Model-Checking In-Lined Reference Monitors*

Meera Sridhar and Kevin W. Hamlen

The University of Texas at Dallas
800 W. Campbell Rd., Richardson, TX 75080, USA
meera.sridhar@student.utdallas.edu,hamlen@utdallas.edu
http://www.cs.utdallas.edu

Abstract. A technique for elegantly expressing In-lined Reference Monitor (IRM) certification as model-checking is presented and implemented. In-lined Reference Monitors (IRM's) enforce software security policies by in-lining dynamic security guards into untrusted binary code. Certifying IRM systems provide strong formal guarantees for such systems by verifying that the instrumented code produced by the IRM system satisfies the original policy. Expressing this certification step as model-checking allows well-established model-checking technologies to be applied to this often difficult certification task. The technique is demonstrated through the enforcement and certification of a URL anti-redirection policy for ActionScript web applets.

1 Introduction

In-Lined Reference Monitors (IRM's) [17] enforce safety policies by injecting runtime security guards directly into untrusted binaries. The guards test whether an impending operation constitutes a policy violation. If so, corrective action is taken to prevent the violation, such as premature termination. The result is *self-monitoring* code that can be safely executed without external monitoring.

IRM's dynamically observe security-relevant events exhibited by the untrusted code they monitor and maintain persistent internal state between these observations, enabling them to accept or reject based on the *history* of events observed. This allows them to enforce powerful security policies, such as safety policies, that are not precisely enforceable by any purely static analysis [14]. Additionally, IRM's afford code consumers the flexibility of specifying or modifying the security policy after receiving the code, whereas purely static analyses typically require the security policy to be known by the code producer.

Certifying IRM systems [1, 13] verify that IRM's generated by a binary *rewriter* are policy-adherent. Since the binary rewriters that in-line security guards into untrusted code can be large and complex, a separate verifier is useful for shifting this complexity out of the trusted computing base. Since the verifier does not perform any code generation, it is typically smaller and less subject to change than a rewriter, and therefore constitutes a more acceptable trusted

* This research was supported by AFOSR YIP award number FA9550-08-1-0044.

component. Past work has implemented IRM certifiers using type-checking [13] and contracts [1].

Model-checking is an extremely powerful software verification paradigm that is useful for verifying properties that are more complex than those typically expressible by type-systems and more semantically flexible and abstract than those typically encoded by contracts. Yet to our knowledge, model-checking has not yet been applied to verify IRM's. In this paper we describe and implement a technique for doing so. The work's main contributions are as follows:

- We present the design and implementation of a prototype IRM model-checking framework for ActionScript bytecode.
- Our design centers around a novel approach for constructing a *state abstraction lattice* from a *security automaton* [2], for precise yet tractable abstract interpretation of IRM code.
- Rigorous proofs of soundness and convergence are formulated for our system using Cousot's abstract interpretation framework [6].
- The feasibility of our technique is demonstrated by enforcing a URL anti-redirection policy for ActionScript bytecode programs.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 gives an overview of our IRM framework, including an operational semantics and the abstract interpretation algorithm. Section 4 provides a formal soundness proof for our algorithm and a proof of fixed point convergence for the abstract machine. Section 5 discusses the details of our implementation of the system for ActionScript bytecode. Finally, Sect. 6 suggests future work.

2 Related Work

In-lined Reference Monitors were first formalized by Erlingsson and Schneider in the development of the PoET/PSLang/SASI systems [10, 17], which implement IRM's for Java bytecode and GNU assembly code. Subsequently, a variety of IRM implementations have been developed. The Java-MOP system [5] allows policy-writers to choose from a sizable collection of formal policy specification languages, including LTL. Mobile [13] targets Microsoft .NET bytecode by transforming untrusted CIL binaries into well-typed Mobile code (a subset of CIL). ConSpec [1] restricts IRM-injected code to effect-free operations, which allows a static analysis to verify that a rewritten program does not violate the intended policy. Finally, SPoX [12] rewrites Java bytecode programs to satisfy declarative, Aspect-Oriented security policies.

To our knowledge, ConSpec [1] and Mobile [13] are the only IRM systems to yet implement automatic certification. The ConSpec verifier performs a static analysis to verify that pre-specified guard code appears at each security-relevant code point; the guard code itself is trusted. Mobile implements a more general certification algorithm by type-checking the resulting Mobile code. While type-checking has the advantage of being light-weight, it comes at the expense of limited computational power. For instance, Mobile cannot enforce certain

dataflow-sensitive security policies since its type-checking algorithm is strictly control-flow based. While the security policies described by these systems are declarative and therefore amenable to a more general verifier, both use a verifier tailored to a specific rewriting strategy.

Related research on general model-checking is vast, but to our knowledge no past work has applied model-checking to the IRM verification problem. A majority of model-checking research has focused on detecting deadlock and assertion violation properties of source code. For example, Java PathFinder (JPF) [15] and Moonwalker [16] verify properties of Java and .NET source programs, respectively. 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]. In prior work [9], we have presented a general model-checking system for ActionScript bytecode implemented using co-logic programming [19]. This paper extends that work by introducing new formalisms specific to the verification of safety policies enforced by IRM's.

ActionScript is a binary virtual machine language by Adobe Systems similar to Java bytecode. It is important as a general web scripting language and is widely used in portable web ads, online games, streaming media, and interactive webpage animations. The ActionScript VM includes standard object-level encapsulation as well as a sandboxing model. While useful, these protections are limited to enforcing a restricted class of low-level, coarse-grained security policies. Several past malware attacks have used ActionScript as a vehicle within the past few years, including several virus families [11], as well as an emerging class of malicious URL-redirection attacks. URL-redirection attacks allow an embedded webpage widget (possibly served by a third party) to redirect the user's browser to a different website. These attacks are particularly problematic in the context of web advertising, since in these scenarios the security policy is typically a fusion of constraints prescribed by multiple independent parties, such as ad distributors and web hosts, who lack access to the applet source code. We apply our certified IRM framework to protect against such attacks in Sect. 5.

3 System Overview

3.1 IRM Framework

Figure 1 depicts the core of our IRM framework, consisting of a collection of rewriters that automatically transform untrusted ActionScript bytecode into self-monitoring ActionScript bytecode, along with a model-checking verifier that certifies the resulting IRM against the original security policy.[1] The untrusted code is obtained from *ShockWave Flash* (SWF) binary archives, which package ActionScript code with related data such as images and sound. Once the raw bytecode is extracted, a Definite Clause Grammar (DCG) [18] parser converts it to an annotated abstract syntax tree (AST) for easy analysis and manipulation.

[1] The IRM framework includes a rewriter per security policy class.

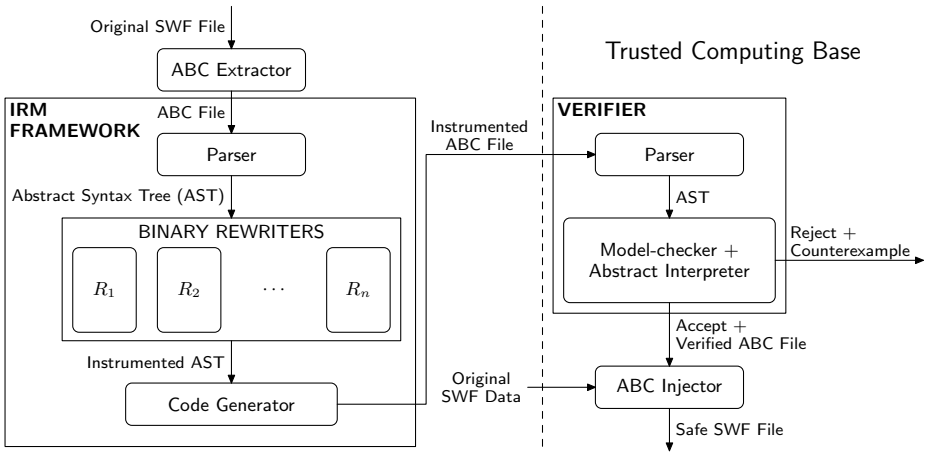


Fig. 1. Certifying ActionScript IRM architecture

We implemented this parser in Prolog so that the same code functions as a code generator due to the reversible nature of Prolog predicates [9]. Modified AST's produced by the rewriter are thereby transformed back into bytecode, and the ABC Injector reconstructs a modified SWF file by packaging the new code with the original data.

In practice it is usually infeasible to develop only one binary rewriter that can efficiently enforce all desired policies for all untrusted applications. Our IRM framework therefore actually consists of a collection of rewriters that have been tailored to different policy classes and rewriting strategies, and that are subject to change as new policies and runtime efficiency constraints arise. All rewriters remain untrusted since their output is certified by a single, trusted verifier. The verifier is more general than the rewriters, and therefore less subject to change. This results in a significantly smaller trusted computing base than if all rewriters were trusted.

The rewriter implementation is discussed in Sect. 5; the remainder of this section discusses the verifier.

3.2 Verifier Overview

The verifier is an abstract machine that non-deterministically explores all control-flow paths of untrusted code, inferring an abstract state for each code point. This process continues, bottom-up, until it converges to a (least) fixed point. The model-checker then verifies that each inferred abstract state is policy-satisfying.

A standard challenge in implementing such an abstract interpreter is to choose an expressive yet tractable language of state abstractions for the abstract machine to consider. A highly expressive state abstraction language allows very precise reasoning about untrusted code, but might cause the iteration process to converge slowly or not at all, making verification infeasible in practice. In

contrast, a less expressive language affords faster convergence, but might result in conservative rejection of many policy-adherent programs due to information lost by the coarseness of the abstraction.

In what follows, we describe a state abstraction that is suitably precise to facilitate verification of typical IRM's, yet suitably sparse to facilitate effective convergence. Section 4 proves these soundness and convergence properties formally. To motivate our choice of abstractions, we begin with a discussion of an important implementation strategy for IRM's—*reified security state*.

In order to enforce history-based security policies, IRM's typically maintain a reified abstraction of the current security state within the modified untrusted code. For example, to enforce a policy that prohibits event e_2 after event e_1 has already occurred, the IRM framework might inject a new boolean variable that is initialized to *false* and updated to *true* immediately after every program operation that exhibits e_1. The framework then injects before every e_2 operation new code that dynamically tests this injected variable to decide whether the impending operation should be permitted.

When security policies are expressed as security automata [2], this reification strategy can be generalized as an integer variable that tracks the current state of the automaton. Security automata encode security policies as Büchi automata that accept the language of policy-satisfying event sequences. Formally, a deterministic security automaton $A = (Q, \Sigma, q_0, \delta)$ can be expressed as a set of states Q, an alphabet of security-relevant events Σ, a start state $q_0 \in Q$, and a transition relation $\delta : Q \times \Sigma \rightarrow Q$. For the purpose of this paper, we assume that Q is finite.[2] The automaton accepts all finite or infinite sequences for which δ has transitions. Security automata therefore accept policies that are prefix-closed. That is, to prove that infinite executions of an untrusted program satisfy such a policy, it suffices to prove that every finite execution prefix satisfies the policy. We therefore define the set of finite prefixes $\mathcal{P}$ of the security policy denoted by a deterministic security automaton as follows.

Definition 1 (Security Policy). *Let $A = (Q, \Sigma, q_0, \delta)$ be a deterministic security automaton. The* security policy *$\mathcal{P}_A$ for automaton A is defined by $\mathcal{P}_A = Res_A(Q)$, where notation $Res_A(q)$ denotes the* residual *[8] of state q in automaton A—that is, the set of finite sequences that cause the automaton to arrive in state q—and we lift Res_A to sets of states via $Res_A(Q) = \cup_{q \in Q} Res_A(q)$. When automaton A is unambiguous, we will omit subscript A, writing $\mathcal{P} = Res(Q)$.*

Our verifier accepts as input security policies expressed as security automata and IRM's that implement reified security state as integer automaton states. To verify that the untrusted code accurately maintains these state variables to track the runtime security state, our abstract states include an *abstract trace* and *abstract program variable values* defined in terms of this automaton.

[2] Any actual implementation of an IRM must have a finite Q since otherwise the IRM would require infinite memory to represent the current security state.

Definition 2 (Abstract Traces). *The language SS of* abstract traces *is* $SS = \{(Res(Q_0), \tau) \mid Q_0 \subseteq Q, \tau \in \Sigma^*, |\tau| \leq k\} \cup \{\top_{SS}\}$ *where* $\top_{SS} = \Sigma^*$. *Abstract traces are ordered by subset relation* $\subseteq$, *forming the lattice* $(SS, \subseteq)$.

Intuitively, Definition 2 captures the idea that an IRM verifier must track abstract security states as two components: a union of residuals $Res(Q_0)$ and a finite sequence τ of literal events. Set $Res(Q_0)$ encodes the set of possible security states that the untrusted program might have been in when the reified state variable was last updated by the IRM to reflect the current security state. The actual current security state of the program can potentially be out of sync with the reified state value at any given program point because IRM's typically cannot update the state value in the same operation that exhibits a security-relevant event. Thus, trace τ models the sequence of events that have been exhibited since the last update of the state value. In general, an IRM may delay updates to its reified state variables for performance reasons until after numerous security-relevant events have occurred. Dynamic tests of reified state variables therefore reveal information about an earlier security state that existed before τ occurred, rather than the current security state. This distinction is critical for accurately reasoning about real IRM code.

We limit the length of τ in our definition to a fixed constant k to keep our abstract interpretation tractable. This means that when an IRM performs more than k security-relevant operations between state variable updates, our verifier will conservatively approximate traces at some program points, and might therefore conservatively reject some policy-adherent programs. The choice of constant k dictates a trade-off between IRM performance and verification efficiency. A low k forces IRM's to update security state variables more frequently in order to pass verification, potentially increasing runtime overhead. A high k relaxes this burden but yields a larger language of abstract states, potentially increasing verification overhead. For our implementation, $k = 1$ suffices.

Reified state values themselves are abstracted as integers or $\top_{VS}$ (denoting an unknown value). For simplicity, our formal presentation treats all program values as integers and abstracts them in the same way.

Definition 3 (Abstract Values). *Define* $VS = \mathbb{Z} \cup \{\top_{VS}\}$ *to be the set of* abstract program values*, and define* value order *relation* $\leq_{VS}$ *by* $(n \leq_{VS} n)$ *and* $(n \leq_{VS} \top_{VS})\ \forall n \in VS$. *Observe that* $(VS, \leq_{VS})$ *forms a height-2 lattice.*

3.3 Concrete Machine

The abstract states described above abstract the behavior of a concrete machine that models the actual behavior of ActionScript bytecode programs as interpreted by the ActionScript virtual machine. We define the concrete machine to be 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. Figure 2 defines a configuration $\chi = \langle L : i, \sigma, \nu, m, \tau \rangle$ as a *labeled instruction* $L : i$, an *operand stack* σ, a *local variable store* ν, a reified security state value m, and

$$
\begin{array}{lr}
\chi ::= \langle L : i, \sigma, \nu, m, \tau \rangle & \text{(configurations)} \\
L & \text{(code labels)} \\
i ::= \textbf{ifle}\ L \mid \textbf{getlocal}\ n \mid \textbf{setlocal}\ n \mid \textbf{jmp}\ L \mid & \text{(instructions)} \\
\quad \textbf{event}\ e \mid \textbf{setstate}\ n \mid \textbf{ifstate}\ n\ L & \\
\sigma ::= \cdot \mid v :: \sigma & \text{(concrete stacks)} \\
v \in \mathbb{Z} & \text{(concrete values)} \\
\nu : \mathbb{Z} \rightarrow v & \text{(concrete stores)} \\
m \in \mathbb{Z} & \text{(concrete reified state)} \\
e \in \Sigma & \text{(events)} \\
\tau \in \Sigma^* & \text{(concrete traces)} \\
\chi_0 = \langle L_0 : p(L_0), \cdot, \nu_0, 0, \epsilon \rangle & \text{(initial configurations)} \\
\nu_0 = \mathbb{Z} \times \{0\} & \text{(initial stores)} \\
P ::= (L, p, s) & \text{(programs)} \\
p : L \rightarrow i & \text{(instruction labels)} \\
s : L \rightarrow L & \text{(label successors)}
\end{array}
$$

Fig. 2. Concrete machine configurations and programs

a trace τ of security-relevant events that have been exhibited so far during the current run. A *program* $P = (L, p, s)$ consists of a program entrypoint label L, a mapping p from code labels to program instructions, and a label successor function s that defines the destinations of non-branching instructions.

To simplify the discussion, we here consider only a core language of ActionScript bytecode instructions. 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** e models a security-relevant operation that exhibits event e.

The **setstate** n and **ifstate** n L instructions set the reified security state and perform a conditional jump based upon its current value, respectively. While the real ActionScript instruction set does not include these last three operations, in practice they are implemented as fixed instruction sequences that perform security-relevant operations (e.g., system calls), store an integer constant in a safe place (e.g., a reserved private field member), and conditionally branch based on that stored value, respectively. The bytecode language's existing object encapsulation and type-safety features are leveraged to prevent untrusted code from corrupting reified security state.

Figure 3 provides a complete small-step operational semantics for the concrete machine. Observe that in Rule (CEvent), policy-violating events cause the concrete machine to enter a stuck state. Thus, security violations are modeled in the concrete domain as stuck states. The concrete semantics have no explicit operation for normal program termination; we model termination as an infinite

$$\frac{n_1 \leq n_2}{\langle L_1 : \textbf{ifle } L_2, n_1{::}n_2{::}\sigma, \nu, m, \tau\rangle \mapsto \langle L_2 : p(L_2), \sigma, \nu, m, \tau\rangle}(\textsc{CIfflePos})$$

$$\frac{n_1 > n_2}{\langle L_1 : \textbf{ifle } L_2, n_1{::}n_2{::}\sigma, \nu, m, \tau\rangle \mapsto \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau\rangle}(\textsc{CIfleNeg})$$

$$\frac{}{\langle L : \textbf{getlocal } n, \sigma, \nu, m, \tau\rangle \mapsto \langle s(L) : p(s(L)), \nu(n){::}\sigma, \nu, m, \tau\rangle}(\textsc{CGetlocal})$$

$$\frac{}{\langle L : \textbf{setlocal } n, n_1{::}\sigma, \nu, m, \tau\rangle \mapsto \langle s(L) : p(s(L)), \sigma, \nu[n := n_1], m, \tau\rangle}(\textsc{CSetlocal})$$

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

$$\frac{\tau e \in \mathcal{P}}{\langle L : \textbf{event } e, \sigma, \nu, m, \tau\rangle \mapsto \langle s(L) : p(s(L)), \sigma, \nu, m, \tau e\rangle}(\textsc{CEvent})$$

$$\frac{}{\langle L : \textbf{setstate } n, \sigma, \nu, m, \tau\rangle \mapsto \langle s(L) : p(s(L)), \sigma, \nu, n, \tau\rangle}(\textsc{CSetstate})$$

$$\frac{}{\langle L_1 : \textbf{ifstate } n\ L_2, \sigma, \nu, n, \tau\rangle \mapsto \langle L_2 : p(L_2), \sigma, \nu, n, \tau\rangle}(\textsc{CIfstatePos})$$

$$\frac{m \neq n}{\langle L_1 : \textbf{ifstate } n\ L_2, \sigma, \nu, m, \tau\rangle \mapsto \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau\rangle}(\textsc{CIfstateNeg})$$

Fig. 3. Small-step operational semantics for the concrete machine

stutter state. The soundness proof in Sect. 4 shows that any program that is accepted by the abstract machine will never enter a stuck state during any concrete run; thus, verification is sufficient to prevent policy violations.

3.4 Abstract Machine

We define our abstract machine as a tuple $(\mathcal{A}, \chi_0, \rightsquigarrow)$, where $\mathcal{A}$ is the set of configurations of the abstract machine, χ_0 is the same initial configuration as the concrete machine, and $\rightsquigarrow$ is the transition relation in the abstract domain. Abstract configurations are formally defined in Fig. 4. Figure 5 lifts the $\leq_{VS}$ relation to operand stacks and stores to form a lattice $(\mathcal{A}, \leq_{\hat{\chi}})$ of abstract states. That is, stacks (stores) are related if their sizes (domains) are identical and their corresponding members are related.

The small-step operational semantics of the abstract machine are given in Fig. 6. When the abstract machine can infer concrete values for operands, as in Rule (AIflePos), it performs a transition resembling the corresponding concrete transition. However, when operand values are unknown, as in Rule (AIfleTop), the abstract machine non-deterministically explores all possible control flows resulting from the operation.

The premises of rules (AEvent), (ASetstate), and (AIfstateNeg) appeal to a model-checker that decides subset relations for abstract states according to Definition 2. Thus, the abstract machine enters a stuck state when it encounters a potential policy violation (see Rule (AEvent)). Abstract stuck states correspond to rejection by the verifier.

$$
\begin{array}{rr}
\hat{\chi} ::= \bot \mid \langle L : i, \hat{\sigma}, \hat{\nu}, m, (Res(q_m), \bar{\tau})\rangle \mid \langle L : i, \hat{\sigma}, \hat{\nu}, \top_{VS}, \hat{\tau}\rangle & \text{(abstract configs)} \\
\hat{\sigma} ::= \cdot \mid \hat{v} :: \hat{\sigma} & \text{(evaluation stacks)} \\
\hat{v} \in VS & \text{(abstract values)} \\
\hat{\nu} : \mathbb{Z} \to \hat{v} & \text{(abstract stores)} \\
\hat{m} \in \mathbb{Z} \cup \top_{VS} & \text{(abstract reified state)} \\
\bar{\tau} \in \cup_{n \le k} \Sigma^n & \text{(bounded traces)} \\
\hat{\tau} \in SS & \text{(abstract traces)}
\end{array}
$$

Fig. 4. Abstract machine configurations

$$
\frac{}{\bot \le_{\hat{\chi}} \hat{\chi}}
\qquad
\frac{}{\cdot \le_{VS} \cdot}
$$

$$
\frac{\hat{\sigma} \le_{VS} \hat{\sigma}' \qquad \hat{\nu} \le_{VS} \hat{\nu}' \qquad R_m\tau \subseteq R_m\tau'}{\langle L : i, \hat{\sigma}, \hat{\nu}, m, (R_m, \tau)\rangle \le_{\hat{\chi}} \langle L : i, \hat{\sigma}', \hat{\nu}', m, (R_m, \tau')\rangle}
\qquad
\frac{\hat{\sigma}_1 \le_{VS} \hat{\sigma}_2 \qquad va_1 \le_{VS} va_2}{va_1 :: \hat{\sigma}_1 \le_{VS} va_2 :: \hat{\sigma}_2}
$$

$$
\frac{\hat{\sigma} \le_{VS} \hat{\sigma}' \qquad \hat{\nu} \le_{VS} \hat{\nu}' \qquad \hat{\tau} \subseteq \hat{\tau}'}{\langle L : i, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \le_{\hat{\chi}} \langle L : i, \hat{\sigma}', \hat{\nu}', \top, \hat{\tau}'\rangle}
\qquad
\frac{\hat{\nu}_1(n) \le_{VS} \hat{\nu}_2(n) \quad \forall n \in \mathbb{Z}}{\hat{\nu}_1 \le_{VS} \hat{\nu}_2}
$$

Fig. 5. State-ordering relation $\le_{\hat{\chi}}$

$$
\frac{n_1 \le n_2}{\langle L_1 : \textbf{ifle}\ L_2, n_1{::}n_2{::}\hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle L_2 : p(L_2), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle}\,(\textsc{AIfflePos})
$$

$$
\frac{n_1 > n_2}{\langle L_1 : \textbf{ifle}\ L_2, n_1{::}n_2{::}\hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle s(L_1) : p(s(L_1)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle}\,(\textsc{AIfleNeg})
$$

$$
\frac{\top_{VS} \in \{va_1, va_2\} \qquad L' \in \{L_2, s(L_1)\}}{\langle L_1 : \textbf{ifle}\ L_2, va_1{::}va_2{::}\hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle L' : p(L'), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle}\,(\textsc{AIfleTop})
$$

$$
\frac{}{\langle L : \textbf{getlocal}\ n, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\nu}(n){::}\hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle}\,(\textsc{AGetlocal})
$$

$$
\frac{}{\langle L : \textbf{setlocal}\ n, va_1{::}\hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}[n := va_1], \hat{m}, \hat{\tau}\rangle}\,(\textsc{ASetlocal})
$$

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

$$
\frac{\hat{\tau}e \subseteq \hat{\tau}' \subseteq \mathcal{P}}{\langle L : \textbf{event}\ e, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}'\rangle}\,(\textsc{AEvent})
$$

$$
\frac{\hat{\tau} \subseteq Res(q_n)}{\langle L : \textbf{setstate}\ n, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}, n, (Res(q_n), \epsilon)\rangle}\,(\textsc{ASetstate})
$$

$$
\frac{\hat{m} \in \{n, \top\}}{\langle L_1 : \textbf{ifstate}\ n\ L_2, \hat{\sigma}, \hat{\nu}, \hat{m}, (S, \tau)\rangle \rightsquigarrow \langle L_2 : p(L_2), \hat{\sigma}, \hat{\nu}, n, (Res(q_n), \tau)\rangle}\,(\textsc{AIfstatePos})
$$

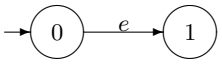
$$
\frac{\hat{m} \neq n \qquad (S - Res(q_n))\tau \subseteq \hat{\tau}}{\langle L_1 : \textbf{ifstate}\ n\ L_2, \hat{\sigma}, \hat{\nu}, \hat{m}, (S, \tau)\rangle \rightsquigarrow \langle s(L_1) : p(s(L_1)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle}\,(\textsc{AIfstateNeg})
$$

Fig. 6. Small-step operational semantics for the abstract machine

Rule (ASetstate) requires that acceptable programs must maintain a reified security state that is consistent with the actual security state of the program during any given concrete execution. This allows the (AIfstatePos) and (AIfstateNeg) rules of the abstract machine to infer useful security information in the positive and negative branches of program operations that dynamically test this state. The verifier can therefore reason that dynamic security guards implemented by an IRM suffice to prevent runtime policy violations.

3.5 An Abstract Interpretation Example

Abstract interpretation involves iteratively computing an abstract state for each code point. Multiple abstract states obtained for the same code point are combined by computing their join in lattice $(\mathcal{A}, \leq_{\hat{\chi}})$. This process continues until a fixed point is reached.



L_1 : **ifstate** 0 L_3	$\langle \ldots, 0, \epsilon \rangle \sqcup \langle \ldots, 1, e \rangle$	$= \langle \ldots, \top, \epsilon + e \rangle$
L_2 : **jmp** L_2	$\bot \sqcup \langle \ldots, \top, e \rangle$	$= \langle \ldots, \top, e \rangle$
L_3 : **event** e	$\langle \ldots, 0, \epsilon \rangle \sqcup \langle \ldots, 0, \epsilon \rangle$	$= \langle \ldots, 0, \epsilon \rangle$
setstate 1	$\langle \ldots, 0, e \rangle \sqcup \langle \ldots, 0, e \rangle$	$= \langle \ldots, 0, e \rangle$
jmp L_1	$\langle \ldots, 1, e \rangle \sqcup \langle \ldots, 1, e \rangle$	$= \langle \ldots, 1, e \rangle$

Fig. 7. An abstract interpretation example

To illustrate this, we here walk the abstract interpreter through the simple example program shown in the first column of Fig. 7, enforcing the policy $\epsilon + e$ whose security automaton is depicted at the top of the figure. Abstract states inferred on *first entry* to each code point are written to the left of the $\sqcup$ in the second column. (All but the reified state value 0 and trace ϵ are omitted from each configuration since they are irrelevant to this particular example.) Abstract states inferred on *second entry* are written after the $\sqcup$, and the resulting join of these states is written in the third column. In this example a fixed point is reached after two iterations.

The abstract interpreter begins at entrypoint label L_1 in initial configuration $\chi_0 = \langle \ldots, 0, \epsilon \rangle$. Since the reified state is known, the abstract machine performs transition (AIfstatePos) and arrives at label L_3. Operation **event** e appends e to the trace, operation **setstate** 1 updates the reified state, and operation **jmp** L_1 returns to the original code point.

The join of these two states yields a new configuration in which the reified state is unknown ($\top$), so on the second iteration the abstract machine nondeterministically transitions to both L_2 and L_3. However, both transitions infer

useful security state information based on the results of the dynamic test. Transition (AIFSTATEPOS) to label L_3 refines the abstract trace from $\epsilon+e$ to $Res(q_0) = \epsilon$, and transition (AIFSTATENEG) to label L_2 refines it to $\epsilon+e-Res(q_0) = e$. These refinements allow the verifier to conclude that all abstract states are policy-satisfying. In particular, the dynamic state test at L_1 suffices to prevent policy violations at L_3.

4 Analysis

4.1 Soundness

The abstract machine defined in Section 3.4 is *sound* with respect to the concrete machine defined in Section 3.3 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. This further implies that if the abstract machine does not enter a stuck state for a given program, nor does the concrete machine. Since concrete stuck states model security violations, this implies that a verifier consistent with the abstract machine will reject all policy-violating programs.

$$\frac{\sigma \leq_{VS} \hat{\sigma} \qquad \nu \leq_{VS} \hat{\nu} \qquad \tau \in \hat{\tau}}{\langle L : i, \sigma, \nu, m, \tau\rangle \sim \langle L : i, \hat{\sigma}, \hat{\nu}, \top, \hat{\tau}\rangle}(\textsc{SoundTop})$$

$$\frac{\sigma \leq_{VS} \hat{\sigma} \qquad \nu \leq_{VS} \hat{\nu} \qquad \tau \in Res(q_m)\tau' \qquad \tau \in S\tau'}{\langle L : i, \sigma, \nu, m, \tau\rangle \sim \langle L : i, \hat{\sigma}, \hat{\nu}, m, (S, \tau')\rangle}(\textsc{SoundInt})$$

Fig. 8. Soundness relation $\sim$

We define the soundness of state abstractions in terms of a *soundness relation* [7] written $\sim \subseteq \mathcal{C} \times \mathcal{A}$ that is defined in Fig. 8. Following the approach of [4], soundness of the operational semantics given in Figs. 3 and 6 is then proved via progress and preservation lemmas. The preservation lemma proves that a bisimulation of the abstract and concrete machines preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the concrete machine does not enter a stuck state. Together, these two lemmas dovetail to form an induction over arbitrary length execution sequences, proving that programs accepted by the verifier will not commit policy violations.

We sketch interesting cases of the progress and preservation proofs below.

Lemma 1 (Progress). *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim \hat{\chi}$, if there exists $\hat{\chi}' \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$, then there exists $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$.*

Proof. Let $\chi = \langle L : i, \sigma, \nu, m, \tau\rangle \in \mathcal{C}$, $\hat{\chi} = \langle L : i, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}\rangle \in \mathcal{A}$, 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 a case distinction

on the derivation of $\hat{\chi} \rightsquigarrow \hat{\chi}'$. The one interesting case is that for Rule (AEvent), since the corresponding (CEvent) rule in the concrete semantics is the only one with a non-trivial premise. For brevity, we show only that case below.

Case (AEvent)**:** From Rule (AEvent) we have $i = \textbf{event}\ e$ and $\hat{\chi}' = \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}' \rangle$, where $\hat{\tau}e \subseteq \hat{\tau}' \subseteq \mathcal{P}$ holds. Choose configuration $\chi' = \langle s(L) : p(s(L)), \sigma, \nu, m, (\tau, e) \rangle$. From $\chi \sim \hat{\chi}$ we have $\tau \in \hat{\tau}$. It follows that $\hat{\tau}e \subseteq \mathcal{P}$ holds. By Rule (CEvent), we conclude that $\chi \mapsto \chi'$ is derivable.

The remaining cases are straightforward, and are therefore omitted. □

Lemma 2 (Preservation). *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim \hat{\chi}$, if there exists a non-empty $\mathcal{A}' \subseteq \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \mathcal{A}'$, then for every $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$ there exists $\hat{\chi}' \in \mathcal{A}'$ such that $\chi' \sim \hat{\chi}'$.*

Proof. Let $\chi = \langle L : i, \sigma, \nu, m, \tau \rangle \in \mathcal{C}$, $\hat{\chi} = \langle L : i, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau} \rangle \in \mathcal{A}$, and $\chi' \in \mathcal{C}$ be given such that $\chi \mapsto \chi'$. Proof is by case distinction on the derivation of $\chi \mapsto \chi'$. For brevity we sketch only the most interesting cases below.

Case (CEvent)**:** From Rule (CEvent) we have $i = \textbf{event}\ e$ and $\chi' = \langle s(L) : p(s(L)), \sigma, \nu, m, \tau e \rangle$. Since $\mathcal{A}'$ is non-empty, we may choose $\hat{\chi}' = \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}' \rangle$ such that $\hat{\tau}e \subseteq \hat{\tau}' \subseteq \mathcal{P}$ by (AEvent). We can then obtain a derivation of $\chi' \sim \hat{\chi}'$ from the derivation of $\chi \sim \hat{\chi}$ by appending event e to all of the traces in the premises of (SoundTop) or (SoundInt), and observing that the resulting premises are provable from $\hat{\tau}e \subseteq \hat{\tau}'$.

Case (CSetstate)**:** From Rule (CSetstate) we have $i = \textbf{setstate}\ n$ and $\chi' = \langle s(L) : p(s(L)), \sigma, \nu, n, \tau \rangle$. Since $\mathcal{A}'$ is non-empty, we may choose $\hat{\chi}' = \langle s(L) : p(s(L)), \hat{\sigma}, \hat{\nu}, n, (Res(q_n), \epsilon) \rangle$ such that $\hat{\tau} \subseteq Res(q_n)$ holds by Rule (ASetstate). From $\chi \sim \hat{\chi}$ we have $\tau \in \hat{\tau}$. Thus, $\tau \in Res(q_n)$ holds and relation $\chi' \sim \hat{\chi}'$ follows from Rule (SoundInt).

Case (CIfstatePos)**:** From Rule (CIfstatePos) we have $i = \textbf{ifstate}\ n\ L_2$ and $\chi' = \langle L_2 : p(L_2), \sigma, \nu, n, \tau \rangle$. If $\hat{m} = n \neq \top$, then $\hat{\tau} = (S, \bar{\tau})$ by (AIfstatePos), so choose $a' = \langle L_2 : p(L_2), \hat{\sigma}, \hat{\nu}, n, (Res(q_n), \bar{\tau}) \rangle$. Relation $\chi \sim \hat{\chi}$ proves $\chi' \sim \hat{\chi}'$ by (SoundInt). Otherwise $\hat{m} = \top$, so choose $\hat{\chi}' = \langle L_2 : p(L_2), \hat{\sigma}, \hat{\nu}, \top, \hat{\tau} \rangle$. Relation $\chi \sim \hat{\chi}$ proves $\chi' \sim \hat{\chi}'$ by (SoundTop).

Case (CIfstateNeg)**:** From Rule (CIfstateNeg) we have $i = \textbf{ifstate}\ n\ L_2$ and $\chi' = \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau \rangle$, where $n \neq m$. If $\hat{m} \neq \top$ then $\hat{\tau} = (S, \bar{\tau})$ by (AIfstateNeg), so choose $\hat{\chi}' = \langle s(L_1) : p(s(L_1)), \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau}' \rangle$ such that $(S - Res(q_n))\bar{\tau} \subseteq \hat{\tau}'$ holds by (AIfstateNeg). In any deterministic security automaton, every residual is disjoint from all others. Thus, $\hat{m} \neq n$ implies that $\hat{\tau} \notin Res(q_n)\bar{\tau}$, and therefore $\hat{\tau} \subseteq (S - Res(q_n))\bar{\tau}$. A derivation of $\chi' \sim \hat{\chi}'$ can therefore be obtained from the one for $\chi \sim \hat{\chi}$ using (SoundInt). Otherwise $\hat{m} = \top$, and the rest of the case follows using logic similar to the case for (CIfstatePos). □

Theorem 1 (Soundness). *If the abstract machine does not enter a stuck state from the initial state χ_0, then for any concrete state $\chi \in \mathcal{C}$ reachable from the*

initial state χ_0, the concrete machine can make progress. If state χ is a security-relevant event then this progress is derived by rule (CEvent) *of Fig. 3, whose premise guarantees that the event does not cause a policy violation. Thus, any program accepted by the abstract machine does not commit a policy violation when executed.*

Proof. The theorem follows from the progress and preservation lemmas by an induction on the length of an arbitrary, finite execution prefix. □

4.2 Convergence

In practice, effective verification depends upon obtaining a fixed point for the abstract machine semantics in reasonable time for any given untrusted program. The convergence rate of the algorithm described in Sect. 3.5 depends in part on the height of the lattice of abstract states. This height dictates the number of iterations required to reach a fixed point in the worst case. All components of the language of abstract states defined in Fig. 4 have height at most 2, except for the lattice SS of abstract traces. Lattice SS is finite whenever security automaton A is finite; therefore convergence is guaranteed in finite time. In the proof that follows we prove the stronger result that lattice SS has non-exponential height—in particular, it has height that is quadratic in the size of the security automaton.

Theorem 2. *Let $A = (Q, \Sigma, \delta)$ be a deterministic, finite security automaton. Lattice $(SS, \subseteq)$ from Definition 2 has height $O(|Q|^2 + k|Q|)$.*

Proof. Let $Q_1, Q_2 \subseteq Q$ and $\tau_1, \tau_2 \in \cup_{n \leq k} \Sigma^n$ be given. For all $i \in \{1, 2\}$ define $L_i = \mathit{Res}(Q_i)\tau_i$, and assume $\emptyset \subsetneq L_1 \subsetneq L_2 \subseteq \mathcal{P}$. Define

$$m(L) = (|Q| + 1)|\mathit{suf}(L)| - |\mathit{Pre}(L)|$$

where $\mathit{suf}(L) = \max\{\tau \in \Sigma^* \mid L \subseteq \Sigma^*\tau\}$ is the largest common suffix of all strings in non-empty language L and $\mathit{Pre}(L) = \{q \in Q \mid \mathit{Res}(q)\mathit{suf}(L) \cap L \neq \emptyset\}$ is the set of possible automaton states that an accepting path for a string in L might be in immediately prior to accepting the common suffix. We will prove that $m(L_1) > m(L_2)$. By the pumping lemma, $|\mathit{suf}(L_i)| = |\mathit{suf}(\mathit{Res}(Q_i)\tau_i)|$ is at most $|Q| + k$, so this proves that any chain in lattice $(SS, \subseteq)$ has length at most $O(|Q|^2 + k|Q|)$.

We first prove that $\mathit{Res}(\mathit{Pre}(L_i))\mathit{suf}(L_i) = L_i \ \forall i \in \{1, 2\}$. The $\supseteq$ direction of the proof is immediate from the definition of Pre; the following proves the $\subseteq$ direction. Let $\tau \in \mathit{Res}(\mathit{Pre}(L_i))\mathit{suf}(L_i)$ be given. There exists $q \in \mathit{Pre}(L_i)$ and $\tau' \in \mathit{Res}(q)$ such that $\tau = \tau'\mathit{suf}(L_i)$. Since $L_i = \mathit{Res}(Q_i)\tau_i$, τ_i is a suffix of $\mathit{suf}(L_i)$, so there exists $\tau_i' \in \Sigma^*$ such that $\mathit{suf}(L_i) = \tau_i'\tau_i$. From $q \in \mathit{Pre}(L_i)$ we obtain $\mathit{Res}(q)\mathit{suf}(L_i) \cap L_i = \mathit{Res}(q)\tau_i'\tau_i \cap \mathit{Res}(Q_i)\tau_i = (\mathit{Res}(q)\tau_i' \cap \mathit{Res}(Q_i))\tau_i \neq \emptyset$. Thus, there is an accepting path for τ_i' from q to some state in $\mathit{Res}(Q_i)$. It follows that $\tau'\tau_i' \in \mathit{Res}(Q_i)$, so $\tau = \tau'\tau_i'\tau_i \in \mathit{Res}(Q_i)\tau_i = L_i$. We conclude that $\mathit{Res}(\mathit{Pre}(L_i))\mathit{suf}(L_i) \subseteq L_i$.

From this result we prove that $m(L_1) > m(L_2)$. Since $L_1 \subsetneq L_2$, it follows that $suf(L_2)$ is a suffix of $suf(L_1)$. If it is a strict suffix then the theorem is proved. If instead $suf(L_1) = suf(L_2) = x$, then we have the following:

$$\begin{gathered} L_1 \subsetneq L_2 \\ Res(Pre(L_1))x \subsetneq Res(Pre(L_2))x \\ Res(Pre(L_1)) \subsetneq Res(Pre(L_2)) \end{gathered}$$

Since A is deterministic and therefore each residual is disjoint, we conclude that $Pre(L_1) \subsetneq Pre(L_2)$ and therefore $m(L_1) > m(L_2)$. □

5 Implementation

We used our IRM framework to enforce and verify a URL anti-redirection policy for ActionScript ad applets. ActionScript bytecode performs a URL redirection using the `navigateToURL` system call, which accepts the URL target as its argument. To protect against malicious redirections, we enforced a policy that requires `check_url(s)` to be called sometime before any call to `navigateToURL(s)`, for each string constant `s`. Here, `check_url` is a trusted implementation provided by the ad distributor and/or web host, and may therefore rely on dynamic information such as the webpage that houses the current ad instance, the current user's identity, etc.

A naïve IRM can satisfy this policy by simply inserting a call to `check_url` immediately before each call to `navigateToURL`. Since calls to `check_url` are potentially expensive, our IRM takes the more efficient approach of reifying a separate security state variable into the untrusted binary for each string constant.[3] The reified state tracks whether that string has yet passed inspection and avoids duplicate calls for the same constant. In the less common case where the untrusted code must call `navigateToURL` with a dynamically generated string, the IRM resorts to the naïve approach described above. Maintaining persistent internal state for dynamically generated strings is left for future work.

Program Tested	Size Before	Size After	Rewriting Time	Verification Time
`countdownBadge.abc`	1.80 KB	1.95 KB	1.429s	0.532s
`NavToURL.abc`	0.93 KB	1.03 KB	0.863s	0.233s

Fig. 9. Experimental Results

The resulting instrumented binaries are independently certified by the model-checking verifier using the original security policy expressed as a security automaton. Figure 9 shows the results of rewriting and verifying binaries extracted from two real-world SWF ads that perform redirections. All tests were performed on

[3] The number of string constants is known at rewriting time based on the size of the constant pool in the ActionScript binary.

an Intel Pentium Core 2 Duo machine running Yap Prolog v5.1.4. In both cases the IRM passed verification and prevented malicious URL redirections.

6 Conclusion

We have presented a technique for certifying IRM's through model-checking. Our technique derives a state abstraction lattice from a security automaton to facilitate precise abstract interpretation of IRM code. Formal proofs of soundness and convergence guarantee reliability and tractability of the verification process. Finally, we demonstrate the feasibility of our technique by enforcing a URL anti-redirection policy for ActionScript bytecode programs.

While our algorithm successfully verifies an important class of IRM implementations involving reified security state, it does not support all IRM rewriting strategies. Reified security state that is per-object [13] instead of global, or that is updated by the IRM before the actual security state changes at runtime rather than after, are two examples of IRM strategies not supported by our model. In future work we intend to investigate ways of generalizing our approach to cover these cases.

We also plan to augment our system with support for recursion and mutual recursion, which is currently not handled by our implementation. Finally, we also plan to extend our technique to other binary languages and the IRM systems that have been implemented for them.

Acknowledgments

The authors thank Peleus Uhley at Adobe Research for providing real-world SWF applets of interest for testing and certification, and R. Chandrasekaran and Feliks Kluzniak for various helpful discussions.

References

1. I. Aktug and K. Naliuka. ConSpec - a formal language for policy specification. *Science of Computer Programming*, 74:2–12, 2008.
2. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1986.
3. G. Balakrishnan, T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum. Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Proc. Computer-Aided Verification*, pages 158–163, 2005.
4. B.-Y. E. Chang, A. Chlipala, and G. C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Proc. Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, 2006.
5. F. Chen. Java-MOP: A monitoring oriented programming environment for Java. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, 2005.

6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Prog. Languages*, pages 234–252, 1977.
7. P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.
8. F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. In *Proc. Annual Symposium on Theor. Aspects of Comput. Sci.*, volume 2010/2001 of *LNCS*, pages 144–157, 2001.
9. B. W. DeVries, G. Gupta, K. W. Hamlen, S. Moore, and M. Sridhar. ActionScript bytecode verification with co-logic programming. In *Proc. ACM Workshop on Prog. Languages and Analysis for Security (PLAS)*, 2009.
10. Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop*, 1999.
11. fukami and B. Fuhrmannek. SWF and the malware tragedy. In *Proc. OWASP Application Security Conference*, 2008.
12. K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proc. ACM Workshop on Prog. Languages and Analysis for Security (PLAS)*, 2008.
13. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. ACM Workshop on Prog. Languages and Analysis for Security (PLAS)*, 2006.
14. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. In *ACM Trans. Prog. Languages and Systems*, 2006.
15. W. Kisser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
16. T. C. Ruys and N. H. M. A. de Brugh. MMC: the Mono Model Checker. *Electron. Notes Theor. Comput. Sci.*, 190(1):149–160, 2007.
17. F. B. Schneider. Enforceable security policies. *ACM Trans. Information and System Security*, 3:30–50, 2000.
18. L. Shapiro and E. Y. Sterling. *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press, 1994.
19. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In *Proc. Int. Conf. on Logic Programming*, 2006.