Hippocratic Binary Instrumentation: First Do No Harm (Extended Version)

Meera Sridhar Richard Wartell Kevin W. Hamlen

The University of Texas at Dallas

Technical Report UTDCS-03-13

February 26, 2013

Abstract

In-lined Reference Monitors (IRMs) cure binary software of security violations by instrumenting them with runtime security checks. Although over a decade of research has firmly established the power and versatility of the in-lining approach to security, its widespread adoption by industry remains impeded by concerns that in-lining may corrupt or otherwise harm intended, safe behaviors of the software it protects. Practitioners with such concerns are unwilling to adopt the technology despite its security benefits for fear that some software may break in ways that are hard to diagnose.

This paper shows how recent approaches for machine-verifying the policy-compliance (soundness) of IRMs can be extended to also formally verify IRM preservation of policy-compliant behaviors (transparency). Feasibility of the approach is demonstrated through a transparency-checker implementation for Adobe ActionScript bytecode. The framework is applied to enforce security policies for Adobe Flash web advertisements and automatically verify that their policy-compliant behaviors are preserved.

1 Introduction

Runtime software monitoring via binary instrumentation (a.k.a., *in-lined reference monitoring*) has gained much attention in the literature as a powerful, flexible, and efficient approach to software security enforcement (e.g., [1, 2, 4, 8, 13, 15, 17–19, 24–26, 28, 32, 38–40, 47, 54, 55]). In-lined reference monitors (IRMs) dynamically enforce security policies by injecting security guards into untrusted binary code. At runtime, the guards check impending program operations and take corrective action if the operations constitute policy violations. The result is a new program that efficiently self-enforces a customized security policy.

Correct IRMs must satisfy two requirements: *soundness* and *transparency* [31, 40]. Soundness demands that the instrumented code satisfy the security policy, whereas transparency demands that it preserve the behavior of policy-compliant code. To formally define policy-compliance, IRM policies are specified using a *policy specification language* (e.g., [4, 8, 18, 23, 26, 30, 38, 55]), which typically

leverages concepts from *aspect-oriented programming* (AOP) [37] to abstractly identify security-relevant program operations. For example, the SPoX IRM system [30] expresses safety policies encoded as aspect-oriented security automata.

Independent certification of instrumented code has become increasingly important for minimizing and stabilizing the trusted computing base (TCB) of IRM systems [4, 5, 33, 50, 51]. Without such certification, TCBs of these systems may be undesirably large and complex for several reasons: (1) The rewriters that perform instrumentation often rely upon heuristic binary reverse-engineering components that are unsound, incomplete, or both. (2) Practical frameworks may include many rewriting systems deployed by different principals with differing security interests. All these rewriters and their future revisions enlarge the TCB. (3) In many contexts policy specifications change rapidly as new attacks appear and new vulnerabilities are discovered. A good example is IRM systems for web ad security [39, 49, 50], which change frequently as new system vulnerabilities emerge. Thus, a separate, light-weight, and policy-general certifier that checks instrumented code on a case-by-case basis is valuable for shifting the larger and less stable rewriting system out of the TCB.

Numerous past works have developed powerful technologies for formally machine-verifying the soundness of IRMs [4, 5, 10, 32, 33, 49–51]. However, none to our knowledge have tackled the dual problem of machine-verifying transparency (cf., [36, 51]). Transparency is a major concern for organizations that stand to lose significant revenue or reputation from temporary functionality losses. For example, despite high concerns about web ad security, ad distribution networks are often unwilling to adopt any protection system that involves binary modification unless there is overwhelming evidence that no safe ads are adversely affected by the process. Even a temporary loss of functionality in a few ads can result in millions of dollars in lost revenue. Proof of transparency is therefore a prerequisite for practical adoption of these security technologies.

The high difficulty of creating fully generalized, program-agnostic IRMs that correctly preserve all safe applications justifies this call for strong evidence of transparency. It is quite common for a monitor that works flawlessly when in-lined into most applications to suddenly malfunction when it is in-lined into an unusual application, such as one that overrides system methods called by the IRM, modifies the class-loader in an unusual way, or adds event listeners that disrupt monitor control-flows. Such conflicts are very difficult to identify manually at the binary level, motivating the need for automated assistance.

While the general problem of verifying program-equivalence is well known to be undecidable, we observe that the special case of verifying IRM transparency is more tractable due to the way IRMs are produced. IRMs are typically generated by automated binary *rewriters*, which transform policy specifications into suitable code insertions. Since the rewriter's code analysis power is limited, it must limit itself to insertions that it can infer are sound and transparent with respect to the target program. All rewriters therefore carry internal, implicit evidence that their code transformations are not harmful. By making this evidence explicit, we show that a verifier can independently confirm that code produced by the rewriter preserves all safe flows (without trusting the rewriter or the evidence it

presents).

This paper therefore presents the design and implementation of the first automated transparency-verifier for IRMs. Our main contributions include:

- We show how prior work on verifying IRM soundness via model-checking [33, 50] can be extended in a natural way to verify IRM transparency.
- We demonstrate how an untrusted, external invariant-generator can reduce the verifier's state-exploration burden and afford it greater generality than the more specialized rewriting systems it checks.
- Prolog unification [48] and Constraint Logic Programming (CLP) [34] are leveraged to keep the verifier implementation simple and closely tied to the underlying verification algorithm.
- Proofs of correctness are formulated for the verification algorithm using Cousots' abstract interpretation framework [16].
- The feasibility of our technique is demonstrated through a prototype implementation that targets the full ActionScript bytecode language.

The rest of the paper is organized as follows. We begin with a summary of prior work that influences our system design in §2. Section 3 presents an overview of our transparency verifier, and §4 details the verification and abstract interpretation algorithms. Section 5 presents implementation and results. Finally, §6 concludes.

2 Related Work

In-lined Reference Monitors IRMs were first formalized in the development of the PoET/PSLang/SASI systems [24, 47], which instrument Java bytecode and Gnu assembly code. Subsequently, numerous IRM frameworks have been developed for Java [4, 8, 13, 17, 26, 30, 38, 40], JavaScript [28, 55], .NET [32], ActionScript [30, 39], Android [19], and x86/64 native code [1, 2, 25, 54] architectures. Our experiments target SPoX-IRMs [30], which rewrite Java and ActionScript bytecode programs to satisfy declarative, aspect-oriented security policies.

All of these systems rewrite untrusted binaries by statically identifying potentially policy-violating program operations, and injecting guard code that dynamically decides whether impending operations are safe. The exact implementation of the guard code varies widely depending on policy, architectural, and application details. For example, to enforce stateful policies (i.e., those in which each event's permissibility depends on the history of past events) the IRM may introduce new program variables, methods, and classes that track the history of security-relevant events at runtime. Guard code then consults these *reified state variables* in order to test for impending violations.

IRMs also typically make some effort to optimize their code insertions for better performance, such as by hoisting checks out of loops or reorganizing basic

blocks. Thus, a mere syntactic comparison of original and rewritten code is not sufficient in general to verify transparency of real IRMs. This influences the design of our verifier, since our goal is to support a wide class of rewriting approaches.

IRM Soundness Certification Several past works have successfully performed certification of IRM soundness. The Mobile system [32] 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. ConSpec [4, 5] adopts a security-by-contract approach to IRM certification. Its certifier performs a static analysis that verifies that contract-specified guard code appears at each security-relevant code point. Our past work presents model-checking as an efficient approach for verifying such IRMs without trusted guard code [33, 50]. The Coq proof assistant has also been applied to implement monitor-generating algorithms that are provably sound [10].

IRM Transparency In contrast, transparency has been less studied. IRM transparency is defined in terms of a trace-equivalence relation that demands that the original and IRM-instrumented code must exhibit equivalent behavior on equal inputs whenever the original obeys the policy [31, 40]. Traces are equivalent if they are equal after erasure of irrelevant events (e.g., stutter steps). Subsequent work has proposed that additionally the IRM should preserve violating traces up to the point of violation [36].

Chudnov and Naumann provide the first formal proof of transparency for a real IRM [15]. Their IRMs enforce information flow properties, so transparency is there defined in terms of program input-output pairs. In lieu of machine-certification, a written proof establishes that all programs yielded by that particular rewriting algorithm are transparent. The proof is therefore specific to one rewriting algorithm and does not necessarily generalize to other IRM systems or policy languages.

Compiler Verification Program equivalence-checking has been studied in the context of *translation validation*, which verifies behavior-preservation of compiler optimization phases [43, 46]. Conceptually, translation validators explore the cross-product space of an abstract bisimulation of original and rewritten code, attempting to prove a semantic equivalence property of each abstract state [56]. By changing the property being checked, one can potentially verify software security properties, such as information flow policies [7].

Our work applies cross-product exploration to the problem of IRM transparency verification. However, unlike compiler translations, IRMs are not obligated to satisfy transparency for policy-violating flows—indeed, they must not. This significantly changes the semantic equivalence properties that a transparency verifier must check. The new property is essentially an implication with policy-adherence as its antecedent and observable semantic equivalence as its

consequent. In addition, IRMs introduce non-trivial, permanent memory state changes (e.g., reified state variables that track security state, modified arguments that flow to potentially unsafe operations, etc.) and interprocedural structural changes (e.g., new classes and methods associated with the monitor) that are atypical of compiler optimizations. These are not supported by existing compiler translation validators to our knowledge.

Secure Protocol Analysis Observational equivalence of abstract processes is often formally verified to ensure data confidentiality of communication protocols [9, 14, 20, 53]. In this context, observational equivalence implies that a secret exchanged by the protocol is not divulged to an attacker. This differs from our work in at least two significant respects: (1) Observational equivalence of IRMs is less strict because our goal is program feature preservation, not privacy. Thus, some visible behavior changes (e.g., timing changes) are acceptable as long as they do not impair desired program functionality. (2) We decide observational equivalence of real binary programs expressed in a real-world language (ActionScript), rather than abstract process descriptions.

Semantic Equivalence for Revision Tracking Differential symbolic execution [44] involves characterizing revision changes to software systems. Its goal is code documentation and comprehension rather than preclusion of observable behavior differences at the binary level.

ActionScript Security 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. ActionScript VM security is based on object-level encapsulation, code-signing, and sandboxing.

ActionScript's pervasive use in web advertisements has made it an attractive vehicle for many malware attacks in recent years. The first ActionScript virus (SWF/LFM-926) was identified in 2002, followed by numerous others including the Troj/SWFexp, Sus/SWFScene, Troj/SwfDL, and Troj/SWFdldr families, to name a few. Though its bytecode language is type-safe, past malware has exploited VM buffer overflows [22], implemented cross-site-scripting attacks, and performed click-jacking [41, 42] to damage browsers or disrupt victim host pages. The difficulty of enforcing rich ActionScript security policies that prevent such attacks in web environments that are aggressively heterogeneous (e.g., composed of *mash-ups* that mix mobile code from many mutually distrusting sources) has led to application of IRM technologies to this challenging problem domain [21, 33, 39, 49–51].

Existing security mechanisms for ActionScript and related technologies (including the ActionScript VM, Flash, Flex, and AIR) mainly fall into two categories: code-signing and sandboxing. While these suffice for certain classes of attacks, code-signing places the code-producer in the TCB, and sandboxing enforces only coarse-grained access control policies built into the VM and runtime

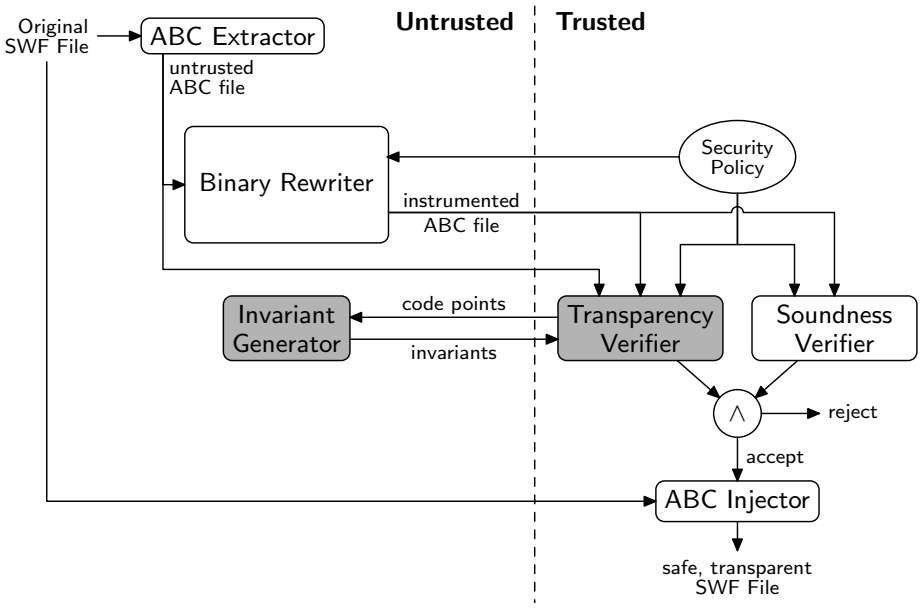

Figure 1: A certifying ActionScript IRM architecture

libraries. System- and application-specific policies, such as those that prohibit write access to files with certain names, or finer-grained policies that constrain arguments to individual instructions (such as those that exploit buffer overruns on older VMs) are not supported.

3 Overview

3.1 Rewriting and Soundness Verification

Figure 1 depicts our certifying IRM framework, consisting of a binary rewriter that automatically transforms untrusted ActionScript bytecode into self-monitoring bytecode, along with verifiers for soundness and transparency. Our main contributions are the transparency-verifier and invariant generator; rewriting and soundness verification are based on prior work [21, 49, 50].

Untrusted code is obtained from *ShockWave Flash* (SWF) binary archives, which package ActionScript code with data, such as images and sound. A binary rewriter rewrites the bytecode according to the security policy. It adopts the typical strategy of injecting guard code around potentially security-relevant bytecode instructions. The guard code dynamically tracks security state and tests arguments of impending operations for security-relevance. Impending violations solicit corrective action, which can include premature termination, raising an alarm, interacting with the user, and/or rolling the application back to a consistent state.

To enforce stateful policies, the IRM introduces reified state variables that track the history at runtime. This is achieved by expressing the policy as a deterministic *security automaton* [6, 47] that accepts the language of permissible traces. By assigning integer labels to the automaton states, the IRM efficiently tracks the security state using integer-valued fields.

Modified bytecode produced by the rewriter is re-packaged with the original data to produce a new, safe SWF file. Policy adherence of the instrumented code is independently verified by the soundness verifier, while behavioral preservation of safe programs is confirmed by the transparency verifier. Only code that passes both tests is reassembled with its data and executed.

3.2 Defining IRM Transparency

Past work defines IRM transparency and policies in terms of traces [31, 40]:

Definition 1 (Events, Traces, and Policies) *A* trace τ *is a (finite or infinite) sequence of* observable events*, where observable events are a distinguished subset of all* program operations*—instructions parameterized by their arguments. Policies $\mathcal{P}$ denote sets of permissible traces.*

Observability can be defined in various ways. In our implementation, observable events include most system API calls and their arguments, which are the only means in ActionScript to affect process-external resources like the display or file system. Policy specifications may identify certain API calls as unobservable by assumption, such as those known to be effect-free.

Transparency can then be defined as equivalence of traces exhibited by a *bisimulation* of the original program with its IRM-instrumented counterpart. Intuitively, the bisimulation runs both programs simultaneously (on equal inputs), non-deterministically stepping one or the other on each computational step. An adequate transparency definition must support non-terminating computations (i.e., infinite traces), it must not demand that programs are lock-step behaviorally equivalent (since IRMs introduce new code and reorder existing code), and it must not permit IRMs to infinitely delay original, safe computations (since that effectively discards the original computation, violating transparency). This leads to the following definitions:

Definition 2 (Progressive) *A bisimulation flow w is* progressive *if both bisimulated programs step infinitely often (i.o.) in w. (To support terminating computations, we model termination as an infinite stutter state.)*

Definition 3 (Shuffle) *Two bisimulation flows w_1 and w_2 are* shuffles *of one another if $\pi_i w_1 = \pi_i w_2$ for all $i \in \{1, 2\}$, where projection $\pi_j w$ denotes the sequence of program i's steps and states in bisimulation flow w.*

Definition 4 (Transparency) *A bisimulation state is* transparent *if its constituent program states have observationally equivalent traces.*[1] *A bisimulation*

[1] Each program state conceptually includes a trace of its observable event history.

is transparent *if for every progressive flow w, there exists a shuffle of w whose states are i.o. transparent.*

In the above, shuffles admit all possible interleavings of the steps and states of the two programs without reordering the steps and states of each individual program. Definition 4 therefore recognizes an IRM as transparent if it preserves the order and reachability of observable events of non-violating traces. This permits IRMs to augment untrusted code with unobservable stutter-steps (e.g., runtime security checks) and observable interventions (e.g., code that takes corrective action when an impending violation is detected), but not new operations that are observably different even when the original code does not violate the policy. The IRM must also not insert non-terminating loops to policy-adherent flows, since these could suppress desired program behaviors.

3.3 Verifying Transparency

Our transparency verifier is an abstract interpreter and model-checker that nondeterministically explores the cross-product of the state spaces of the original and rewritten programs. To accommodate IRMs that introduce new methods, abstract interpretation is fully interprocedural; calls flow into the bodies of callees. (Recursive and mutually recursive callees require a loop invariant, discussed in §3.4, in order for this process to converge.)

Each abstract state includes a store that maps fields and local variables to symbolic expressions, various other structures that model ActionScript VM states (e.g., stacks), and an abstract trace that describes the language of possible traces exhibited prior to the current state. They additionally include linear constraints introduced by conditional instructions. For example, the abstract states of control-flow nodes dominated by the positive branch of a conditional instruction that tests `(x<=y)` may contain the constraint $x \leq y$.

Our approach to transparency verification is based on the observation that in order to prove transparency of a particular program pair, it suffices to prove that there exists a set S of transparent, abstract, bisimulation states that are visited infinitely often in every *policy-compliant* bisimulation of the two programs. (Recall that policy-violating runs are intentionally modified by the IRM, and therefore exempt from transparency obligations.) Such a set inductively establishes that every safe computation is preserved. This observation is formalized by the following theorem:

Theorem 1 (Transparency) *A bisimulation is transparent if there exists a set S of abstract bisimulation states such that*

(1) S includes the abstract bisimulation start state $\hat{\Gamma}_0$;

(2) every state in S is transparent; and

(3) for every policy-compliant, progressive flow $\hat{\Gamma}_0 \cdots \hat{\Gamma}\, w$ where $\hat{\Gamma} \in S$, there exists a shuffle of suffix w that includes a member of S.

Proof 1 *Assume there exists such a set S, and let w be a policy-compliant, progressive bisimulation flow. Flow w begins with start state $\hat{\Gamma}_0$, and $\hat{\Gamma}_0 \in S$ by (1). Applying (3) inductively proves that w has a shuffle in which states of S appear i.o. States of S are transparent by (2), so w is transparent. Thus, all such flows are transparent, so the bisimulation is transparent.*

By exhibiting such a set S, a rewriter can prove to an independent verifier that IRMs it produces are transparent. The verifier confirms that S satisfies properties 1–3 of Theorem 1. To confirm property 3, it abstractly bisimulates all flows from each state in S, confirming that each has a shuffle that revisits a state in S.

3.4 Invariant Generation

Set S is feasible for existing IRM systems to infer and expose because it conceptually corresponds to the code points where the in-lined IRM code ends and the application's original programming resumes. Thus, while general-purpose invariant-generation is not tractable for arbitrary software, our approach benefits from the fact that IRM systems leave large portions of the untrusted programs they modify unchanged for practical reasons. Their modifications tend to be limited to small, isolated blocks of guard code scattered throughout the modified binary. Past work has observed that the unmodified sections of code tend to obey relatively simple invariants that facilitate tractable proofs of soundness for the resulting IRM [33, 50].

We observe that a similar strategy suffices to generate invariants that prove transparency for these IRMs. Specifically, an invariant-generator for a typical IRM system can assert that if the two programs are observably equivalent on entry to each block of guard code, and the original program does not violate the policy during the guarded block, then the traces are equivalent on exit from the block. Moreover, the abstract states remain step-wise equivalent outside these blocks. This strategy reduces the vast majority of the search space that is unaffected by the IRM to a simple, linear scan that confirms that the IRM remains dormant outside these blocks (i.e., its state does not leak into the observable events exhibited by the rewritten code).

Our framework lazily reveals S via an untrusted invariant-generator that gives the verifier hints that help it more quickly confirm that S satisfies Theorem 1. For each abstract code point $\hat{\Gamma}$ in the cross-product state space, the invariant-generator suggests (1) a state from S that abstracts $\hat{\Gamma}$, and (2) a subset of S that post-dominates $\hat{\Gamma}$ (i.e., where flows that pass through $\hat{\Gamma}$ later exhibit equivalent shuffles). The former abstracts away extraneous information inferred by the abstract interpreter that is irrelevant for proving transparency. The latter is a witness that proves property 3 of Theorem 1.

3.5 Invariant Verification

Hints provided by the invariant-generator remain strictly untrusted by the verifier. They are only accepted if they are implied by information already inferred by the verifier's abstract interpreter. Over-abstractions can cause the verifier to discard information needed to prove transparency, resulting in conservative rejection of the code; but they never result in acceptance of non-transparent code. This allows invariant-generation to potentially rely on untrusted information, such as the binary rewriting algorithm, without including that information in the TCB of the system.

To verify abstract bisimulation states suggested by the untrusted invariant-generator, prune policy-violating flows, and check trace-equality, the heart of the transparency verifier employs a model-checking algorithm that proves implications of the form $A \Rightarrow B$, where A is an abstract bisimulation state inferred by the abstract interpreter, and B is an untrusted abstraction suggested by the invariant-generator. Model-checking consists of two stages:

1. *Unification.* Program states include data structures, such as ActionScript bytecode operand stacks, objects, and traces. These are first mined for equality constraints through unification. For example, if state A includes constraints $\hat{\rho}_1 = v_1::\hat{s}_1$, $\hat{\rho}_2 = v_2::\hat{s}_2$, and $\hat{\rho}_1 = \hat{\rho}_2$, then unification infers additional equalities $v_1 = v_2$ and $\hat{s}_1 = \hat{s}_2$.

2. *Linear constraint solving.* The equality constraints inferred by step 1 are then combined with any inequality constraints in each state to form a pure linear constraint satisfaction problem without structures. A linear constraint solver verifies that sentence $A' \wedge \neg B'$ is unsatisfiable, where A' and B' are the linear constraints from A and B, respectively.

Both unification and linear constraint solving can be elegantly realized in Prolog with Constraint Logic Programming (CLP) [34], making this an ideal language for our verifier implementation.

Verification assumes bytecode type-safety of both original and rewritten code as a prerequisite. This assumption is checked by the ActionScript VM type-checker. Assuming type-safety allows the IRM and verifier to leverage properties such as object encapsulation, memory safety, and control-flow safety to reduce the space of executions that must be anticipated.

3.6 Limitations

We demonstrate experimentally (see §5) that generation of adequate invariants is tractable for typical IRMs; however, the power of our approach remains limited by the power of the model-checker's constraint language. For example, an IRM that stores object security states in a hash table cannot be verified by our system because our constraint language is not sufficiently powerful to express collision properties of hash functions that are necessary for proving that such an IRM only undertakes observable, corrective actions when a policy violation would otherwise result.

Our verifier also cannot verify IRMs that insert non-trivial loops into policy-adherent flows. The verifier conservatively rejects such loops because they lead to potentially infinite flows with no finite prefix where the traces are equal. IRMs may, however, safely introduce loops as part of interventions, since they are under no obligation to maintain transparency for policy-violating flows. Loops in the original, unmodified code are also supported because the verifier does not need to prove that they terminate; it simply proves that their termination conditions are unchanged by the IRM.

ActionScript does not presently support concurrency or multithreading (although the underlying VM/OS may use threading to concurrently execute multiple applets). Therefore, our transparency verification algorithm restricts its attention to purely serial flows.

Introspective (e.g., reflective) code has some interesting implications for transparency. Rather than conservatively prohibiting introspection, we model introspection as an input from the external computing environment. Transparency asserts that the original and IRM-instrumented programs exhibit equivalent policy-compliant behaviors in identical environments. However, IRM instrumentation may change the part of the environment that introspective queries consult, resulting in a permissible behavior change. As a simple example, an applet that prints its own size will print its new (usually larger) size after instrumentation. Even though this is an observable behavior change, it obeys the definition (and the spirit) of transparency because the program's *introspective logic* has been preserved by the IRM, even though the input to that logic has changed. Thus, our verifier proves that each applet's logic is not corrupted by the IRM, and it permits that logic to recognize and respond to the IRM's presence with different observable behavior. This is useful because it allows applets to intentionally acommmodate the IRM. For example, applets that discover the IRM's presence through introspection may voluntarily disable features that conflict with the local security policy.

4 Formal Approach

4.1 ActionScript Bytecode Core Subset

For expository simplicity, we express the verification algorithm and proof of correctness in terms of a small (but Turing-complete), stack-based toy language that includes standard arithmetic operations, conditional and unconditional jumps, integer-valued local registers, and the special instructions listed in Fig. 2. The implementation described in §5 supports the full ActionScript bytecode language (subject to the limitations in §3.6).

Instruction $\mathbf{api}_m n$ models a system API call, where m is a method identifier and n is the method's arity. Most API calls are assumed to be observable; these are modeled by an additional **apptrace** instruction that explicitly appends the API call event to the trace. Observable events can therefore be modeled as a macro $\mathbf{obsevent}_m n$ whose expansion is given in Fig. 3. In the rewritten code,

$\mathbf{api}_m n$	system API calls
$\mathbf{apptrace}_m n$	append to trace
$\mathbf{assert}_m n$	assert policy-adherence of event

Figure 2: Non-standard core language instructions

Original	Rewritten
$\mathbf{obsevent}_m n \equiv \mathbf{assert}_m n$ $\mathbf{apptrace}_m n$ $\mathbf{api}_m n$	$\mathbf{obsevent}_m n \equiv$ $\mathbf{apptrace}_m n$ $\mathbf{api}_m n$

Figure 3: Semantics of the **obsevent** pseudo-instruction

the expansion appends the event to the trace and performs the event. In the original code, it additionally asserts that the flow is unreachable if the event violates the policy. This models our premise that transparency obligations are waived when the IRM must intervene to prevent a violation, and prunes such flows from the verifier's search space.

The toy language models objects and their instance fields by reducing them to integer encodings, and exceptions are modeled as conditional branches in the typical way. A formal treatment of these is here omitted; their implementation in the transparency verifier is that of a standard abstract interpreter.

The trace accumulated by **apptrace** instructions is conceptual; it is not actually implemented and therefore not directly readable by programs. To track it, IRMs typically introduce reified state variables as described in §3.1.

4.2 Concrete and Abstract Machines

Concrete and abstract interpretation of programs is expressed as the small-step operational semantics of a concrete and an abstract machine, respectively. Figure 4 defines a concrete machine (program) state χ as a tuple consisting of a labeled bytecode instruction $L{:}i$, a concrete operand stack ρ, a concrete store σ, and a concrete trace of observable events τ. The store σ maps reified security state variables r and local variables ℓ to their integer values [50]. Abstract machine (program) states $\hat{\chi}$ are defined similarly, except that abstract stacks, stores, and traces are defined over symbolic expressions instead of values. Expressions include integer-valued *meta-variables* $\hat{v}$ and return values $\mathbf{rval}_m(e_1{::}\cdots{::}e_n)$ of API calls. Meta-variables $\hat{s}$ and $\hat{t}$ denote entire abstract stacks and traces, respectively.

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.

Since transparency verification involves bisimulating the original and instru-

L	(Code Labels)
i	(Instructions)
$P ::= (L, p, s)$	(Programs)
$p : L \to i$	(Instruction Labels)
$s : L \rightharpoonup L$	(Label Successors)
$m \in \mathbb{N}$	(Method Identifiers)
$n \in \mathbb{N}$	(Method Arities)
$\sigma : (r \uplus \ell) \rightharpoonup \mathbb{Z}$	(Concrete Stores)
$\rho ::= \cdot \mid x::\rho$	(Concrete Stacks)
$x \in \mathbb{Z}$	(Values)
$\tau ::= \epsilon \mid \tau\mathbf{api}_m(x_1::\cdots::x_n)$	(Concrete Traces)
$sys : \mathbb{N} \times \mathbb{Z}^* \to \mathbb{Z}$	(API Return Values)
$\mathfrak{a} : \tau \rightharpoonup \mathbb{N}$	(Security Automaton State)
$\chi ::= \langle L : i, \rho, \sigma, \tau\rangle$	(Concrete Configurations)
$e ::= n \mid \hat{v} \mid e_1{+}e_2 \mid \ldots \mid$ $\mathbf{rval}_m(e_1::\cdots::e_n) \mid \hat{\mathfrak{a}}(\hat{\tau})$	(Symbolic Expressions)
$\hat{v}, \hat{s}, \hat{t}$	(Value, Stack, & Trace Variables)
$\hat{\rho} ::= \cdot \mid \hat{s} \mid e::\hat{\rho}$	(Abstract Stacks)
$\hat{\sigma} : (r \uplus \ell) \to e$	(Abstract Stores)
$\hat{\tau} ::= \epsilon \mid \hat{t} \mid \hat{\tau}\mathbf{api}_m(e_1::\cdots::e_n)$	(Abstract Traces)
$\hat{\chi} ::= \langle L : i, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle$	(Abstract Configurations)
$\hat{\chi}_0 = \langle L_0 : p(L_0), \cdot, \hat{\sigma}_0, \epsilon\rangle$	(Initial Abstract Configurations)

Figure 4: Concrete and abstract program states

mented programs, Fig. 5 extends the concrete and abstract program states described above to *bisimulation machine* states. Each such bisimulation state includes both an original and a rewritten program state. The abstract bisimulation state additionally includes a constraint list ζ consisting of a conjunction of linear inequalities over expressions.

The concrete machine semantics are modeled after the ActionScript VM 2 (AVM2) semantics [3]; Fig. 6 shows the semantics of the special instructions of Fig. 2. Relation $\chi \mapsto_P^n \chi'$ denotes n steps of concrete interpretation of program P. Subscript P is omitted when the program is unambiguous, and when n is omitted it defaults to 1 step.

Rule CAPI models calls to the system API using an opaque function *sys* that maps method identifiers and arguments to return values. Any non-determinism in the system API is modeled by extending the prototypes of system API functions with additional arguments. Rule CBisim lifts the single-machine semantics to a

$$\zeta ::= \bigwedge_{i=1..n} t_i \quad (n \geq 1) \qquad \text{(Constraints)}$$

$$t ::= T \mid F \mid e_1 \leq e_2 \mid \hat{\tau}_1 = \hat{\tau}_2 \qquad \text{(Clauses)}$$

$$\Gamma = \langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \qquad \text{(Concrete Interpreter States)}$$

$$\hat{\Gamma} = \langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta \rangle \qquad \text{(Abstract Interpreter States)}$$

$$\langle \mathcal{C}, \langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle, \mapsto_P^n \rangle \qquad \text{(Concrete Interpreter)}$$

$$\langle \mathcal{A}, \langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle, \rightsquigarrow_P^n \rangle \qquad \text{(Abstract Interpreter)}$$

Figure 5: Concrete and abstract bisimulation machines

$$\frac{x' = sys(m, x_1{::}x_2{::}\cdots{::}x_n)}{\langle L : \mathbf{api}_m n, x_1{::}x_2{::}\cdots{::}x_n{::}\rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), x'{::}\rho, \sigma, \tau \rangle} \text{(CAPI)}$$

$$\frac{\rho = x_1{::}x_2{::}\cdots{::}x_n{::}\rho'}{\langle L : \mathbf{apptrace}_m n, \rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), \rho, \sigma, \tau \mathbf{api}_m(x_1{::}x_2{::}\cdots{::}x_n) \rangle} \text{(CAppTrace)}$$

$$\frac{\rho = x_1{::}\cdots{::}x_n{::}\rho' \qquad \tau \mathbf{api}_m(x_1{::}\cdots{::}x_n) \in \mathcal{P}}{\langle L : \mathbf{assert}_m n, \rho, \sigma, \tau \rangle \mapsto \langle s(L) : p(s(L)), \rho, \sigma, \tau \rangle} \text{(CAssert)}$$

$$\frac{\chi_i \mapsto_1 \chi_i' \qquad \chi_j = \chi_j' \qquad i \neq j}{\langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \mapsto \langle \chi_{\mathcal{O}}', \chi_{\mathcal{R}}' \rangle} \text{(CBisim)}$$

Figure 6: Concrete small-step operational semantics

bisimulation machine that non-deterministically chooses which machine to step next.

Figure 7 gives the corresponding semantics for abstract interpretation. Each step $\hat{\chi} \rightsquigarrow \hat{\chi}', \zeta$ of abstract interpretation yields both a new program state $\hat{\chi}'$ and a list ζ of new constraints. These are conjoined into the master list of constraints by rule ABisim.

Rule AAPI uses expression $\mathbf{rval}_m(\cdots)$ to abstractly denote the return value of API call m. Rule AAssert introduces a new constraint that asserts that appending API call m to the current trace yields a policy-adherent trace. The constraint uses the expression $\hat{\mathfrak{a}}(\hat{\tau}')$ to denote the security automaton state. Rule Abstraction allows the abstract interpreter to discard information at any point by abstracting the current state. This facilitates pruning the search space in response to hints from the invariant-generator. Discarding too much information can result in conservative rejection, but it never results in incorrect acceptance of non-transparent code.

4.3 Verification Algorithm

Algorithms 1–2 present our transparency verification algorithm in terms of the abstract interpretation semantics. Algorithm 2 verifies an individual abstract

$$\frac{e' = \mathbf{rval}_m(e_1 :: e_2 :: \cdots :: e_n)}{\langle L : \mathbf{api}_m n, e_1 :: e_2 :: \cdots :: e_n :: \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), e' :: \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle, T} (\text{AAPI})$$

$$\frac{\hat{\rho} = e_1 :: \cdots :: e_n :: \hat{\rho}'}{\langle L : \mathbf{apptrace}_m n, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\rho}, \hat{\sigma}, \hat{\tau}\mathbf{api}_m(e_1 :: \cdots :: e_n)\rangle, T} (\text{AAppTrace})$$

$$\frac{\zeta = \left(0 \leq \hat{\mathfrak{a}}(\hat{\tau}\mathbf{api}_m(e_1 :: \cdots :: e_n))\right)}{\langle L : \mathbf{assert}_m n, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle \rightsquigarrow \langle s(L) : p(s(L)), \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle, \zeta} (\text{AAssert})$$

$$\frac{\hat{\chi}_{\mathcal{O}} \subseteq \hat{\chi}'_{\mathcal{O}} \qquad \hat{\chi}_{\mathcal{R}} \subseteq \hat{\chi}'_{\mathcal{R}} \qquad \zeta \Rightarrow \zeta'}{\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta\rangle \rightsquigarrow \langle \hat{\chi}'_{\mathcal{O}}, \hat{\chi}'_{\mathcal{R}}, \zeta'\rangle} (\text{Abstraction})$$

$$\frac{\hat{\chi}_i \rightsquigarrow \hat{\chi}'_i, \zeta' \qquad \hat{\chi}_j = \hat{\chi}'_j \qquad i \neq j}{\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta\rangle \rightsquigarrow \langle \hat{\chi}'_{\mathcal{O}}, \hat{\chi}'_{\mathcal{R}}, \zeta \wedge \zeta'\rangle} (\text{ABisim})$$

Figure 7: Abstract small-step operational semantics

Algorithm 1 Verification

Input: *Cache* $= \{\}$, *Horizon* $= \{\hat{\Gamma}_0\}$
Output: *Accept* or *Reject*

1: **while** *Horizon* $\neq \emptyset$ **do**
2: $\quad\hat{\Gamma} \leftarrow$ *choose*(*Horizon*)
3: $\quad S_{\hat{\Gamma}} \leftarrow$ *VerificationSingleCodePoint*$(\hat{\Gamma})$
4: $\quad$**if** $S_{\hat{\Gamma}} =$ *Reject* **then return** *Reject*
5: $\quad$*Cache* $\leftarrow$ *Cache* $\cup \{\hat{\Gamma}\}$
6: $\quad$*Horizon* $\leftarrow$ (*Horizon* $\cup S_{\hat{\Gamma}}$)$\backslash$*Cache*
7: **end while**
8: **return** *Accept*

bisimulation state, and Algorithm 1 calls it as a subroutine to verify transparency of all reachable control-flows. We discuss each algorithm below.

Algorithm 1 takes as input a cache of previously explored abstract bisimulation states and a horizon of unexplored abstract bisimulation states. Upon successful verification of all control flows, it returns *Accept*; otherwise it returns *Reject*. It begins by drawing an arbitrary unexplored bisimulation state $\hat{\Gamma}$ from the *Horizon* (line 2) and passing it to Algorithm 2. Algorithm 2 returns a set $S_{\hat{\Gamma}}$ of abstract bisimulation states where bisimulation must continue in order to verify all control-flows proceeding from $\hat{\Gamma}$ (line 3). Every state of $S_{\hat{\Gamma}}$ that is not already in the *Cache* is added to the *Horizon* (line 6). Verification concludes when all states in the *Horizon* have been explored.

Algorithm 2 takes an abstract bisimulation state $\hat{\Gamma}$ as input. It begins by asking the invariant-generator for a hint (line 1), consisting of: (1) a new (possibly more abstract) bisimulation state $\hat{\Gamma}_H$ for $\hat{\Gamma}$, (2) a finite, *generalized post-dominating set* $D_{\hat{\Gamma}}$ for $\hat{\Gamma}$ whose members are all transparent code points, and (3) a stepping-bound n. A set S of abstract bisimulation states is said to be generalized post-dominating for $\hat{\Gamma}$ if every complete control-flow that includes $\hat{\Gamma}$ later includes at least one member of S [29]. In our case, the complete flows

Algorithm 2 VerificationSingleCodePoint

Input: $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$
Output: $S_{\hat{\Gamma}}$ or *Reject*

1: $(\hat{\Gamma}_H, D_{\hat{\Gamma}}, n) \leftarrow \mathit{InvariantGen}(\hat{\Gamma})$
2: $\mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}, \hat{\Gamma}_H)$
3: **if** $\mathit{SatValue} = \mathit{Reject}$ **then return** *Reject*
4: $S_{\hat{\Gamma}} \leftarrow \mathit{AbsIn}^n(\{\hat{\Gamma}_H\}, D_{\hat{\Gamma}})$
5: **if** $\mathit{labels}(S_{\hat{\Gamma}}) \not\subseteq D_{\hat{\Gamma}}$ **then return** *Reject*
6: **for each** $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle \in S_{\hat{\Gamma}}$ **do**
7: $\quad \langle \_, \_, \_, \hat{\tau}'_1 \rangle = \hat{\chi}'_1$
8: $\quad \langle \_, \_, \_, \hat{\tau}'_2 \rangle = \hat{\chi}'_2$
9: $\quad \mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}', \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \wedge (\hat{\tau}'_1 = \hat{\tau}'_2) \rangle)$
10: $\quad$ **if** $\mathit{SatValue} = \mathit{Reject}$ **then return** *Reject*
11: **end for**
12: **return** $S_{\hat{\Gamma}}$

are the infinite ones (since termination is modeled as an infinite stutter state). The stepping bound n is an upper bound on the number of steps required to reach any state in $D_{\hat{\Gamma}}$ from $\hat{\Gamma}_H$. Note that we express the stepping-bound here as a single integer for simplicity. For efficient implementation, the bound can be replaced with a pair of integers (n_1, n_2), n_i representing the exact number of steps machine i should step in order to reach any state in $D_{\hat{\Gamma}}$ from $\hat{\Gamma}_H$.

The hint obtained in line 1 is not trusted; it must therefore be verified. To do so, model-checking first confirms that $\hat{\Gamma}_H$ is a sound abstraction of $\hat{\Gamma}$ according to the ABSTRACTION rule of the operational semantics (see Fig. 7). Next, it abstract interprets for n steps from $\hat{\Gamma}$ to confirm post-dominance of $D_{\hat{\Gamma}}$. Function *AbsIn* in line 4 is defined by

$$\begin{aligned}\mathit{AbsIn}(S, E) = &\{\hat{\Gamma}' \mid \hat{\Gamma} \in S, \mathit{labels}(\hat{\Gamma}) \notin E, \hat{\Gamma} \rightsquigarrow \hat{\Gamma}'\} \cup \\ &\{\hat{\Gamma} \in S \mid \mathit{labels}(\hat{\Gamma}) \in E\}\end{aligned}$$

where $\mathit{labels}(\langle L_1 \colon \ldots \rangle, \langle L_2 \colon \ldots \rangle) = (L_1, L_2)$ extracts the code labels of an abstract bisimulation state. Function $\mathit{AbsIn}(S, E)$ therefore abstract interprets all states in S for one step, except that states already at end code-points in E are not interpreted further. Finally, the model-checker confirms transparency of all members of $S_{\hat{\Gamma}}$ (line 10). If successful, set $S_{\hat{\Gamma}}$ is returned.

4.4 Model-Checking

Verification of abstract bisimulation states suggested by the invariant-generator, pruning of policy-violating flows, and verification of transparency are all reduced by Algorithm 2 to proving implications of the form $A \Rightarrow B$. These are proved by the two-stage model-checking procedure in Algorithm 3, consisting of unification followed by linear constraint solving.

Algorithm 3 ModelCheck

Input: $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$, $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle$
Output: *Accept* or *Reject*
1: $\zeta_U \leftarrow \mathit{Unify}(\hat{\Gamma}, \hat{\Gamma}')$
2: **if** $\zeta_U = \mathit{Fail}$ **then return** *Reject*
3: $\mathit{SatValue} \leftarrow \mathit{CLP}(\zeta_U \wedge \zeta \wedge \neg\zeta')$
4: **if** $\mathit{SatValue} = \mathit{False}$ **then**
5: **return** *Accept*
6: **else**
7: **return** *Reject*
8: **end if**

Unification Each abstract state $\hat{\chi}$ can be viewed as a set of equalities that relate state components to their values. Many of these equalities relate structures; for example, each operand stack is an ordered list of expressions. Given two abstract bisimulation states $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$ and $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle$, the model-checker first uses Prolog unification to mine all structural equalities for equalities over their contents. For example, if $\hat{\rho}_1 = e_1 :: \hat{s}$ and $\hat{\rho}'_1 = e'_1 :: e'_2 :: \hat{s}'$, then unification infers $e_1 = e'_1$ and $\hat{s} = e'_2 :: \hat{s}'$. If unification fails, the model-checker rejects. Successful unification yields a collection ζ_U of purely integer equalities.

Linear Constraint Solving The model-checker then verifies implication $\zeta \Rightarrow \zeta'$ by applying constraint logic programming (CLP) to verify the unsatisfiability of sentence $\zeta_U \wedge \zeta \wedge \neg\zeta'$. That is, it confirms that under the hypothesis ζ_U that $\hat{\Gamma}$ and $\hat{\Gamma}'$ abstract the same concrete bisimulation state, there is no instantiation of the free variables that falsifies $\zeta \Rightarrow \zeta'$.

The constraints that populate lists ζ arise from four major sources: conditional branches, observable events in the original code, trace-equality checks, and hints provided by the invariant-generator. Each plays an important role in practical transparency verification, so we discuss each.

Conditional branches introduce constraints of the form $e_1 \leq e_2$ to the abstract bisimulation state. These are important for verifying that some outgoing branches from IRM conditional guards are only traversed when the original code violates the policy; the transparency obligation is therefore waived for such flows. This permits IRMs to implement transparency-violating intervention code as long as it does not observably affect non-violating runs of the program.

Observable events in the original code introduce constraints of the form $0 \leq \hat{\mathfrak{a}}(\hat{\tau})$ (see rule AAssert of Fig. 7) which assert that the security automaton that encodes the policy is in a well-defined state (i.e., it has not rejected the original program). This prunes flows in which the original program violates security, avoiding conservative rejection of rewritten code that intervenes appropriately.

Trace-equality checks are performed by line 9 of Algorithm 2, which queries the model-checker with constraints of the form $\hat{t}_1 = \hat{t}_2$. These inquire whether equivalence of two traces is provable from information that is known about the current bisimulation state. If the constraint is falsifiable, the verifier conserva-

tively rejects.

The untrusted invariant-generator may also introduce constraints that encode loop invariants relevant to proving transparency. This is critical for tractably exploring the full state space and keeping the TCB small. Invariant-generation is discussed in greater detail in the next subsection.

4.5 Invariant Generation

Recall from §4.3 that for every reachable code point (L_1, L_2) in the bisimulation state space, the verifier requires an (untrusted) hint consisting of: (1) an invariant for (L_1, L_2) in the form of an abstract bisimulation state, (2) a finite, generalized post-dominating set for (L_1, L_2) whose members are all transparent code points, and (3) a stepping-bound n. These hints may come from any untrusted source, since they are independently verified by the trusted model-checker. This is important for both verifier generality and TCB minimization. General-purpose invariant-generation is well known to be difficult, so it is unlikely that any one invariant-generation strategy suffices for all IRMs. Shifting invariant-generation outside the TCB addresses this dilemma by allowing it to be tailored to a specific rewriting algorithm without re-proving correctness of the verification algorithm for each new invariant-generator. In this section we outline a strategy for generating these invariants that suffices for the SPoX rewriting system [30], and that can be used as a basis for transparency verification of many other similar IRM systems.

SPoX implements IRMs as collections of small code blocks that guard security-relevant operations. It also introduces new classes and methods that maintain and track reified security state variables implemented as private class fields. The relationship between the reified state and the security state of the program constitutes an invariant, termed *synchronization* (SYNC), that has been used to verify its soundness [33]. We observe that extending this invariant with an obligation to restore trace-equivalence at a certain subset of synchronized points suffices to also verify transparency.

Algorithms 4–5 generate such invariants by consulting a set of *Marked* code labels that the IRM claims it has semantically modified. Marked regions include IRM guard code and the security-relevant instructions they guard, but not IRM intervention code that responds to impending violations. (Interventions remain unmarked since the verifier proves them unreachable when the original code satisfies the policy.) The invariant-generator chooses which invariant to return depending on whether the bisimulation state is marked.

Outside marked regions it uses Algorithm 4 to generate a hint that asserts that the original and rewritten machines are step-wise equivalent. That is, all original and rewritten state components are equal except for state introduced by the IRM. It additionally asserts that reified state variables introduced by the IRM accurately encode the current security state; this is captured by clause $r = \hat{\mathfrak{a}}(\hat{t}_{\mathcal{R}})$ in line 6. This property is necessary to prove that interventions are unreachable and therefore exempt from transparency.

Algorithm 4 GenericInvariant

Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$

1: choose fresh meta-variables $\hat{v}_\ell$, $\hat{v}'_\ell$, $\hat{s}$, $\hat{s}'$, $\hat{t}$, and $\hat{t}'$
2: $\hat{\sigma} \leftarrow \{(\ell, \hat{v}_\ell) \mid \hat{\sigma}_1(\ell) = e_1\}$
3: $\hat{\sigma}' \leftarrow \{(\ell, \hat{v}'_\ell) \mid \hat{\sigma}_2(\ell) = e_2\} \cup \{(r, \hat{\sigma}_2(r))\}$
4: $\hat{\chi} \leftarrow \langle L_1 : i_1, \hat{s}, \hat{\sigma}, \hat{t} \rangle$
5: $\hat{\chi}' \leftarrow \langle L_2 : i_2, \hat{s}', \hat{\sigma}', \hat{t}' \rangle$
6: $\zeta = (\hat{s}{=}\hat{s}') \wedge \left(\bigwedge_{\ell \in \hat{\sigma} \leftarrow \cap \hat{\sigma}' \leftarrow} \hat{v}_\ell{=}\hat{v}'_\ell\right) \wedge (r{=}\hat{\mathfrak{a}}(\hat{t}')) \wedge (\hat{t}{=}\hat{t}')$
7: **return** $\mathbf{I} = \langle \hat{\chi}, \hat{\chi}', \zeta \rangle$

Algorithm 5 InvariantGen

Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$
Output: $\hat{\Gamma}_H$,$D_{\hat{\Gamma}}$,n

1: **if** $L_2 \in \mathit{Marked}$ **then**
2: **return** $(\mathit{GenericInvariant}(\hat{\chi}_1, \hat{\chi}_2), \{(L_1, L_2)\}, 1)$
3: **else**
4: $\hat{\Gamma} \leftarrow \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$
5: $n \leftarrow \min\{n \mid \mathit{AbsIn}^n(\{\hat{\Gamma}\}, \mathit{Marked}) = \mathit{AbsIn}^{n+1}(\{\hat{\Gamma}\}, \mathit{Marked})\}$
6: **return** $(\hat{\Gamma}, \mathit{labels}(\mathit{AbsIn}^n(\{\hat{\Gamma}\}, \mathit{Marked})), n)$
7: **end if**

Within marked regions, the invariant-generator uses the last half of Algorithm 5, which asserts that transparency is restored once bisimulation exits the marked region. To prove that execution does eventually exit the marked region, line 5 uses abstract interpretation to find each control-flow's exit point. As mentioned in §3.6, IRMs implementing non-trivial loops outside of interventions may cause this step to conservatively fail.

While the invariant-generation algorithm presented here is specific to SPoX, it can be adapted to suit other similar instrumentation algorithms by replacing constraint $r = \hat{\mathfrak{a}}(\hat{t}_{\mathcal{R}})$ in Algorithm 4 with a different constraint that models the way in which the IRM reifies the security state. Similarly, appeals to the *Marked* set can be replaced with alternative logic that identifies code points where the transparency invariant is restored after the IRM has completed any maintenance associated with security-relevant operations.

4.6 A Verification Example

To illustrate, we briefly describe transparency verification of the simple pseudo-bytecode listing in Fig. 8, which implements an IRM that prohibits more than 100 calls to security-relevant method `NavigateToURL`. The listing on the left is the original program, and the one on the right is the IRM-instrumented one. Lines with an × are those in the *Marked* set described in §4.5. This IRM tracks the number of calls to `NavigateToURL` in global field c. (Real IRMs are typically much more complex, but we use this simplified example for clarity.)

The abstract interpreter begins exploring the cross-product space from point $(L1, L4)$, where both original and rewritten programs start. The initial state

L1: **push** `"http:// ..."`		L4: **push** `"http:// ..."`
	×	L5: **get** c
	×	L6: **iflt** 100, L8 // if $c \leq 100$ goto L8
		L7: **call** exit
L2: **call** NavigateToURL	×	L8: **call** NavigateToURL
	×	L9: **get** c
	×	L10: **push** 1
	×	L11: **add**
	×	L12: **set** c
L3: **jmp** L1		L13: **jmp** L4

Figure 8: An IRM that prohibits more than 100 URL navigations

asserts equivalence of all state components except c, which does not exist in the original code and is initialized to 0 in the rewritten code. The (untrusted) invariant-generator suggests a hint that abstracts this to a clause of the form $c = \hat{\mathfrak{a}}(\hat{t})$, where meta-variable $\hat{t}$ denotes the current trace. This invariant recommends that the only information necessary at point $(L1, L4)$ to infer transparency is that c correctly reflects the security automaton state. Since initially $\hat{t} = \hat{\tau} = \epsilon$ in both machines, the verifier confirms that this abstraction hint is sound and uses it henceforth as an invariant for point $(L1, L4)$.

The invariant-generator next supplies a post-dominating set $\{(L3, L13)\}$ and stepping bound 11, which asserts that all realizable flows from $(L1, L4)$ take both machines to $(L3, L13)$ within 11 steps. The verifier confirms this by non-deterministically interpreting all paths from $(L1, L4)$ for 11 steps. When interpretation reaches the conditional at $L6$, clause $c = \hat{\mathfrak{a}}(\hat{t})$ is critical for inferring that $L7$ is unreachable when the original code satisfies the policy. Specifically, the policy-adherence assumption yields constraint $\hat{\mathfrak{a}}(\hat{\tau}) < 100$ after $L8$, which contradicts negative branch condition $c \geq 100$ introduced by $L7$ of the rewritten code when $c = \hat{\mathfrak{a}}(\hat{\tau})$.

Once the interpreter reaches point $(L3, L13)$, the invariant-generator supplies the same abstract bisimulation state as it did for $(L1, L4)$. That is, the state opaquely asserts that all common state components (including traces) are equal, and reified state c equals security automaton state $\hat{\mathfrak{a}}(\hat{t})$. The linear constraint solver confirms that the incremented c ($L11$) matches the incremented state $\hat{\mathfrak{a}}(\hat{t}\,\mathbf{api}_{\text{NavigateToURL}})$, and therefore accepts the new invariant. Abstract interpreting for two additional steps, it confirms that this matches the loop invariant supplied for $(L1, L4)$, and accepts the program-pair as transparent.

4.7 Proof of Verifier Correctness

Theorem 1 reduces correctness of the transparency verification algorithm to soundness of the abstract interpreter and model-checker. That is, if the verifier's abstract bisimulation of the two programs (including the interpretation rules that perform model-checking) soundly abstracts the programs' actual, concrete executions, and if the verifier accepts, then the set S described in Theorem 1

exists, and we conclude (from Theorem 1) that the IRM is transparent.

This notion of soundness can be formally defined in terms of a denotational semantics $\mathcal{D}$ of abstract states given in Fig. 9. *Soundness relation* $\sim_{\mathbf{T}} \subseteq \mathcal{C} \times \mathcal{A}$ is then defined by

$$\frac{\langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \in \mathcal{D}[\![\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta \rangle]\!]}{\langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \sim_{\mathbf{T}} \langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta \rangle} (\textsc{Sound})$$

Following the approach of [12], soundness of the abstract interpreter is proved via two lemmas that establish preservation and progress (respectively) 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 interpreter covers all realizable flows. Together, these two lemmas dovetail to form an induction over arbitrary length execution sequences. Both lemmas are proved by induction over the respective operational semantics (Figs. 6 and 7). Soundness of the model-checker follows from soundness of the Prolog CLP engine [11].

The full proofs are lengthy, so we sketch only the most interesting cases below.

Lemma 1 (Progress) *For all $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim_{\mathbf{T}} \hat{\chi}$, if there exists $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$, then there exists $\hat{\chi}' = \langle L_{\hat{\chi}'} : i_{\hat{\chi}'}, \hat{\rho}', \hat{\sigma}', \hat{\tau}' \rangle \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$.*

Proof 2 *The only non-trivial case is the one for* **assert**. *In that case, the second premise of rule* CAssert *(Fig. 6) proves that the asserted event does not violate the policy. This suffices to prove that the denotation of $\hat{\mathfrak{a}}$ in the premise of rule* AAssert *(Fig. 7) is well-defined, and therefore the abstract machine takes a corresponding step.*

Lemma 2 (Preservation) *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim_{\mathbf{T}} \hat{\chi}$, 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 $\chi' \sim_{\mathbf{T}} \hat{\chi}'$.*

Proof 3 *The proof first eliminates the* Abstraction *rule from consideration through a structural cut elimination argument (cf., [45]). Two interesting cases remain: that for* **api** *instructions, and that for* **assert** *instructions. The case for* **assert** *is similar to Proof 2. Preservation of* **api** *instructions follows from relating the denotation of the* **rval** *expression in the premise of rule* AAPI *(Fig. 7) to the system-modeling function sys in the premise of rule* CAPI *(Fig. 6). This follows from the definition of $\mathcal{E}[\![\mathbf{rval}_m(e_1 :: \cdots :: e_n)]\!]$ in Fig. 9.*

Theorem 2 (Soundness) *Starting from the initial abstract interpreter state, $\langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle$, if the abstract interpreter does not reject, then, for each concrete interpreter state in each realizable flow starting from the initial concrete interpreter state $\langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle$, there exists an abstract interpreter state that soundly abstracts that concrete state.*

Proof 4 *By the definition of abstract interpreter acceptance, starting from initial state* $\langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle$ *the abstract interpreter continually makes progress. By a trivial induction over the set of finite prefixes of this abstract transition chain, the progress and preservation lemmas prove that the concrete interpreter also continually makes progress from initial state* $\langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle$*, and the abstract interpreter infers a sound abstraction at every code point.*

5 Implementation and Results

Our implementation of the transparency verification algorithm detailed in §4 targets the full ActionScript bytecode language. It consists of 2500 lines of Prolog for 32-bit Yap 6.2 that parses and verifies pairs of Shockwave Flash File (SWF) binary archives. YAP CLP(R) [35] is used for constraint solving and Yap's tabling for memoization. We have made our tool, called FlashTrack (Flash TRAnsparency ChecKer), available for download online [52].

IRM instrumentation is accomplished via a collection of small binary-to-binary rewriters. They each augment untrusted ActionScript code with security guards according to a security policy, specified as a SPoX security automaton. For ease of implementation, each rewriter is specialized to a particular policy class. For example, one rewriter enforces *resource bound* policies that limit the number of accesses to policy-specified system API functions per run. It augments untrusted code with counters that track accesses, and halts the applet when an impending operation would exceed the bound. The rewriters are each about 200 lines of Prolog (not including parsing) and the invariant-generaters are about 100 lines each.

Each rewriter is accompanied by an invariant-generator that follows the algorithm described in §4.5. The generated invariants match the details of each rewriter's code-transformation strategy, exhibiting no conservative rejection that we know of for any code that the rewriters produce. We expect that adapting invariant generation to other similar IRM systems will only require small modifications.

To demonstrate the versatility of FlashTrack, rewriters in our framework perform localized binary optimizations during rewriting when convenient. For example, when original code followed by IRM code forms a sequence of consecutive conditional branches, the entire sequence (including the original code) is replaced with an ActionScript multi-way jump instruction (`lookupswitch`). Certifying transparency of the instrumented code therefore requires the verifier to infer semantic equivalence of these transformations.

When implementing our IRMs we found the transparency verifier to be a significant aid to debugging. Bugs that we encountered included IRMs that fail transparency when in-lined into unusual code that overrides IRM-called methods (e.g., `toString`), IRMs that throw uncaught exceptions (e.g., null pointer) in rare cases, IRMs that inadvertently trigger class initializer code that contains an observable operation, and broken IRM instructions that corrupt a register or

$$I \in \mathbb{I} = (\hat{v} \to \mathbb{Z}) \cup (\hat{s} \to \rho) \cup (\hat{t} \to \tau) \qquad \text{(Interpretations)}$$

$$\mathcal{E} : e \to \mathbb{I} \to \mathbb{Z} \qquad \text{(Expression Denotations)}$$

$$\mathcal{E}[\![n]\!]I = n$$

$$\mathcal{E}[\![\hat{v}]\!]I = I(\hat{v})$$

$$\mathcal{E}[\![e_1 + e_2]\!]I = \mathcal{E}[\![e_1]\!]I + \mathcal{E}[\![e_2]\!]I$$

$$\mathcal{E}[\![\mathbf{rval}_m(e_1 :: \cdots :: e_n)]\!]I = sys(m, \mathcal{E}[\![e_1]\!]I :: \cdots :: \mathcal{E}[\![e_n]\!]I)$$

$$\mathcal{E}[\![\hat{\mathfrak{a}}(\hat{\tau})]\!]I = \mathfrak{a}(\mathcal{E}[\![\hat{\tau}]\!]I)$$

$$\mathcal{D}_{\mathrm{K}} : \hat{\rho} \to \mathbb{I} \to \rho \qquad \text{(Stack Denotations)}$$

$$\mathcal{D}_{\mathrm{K}}[\![\cdot]\!]I = \cdot$$

$$\mathcal{D}_{\mathrm{K}}[\![\hat{s}]\!]I = I(\hat{s})$$

$$\mathcal{D}_{\mathrm{K}}[\![e :: \hat{\rho}]\!]I = \mathcal{E}[\![e]\!]I :: \mathcal{D}_{\mathrm{K}}[\![\hat{\rho}]\!]I$$

$$\mathcal{D}_{\mathrm{S}} : \hat{\sigma} \to \mathbb{I} \to \sigma \qquad \text{(Store Denotations)}$$

$$\mathcal{D}_{\mathrm{S}}[\![\hat{\sigma}]\!]Ix = \mathcal{E}[\![\hat{\sigma}(x)]\!]I \qquad \text{(Store Denotations)}$$

$$\mathcal{D}_{\mathrm{T}} : \hat{\tau} \to \mathbb{I} \to \tau \qquad \text{(Trace Denotations)}$$

$$\mathcal{D}_{\mathrm{T}}[\![\epsilon]\!]I = \epsilon$$

$$\mathcal{D}_{\mathrm{T}}[\![\hat{t}]\!]I = I(\hat{t})$$

$$\mathcal{D}_{\mathrm{T}}[\![\mathbf{api}_m(e_1 :: \cdots :: e_n)\hat{\tau}]\!]I = \mathbf{api}_m(\mathcal{E}[\![e_1]\!]I :: \cdots :: \mathcal{E}[\![e_n]\!]I)\mathcal{D}_{\mathrm{T}}[\![\hat{\tau}]\!]$$

$$\mathcal{D}_{\mathrm{C}} : \hat{\chi} \to \mathbb{I} \to \chi \qquad \text{(Config. Denotations)}$$

$$\mathcal{D}_{\mathrm{C}}[\![\langle L : i, \hat{\rho}, \hat{\sigma}, \hat{\tau}\rangle]\!]I = \langle L : i, \mathcal{D}_{\mathrm{K}}[\![\hat{\rho}]\!]I, \mathcal{D}_{\mathrm{S}}[\![\hat{\sigma}]\!]I, \mathcal{D}_{\mathrm{T}}[\![\hat{\tau}]\!]I\rangle$$

$$\mathcal{T} : e \to 2^{\mathbb{I}} \qquad \text{(Term Denotations)}$$

$$\mathcal{T}[\![T]\!] = \mathbb{I}$$

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

$$\mathcal{T}[\![e_1 \leq e_2]\!] = \{I \mid \mathcal{E}[\![e_1]\!]I \leq \mathcal{E}[\![e_2]\!]I\}$$

$$\mathcal{C} : \zeta \to 2^{\mathbb{I}} \qquad \text{(Constraint Denotations)}$$

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

$$\mathcal{D}[\![\langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta\rangle]\!] = \{\langle \mathcal{D}_{\mathrm{C}}[\![\hat{\chi}_{\mathcal{O}}]\!]I, \mathcal{D}_{\mathrm{C}}[\![\hat{\chi}_{\mathcal{R}}]\!]I\rangle \mid I \in \mathcal{C}[\![\zeta]\!]\} \qquad \text{(Bisim. State Denotations)}$$

Figure 9: Denotational semantics for verifier states

Table 1: Experimental Results

Program	Policy	File Size (KB) old	File Size (KB) new	Number of Methods	Rewriting Time (ms)	Verification Time (ms)
adult1	ResBnds	1	2	4	< 1	< 1
adult2		18	18	102	127	1201
atmos		1	1	6	< 1	< 1
att		22	22	147	156	1434
ecls		2	3	6	16	< 1
eco		2	3	6	< 1	16
flash		3	4	12	< 1	62
fxcm		2	2	12	16	16
gm		21	22	142	157	1245
gucci		2	2	6	15	16
iphone		2	2	6	< 1	< 1
IPLad		2	2	15	31	15
jlopez		17	17	151	95	560
lowes		34	34	181	218	16549
men1		33	34	237	203	3757
men2		40	40	270	297	4964
prius		71	71	554	516	10359
priusm		70	71	542	468	9951
sprite		34	34	324	234	3075
utv		21	21	155	151	1171
verizon1		3	4	25	< 1	37
verizon2		3	3	12	31	15
weightwatch		4	4	34	47	47
wines		185	185	926	904	35926
expandall	NoExpands	3	4	17	47	79
cookie	NoCookieSet	3	3	8	31	16
CookieSet		1	1	4	< 1	< 1
HeapSprAttk	NoHeapSpray	1	1	4	15	15

stack slot that flows to an observable operation. All of these were immediately detected by the transparency verifier.

We applied our prototype framework to rewrite and verify numerous real-world Flash ads drawn from public web sites. The results are summarized in Table 1. For each ad, the table columns report the policy type, bytecode size before and after rewriting, the number of methods in the original code, and the rewriting and verification times. All tests were performed on a Lenovo Z560 notebook computer running Windows 7 64-bit with an Intel Core I5 M480 dual core processor, 2.67 GHz processor speed, and 4GB of memory.

Except for `HeapSprayAttack` (a synthetic attack discussed below) all tested ads were being served by public web sites when we collected them. Some came from popular business and e-commerce websites, but the more obtrusive ones with potentially undesirable actions tended to be hosted by less reputable sites, such as adult entertainment and torrent download pages. Potentially undesirable actions include unsolicited URL redirections, large pop-up expansions, tracking cookies, and excessive memory usage. Ad complexity was not necessarily indicative of maliciousness; some of the most complex ads were benign. For example, `wine` implements complex interactive menus showcasing wines and ultimately offering navigation to the seller's site.

All programs are classified into one of four case study classes:

Bounding URL Navigations We enforced a resource bound policy that restricts the number of times an ad may navigate away from the hosting site. This helps to prevent unwanted chains of pop-up windows. The IRM enforces the policy by counting calls to the `NavigateToURL` system API function. When an impending call would exceed the bound, the call is suppressed at runtime by a conditional branch. To verify transparency of the resulting IRM, the verifier proves that such branches are only reachable in the event of a policy violation by the original code.

Bounding Cookie Storage For another resource bounds policy, we limited the number of cookie creations per ad. This was achieved by guarding calls to the `SetCookie` API function. Impending violations cause the IRM to prematurely halt the applet.

Preventing Pop-up Expansions Some Flash ads expand to fill a large part of the web page whenever the user clicks or mouses over the ad space. This is frequently abused for click-jacking. Even when ad clicks solicit non-malicious behavior, many web publishers and users regard excessive expansion as a denial-of-service attack upon the embedding page. There is therefore high demand for a means of disabling it. Our expansion-disabling policy does so by denying access to the `GoToAndPlay` system API function.

Heap Spray Attacks *Heap spraying* is a technique for planting malicious payloads by allocating large blocks of memory containing sleds to dangerous code. Cooperating malware (often written in an alternative, less safe language) can then access the payload to do damage, for example by exploiting a buffer overrun to jump to the sled. By separating the payload injector and exploit code in different applications, the attack becomes harder to detect.

ActionScript has been used as a heap spraying vehicle in several past attacks [27]. The spray typically allocates a large byte array and inserts the payload into it one byte at a time, making it more difficult to reliably detect the payload's signature via purely static inspection of the ActionScript binary.

To inhibit heap sprays, we enforced a policy that bounds the number of byte-write operations that an ad may perform on any given run. We then implemented a heap spray (`HeapSprAttk`) and verified that the IRM successfully prevented the attack. Applying the policy to all other ads in Table 1 resulted in no behavioral changes, as confirmed by the verifier.

6 Conclusions and Future Work

Concerns about program behavior-preservation (transparency) have impeded the practical adoption of IRM systems for enforcing mobile code security. Code producers and consumers both desire the powerful and flexible policy-enforcement offered by IRMs, but are unwilling to accept unintended corruption of non-malicious program behaviors.

To address these concerns, we presented the design and implementation of the first automated transparency-verifier for IRMs, and demonstrated how safety-verifiers based on model-checking can be extended in a natural way to additionally verify IRM transparency. To minimize the TCB and keep verification tractable, an untrusted, external invariant-generator safely leverages rewriter-specific instrumentation information during verification. Hints from the invariant-generator reduce the state-exploration burden and afford the verifier greater generality than the more specialized rewriting systems it checks. Prolog unification and Constraint Logic Programming (CLP) keeps the verifier implementation simple and closely tied to the underlying verification algorithm, which is supported by proofs of correctness and abstract interpretation soundness. Practical feasibility is demonstrated through experiments on a variety of real-world ActionScript bytecode applets.

Future work should test the generality of the approach by applying automated transparency-verification to other IRM systems, such as those that target Java bytecode. Since Java includes concurrency, such work should consider the open problem of tractable transparency-checking for multithreaded bytecodes.

Recent work has also observed that in some usage scenarios IRMs must additionally preserve the behavior of *policy-violating* traces up to the point of violation; this is called *corrective enforcement* [36]. Our verifier imposes no transparency obligation on violating flows, but could be extended to do so. Future work should also investigate such an extension to support this class of IRMs.

Acknowledgments

The research reported herein was supported in part by NSF awards #1054629 and #1065216, and by AFOSR award FA9550-10-1-0088. Any opinions, recommendations, or conclusions expressed are those of the authors and do not necessarily reflect those of the NSF or AFOSR.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Informations and Systems Security*, 13(1), 2009.

[3] ActionScript Virtual Machine 2 Overview. ActionScript virtual machine 2 overview, 2007. http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf.

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

[5] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proceedings of the International Symposium on Formal Methods*, pages 262–277, 2008.

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

[7] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the Computer Security Foundations Workshop*, pages 100–114, 2004.

[8] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 305–314, 2005.

[9] Bruno Blanchet, Martín Abadi, and Cédric Fournet. Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming*, 75(1):3–51, 2008.

[10] Jan Olaf Blech, Yliès Falcone, and Klaus Becker. Towards certified runtime verification. In *Proceedings of the International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, pages 494–509, 2012.

[11] Egon Börger and Rosario F. Salamone. CLAM specification for provably correct compilation of CLP(R) programs. In Egon Börger, editor, *Specification and Validation Methods*, pages 96–130. Oxford University Press, 1995.

[12] Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. A framework for certified program analysis and its applications to mobile-code safety. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 174–189, 2006.

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

[14] Vincent Cheval, Hubert Comon-Lundh, and Stéphanie Delaune. Trace equivalence decision: negative tests and non-determinism. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 321–330, 2011.

[15] Andrey Chudnov and David A. Naumann. Information flow monitor inlining. In *Proceedings of the Computer Security Foundations Symposium*, pages 200–214, 2010.

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

[17] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreated Java. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 546–569, 2009.

[18] Daniel S. Dantas and David Walker. Harmless advice. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 383–396, 2006.

[19] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A rewriting framework for in-app reference monitors for Android applications. In *IEEE Mobile Security Technologies*, 2012.

[20] Stéphanie Delaune, Steve Kremer, and Mark D. Ryan. Symbolic bisimulation for the applied pi-calculus. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 133–145, 2007.

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

[22] Mark Dowd. Application-specific attacks: Leveraging the ActionScript virtual machine. IBM Global Technology Services, White Paper, April 2008.

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

[24] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, 1999.

[25] Ulfar Erlingsson, Martin Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 75–88, 2006.

[26] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[27] FireEye. Heap spraying with ActionScript, 2009. http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html.

[28] Matthew Fredrikson, Richard Joiner, Somesh Jha, Thomas Reps, Phillip Porras, Hassen Saïdi, and Vinod Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proceedings of the International Conference on Computer Aided Verification*, pages 548–563, 2012.

[29] Rajiv Gupta. Generalized dominators and post-dominators. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 246–257, 1992.

[30] Kevin W. Hamlen and Micah Jones. Aspect-oriented in-lined reference monitors. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, pages 11–20, 2008.

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

[32] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, pages 7–16, 2006.

[33] Kevin W. Hamlen, Micah M. Jones, and Meera Sridhar. Aspect-oriented runtime monitor certification. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 126–140, March–April 2012.

[34] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, pages 503–581, 1994.

[35] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14:339–395, May 1992.

[36] Raphael Khoury and Nadia Tawbi. Corrective enforcement: A new paradigm of security policy enforcement by monitors. *ACM Transactions on Information and Systems Security*, 15(2), 2012.

[37] Gregor Kiczales, John Lamping, Anurag Medhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[38] Moonzoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[39] Zhou Li and XiaoFeng Wang. FIRM: Capability-based inline mediation of Flash behaviors. In *Proceedings of the Annual Computer Security Applications Conference*, pages 181–190, 2010.

[40] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the European Symposium on Research in Computer Security*, pages 355–373, 2005.

[41] Mitre. CVE-2010-2216, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2216.

[42] Mitre. CVE-2010-2215, 2010. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2215.

[43] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 83–94, 2000.

[44] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[45] Frank Pfenning. Structural cut elimination. In *Proc. Annual Sym. Logic in Comp. Sci. (LICS)*, pages 156–166, 1995.

[46] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–166, 1998.

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

[48] Leon Shapiro and Ehud Y. Sterling. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, 1994.

[49] Meera Sridhar and Kevin W. Hamlen. In-lined reference monitoring in Prolog. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, pages 149–151, 2010.

[50] Meera Sridhar and Kevin W. Hamlen. Model-checking in-lined reference monitors. In *Proceedings of the International Conference on Verification, Model-Checking and Abstract Interpretation*, pages 312–327, 2010.

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

[52] Meera Sridhar, Richard Wartell, and Kevin W. Hamlen. FlashTrack: A transparency checker tool for Flash applications. http://code.google.com/p/flashtrack, 2013.

[53] Alwen Tiu and Jeremy E. Dawson. Automating open bisimulation checking for the spi calculus. In *Proceedings of the Computer Security Foundations Symposium*, pages 307–321, 2010.

[54] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

[55] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–249, 2007.

[56] Anna Zaks and Amir Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proceedings of the International Symposium on Formal Methods*, pages 35–51, 2008.