow}. \mathcal{E}[\![\hat{\sigma}(x)]\!]\mu = \sigma(x)}{\mu, \hat{\sigma} \models \sigma} (\textsc{Str-Sound})$$

$$\frac{\mu \in \mathcal{C}[\![\zeta]\!] \qquad \mu, \hat{\rho} \models \rho \qquad \mu, \hat{\sigma} \models \sigma}{\langle L : i, \rho, \sigma \rangle \sim \langle L : i, \zeta, \hat{\rho}, \hat{\sigma} \rangle} (\textsc{Sound})$$

Figure 17: Soundness relation $\sim$

multiple different abstract states are inferred for the same code point, the state space is pruned by computing the join of the abstract states. State lattice $(\mathcal{A}, \preceq_{\hat{\chi}})$ is defined in Fig. 16. This reduces the number of control-flows that an implementation of the abstract machine must explore.

4 Soundness

The abstract machine (defined in §3.4) is *sound* with respect to the concrete machine (defined in §3.2) in the sense that each inferred abstract state $\hat{\chi}$ conservatively approximates all concrete states χ that can arise at the same program point during an execution of the concrete machine on the same program. The soundness of state abstractions is formally captured in terms of a *soundness relation* [8] written $\sim \subseteq \mathcal{C} \times \mathcal{A}$, defined in Fig. 17.

Our proof of soundness relies upon a soundness relationship between the concrete and abstract denotational semantics of SPoX policies. This soundness relation is described by the following theorem.

Theorem 1 (SPoX Soundness)**.** *If $\mathcal{P}[\![pol]\!] = (\ldots, \delta)$ and $(\psi, \sigma), \widehat{jp} \models jp$ holds, then $\sigma' \in \delta(\sigma|_a, jp)$ if and only if there exist $\zeta' \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\widehat{jp}$ and $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta']\!]$ such that $\psi''(a_0) = \sigma(a)$ and $\sigma''(a) = \sigma'(a)$.*

Proof. The proof can be decomposed into the following series of lemmas that correspond to each of the SPoX policy syntax forms. Without loss of generality, we assume for simplicity that alpha-conversion θ is the identity function. □

Lemma 1. *If $\mathcal{EP}[\![ep]\!]\psi = (\sigma, \sigma')$ then there exists $(\psi'', \sigma') \in \mathcal{C}[\![\widehat{\mathcal{EP}}[\![ep]\!]]\!]$ such that $\psi'' \sqsubseteq \psi[a_0 = \sigma(a)]$ and $\psi''(a_0) = \sigma(a)$.*

Lemma 2. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *then* $\mathcal{PC}[\![or \ldots]\!]jp = Succ$ *if and only if there exists* $\zeta \in \widehat{\mathcal{PCD}}[\![or \ldots]\!]\widehat{jp}$ *such that* $(\psi'', \bot) \in \mathcal{C}[\![\zeta]\!]$ *and* $\psi'' \sqsubseteq \psi$.

Lemma 3. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *then* $\mathcal{PC}[\![pcc]\!]jp = Succ$ *if and only if* $(\psi'', \bot) = \mathcal{C}[\![\widehat{\mathcal{PCC}}[\![pcc]\!]\widehat{jp}]\!]$ *where* $\psi'' \sqsubseteq \psi$.

Lemma 4. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *and* $f(jp) = Succ$ *then* $(f, \sigma|_a, \sigma') \in \mathcal{ES}[\![edg]\!]\psi$ *if and only if there exists* $\zeta' \in \widehat{\mathcal{ES}}[\![edg]\!]\widehat{jp}$ *such that* $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta']\!]$, $\sigma''(a) = \sigma'(a)$, *and* $\psi''(a) = \sigma(a)$.

Proof. Proofs of Lemmas 1–4 follow from a straightforward expansion of the definitions in Figs. 11 and 7. □

Soundness of the abstract machine with respect to the concrete machine is proved via preservation and progress lemmas for a bisimulation of the abstract and concrete machines. The preservation lemma proves that the bisimulation 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. 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.

Lemma 5 (Progress)**.** *For every* $\chi \in \mathcal{C}$ *and* $\hat{\chi} \in \mathcal{A}$ *such that* $\chi \sim \hat{\chi}$, *if there exists* $\hat{\chi}' = \langle L_{\hat{\chi}'} : i_{\hat{\chi}'}, \zeta', \hat{\rho}', \hat{\sigma}' \rangle \in \mathcal{A}$ *such that* $\hat{\chi} \rightsquigarrow \hat{\chi}'$ *and* $\mathcal{C}[\![\zeta']\!] \neq \emptyset$, *then there exists* $\chi' \in \mathcal{C}$ *such that* $\chi \mapsto \chi'$.

Proof. Let $\chi = \langle L : i, \rho, \sigma \rangle \in \mathcal{C}$, $\hat{\chi} = \langle L : i, \zeta, \hat{\rho}, \hat{\sigma} \rangle \in \mathcal{A}$, and $\hat{\chi}' \in \mathcal{A}$ be given, and assume $\chi \sim \hat{\chi}$ and $\hat{\chi} \rightsquigarrow \hat{\chi}'$ both hold. Proof is by case distinction on the derivation of $\hat{\chi} \rightsquigarrow \hat{\chi}'$.

Case (AIflePos)**:** The rule's premises prove that $\hat{\chi} = \langle L : \textbf{ifle } L', \zeta, e_1{::}e_2{::}\hat{\rho}_r, \hat{\sigma} \rangle$ and $\hat{\chi}' = \langle L' : p(L'), \zeta \wedge (e_2 \leq e_1), \hat{\rho}_r, \hat{\sigma} \rangle$, Relation $\chi \sim \hat{\chi}$ implies that ρ is of the form $x_1{::}x_2{::}\rho_r$. Choose configuration $\chi' = \langle L' : p(L'), \rho_r, \sigma \rangle$. If $x_2 \leq x_1$, then $\chi \mapsto \chi'$ is derivable by Rule (CIflePos). If $x_2 > x_1$, then $\chi \mapsto \chi'$ is derivable by Rule (CIfleNeg).

Case (AIfleNeg)**:** Similar to (AIflePos), omitted.

Case (AGetLocal)**:** The rule's premises prove that $\hat{\chi} = \langle L : \textbf{getlocal } \ell, \zeta, \hat{\rho}, \hat{\sigma} \rangle$ and $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta, \hat{\sigma}(\ell){::}\hat{\rho}, \hat{\sigma} \rangle$. Relation $\chi \sim \hat{\chi}$ implies that $\ell \in \sigma^{\leftarrow}$. Choosing configuration $\chi' = \langle s(L) : p(s(L)), \sigma(\ell){::}\rho, \sigma \rangle$ allows $\chi \mapsto \chi'$ to be derived by Rule (CGetLocal).

Case (ASetLocal)**:** The rule's premises prove that $\hat{\chi} = \langle L : \textbf{setlocal } \ell, \zeta, e{::}\hat{\rho}, \hat{\sigma} \rangle$ and $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta[\hat{v}/\ell], \hat{\rho}[\hat{v}/\ell], \hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]] \rangle$, where $\hat{v}$ is fresh. Relation $\chi \sim \hat{\chi}$ implies that ρ has the form $x{::}\rho_r$. Choosing configuration $\chi' = \langle s(L) : p(s(L)), \rho_r, \sigma[\ell := x] \rangle$ allows $\chi \mapsto \chi'$ to be derived by Rule (CSetLocal).

Case (AJmp)**:** Trivial, omitted.

Case (AEvent)**:** The rule's premises prove that abstract configuration $\hat{\chi} = \langle L : \mathbf{event}_y\ n,\ \zeta_1,\ e_1{::}e_2{::}\cdots{::}e_y{::}\hat{\rho}_r,\ \hat{\sigma}\rangle$ and $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta', \hat{\rho}_r, \hat{\sigma}\rangle$, where $\zeta' = \zeta_1[\theta(a)/a] \wedge \zeta_2[\theta(a)/a_0]$ with $\zeta_2 \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\widehat{jp}$, and $\widehat{jp} = \langle \mathbf{event}_y\ n,\ e_1{::}e_2{::}\cdots{::}e_y{::}\cdot, \langle\rangle\rangle$.

To derive $\chi \mapsto \chi'$ using Rule (CEvent), one must prove that there exists $\sigma' \in \delta(\sigma|_a, jp)$ where $jp = \langle \mathbf{event}_y\ n,\ x_1{::}x_2{::}\cdots{::}x_y{::}\cdot, \langle\rangle\rangle$. Once this is established, we may choose configuration $\chi' = \langle s(L) : p(s(L)), \rho_r, \sigma[a := \sigma'(a)]\rangle$ to derive $\chi \mapsto \chi'$ by Rule (CEvent).

We will prove $\sigma' \in \delta(\sigma|_a, jp)$ using Theorem 1. The premises of the derivation of $\chi \sim \hat{\chi}$ suffice to derive $\mu, \widehat{jp} \models jp$ by Rule (Jp-Sound). Denotation $\mathcal{C}[\![\zeta']\!]$ is non-empty by assumption; therefore we may choose $(\psi_0, \sigma_0) \in \mathcal{C}[\![\zeta']\!]$ and define $\psi'' = \psi_0[a_0 := \sigma(a)]$ and $\sigma'' = \sigma_0$. Observe that the definition of ζ' in terms of ζ_2 proves that $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta_2]\!]$. Furthermore, since σ' is heretofore unconstrained, we may define $\sigma'(a) = \sigma''(a)$. Theorem 1 therefore proves that $\sigma' \in \delta(\sigma|_a, jp)$. □

The following substitution lemma aids in the proof of the Preservation Lemma that follows it.

Lemma 6. *For any expression e_0, mappings (ψ, σ), variables $\ell \in \sigma^{\leftarrow}$ and $\hat{v} \notin \sigma^{\leftarrow}$, and value x, $\mathcal{E}[\![e_0]\!](\psi, \sigma) = \mathcal{E}[\![e_0[\hat{v}/\ell]]\!](\psi[\hat{v} := \sigma(\ell)], \sigma[\ell := x])$.*

Proof. Proof is by a straightforward induction over the structure of e_0, and is therefore omitted. □

Lemma 7 (Preservation)**.** *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim \hat{\chi}$, for every $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$ there exists $\hat{\chi}' \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ and $\chi' \sim \hat{\chi}'$.*

Proof. Let $\chi = \langle L : i, \rho, \sigma\rangle \in \mathcal{C}$, $\hat{\chi} \in \mathcal{A}$, and $\chi' \in \mathcal{C}$ be given such that $\chi \mapsto \chi'$. Proof is by case distinction over the derivation of $\chi \mapsto \chi'$.

Case (CIflePos)**:** Rule (CIflePos) implies that $i = \mathbf{ifle}\ L'$, stack ρ has the form $x_1{::}x_2{::}\rho_r$, and $\chi' = \langle s(L) : p(s(L)), \rho_r, \sigma\rangle$. Relation $\chi \sim \hat{\chi}$ proves that $\hat{\chi}$ has the form $\langle L : \mathbf{ifle}\ L', \zeta,\ e_1{::}e_2{::}\hat{\rho}_r,\ \hat{\sigma}\rangle$. Choose $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta \wedge (e_2 \leq e_1), \hat{\rho}_r, \hat{\sigma}\rangle$ and observe that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ is derivable by Rule (AIflePos).

Relation $\chi \sim \hat{\chi}$ implies (1) $\mu \in \mathcal{C}[\![\zeta]\!]$, (2) $\mu, \hat{\rho} \models \rho$, and (3) $\mu, \hat{\sigma} \models \sigma$. Proving $\chi' \sim \hat{\chi}'$ requires deriving the three premises of Rule (Sound):

(A) To derive $\mu \in \mathcal{C}[\![\zeta \wedge (e_2 \leq e_1)]\!]$, observe that $\mathcal{C}[\![\zeta \wedge (e_2 \leq e_1)]\!] = \mathcal{C}[\![\zeta]\!] \cap \mathcal{T}[\![e_2 \leq e_1]\!]$. It follows from (1) above that $\mu \in \mathcal{C}[\![\zeta]\!]$. By definition, $\mathcal{T}[\![e_2 \leq e_1]\!] = \{\mu' \in \Psi \times \Sigma \mid \mathcal{E}[\![e_2]\!]\mu' \leq \mathcal{E}[\![e_1]\!]\mu'\}$. Rule (Str-Sound) proves that $\mathcal{E}[\![\hat{\sigma}(n)]\!]\mu = \sigma(n)\ \forall n \in \{1, 2\}$. Thus, $\mathcal{E}[\![e_1]\!]\mu = x_1$ and $\mathcal{E}[\![e_2]\!] = x_2$. Since $x_2 \leq x_1$ (from Rule (CIflePos)), this implies $\mathcal{E}[\![e_2]\!]\mu \leq \mathcal{E}[\![e_1]\!]\mu$. From the definition of $\mathcal{T}$, it follows that $\mu \in \mathcal{T}[\![e_2 \leq e_1]\!]$.

(B) $\mu, \hat{\rho}_r \models \rho_r$ follows directly from (2) above and Rule (Stk-Sound).

(C) $\mu, \hat{\sigma} \models \sigma$ follows directly from (3) above.

Case (CIfleNeg)**:** Similar to (CIflePos), omitted.

Case (CGetLocal)**:** Rule (GetLocal) proves that $i = \textbf{getlocal}\ \ell$, and $\chi' = \langle s(L) : p(s(L)), \sigma(\ell)::\rho, \sigma\rangle$. Relation $\chi \sim \hat{\chi}$ proves that $\hat{\chi}$ has the form $\langle L : \textbf{getlocal}\ \ell, \zeta, \hat{\sigma}(\ell)::\hat{\rho}, \hat{\sigma}\rangle$. Choose $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta, \hat{\rho}, \hat{\sigma}\rangle$, and observe that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ is derivable by Rule (AGetLocal).

Relation $\chi \sim \hat{\chi}$ implies (1) $\mu \in \mathcal{C}[\![\zeta]\!]$, (2) $\mu, \hat{\rho} \models \rho$, and (3) $\mu, \hat{\sigma} \models \sigma$. Proving $\chi' \sim \hat{\chi}'$ requires deriving the three premises of Rule (Sound):

(A) $\mu \in \mathcal{C}[\![\zeta]\!]$ follows directly from (1) above.

(B) $\mu, \hat{\sigma}(n)::\hat{\rho} \models \sigma(\ell)::\rho$ can be derived with Rule (Stk-Sound) by combining $\mathcal{E}[\![\hat{\sigma}(\ell)]\!]\mu = \sigma(\ell)$ (from Rule (Str-Sound)) and (2) above.

(C) $\mu, \hat{\sigma} \models \sigma$ follows directly from (3) above.

Case (CSetLocal)**:** Rule (SetLocal) proves that $i = \textbf{setlocal}\ \ell$, that ρ has the form $x::\rho_r$, and that $\chi' = \langle s(L) : p(s(L)), \rho_r, \sigma[\ell := x]\rangle$. Relation $\chi \sim \hat{\chi}$ implies that $\hat{\chi}$ has the form $\langle L : \textbf{setlocal}\ \ell, \zeta, e::\hat{\rho}_r, \hat{\sigma}\rangle$.

Choose $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta[\hat{v}/\ell], \hat{\rho}_r[\hat{v}/\ell], \hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]]\rangle$ where $\hat{v}$ is a fresh meta-variable, and observe that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ is derivable by Rule (ASetLocal).

Relation $\chi \sim \hat{\chi}$ implies (1) $\mu \in \mathcal{C}[\![\zeta]\!]$, (2) $\mu, e::\hat{\rho}_r \models x::\rho_r$, and (3) $\mu, \hat{\sigma} \models \sigma$. Proving $\chi' \sim \hat{\chi}'$ requires deriving the three premises of Rule (Sound), where $\mu' = (\psi[\hat{v} := \sigma(\ell)], \sigma[\ell := x])$:

(A) By a trivial induction over the structure of ζ, if $\mu = (\psi, \sigma) \in \mathcal{C}[\![\zeta]\!]$ and $\hat{v}$ does not appear in ζ, then $\mu' = (\psi[\hat{v} := \sigma(\ell)], \sigma[\ell := x]) \in \mathcal{C}[\![\zeta[\hat{v}/\ell]]\!]$.

(B) By Rule (Stk-Sound), the derivation of (2) contains a sub-derivation of $\mu, \hat{\rho}_r \models \rho_r$. A trivial induction over $\hat{\rho}_r$ therefore proves that $\mu', \hat{\rho}_r[\hat{v}/\ell] \models \rho_r$.

(C) Deriving $\mu', \hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]] \models \sigma[\ell := x]$ requires deriving the two premises of Rule (Str-Sound):

(C1) To prove $\sigma[\ell := x]^{\leftarrow} = \hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]]^{\leftarrow}$, observe that $\sigma[\ell := x]^{\leftarrow} = \sigma^{\leftarrow} \cup \{\ell\}$ and $\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]]^{\leftarrow} = \hat{\sigma}^{\leftarrow} \cup \{\ell\}$. From (3) above and Rule (Str-Sound), it follows that $\sigma^{\leftarrow} = \hat{\sigma}^{\leftarrow}$; therefore $\sigma^{\leftarrow} \cup \{\ell\} = \hat{\sigma}^{\leftarrow} \cup \{\ell\}$.

(C2) To prove $\forall y \in \sigma[\ell := x]^{\leftarrow}\,.\,\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell](y)]\!]\mu' = \sigma[\ell := x](y)$, let $y \in \sigma^{\leftarrow} \cup \{\ell\}$ be given:

- If $y = \ell$ then $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell](\ell)]\!] = \mathcal{E}[\![e[\hat{v}/\ell]]\!]$. Applying Lemma 6 with $e_0 = e$ yields $\mathcal{E}[\![e[\hat{v}/\ell]]\!]\mu' = \mathcal{E}[\![e]\!]\mu$. By (2) above, and Rule (Stk-Sound), $\mathcal{E}[\![e]\!]\mu = x = \sigma[\ell := x](\ell)$.

- If $y \neq \ell$ then $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]](y)]\!] = \mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell](y)]\!]$. Applying Lemma 6 with $e_0 = \hat{\sigma}(y)$ yields $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell](y)]\!]\mu' = \mathcal{E}[\![\hat{\sigma}(y)]\!]\mu$. By Rule (Str-Sound) and (3) above, $\mathcal{E}[\![\hat{\sigma}(y)]\!]\mu = \sigma(y) = \sigma[\ell := x](y)$.

Case (CJmp)**:** Trivial, omitted.

Case (CEvent)**:** Rule (CEvent) proves that $i = \mathbf{event}_y\ n$, that ρ has the form $x_1{::}x_2{::}\cdots{::}x_y{::}\rho_r$, and that $\chi' = \langle s(L) : p(s(L)), \rho_r, \sigma[a := \sigma'(a)]\rangle$, where $\sigma' \in \delta(\sigma|_a, \langle \mathbf{event}_y\ n,\ x_1{::}x_2{::}\cdots{::}x_y{::}\cdot, \langle\rangle\rangle)$.

Relation $\chi \sim \hat{\chi}$ implies that $\hat{\chi} = \langle L : \mathbf{event}_y\ n,\ \zeta_1,\ e_1{::}e_2{::}\cdots{::}e_y{::}\hat{\rho}_r,\ \hat{\sigma}\rangle$ and that for some $\mu = (\psi, \sigma)$: (1) $\mu \in \mathcal{C}[\![\zeta_1]\!]$, (2) $\mu, e_1{::}e_2{::}\cdots{::}e_y{::}\hat{\rho}_r \models x_1{::}x_2{::}\cdots{::}x_y{::}\rho_r$, and (3) $\mu, \hat{\sigma} \models \sigma$. Theorem 1 therefore implies that there exists $\zeta' \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\widehat{jp}$ and $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta']\!]$ such that (4) $\sigma''(a) = \sigma'(a)$ and (5) $\psi''(a_o) = \sigma(a)$.

Choose $\hat{\chi}' = \langle s(L) : p(s(L)), \zeta_1[\theta(a)/a] \wedge \zeta'[\theta(a)/a_0], \hat{\rho}_r, \hat{\sigma}\rangle$ and observe that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ is derivable by Rule (AEvent). Deriving $\chi' \sim \hat{\chi}'$ requires deriving the three premises of Rule (Sound), where $\mu' = (\psi \uplus \psi''[\theta(a)/a_0], \sigma[a := \sigma'(a)])$:

(A) $\mu' \in \mathcal{C}[\![\zeta_1[\theta(a)/a] \wedge \zeta'[\theta(a)/a_0]]\!]$ is provable in two steps:
- $\mu' \in \mathcal{C}[\![\zeta_1[\theta(a)/a]]\!]$ follows from (1) and (5) above.
- $\mu' \in \mathcal{C}[\![\zeta'[\theta(a)/a_0]]\!]$ follows from (4) above.

(B) $\mu', \hat{\rho}_r \models \rho_r$ is derivable by an induction on the height of stack $\hat{\rho}_r$ (which is equal to the height of stack ρ_r by (2) above). The base case of the induction follows trivially from Rule (EmpStk-Sound). The inductive case is derivable from Rule (Stk-Sound) provided that $\mathcal{E}[\![e]\!](\psi \uplus \psi''[\theta(a)/a_0], \sigma[a := \sigma'(a)]) = \mathcal{E}[\![e]\!](\psi, \sigma)$. To prove this, observe that e mentions neither a_0 (because by the definition of $\hat{\mathcal{P}}$, a_0 is a reserved meta-variable name that is not available to programs) nor a (because the abstract state is not directly readable by programs, and therefore cannot leak to the stack). A formal proof of both follows from an inspection of the rules in Fig. 14.

(C) $\mu', \hat{\sigma} \models \sigma[a := \sigma'(a)]$ is derivable by Rule (Str-Sound) by deriving its two premises:
- $\sigma[a := \sigma'(a)]^{\leftarrow} = \hat{\sigma}^{\leftarrow}$ follows trivially from $a \in \sigma^{\leftarrow}$.
- $\forall x \in \sigma[a := \sigma'(a)]^{\leftarrow}\,.\,\mathcal{E}[\![\hat{\sigma}(x)]\!]\mu' = \sigma[a := \sigma'(a)](x)$ follows from (3) above, whose derivation includes a derivation of premise $\forall x \in \sigma^{\leftarrow}\,.\,\mathcal{E}[\![\hat{\sigma}(x)]\!]\mu = \sigma(x)$. □

Theorem 2 (Soundness)**.** *Every program accepted by the abstract machine does not commit a policy violation when executed.*

Proof. By definition of abstract machine acceptance, starting from initial state $\hat{\chi}_0$ the abstract machine continually makes progress. By a trivial induction over

Program	Policy	File Sizes (KB) old/new/libs	No. Classes old/libs	Rewrite Time (s)	# Evts	Total Verif. Time (s)	Model-check Time (s)
EJE	NoExecSaves	434/ 439/ 0	147/ 0	6.1	1	202.8	16.3
RText		1337/1266/ 835	448/ 680	52.1	7	2797.5	54.5
JSesh		2054/1924/20878	863/ 1849	57.8	1	5488.1	196.0
vrenamer	NoExecRename	1048/ 927/ 0	583/ 0	50.1	9	1956.8	41.0
jconsole	NoUnsafeDel	34/ 36/ 0	33/ 0	0.6	2	115.7	15.1
jWeatherWatch	NoSendsAfterReads	390/ 294/ 0	186/ 0	12.3	46	308.2	156.7
YouTubeDownloader		402/ 281/ 0	148/ 0	17.8	20	219.0	53.6
jfilecrypt	NoGui	343/ 303/ 0	164/ 0	9.7	1	642.2	2.8
jknightcommander	OnlySSH	158/ 166/ 4753	146/ 2675	4.5	1	650.1	3.0
Multivalent	EncrpytPDF	1108/1116/ 0	559/ 0	129.9	7	3567.0	26.9
tn5250j	PortRestrict	623/ 646/ 0	416/ 0	85.4	2	2598.2	23.6
jrdesktop	SafePort	382/ 343/ 0	163/ 0	8.3	5	483.0	17.8
JVMail	TenMails	29/ 25/ 0	21/ 0	1.6	2	35.1	8.0
JackMail		214/ 166/ 369	30/ 269	2.5	1	626.7	8.9
Jeti	CapLoginAttempts	545/ 484/ 0	422/ 0	15.3	1	524.3	8.8
ChangeDB	CapMembers	89/ 83/ 404	63/ 286	4.3	2	995.3	12.0
projtimer	CapFileCreations	36/ 34/ 0	25/ 0	15.3	1	56.2	6.1
xnap	NoFreeRide	1321/1251/ 0	878/ 0	24.8	4	1496.2	56.4
Phex		4861/4586/ 3799	1353/ 830	69.4	2	5947.0	172.7
Webgoat	NoSqlXss	500/ 431/ 6338	159/ 3579	16.7	2	10876.0	120.0
OpenMRS	NoSQLInject	2075/1783/24279	932/ 17185	78.7	6	2897.0	37.3
SQuirreL	SafeSQL	1962/1789/ 1003	1328/ 626	140.2	1	3352.1	37.3
JVMail	LogEncrypt	29/ 26/ 0	22/ 0	1.8	6	71.3	43.2
jvs-vfs	CheckDeletion	310/ 277/ 0	127/ 0	4.4	2	193.9	6.3
sshwebproxy	EncryptPayload	43/ 37/ 389	19/ 16	1.1	5	66.7	7.0

Table 1: Experimental Results

the set of finite prefixes of this abstract transition chain, the progress and preservation lemmas prove that the concrete machine also continually makes progress from initial state χ_0. Every security-relevant event in this concrete transition chain therefore satisfies Rule (CEvent) of Fig. 9, whose premise guarantees that the event does not violate the policy. □

5 Implementation

Our prototype verifier implementation consists of 5200 lines of Prolog code and 9100 lines of Java code. The Prolog code runs under 32-bit SWI-Prolog 5.10.4, which communicates with our Java libraries using the JPL interface. The Java side handles the parsing of SPoX policies and input Java binaries, and compares Java 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 contains no potential pointcut matches or possible writes to reified security state.

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

Figure 18: `NoExecSaves` policy

We have used our prototype implementation to rewrite and subsequently verify numerous Java applications. These case-studies are discussed throughout the remainder of the section, and 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.

In Table 1, each cell in the File Sizes column has three parts: the original size of the main program before rewriting, the size after rewriting, and the size of included system libraries that needed to be verified (but not rewritten) as part of verifying the rewritten code. Verification of system library code is required to verify the safety of various control-flows that pass through them. Likewise, each cell in the No. Classes column has two parts: the number of classes in the main program and the number of classes in the libraries.

Rewriting actually reduced the size of many programs, even though code was added and no code was removed. This is because SPoX removes unnecessary metadata from Java class files during parsing and code-generation.

This section partitions our case-studies into eight policy classes. SPoX code is provided for each class in a general form representative of the various instantiations of the policy that we used for specific applications. The instantiations replace the simple pointcut expressions in each figure with more complex, application-specific pointcuts that are here omitted for space reasons.

Filename guards. Figure 18 shows a generalized SPoX policy that prevents file-creation operations from specifying a file name with an executable extension. This could be used to prevent malware propagation.

The regular expression in the `streq` predicate on Line 4 matches any string that ends in "`.exe`", "`.bat`", or a number of other disallowed file extensions. There is a very large number of file extensions that are considered to be executable on Windows. For our implementation, we included every extension listed at [14].

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

```
(state name="s")
(edge name="FileRead"
  (nodes "s" 0,1)
  (and (call "java.io.File.*")
       (argval 1 (streq "[A-Za-z]*:\\windows\\.*"))))
(edge name="NetworkSend"
  (nodes "s" 1,#)
  (call "java.net.Socket.getOutputStream"))
```

Figure 19: `NoSendsAfterReads` policy

of protected directories.

We enforced a similar policy on `vrenamer`, a mass file-renaming application, prohibiting renaming of files to names with executable extensions. Our initial attempt to verify `vrenamer` failed. Analysis of the failure revealed that the original application implements a global exception handler that can potentially hijack control-flows within certain kinds of SPoX-generated IRM guard code if the guard code throws an exception. This could allow the abstract and reified security state to become desynchronized, leading to policy violations.

The verification failure is therefore attributable to a security flaw in the SPoX rewriter. The flaw could be fixed by in-lining inner exception handlers for guard code to protect them from interception by a pre-existing outer handler. In order to verify the application for this instance, we manually edited the rewritten bytecode to exclude the guard code from the scope of the outer exception handler. This resulted in successful verification.

Event ordering. Figure 19 encodes a canonical information flow policy in the IRM literature that prohibits all network-send operations after a sensitive file has been read. Specifically, this policy prevents calls to `Socket.getOutputStream` after any `java.io.File` method call whose first parameter accesses the `windows` directory.

We enforced this policy on `jWeatherWatch`, a weather widget application, and `YouTube Downloader`, 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 security automaton for this policy is similar to the one in the figure.

Pop-up protection. The `NoGui` policy in Fig. 20 prevents applications from opening windows on the user's desktop. We enforced the `NoGui` policy on `jfilecrypt`, a file encrypt/decrypt application. Similar policies can be used

```
(state name="s")
(edge name="no_gui"
  (nodes s 0,#)
    (and (call "jfilecrypt.GuiMainController.new")
         (withincode "jfilecrypt.Application.main")))
```

Figure 20: `NoGui` policy

```
(state name="s")
(edge name="badPort"
  (nodes "s" 0,#)
  (and (set "Config.port")
       (or (argval 1 (intgt 29))
           (argval 1 (intlt 20)))))
```

Figure 21: `SafePort` policy

to prohibit access to other system API methods and place constraints upon their arguments.

Port restriction. Policies such as the one in Fig. 21 limit which remote network ports an application may access. This particular policy, which we enforced on the Telnet client `tn5250j`, restricts the port to the range from 20 to 29, inclusive. Attempting to open a connection on any port outside that range causes a policy violation.

We also enforced a similar policy on `jrdesktop`, a remote desktop client, prohibiting the use of ports less than 1000. For `jknightcommander`, an FTP-capable file manager currently in the pre-alpha release stage, we enforced a policy that prohibits access to any port other than 22, restricting its network access to SFTP ports.

Resource bounds. In §2.1, we described a policy which prohibits an email client from sending more than 10 emails in a given execution, as seen in Fig. 3. We enforced this policy on the email clients `JVMail` and `JackMail`.

We enforced similar resource bound policies on various other programs. For `Jeti`, a Jabber instant messaging client, we limited the number of login attempts to 5 in order to deter brute-force attempts to guess a password for another user's account. For `ChangeDB`, a simple database system, we limited the number of member additions to 10. For `projtimer`, a time management system, we limited the number of automatic file save operations to 5, preventing the application from exhausting the user's file quota.

No freeriding. Figure 22 specifies a more complex counting policy that prohibits freeriding in the file-sharing clients `xnap` and `Phex`. State variable `s` tracks the difference between the number of downloads and the number of uploads that

```
(state name="s")
(forall "i" from -10000 to 1
  (edge name="download"
    (nodes "s" i,i+1)
    (call "Download.download")))
(forall "i" from -9999 to 2
  (edge name="upload"
    (nodes "s" i,i-1)
    (call "Upload.upload")))
(edge name="too_many_downloads"
  (nodes "s" 2,#)
  (call "Download.download"))
```

Figure 22: `NoFreeRide` policy

```
(state name="s")
(edge name="SQL_Injection_occurred"
  (nodes "s" 0,#)
  (and (call "Login.login")
       (not (argval 1 (streq "[a-zA-Z0-9]*")))))
(edge name="XSS_injection_occurred"
  (nodes "s" 0,#)
  (and (call "Employee.new")
       (not (and (argval 2 (streq "[A-Za-z_0-9,.\-\s]*"))
                 (argval 3 (streq "[A-Za-z_0-9,.\-\s]*"))
                 ...
                 (argval 16 (streq "[A-Za-z_0-9,.\-\s]*"))))))
```

Figure 23: `NoSqlXss` policy

the application has completed. That is, downloads increment `s`, while uploads decrement it. If the number of downloads exceeds 2 greater than the number of uploads, a policy violation occurs. This forces the software to share as much as it receives.

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 given in Fig. 23. This figure is a simplified form of a policy that we enforced on `Webgoat`, an educational web application that is designed to be vulnerable to such attacks. The policy uses whitelisting to exclude all input characters except for those listed by the regular expressions (alphabetical, numeric, etc.).

The `XSS_injection_occurred` edge starting on Line 6 includes a large number of dynamic `argval` pointcuts—12 in the actual policy. 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 in `OpenMRS`, a web-based medical database system. Injection was prevented for the patient search feature. The library portion of this application is extremely large but contains no security-relevant events. Thus, the separate, non-stateful verification approach for unmarked code regions was critical for avoiding state-space explosions in this case.

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. Specifically, the policy identified all SQL commands matching the regular expression

```
.*(drop|alter|rename).*(table|database).*
```

as policy violations.

Ensuring advice execution. Most other aspectual policy languages, such as Java-MOP [6], allow explicit prescription of advice code that implements IRM guards and interventions. Such advice is typically intended to enforce some higher-level security property, though it can be difficult to prove that it enforces the actual policy that was intended. SPoX excludes trusted advice for this reason; however, untrusted advice that is not part of the policy specification may nevertheless be in-lined by rewriters to enforce SPoX policies. Chekov✓ can then be applied to verify that this advice actually executes under policy-prescribed circumstances to enforce the policy. This promotes separation of concerns, reducing the TCB so that it does not include the advice.

We simulated the use of advice by manually inserting calls to specific encrypt and log methods prior to email-send events in `JVMail`. Our intent was for each email to be encrypted, then logged, then sent, and in that order. A simplified SPoX specification for the policy is given in Fig. 24. After inserting the advice, we applied the ordering policy using the rewriter, and then 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`), which prohibits deletion of files without first executing advice code that consults the user.

Finally, we enforced a mandatory encryption policy for the `sshwebproxy` application, which allows users to use a web browser to access SSH sessions and perform secure file transfers. To prevent the application from sending plaintext message payloads to the remote host, our `EncryptPayload` policy requires the proxy to encrypt each message payload before sending.

6 Related Work

IRMs were first formalized in the PoET/PSLang/SASI systems [13, 30, 12], which implement IRMs for Java bytecode and GNU assembly code. Subsequently, there has been a growing history of effective IRM systems for many

```
(state name="logged")
(state name="encrypted")
(forall "i" from 0 to 1
  (edge name="encrypt"
    (nodes "encrypted" 0,1)
    (nodes "logged" 0,0)
    (call "Logger.encrypt"))
  (edge name="badOrderEncryptSecond"
    (nodes "encrypted" 0,#)
    (nodes "logged" 1,#)
    (call "Logger.encrypt"))
  (edge name="transaction"
    (nodes "encrypted" 1,0)
    (call "SMTPConnection.sendMail"))
  (edge name="badEncrypt"
    (nodes "encrypted" 1,#)
    (nodes "logged" i,i)
    (call "Logger.encrypt"))
  (edge name="bad_transaction1"
    (nodes "encrypted" 0,#)
    (call "SMTPConnection.sendMail"))
  (edge name="log"
    (nodes "logged" 0,1)
    (nodes "encrypted" 1,1)
    (call "Logger.log"))
  (edge name="badOrderLogFirst"
    (nodes "logged" 0,#)
    (nodes "encrypted" 0,#)
    (call "Logger.log"))
  (edge name="bad_log"
    (nodes "logged" 1,#)
    (nodes "encrypted" i,i)
    (call "Logger.log"))
  (edge name="bad_transaction2"
    (nodes "logged" 0,#)
    (call "SMTPConnection.sendMail")))
```

Figure 24: `LogEncrypt` policy

architectures (cf., [23, 6]). Almost all 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 [30, 24] 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.

Separate certification of IRMs was first implemented by the Mobile system [17], which 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 to execute. The type-based IRM certification approach is efficient and elegant but does not yet support AOP-style IRMs, whose pointcuts typically specify security properties that are not expressible in Mobile's type system.

ConSpec [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; however, the guard code itself remains a trusted part of the policy specification.

Our prior work [33] is the first to adopt a model-checking approach to verify such IRMs without trusted guard code. It presents a prototype IRM certifier for ActionScript bytecode that supports reified security state but not dynamic pointcuts or after-advice. These are non-trivial additions, as discussed in §2.

This paper extends that prior work to support SPoX [15], a purely declarative AOP IRM system for Java bytecode. SPoX policies are advice-free; any advice implemented as part of the IRM remains untrusted and must therefore undergo verification. Policy specifications consist of pointcuts and declarative specifications of how events that match the pointcuts affect the security state. Thus, SPoX policies specify program properties, not rewriting recipes. Past work has used SPoX to specify and enforce a variety of real-world security policies [19, 20].

Related research on general model-checking is vast, but mostly focuses on detecting deadlock and assertion violation properties of source code. For example, Java PathFinder (JPF) [21] and Moonwalker [29] verify properties of Java and .NET source programs, respectively. Other model-checking research [28, 4] has targeted abstract languages, such as the π-calculus [26] or Calculus of Communicating Systems (CCS). Model-checking of binary code is useful in situations where the code consumer may not have access to source code. For example, CodeSurfer/x86 and WPDS++ have been used to extract and check models for x86 binary programs [3].

7 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 Chekov✓—the first automated, model-checking-based certifier for an aspect-oriented, real-world IRM system [15]. Chekov✓ 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 support for 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 Chekov✓ 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 explore methods of leveraging more powerful rewriter-supplied hints that express richer invariants relating abstract and reified security state. Such advances will provide greater flexibility for alternative IRM implementations of stateful policies.

References

[1] Irem Aktug and Katsiaryna Naliuka. ConSpec - a formal language for policy specification. *Science of Computer Programming*, 74:2–12, 2008.

[2] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1986.

[3] Gogul Balakrishnan, Thomas W. Reps, Nicholas Kidd, Akash Lal, Junghee Lim, David Melski, Radu Gruian, Suan Hsi Yong, Chi-Hua Chen, and Tim Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Proc. International Conference on Computer-Aided Verification (CAV)*, pages 158–163, 2005.

[4] Samik Basu and Scott A. Smolka. Model checking the Java metalocking algorithm. *ACM Trans. Software Engineering and Methodology*, 16(3), 2007.

[5] Eric Bodden and Klaus Havelund. Racer: Effective race detection using AspectJ. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–166, 2008.

[6] Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, 2005.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages*, pages 234–252, 1977.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[9] Daniel S. Dantas and David Walker. Harmless advice. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 383–396, 2006.

[10] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Trans. Programming Languages and Systems*, 30(3), 2008.

[11] Brian W. DeVries, Gopal Gupta, Kevin W. Hamlen, Scott Moore, and Meera Sridhar. ActionScript bytecode verification with co-logic programming. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 9–15, 2009.

[12] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, 2004.

[13] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop (NSPW)*, pages 87–95, 1999.

[14] FileInfo.com. Executable file types, 2011. `www.fileinfo.com/filetypes/executable`.

[15] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 11–20, 2008.

[16] Kevin W. Hamlen, Vishwath Mohan, Mohammad M. Masud, Latifur Khan, and Bhavani Thuraisingham. Exploiting an antivirus interface. *Computer Standards & Interfaces Journal*, 31(6):182–1189, 2009.

[17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 7–16, 2006.

[18] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Programming Languages and Systems*, 28(1):175–205, 2006.

[19] Micah Jones and Kevin W. Hamlen. Enforcing IRM security policies: Two case studies. In *Proc. IEEE Intelligence and Security Informatics (ISI) Conference*, pages 214–216, 2009.

[20] Micah Jones and Kevin W. Hamlen. Disambiguating aspect-oriented policies. In *Proc. International Conference on Aspect-Oriented Software Development (AOSD)*, pages 193–204, 2010.

[21] W. Kisser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), 2003.

[22] Zhou Li and XiaoFeng Wang. FIRM: Capability-based inline mediation of Flash behaviors. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 181–190, 2010.

[23] Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, Princeton, New Jersey, 2006.

[24] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005.

[25] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Information and Systems Security*, 12(3), 2009.

[26] Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[27] George C. Necula. Proof-Carrying Code. In *Proc. ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[28] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Terrance Swift, and David S. Warren. Efficient model checking using tabled resolution. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 143–154, 1997.

[29] Theo C. Ruys and Niels H. M. Aan de Brugh. MMC: The Mono model checker. *Electronic Notes in Theoretical Computer Science*, 190(1):149–160, 2007.

[30] Fred B. Schneider. Enforceable security policies. *ACM Trans. Information and Systems Security*, 3(1):30–50, 2000.

[31] Viren Shah and Frank Hill. An aspect-oriented security framework. In *Proc. DARPA Information Survivability Conference and Exposition*, volume 2, 2003.

[32] Meera Sridhar and Kevin W. Hamlen. Actionscript in-lined reference monitoring in prolog. In *Proc. International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 149–151, 2010.

[33] Meera Sridhar and Kevin W. Hamlen. Model-checking in-lined reference monitors. In *Proc. Verification, Model-Checking and Abstract Interpretation (VMCAI)*, pages 312–327, 2010.

[34] Meera Sridhar and Kevin W. Hamlen. Flexible in-lined reference monitor certification: Challenges and future directions. In *Proc. ACM Workshop on Programming Languages meets Program Verification (PLPV)*, pages 55–60, 2011.

[35] The AspectJ Team. The AspectJ programming guide. `www.eclipse.org/aspectj/doc/released/progguide/index.html`, 2003.

[36] John Viega, J.T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2), 2001.

[37] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Programming Languages and Systems*, 26(5):890–910, 2004.