Hippocratic Binary Instrumentation: First Do No Harm

Meera Sridhar[a,*], Richard Wartell[b], Kevin W. Hamlen[a]

[a] *University of Texas at Dallas, TX, USA*
[b] *Mandiant, USA*

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.

Keywords: In-lined Reference Monitors, Transparency, Verification, Model-Checking, Symbolic Interpretation, ActionScript

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., [1–20]). 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.

For example, Fig. 1 shows the implementation of a simple IRM in ActionScript (AS) pseudo-code. The original bytecode on the left has been *instrumented* (i.e., rewritten) with an IRM as shown on the right. The IRM prohibits more than 100 calls to security-relevant API method `NavigateToURL` by counting its calls in program variable c and halting the program when c exceeds bound 100. The AS VM is stack-based, so instruction **get** c pushes c's value onto the stack, and **set** c assigns c a value popped from the stack. (Real IRMs are typically much more complex, but we use this simple example as a running illustration for clarity. The × marks are referenced in §4.6.)

Correct IRMs must satisfy two requirements: *soundness* and *transparency* [3, 21]. Soundness demands that the instrumented code satisfy the security policy, whereas transparency demands that it preserve the behavior of policy-compliant code. That is, adding the IRM to a program must not "break" its policy-compliant behaviors. To formally define policy-compliance, IRM policies are specified using a *policy specification language* (e.g., [8, 11–13, 17, 18, 22, 23]), which typically leverages concepts from *aspect-oriented programming* (AOP) [24] to abstractly identify security-relevant program operations. For example, the SPoX IRM system [23] expresses safety policies encoded as aspect-oriented security automata.

Numerous past works have developed powerful technologies for formally machine-verifying the soundness of IRMs [7, 8, 25–30]. This is important for establishing high assurance, and for minimizing and stabilizing the trusted computing base (TCB) of IRM systems. However, none have tackled the dual problem of machine-verifying transparency

*Corresponding Author
Email addresses: `meera.sridhar@utdallas.edu` (Meera Sridhar), `richard.wartell@mandiant.com` (Richard Wartell), `hamlen@utdallas.edu` (Kevin W. Hamlen)

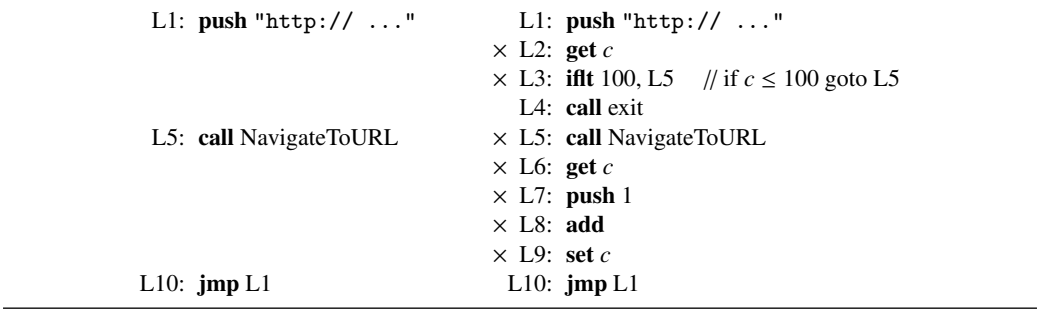

Fig. 1. Original bytecode (left) that has been rewritten (right) with an IRM that prohibits more than 100 URL navigations

(cf., [28, 31]). 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 advertisement security, advertisement distribution networks are often unwilling to adopt any protection system that involves binary modification unless there is overwhelming evidence that no safe advertisements are adversely affected by the process. Even a temporary loss of functionality in a few advertisements could potentially result in millions of dollars in lost revenue [32]. 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 [26, 27] can be extended in a natural way to verify IRM transparency.
- We introduce the design and implementation of an untrusted, external invariant-generator that can reduce the verifier's state-exploration burden and afford it greater generality than the more specialized rewriting systems it checks.
- Prolog unification [33] 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 the Cousots' abstract interpretation framework [35].
- The feasibility of our technique is demonstrated through a prototype implementation that targets the full AS 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 symbolic interpretation algorithms. Section 5 presents implementation and results. Finally, §6 concludes.

2. Background and Related Work

In-lined Reference Monitors. IRMs were first formalized in the development of the PoET/PSLang/SASI systems [4, 6], which instrument Java bytecode and GNU assembly code. Subsequently, numerous IRM frameworks have been developed for Java [2, 3, 8, 10–13, 23], JavaScript [17, 19], .NET [7], AS [9, 23], Android [20], and x86/64 native code [1, 14–16] architectures. Our experiments target SPoX-IRMs [23], which rewrite Java and AS 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. In Fig. 1, c is an example of a reified state variable.

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.

ActionScript. AS is a binary virtual machine language by Adobe Systems similar to Java bytecode. It is an object-oriented, single-threaded, stack-based language, with bytecode type-safety, managed memory, and exceptions. Compiled bytecode runs on a VM [36], usually in a web browser. It is important as a general web scripting language, and is widely used in portable web advertisements, online games, streaming media, and interactive web page animations. AS VM security is based on object-level encapsulation, code-signing, and sandboxing.

AS's pervasive use in web advertisements has made it an attractive vehicle for many malware attacks in recent years. Though its bytecode language is type-safe, past malware has exploited VM buffer overflows [37], implemented cross-site-scripting attacks, and performed click-jacking [38, 39] to damage browsers or disrupt victim host pages. The difficulty of enforcing rich AS 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 [9, 25–28, 40].

IRM Soundness Certification. Several past works have successfully performed certification of IRM soundness. The Mobile system [7] 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 [8, 29] 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 [26, 27].

An alternative to verifying IRMs is to prove the soundness of each rewriting implementation once and for all. For example, the Coq proof assistant has been applied to implement provably sound monitor-generating algorithms for OCaml [30]. However, extending this to production-level IRM systems requires proving the correctness of the entire IRM-synthesis tool chain, which can be considerable. For example, Java-MOP, which includes AspectJ, consists of almost a million lines of Java source code [2]. Moreover, proofs of rewriter soundness are inapplicable to architectures where code-recipients must verify IRMs produced by an untrusted third party [41]. For these reasons, automated IRM verification has become the dominant approach.

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 [3, 21]. 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 [31].

Chudnov and Naumann provide the first formal IRM transparency proof [5]. 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 one particular rewriting algorithm are transparent. The proof is therefore specific to one rewriting algorithm and does not necessarily generalize to other IRM systems or policies.

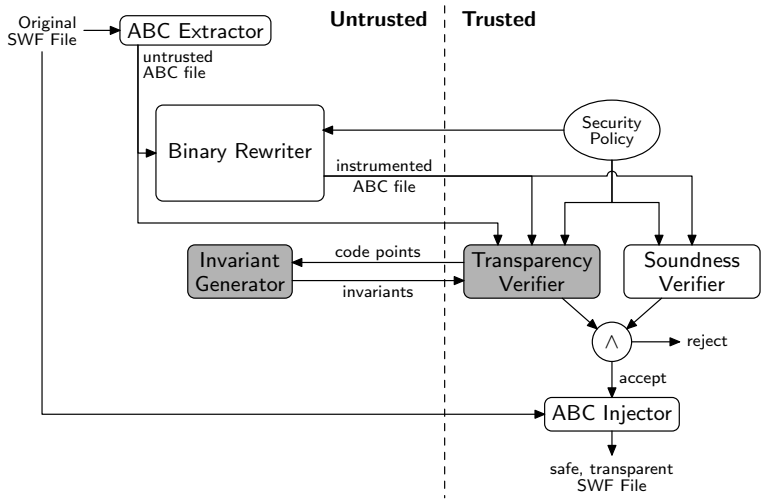

Fig. 2. A certifying ActionScript IRM architecture

Compiler Verification. Program equivalence-checking has been studied in the context of *translation validation*, which verifies behavior-preservation of compiler optimization phases [42–45]. 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 [46]. By changing the property being checked, one can potentially verify software security properties, such as information flow policies [47].

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 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 translation validators to our knowledge.

3. Overview

3.1. Rewriting and Soundness Verification

Figure 2 depicts our certifying IRM framework, consisting of a binary rewriter that automatically transforms untrusted AS 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 [25, 27, 40].

The AS Bytecode (ABC) Extractor tool extracts untrusted code from *ShockWave Flash* (SWF) binary archives, which package AS code with data, such as images and sound. A SPoX 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* [4, 48] 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. For example, the policy enforced in Fig. 1 is expressible as a security automaton with 100 states numbered 0 to 99. The start state is 0, and each state $i \in [0, 98]$ has an outgoing edge to state $i + 1$ labeled `NavigateToURL(*)`. Thus, the automaton accepts the language of traces that include at most 100 calls to `NavigateToURL`. Prior work has shown that all safety policies are expressible as security automata [4, 48].

The ABC Injector tool re-packages the modified bytecode produced by the rewriter 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 [3, 21]:

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.*

The distinction between observable and unobservable events distinguishes IRM operations that violate transparency from those that do not. For example, the IRM in Fig. 1 safely introduces unobservable operation $L9$ to policy-compliant runs, but must not introduce observable operation $L4$ to such runs. 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 AS 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 parallel simulation of the original program and its IRM-instrumented counterpart. Intuitively, the simulation runs both programs on equal inputs, non-deterministically stepping one or the other on each computational step.

A state of such a simulation consists of a pair of VM states (one for the original program and one for the rewritten one) and a pair of traces recording the observable events that led to each state. We conceptually consider these traces to be fields of their respective VM states, for which interpreted programs have only one operation: append. Adequate formal definitions of parallel simulations and their transparency must support non-terminating computations (i.e., infinite traces), they must not demand that programs are lock-step behaviorally equivalent (since IRMs introduce new code and reorder existing code), and they 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 flow w (i.e., sequence of consecutive states) in a parallel simulation is* progressive *if both simulated programs step infinitely often (i.o.) in w. (To support terminating computations, we model termination as an infinite stutter state.)*

Definition 3 (Shuffle). *Two 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 $\pi_i w$ denotes the sequence of program i's steps and states in flow w of the parallel simulation.*

Definition 4 (Transparency). *A state of a parallel simulation is* transparent *if its constituent program states have observationally equivalent traces. The full simulation is* transparent *if for every progressive flow w, there exists a shuffle of w whose states are i.o. transparent.*

This definition of transparency 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 potentially non-terminating loops to policy-adherent flows, since these could suppress desired program behaviors.

As an illustration, the programs in Fig. 1 exhibit traces consisting of `navigateToURL` calls (the only observable event in that example). A simulation of that program-pair is transparent because even though some of its possible flows are not observationally equivalent (e.g., simulations that run the original program twice as fast as the rewritten one yield persistently inequivalent pairs of traces), every such flow can nevertheless be reshuffled (e.g., to run both programs at roughly the same rate) so that the traces are infinitely often equivalent.

3.3. Verifying Transparency

Our transparency verifier is a symbolic interpreter and model-checker that non-deterministically explores the cross-product of the state spaces of the original and rewritten programs. To accommodate IRMs that introduce new methods, symbolic 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 AS 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)` typically 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, states that are visited infinitely often in every *policy-compliant* parallel simulation of the two programs. (Recall that policy-violating runs are intentionally modified by the IRM, and therefore exempt from this obligation.) Such a set inductively establishes that every safe computation is preserved, since it proves that the history of observable events after rewriting is infinitely often equivalent to that of the original program. This observation is formalized as follows:

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

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

(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 flow of the parallel simulation. 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 parallel simulation 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 abstract-interprets all flows from each state in S, confirming that each has a shuffle that revisits S.

For example, one suitable set S for Fig. 1 consists of all abstract states in which both programs are at $L1$ and their traces are equal. Every policy-satisfying simulation that starts in such a state eventually revisits it (with an appropriately progressive interleaving of the two programs' steps). This S therefore constitutes a loop invariant that proves the IRM's transparency.

3.4. Invariant Generation

It is feasible for IRM systems to infer and expose set S because it intuitively 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 (conjunctions of inequality relations over integers, and prefix relations over traces) that facilitate tractable proofs of soundness for the resulting IRM [26, 27].

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. When the blocks occur within a loop, the generated invariant is an invariant for the loop. 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 properties (1–3) of 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 [49] $\hat{\Gamma}$ (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 symbolic 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 parallel simulation 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 parallel simulation state inferred by the symbolic 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 AS 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 AS 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 that enforce safety properties [4]; 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 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.

AS does not presently support concurrency or threading; therefore, our verification algorithm restricts its attention to purely serial flows.

Introspective (e.g., reflective) code has some interesting implications for transparency, because code that self-inspects could discover the IRM and behave differently (without violating the policy). Such behavioral changes are often desirable; for example, a program that reports its own memory consumption should be permitted to report its new memory consumption after rewriting. Our transparency verifier permits such behavioral changes by modeling introspection results as program inputs. That is, it verifies that the IRM preserves the program logic that processes introspective inputs (e.g., printing them) even if the inputs (e.g., the size) may change due to rewriting.

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

Fig. 3. Non-standard core language instructions

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

Fig. 4. Semantics of the **obsevent** pseudo-instruction

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 subset of AS that includes standard arithmetic operations, conditional and unconditional jumps, integer-valued local registers, and the special instructions listed in Fig. 3. The implementation described in §5 supports the full AS 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. 4. In the rewritten code, 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 symbolic interpreter. For example, object references are modeled as integer Skolem constants, and references to identically-named fields of compatibly-typed objects may-alias. Field-writes assign fresh Skolem constants to all possible aliases.

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 symbolic interpretation of programs are expressed as the small-step operational semantics of a concrete and an abstract machine, respectively. Figure 5 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 [27]. 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 Skolem constants $\hat{v}$ and return values $\mathbf{rval}_m(e_1 :: \cdots :: e_n)$ of API calls. Skolem constants $\hat{s}$ and $\hat{t}$ denote entire abstract stacks and traces, respectively.

A *program* $P = (L, p, s)$ consists of a program entry point 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 simulating the original and instrumented programs, Fig. 6 extends the concrete and abstract program states described above to states of a parallel simulation. Each such state includes both an original and a rewritten program state. The abstract parallel-simulation state additionally includes a constraint list ζ consisting of a conjunction of linear inequalities over expressions.

The concrete machine semantics are modeled after the AS VM 2 (AVM2) semantics [36]; Fig. 7 shows the semantics of the special instructions of Fig. 3. 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 n defaults to 1 step when omitted.

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

Fig. 5. Concrete and abstract program states

$$
\begin{array}{ll}
\zeta ::= \bigwedge_{i=1..n} t_i \quad (n \geq 1) & \text{(Constraints)} \\
t ::= T \mid F \mid e_1 \leq e_2 \mid \hat{\tau}_1 = \hat{\tau}_2 & \text{(Clauses)} \\
\Gamma = \langle \chi_O, \chi_R \rangle & \text{(Concrete Interpreter States)} \\
\hat{\Gamma} = \langle \hat{\chi}_O, \hat{\chi}_R, \zeta \rangle & \text{(Symbolic Interpreter States)} \\
\langle C, \langle \chi_{O_0}, \chi_{R_0} \rangle, \mapsto_P^n \rangle & \text{(Concrete Interpreter)} \\
\langle \mathcal{A}, \langle \hat{\chi}_{O_0}, \hat{\chi}_{R_0}, \zeta_0 \rangle, \rightsquigarrow_P^n \rangle & \text{(Symbolic Interpreter)}
\end{array}
$$

Fig. 6. Concrete and abstract parallel-simulation machines

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

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

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

$$
\frac{\chi_i \mapsto_1 \chi_i' \qquad \chi_j = \chi_j' \qquad i \neq j}{\langle \chi_O, \chi_R \rangle \mapsto \langle \chi_O', \chi_R' \rangle} \text{(CBisim)}
$$

Fig. 7. Concrete small-step operational semantics

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

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

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

$$\frac{\hat{\chi}_O \subseteq \hat{\chi}'_O \qquad \hat{\chi}_R \subseteq \hat{\chi}'_R \qquad \zeta \Rightarrow \zeta'}{\langle \hat{\chi}_O, \hat{\chi}_R, \zeta\rangle \rightsquigarrow \langle \hat{\chi}'_O, \hat{\chi}'_R, \zeta'\rangle}\ (\text{Abstraction})$$

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

Fig. 8. Abstract small-step operational semantics

Algorithm 1 Verification

```
Input: Cache = {}, Horizon = {Γ̂0}                    // explored and unexplored states, respectively
Output: Accept or Reject
1: while Horizon ≠ ∅ do                               // while reachable, unverified states remain
2:     Γ̂ ← choose(Horizon)                            // choose unexplored abstract state
3:     S_Γ̂ ← VerificationSingleCodePoint(Γ̂)           // reduce state to subgoals
4:     if S_Γ̂ = Reject then return Reject
5:     Cache ← Cache ∪ {Γ̂}                            // mark state as explored
6:     Horizon ← (Horizon ∪ S_Γ̂)\Cache                // mark subgoals as unexplored
7: end while
8: return Accept
```

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 parallel-simulation machine that non-deterministically chooses which machine to step next.

Figure 8 gives the corresponding semantics for symbolic interpretation. Each step $\hat{\chi} \rightsquigarrow \hat{\chi}', \zeta$ of symbolic 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 symbolic expression $\hat{\mathfrak{a}}(\hat{\tau}')$ to denote the security automaton state. Rule Abstraction allows the symbolic 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 symbolic interpretation semantics. Algorithm 2 verifies an individual abstract parallel simulation 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 states and a horizon of unexplored abstract states. Upon successful verification of all control flows, it returns *Accept*; otherwise it returns *Reject*. It begins by drawing an arbitrary unexplored state $\hat{\Gamma}$ from the *Horizon* (line 2) and passing it to Algorithm 2. Algorithm 2 returns a set $S_{\hat{\Gamma}}$ of abstract states where simulation 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 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) state $\hat{\Gamma}_H$ for $\hat{\Gamma}$, (2) a finite, *generalized post-dominating set* $D_{\hat{\Gamma}}$ for $\hat{\Gamma}$ whose

Algorithm 2 VerificationSingleCodePoint

Input: $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$ *// abstract simulation state*
Output: $S_{\hat{\Gamma}}$ or *Reject* *// set of proof subgoals*
1: $(\hat{\Gamma}_H, D_{\hat{\Gamma}}, n) \leftarrow \mathit{InvariantGen}(\hat{\Gamma})$ *// query untrusted invariant-generator*
2: $\mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}, \hat{\Gamma}_H)$ *// verify that $\hat{\Gamma}_H$ abstracts $\hat{\Gamma}$*
3: **if** $\mathit{SatValue} = \mathit{Reject}$ **then return** *Reject*
4: $S_{\hat{\Gamma}} \leftarrow \mathit{AbsIn}^n(\{\hat{\Gamma}_H\}, D_{\hat{\Gamma}})$ *// interpret from $\hat{\Gamma}_H$ to get subgoals $S_{\hat{\Gamma}}$*
5: **if** $\mathit{labels}(S_{\hat{\Gamma}}) \not\subseteq D_{\hat{\Gamma}}$ **then return** *Reject* *// verify that $D_{\hat{\Gamma}}$ post-dominates $\hat{\Gamma}_H$*
6: **for each** $\hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle \in S_{\hat{\Gamma}}$ **do** *// for each subgoal*
7: $\langle \_, \_, \_, \hat{\tau}'_1 \rangle = \hat{\chi}'_1$
8: $\langle \_, \_, \_, \hat{\tau}'_2 \rangle = \hat{\chi}'_2$
9: $\mathit{SatValue} \leftarrow \mathit{ModelCheck}(\hat{\Gamma}', \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \wedge (\hat{\tau}'_1 = \hat{\tau}'_2) \rangle)$
10: **if** $\mathit{SatValue} = \mathit{Reject}$ **then return** *Reject* *// verify goal $\hat{\Gamma}'$ is transparent*
11: **end for**
12: **return** $S_{\hat{\Gamma}}$ *// return subgoals*

Algorithm 3 ModelCheck

Input: $\hat{\Gamma} = \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle, \hat{\Gamma}' = \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \rangle$ *// trusted abstract state, and untrusted abstraction of it*
Output: *Accept* or *Reject*
1: $\zeta_U \leftarrow \mathit{Unify}(\hat{\Gamma}, \hat{\Gamma}')$ *// check structural compatibility*
2: **if** $\zeta_U = \mathit{Fail}$ **then return** *Reject*
3: $\mathit{SatValue} \leftarrow \mathit{CLP}(\zeta \wedge \neg\zeta' \wedge \zeta_U)$ *// verify unsatisfiability of implication negation*
4: **if** $\mathit{SatValue} = \mathit{False}$ **then**
5: **return** *Accept*
6: **else**
7: **return** *Reject*
8: **end if**

members are all transparent code points, and (3) a stepping-bound n. A set S of abstract 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 [49]. In our case, the complete flows 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) with $n_1 + n_2 = n$, where n_i represents the exact number of steps machine i should step 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. 8). Next, it performs symbolic interpretation for n steps from $\hat{\Gamma}$ to confirm post-dominance of $D_{\hat{\Gamma}}$. Function $\mathit{AbsIn}^n(S, E)$ in line 4 performs symbolic interpretation from S for n steps or until reaching a code label in E. 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 parallel-simulation 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.

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 parallel-simulation 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. If unification fails, the model-checker rejects. Successful unification yields a collection ζ_U of purely integer equalities.

Algorithm 4 GenericInvariant

Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$ *// abstract state*
Output: $\langle \hat{\chi}, \hat{\chi}', \zeta \rangle$ *// (more abstract) state*
1: choose fresh Skolem constants $\hat{v}_\ell$, $\hat{v}'_\ell$, $\hat{s}$, $\hat{s}'$, $\hat{t}$, and $\hat{t}'$
2: $\hat{\sigma} \leftarrow \{(\ell, \hat{v}_\ell) \mid \hat{\sigma}_1(\ell) = e_1\}$ *// abstract all local and reified state variables to fresh vars*
3: $\hat{\sigma}' \leftarrow \{(\ell, \hat{v}'_\ell) \mid \hat{\sigma}_2(\ell) = e_2\} \cup \{(r, \hat{\sigma}_2(r))\}$
4: $\hat{\chi} \leftarrow \langle L_1 : i_1, \hat{s}, \hat{\sigma}, \hat{t} \rangle$ *// abstract traces to trace vars*
5: $\hat{\chi}' \leftarrow \langle L_2 : i_2, \hat{s}', \hat{\sigma}', \hat{t}' \rangle$
6: $\zeta' \leftarrow (\hat{s}{=}\hat{s}') \wedge \left(\bigwedge_{\ell \in \hat{\sigma} \leftarrow \cap \hat{\sigma}' \leftarrow} \hat{v}_\ell{=}\hat{v}'_\ell\right) \wedge (r{=}\hat{\mathfrak{a}}(\hat{t}')) \wedge (\hat{t}{=}\hat{t}')$ *// assert SYNC and transparency*
7: **return** $\mathbf{I} = \langle \hat{\chi}, \hat{\chi}', \zeta' \rangle$

Algorithm 5 InvariantGen

Input: $\hat{\chi}_1 = \langle L_1 : i_1, \hat{\rho}_1, \hat{\sigma}_1, \hat{\tau}_1 \rangle, \hat{\chi}_2 = \langle L_2 : i_2, \hat{\rho}_2, \hat{\sigma}_2, \hat{\tau}_2 \rangle, \zeta$ *// abstract state*
Output: $\hat{\Gamma}_H, D_{\hat{\Gamma}}, n$ *// more abstract state, post-dominating set, and step bound*
1: **if** $L_2 \notin Marked$ **then** *// outside marked region*
2: **return** $(GenericInvariant(\hat{\chi}_1, \hat{\chi}_2), \{(L_1, L_2)\}, 1)$ *// use Algorithm 4*
3: **else**
4: $\hat{\Gamma} \leftarrow \langle \hat{\chi}_1, \hat{\chi}_2, \zeta \rangle$ *// don't abstract the state*
5: $n \leftarrow \min\{n \mid AbsIn^n(\{\hat{\Gamma}\}, Marked) = AbsIn^{n+1}(\{\hat{\Gamma}\}, Marked)\}$ *// find step bound*
6: **return** $(\hat{\Gamma}, labels(AbsIn^n(\{\hat{\Gamma}\}, Marked)), n)$
7: **end if**

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 state, there is no instantiation of the free variables that falsifies $\zeta \Rightarrow \zeta'$.

4.5. Invariant Generation

Recall from §4.3 that for every reachable code point (L_1, L_2) in the parallel simulation's state space, the verifier requires an (untrusted) hint consisting of: (1) an invariant for (L_1, L_2) in the form of an abstract 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.

In this section we outline a strategy for generating these invariants that allows our verifier to prove transparency for IRMs produced by the SPoX rewriting system [23], 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 variable and the state of the security automaton that encodes the policy constitutes an invariant, termed *synchronization* (SYNC), that has been used to verify its soundness [26]. 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 parallel-simulation 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.

Within marked regions, the invariant-generator uses the last half of Algorithm 5, which asserts that transparency is restored once parallel simulation exits the marked region. To prove that execution does eventually exit the marked region, line 5 uses symbolic 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 $r = \hat{\mathfrak{a}}(\hat{t}_R)$ 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 transparency verification, we revisit the pseudo-bytecode listing in Fig. 1. Recall that the figure depicts an IRM that prohibits more than 100 calls to security-relevant method `NavigateToURL`. Lines with an × are those in the *Marked* set described in §4.5.

The verifier (Algorithm 1) begins exploring the cross-product space from point $(L1, L1)$, where both original and rewritten programs are at $L1$. Line 1 of Algorithm 2 consults the (untrusted) invariant-generator, which suggests a hint that abstracts this to a clause asserting $c = \hat{\mathfrak{a}}(\hat{t})$ (Algorithm 4, line 6), where Skolem constant $\hat{t}$ denotes the current trace and $\hat{\mathfrak{a}}(\hat{t})$ is the current security automaton state. This invariant recommends that the only information necessary at $L1$ to infer transparency is that c correctly reflects the security automaton state, and that all other state components are unchanged by the IRM. Since initially $\hat{t} = \hat{\tau} = \epsilon$ in both machines, the verifier confirms that $c = 0$ and $\hat{\mathfrak{a}}(\hat{\tau}) = 0$ using Algorithm 3, and therefore tentatively accepts $c = \hat{\mathfrak{a}}(\hat{\tau})$ as a possible invariant for point $(L1, L1)$. The invariant-generator next supplies a post-dominating set $\{(L1, L2)\}$ and stepping bound 1 (see Algorithm 5, line 2) to assert that all realizable flows from $(L1, L1)$ have a shuffle containing $(L1, L2)$, and that $(L1, L2)$ is reachable in 1 step. The verifier confirms this (line 2 of Algorithm 2) and continues verification at $(L1, L2)$.

When this process repeats at $(L1, L2)$, we find that $L2$ is marked (×) so lines 4–6 of Algorithm 5 generate the invariant of Algorithm 4. This returns post-dominating set $\{(L10, L10)\}$ and stepping bound 9, which advise the verifier to continue symbolic interpreting without further abstracting the state until exiting the marked region. When interpretation reaches the conditional at line 3, clause $c = \hat{\mathfrak{a}}(\hat{t})$ (see above) is critical for inferring that $L4$ is unreachable when the original code satisfies the policy. Specifically, the policy-adherence assumption yields constraint $\hat{\mathfrak{a}}(\hat{\tau}) < 100$ after $L5$, which contradicts negative branch condition $c \geq 100$ introduced by $L4$ when $c = \hat{\mathfrak{a}}(\hat{\tau})$, causing line 3 of Algorithm 3 to return *False*.

Once the interpreter reaches $(L10, L10)$, the invariant-generator supplies the same abstract state as it did for $L1$. That is, it asserts that all shared state components (including traces) are equal, and reified state variable c equals security automaton state $\hat{\mathfrak{a}}(\hat{t})$. The linear constraint solver confirms that the incremented c (see $L8$) matches the incremented state $\hat{\mathfrak{a}}(\hat{t}\,\mathbf{api}_{\text{NavigateToURL}})$, and therefore accepts the new invariant. Symbolic interpreting for an additional step, it confirms that this matches the earlier invariant for $L1$, 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 symbolic interpreter and model-checker. That is, if the verifier's abstract simulation of the two programs (including the interpretation rules that perform model-checking) is a bisimulation—i.e., it 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. We have proved soundness of our symbolic interpreter using the Cousots' abstract interpretation framework [35]. Due to space limitations, the full proof is deferred to the companion technical report [50], but we summarize the high-level approach below.

Following the approach of [51], soundness of the abstract interpreter is proved via two lemmas that establish preservation and progress (respectively) for a parallel simulation of the abstract and concrete machines. The preservation lemma proves that the simulation preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the symbolic 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. 7 and 8). Soundness of the model-checker follows from soundness of the Prolog CLP engine [52].

5. Implementation and Results

Our implementation of the transparency verification algorithm detailed in §4 targets the full AS 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) [53] 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 [54].

IRM instrumentation is accomplished via a collection of small binary-to-binary rewriters. They each augment untrusted AS 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-generators 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 AS 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 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 advertisements drawn from public web sites. The results are summarized in Table 1. For each advertisement, 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 advertisements 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. Advertisement complexity was not necessarily indicative of maliciousness; some of the most complex advertisements 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 advertisement 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 advertisement. 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 advertisements expand to fill a large part of the web page whenever the user clicks or mouses over the advertisement space. This is frequently abused for click-jacking. Even when advertisement 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.

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

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.

AS has been used as a heap spraying vehicle in several past attacks [55]. 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 AS binary.

To inhibit heap sprays, we enforced a policy that bounds the number of byte-write operations that an advertisement 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 advertisements in Table 1 resulted in no behavioral changes, as confirmed by the verifier.

6. Conclusions

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 AS bytecode applets.

In future work, we would like to extend our approach to support user-written IRM implementations (e.g., those implemented in AspectJ [2]) in addition to IRMs synthesized purely automatically. This requires an IRM development environment that includes program-proof co-development, such as Coq [30]. Such research will facilitate easier, more reliable development of customized IRMs with machine-checkable proofs of soundness and transparency.

Acknowledgments

The research reported herein was supported in part by the National Science Foundation (NSF) under grants #1065216 and #1054629, and by the Office of Naval Research (ONR) under grant N00014-14-1-0030. All opinions and conclusions expressed are those of the authors and not necessarily of the NSF or ONR.

References

[1] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, N. Fullagar, Native Client: A Sandbox for Portable, Untrusted x86 Native Code, in: Proc. 30th IEEE Sym. Security & Privacy (S&P), 79–93, 2009.

[2] F. Chen, G. Roşu, Java-MOP: A Monitoring Oriented Programming Environment for Java, in: Proc. 11th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 546–550, 2005.

[3] J. Ligatti, L. Bauer, D. Walker, Enforcing Non-safety Security Policies with Program Monitors, in: Proc. 10th European Sym. Research in Computer Security (ESORICS), 355–373, 2005.

[4] F. B. Schneider, Enforceable Security Policies, ACM Trans. Information and Systems Security (TISSEC) 3 (1) (2000) 30–50.

[5] A. Chudnov, D. A. Naumann, Information Flow Monitor Inlining, in: Proc. 23rd IEEE Computer Security Foundations Sym. (CSF), 200–214, 2010.

[6] Ú. Erlingsson, F. B. Schneider, SASI Enforcement of Security Policies: A Retrospective, in: Proc. New Security Paradigms Workshop (NSPW), 87–95, 1999.

[7] K. W. Hamlen, G. Morrisett, F. B. Schneider, Certified In-lined Reference Monitoring on .NET, in: Proc. 1st ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS), 7–16, 2006.

[8] I. Aktug, K. Naliuka, ConSpec – a Formal Language for Policy Specification, Science Computer Programming (SCP) 74 (2008) 2–12.

[9] Z. Li, X. Wang, FIRM: Capability-based Inline Mediation of Flash Behaviors, in: Proc. 26th Annual Computer Security Applications Conf. (ACSAC), 181–190, 2010.

[10] M. Dam, B. Jacobs, A. Lundblad, F. Piessens, Security Monitor Inlining for Multithreated Java, in: Proc. 23rd European Conf. Object-Oriented Programming (ECOOP), 546–569, 2009.

[11] D. Evans, A. Twynman, Flexible Policy-directed Code Safety, in: Proc. 20th IEEE Sym. Security & Privacy (S&P), 32–45, 1999.

[12] M. Kim, M. Viswanathan, S. Kannan, I. Lee, O. Sokolsky, Java-MaC: A Run-time Assurance Approach for Java Programs, Formal Methods in System Design 24 (2) (2004) 129–155.

[13] L. Bauer, J. Ligatti, D. Walker, Composing Security Policies with Polymer, in: Proc. 26th ACM Conf. Programming Language Design and Implementation (PLDI), 305–314, 2005.

[14] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti, Control-flow Integrity Principles, Implementations, and Applications, ACM Trans. Information and Systems Security (TISSEC) 13 (1).

[15] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti, Control-flow Integrity, in: Proc. 12th ACM Conf. Computer and Communications Security (CCS), 340–353, 2005.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, G. C. Necula, XFI: Software Guards for System Address Spaces, in: Proc. USENIX Sym. Operating Systems Design and Implementation (OSDI), 75–88, 2006.

[17] D. Yu, A. Chander, N. Islam, I. Serikov, JavaScript Instrumentation for Browser Security, in: Proc. 35th ACM Sym. Principles Of Programming Languages (POPL), 237–249, 2007.

[18] D. S. Dantas, D. Walker, Harmless Advice, in: Proc. 34th ACM Sym. Principles Of Programming Languages (POPL), 383–396, 2006.

[19] M. Fredrikson, R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, V. Yegneswaran, Efficient Runtime Policy Enforcement Using Counterexample-guided Abstraction Refinement, in: Proc. 24th Int. Conf. Computer Aided Verification (CAV), 548–563, 2012.

[20] B. Davis, B. Sanders, A. Khodaverdian, H. Chen, I-ARM-Droid: A Rewriting Framework for In-app Reference Monitors for Android Applications, in: IEEE Mobile Security Technologies (MoST), 2012.

[21] K. W. Hamlen, G. Morrisett, F. B. Schneider, Computability Classes for Enforcement Mechanisms, ACM Trans. On Programming Languages And Systems (TOPLAS) 28 (1) (2006) 175–205.

[22] Ú. Erlingsson, The Inlined Reference Monitor Approach to Security Policy Enforcement, Ph.D. thesis, Cornell University, Ithaca, New York, 2004.

[23] K. W. Hamlen, M. Jones, Aspect-oriented In-lined Reference Monitors, in: Proc. 3rd ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS), 11–20, 2008.
[24] G. Kiczales, J. Lamping, A. Medhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented Programming, in: Proc. 11th European Conf. Object-Oriented Programming (ECOOP), 220–242, 1997.
[25] M. Sridhar, K. W. Hamlen, In-lined Reference Monitoring in Prolog, in: Proc. 12th Int. Sym. Practical Aspects of Declarative Languages (PADL), 149–151, 2010.
[26] K. W. Hamlen, M. M. Jones, M. Sridhar, Aspect-oriented Runtime Monitor Certification, in: Proc. 18th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 126–140, 2012.
[27] M. Sridhar, K. W. Hamlen, Model-checking In-lined Reference Monitors, in: Proc. 11th Int. Conf. Verification, Model-Checking and Abstract Interpretation (VMCAI), 312–327, 2010.
[28] M. Sridhar, K. W. Hamlen, Flexible In-lined Reference Monitor Certification: Challenges and Future Directions, in: Proc. 5th ACM SIGPLAN Workshop Programming Languages meets Program Verification (PLPV), 55–60, 2011.
[29] I. Aktug, M. Dam, D. Gurov, Provably Correct Runtime Monitoring, in: Proc. 15th Int. Sym. Formal Methods (FM), 262–277, 2008.
[30] J. O. Blech, Y. Falcone, K. Becker, Towards Certified Runtime Verification, in: Proc. 10th Int. Conf. Formal Engineering Methods: Formal Methods and Software Engineering (SEFM), 494–509, 2012.
[31] R. Khoury, N. Tawbi, Corrective Enforcement: A New Paradigm of Security Policy Enforcement By Monitors, ACM Trans. Information and Systems Security (TISSEC) 15 (2).
[32] Internet Advertising Bureau, Internet Advertising Revenue Report FY 2012, http://www.iab.net/media/file/IAB_Internet_Advertising_Revenue_Report_FY_2012_rev.pdf, 2013.
[33] L. Shapiro, E. Y. Sterling, The Art of Prolog: Advanced Programming Techniques, MIT Press, 1994.
[34] J. Jaffar, M. J. Maher, Constraint Logic Programming: A Survey, J. Logic Programming (JLP) (1994) 503–581.
[35] P. Cousot, R. Cousot, Abstract Interpretation Frameworks, J. Logic and Computation 2 (4) (1992) 511–547.
[36] ActionScript Virtual Machine 2 Overview, http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf, 2007.
[37] M. Dowd, Application-specific Attacks: Leveraging the ActionScript Virtual Machine, IBM Global Technology Services, White Paper, 2008.
[38] CVE-2010-2216, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2216, 2010.
[39] CVE-2010-2215, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2215, 2010.
[40] B. W. DeVries, G. Gupta, K. W. Hamlen, S. Moore, M. Sridhar, ActionScript Bytecode Verification with Co-logic Programming, in: Proc. 4th ACM SIGPLAN Workshop Programming Languages and Analysis for Security (PLAS), 9–15, 2009.
[41] M. Jones, K. W. Hamlen, A Service-oriented Approach to Mobile Code Security, in: Proc. 8th Int. Conf. Mobile Web Information Systems (MobiWIS), 531–538, 2011.
[42] A. Pnueli, M. Siegel, E. Singerman, Translation Validation, in: Proc. 4th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 151–166, 1998.
[43] G. C. Necula, Translation Validation for an Optimizing Compiler, in: Proc. 21st ACM Conf. Programming Language Design and Implementation (PLDI), 83–94, 2000.
[44] Program-Transformation.Org, Program Optimization, http://www.program-transformation.org/Transform/ProgramOptimization, 2013.
[45] X. Leroy, Formal Verification of a Realistic Compiler, Communications ACM (CACM) 52 (7) (2009) 107–115.
[46] A. Zaks, A. Pnueli, CoVaC: Compiler Validation By Program Analysis of the Cross-product, in: Proc. 15th Int. Sym. Formal Methods (FM), 35–51, 2008.
[47] G. Barthe, P. R. D'Argenio, T. Rezk, Secure Information Flow By Self-composition, in: Proc. 17th IEEE Computer Security Foundations Workshop (CSF), 100–114, 2004.
[48] B. Alpern, F. B. Schneider, Recognizing Safety and Liveness, Distributed Computing 2 (1986) 117–126.
[49] R. Gupta, Generalized Dominators and Post-dominators, in: Proc. 20th ACM Sym. Principles Of Programming Languages (POPL), 246–257, 1992.
[50] M. Sridhar, R. Wartell, K. W. Hamlen, Hippocratic Binary Instrumentation: First Do No Harm (Extended Version), Tech. Rep., Dept. Of Comput. Science, U. Texas at Dallas, 2013.
[51] B.-Y. E. Chang, A. Chlipala, G. C. Necula, A Framework for Certified Program Analysis and its Applications to Mobile-code Safety, in: Proc. 7th Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI), 174–189, 2006.
[52] E. Börger, R. F. Salamone, CLAM Specification for Provably Correct Compilation of CLP(R) Programs, in: E. Börger (Ed.), Specification and Validation Methods, Oxford University Press, 96–130, 1995.
[53] J. Jaffar, S. Michaylov, P. J. Stuckey, R. H. C. Yap, The CLP(R) Language and System, ACM Trans. On Programming Languages And Systems (TOPLAS) 14 (3) (1992) 339–395.
[54] M. Sridhar, R. Wartell, K. W. Hamlen, FlashTrack: A Transparency Checker Tool for Flash Applications, http://code.google.com/p/flashtrack, 2013.
[55] Heap Spraying with ActionScript, http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html, 2009.