# Flexible In-lined Reference Monitor Certification:

## Challenges and Future Directions

Meera Sridhar
University of Texas at Dallas
meera.sridhar@student.utdallas.edu

Kevin W. Hamlen
University of Texas at Dallas
hamlen@utdallas.edu

## Abstract

Over the last few years, *in-lined reference monitors* (IRM's) have gained much popularity as successful security enforcement mechanisms. Aspect-oriented programming (AOP) provides one elegant paradigm for implementing IRM frameworks. There is a foreseen need to enhance both AOP-style and non-AOP IRM's with static *certification* due to two main concerns. Firstly, the Trusted Computing Base (TCB) can grow large quickly in an AOP-style IRM framework. Secondly, in many practical settings, such as in the domain of web-security, aspectually encoded policy implementations and the *rewriters* that apply them to untrusted code are subject to frequent change. Replacing the rewriter with a small, light-weight, yet powerful *certifier* that is policy-independent and less subject to change addresses both these concerns.

The goal of this paper is two-fold. First, interesting issues encountered in the process of building certification systems for IRM frameworks, such as policy specification, certifier soundness, and certifier completeness, are explored in the light of related work. In the second half of the paper, three prominent unsolved problems in the domain of IRM certification are examined: runtime code-generation via `eval`, IRM certification in the presence of concurrency, and formal verification of transparency. Promising directions suggested by recent work related to these problems are highlighted.

***Categories and Subject Descriptors*** D.2.4 [*Software Engineering*]: [Software/Program Verification]; D.4.6 [*Operating Systems*]: [Security and Protection—access controls]; F.3.1 [*Logics and Meanings of Programs*]: [Specifying and Verifying and Reasoning about Programs—mechanical verification]; F.3.2 [*Logics and Meanings of Programs*]: [Semantics of Programming Languages—program analysis]

***General Terms*** Languages, Security

***Keywords*** Aspect-oriented programming, In-lined reference monitors, Runtime verification, Static verification

## 1. Introduction

Over the last few years, in-lined reference monitoring (c.f., [10, 39, 46, 55]) has become increasingly popular as a software security enforcement mechanism. In-Lined Reference Monitors (IRM's) [46] enforce safety policies by injecting runtime security guards directly into untrusted binaries. The guards test whether an impending operation constitutes a policy violation. If so, corrective action (e.g., premature termination) is taken to prevent the violation. The result is *self-monitoring* code that can be executed safely without external monitoring. A canonical example policy in the literature is one that prohibits all network-send operations after the program has read from a confidential file [46]. The IRM enforces the policy by tracking the history of security-relevant operations at runtime and consulting it in runtime guards. The approach is both flexible and powerful, being shown capable of enforcing policies not enforceable by any purely static analysis [27].

*Aspect-Oriented Programming* (AOP) [35] has been identified as a natural and elegant means of implementing IRM's (e.g., [14, 15, 26, 47, 54]). *Aspects* encode an IRM's implementation as a collection of *pointcuts*, which identify potentially security-relevant program operations, each paired with *advice*, which prescribes a local code transformation sufficient to guard the operation. This localized code-rewriting approach has been shown to be sufficient to enforce large, important classes of policies, including the safety policies [27, 46] and some liveness policies [39]. Additionally, AOP enjoys an extensive support system and tool base, making it popular in many academic and industrial settings. An important caveat to the approach is that there must be some way of preventing circumvention of the injected guards by unrestricted control-flows or corruption of the guard code by unrestricted memory accesses. Such protection can be afforded by a type-safe bytecode language, such as Java [10, 26], .NET [28], or ActionScript [49], or by applying a sandboxing mechanism such as program shepherding [37] or software fault isolation [23, 40, 55].

As AOP-based IRM systems gain prominence, there is a foreseen need to enhance them with *certification*. This need arises due to two main concerns: Firstly, the Trusted Computing Base (TCB) of an AOP-style IRM framework can quickly become extremely large as the size of the aspect library grows. When the IRM is intended to apply to large classes of untrusted binaries rather than just one particular application, the required generality makes them extremely difficult to write correctly, as past case-studies have demonstrated [34]. Moreover, the TCB also includes the compiler, *aspect-weaver*, and possibly other support tools that can be difficult to verify formally. This frustrates attempts to provide strong formal guarantees about the instrumented code produced by these systems.

Secondly, in many practical settings aspectually encoded policy implementations and the *rewriters* that apply them to untrusted code are subject to frequent change. A good example is the problem domain of web ad applet security (c.f., [38, 48, 49]). As new attacks appear and new vulnerabilities are discovered, these IRM implementations rapidly change in their technical details (though not in their high-level approach of guarding potentially dangerous operations with dynamic checks). Thus, the considerable effort


*PLPV'11*, January 29, 2011, Austin, Texas, USA.

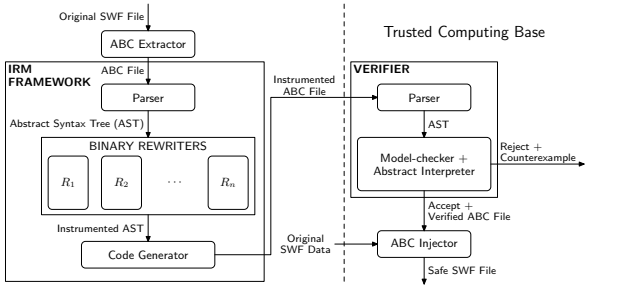


**Figure 1.** Certifying in-lined reference monitoring framework for ActionScript Bytecode

that might be devoted to verifying formally one particular IRM implementation quickly becomes obsolete when the IRM is revised in response to a new threat.

Therefore, rather than proving that a particular IRM framework correctly modifies all untrusted code instances, we instead consider the challenge of machine-certifying individual instrumented code instances with respect to the original policy and untrusted code whence they were derived. Specifically, we would like to prove that, for a given policy $\mathcal{P}$, untrusted code instance $\mathsf{e}$, and rewritten code instance $\mathsf{e}'$,

- **Soundness:** $\mathsf{e}' \in \mathcal{P}$; i.e., rewritten code is policy-satisfying, and
- **Transparency:** if $\mathsf{e} \in \mathcal{P}$ then $\mathsf{e}' \approx \mathsf{e}$, where $\approx$ denotes semantic program-equivalence; i.e., the behavior of policy-satisfying code is preserved across rewriting.

Figure 1 shows an example of a certifying in-lined reference monitoring system for ActionScript Bytecode used in Adobe's Flash, Adobe Integrated Runtime (AIR), and related technologies [49].

The problem of certifying IRM's differs substantially from the more general problem of verifying the safety of arbitrary code. This is because IRM certifiers need only be powerful enough to provide rewriters a reasonable range of certifiable code to which to map untrusted code instances. For example, while general-purpose model-checkers are often very large (e.g., [30]), prior work has shown that in the context of an IRM systems it is possible to create extremely small, efficient model-checkers that are powerful enough to verify large classes of IRM's formally [16, 49]. The rewriters for these model-checking IRM systems simplify the certifier's task when necessary by inserting extra dynamic guards that obviate the proof of safety (at the expense of some additional runtime overhead for the rewritten code). The verifier therefore need only be sufficiently sophisticated to identify the dynamic checks used by the IRM to guard dangerous operations, and verify that they are not circumvented by unguarded control flows.

IRM certification also differs from proof-carrying code (PCC) [43] in that PCC rewriters (*certifying compilers*) leverage source-level information that is typically unavailable to binary rewriters. For example, a certifying compiler may prove control-flow safety by refining a general proof of source-language control-flow safety down to the compiled object code, whereas a binary-level rewriter lacks this source-level information. This has interesting implications for IRM certification because effective IRM certifiers must support a different class of rewritten code—those that are reliably implementable by binary-level IRM rewriters but not necessarily typical of source-level rewriters. This interplay between rewriter and certifier power leads to interesting research opportunities and challenges, which we discuss here.

Our goal in this paper is two-fold: We begin by discussing some interesting issues that we have encountered in the process of building certification systems for IRM frameworks, and explore related works in the area. Section 2 discusses the problem of formulating policy specification languages that are suitable for both automated rewriting and automated certification, and that are also reasonable for writing realistic policies. In Section 3, we explain soundness, completeness, and their relationship in the context of IRM certification. The intractability of full completeness leads to a definition of $\mathcal{P}$*-verifiability*, which captures the notion of an IRM certifier that is sufficiently complete to allow certifiable IRM enforcement of important classes of security policies.

In the second half of the paper, we present three prominent unsolved problems in domain of IRM certification, and discuss why these are particularly challenging: Section 4 considers the issue of runtime code generation via `eval`. Section 5 discusses challenges related to certifying IRM's in the presence of concurrency, and Section 6 considers the challenge of formally verifying transparency. Related work that seems well-poised to address difficult facets of these problems is highlighted.

## 2. IRM Policy Specification

Effective IRM certification requires a means of specifying policies in a way that admits both effective enforcement by an IRM and separate, independent certification by a verifier. Different approaches to this problem give rise to different notions of what it means to certify an IRM against a policy.

One approach adopted by a large number of IRM systems is to express not just policy enforcement as an aspect but the policies themselves as aspects or in an aspect-like language (e.g., [1, 7, 10, 21, 24, 36]). A distinguishing characteristic of these specification languages is that at least part of the specification consists of code fragments (i.e., advice) that implements dynamic security checks, violation-precluding interventions, or IRM state updates. These fragments are woven into the untrusted code by the rewriter; the specification therefore defines a code-transformation recipe.

While this approach lends itself to rewriting, it makes meaningful certification difficult because rewriting becomes a subproblem of certification and therefore leaks back into the TCB. This is illustrated by recent work on IRM verification-by-contract [1], which casts policy specifications as *contracts* that describe the intended security-relevant behavior of the program. The contract is essentially an aspect-oriented program; it consists of `event` clauses that function as pointcuts, and `after` clauses that supply advice. A rewritten program $\mathsf{e}'$ is checked against such a contract $R$ by a process that essentially applies $R$ to the original code $\mathsf{e}$ and tests whether $R(\mathsf{e}) = \mathsf{e}'$.

However, this approach has three drawbacks: (1) Contracts constitute a potentially significant addition to the TCB because they must encode the essence of the rewriting algorithm. (2) The certifier must therefore duplicate large portions of the rewriter in order to compute $R(\mathsf{e})$. (3) Verifying that the contract is sound returns us to the original problem of proving that a general rewriting strategy is sound in all cases—a problem that we argued in Section 1 is more difficult than IRM certification that verifies the safety of rewritten code on a case-by-case basis. It is therefore unclear that this approach constitutes a meaningful reduction to the TCB.

A viable alternative approach is to express policies as types, so that certification can be formalized as type-checking [28]. However, the resulting policy language can be somewhat limiting in practice. Our experience indicates that while certification is extremely elegant in such a system, it is not always easy to find types that express realistic, high-level policies that constrain method argument values, field values, and relationships between object instances within complex data structures. (But see related work on *low-level liquid types* [45] that may offer creative solutions to some of these problems.)

A third approach is to specify policies purely declaratively using a temporal logic such as LTL, or an automaton encoding such as *security automata* [3]. To express low-level binary program properties in such a language, atomic propositions and edge labels can be written as pointcuts that identify security-relevant program operations. Past work on this has yielded formal denotational semantics that reduce such specifications to pure program properties [26].

These specification languages seem well-suited as input to IRM certification systems that decide soundness of rewritten code independently of the original untrusted code, and without the need to duplicate the rewriter implementation within the certifier. Typically the task of synthesizing a program that satisfies such a property is more complex than verifying that an existing program satisfies the property, mainly because the rewriter must support a much larger domain (arbitrary untrusted code) than the domain supported by the certifier (rewriter-supplied, self-monitoring code). Moreover, the certifier in such a framework does not regard the original untrusted code, leading to less code duplication between the rewriter and the certifier. There is therefore good reason to believe that such certification constitutes a meaningful reduction to the TCB, and a meaningful second line of defense.

Additionally, in a purely declarative policy specification language the policies define what security property to enforce without overspecifying *how* it is to be enforced. This characteristic allows for a clear-cut separation between the IRM implementation and the certifier, providing the former the choice of an optimal rewriting strategy customized according to information available at rewrite-time but not necessarily certification-time.

## 3. Certifier Soundness and Completeness

We preface our discussion of IRM certification challenges with some preliminaries related to soundness and completeness of rewriters and certifiers. This leads to a definition of $\mathcal{P}$-verifiability that relates the two in the context of a certifying IRM framework.

A total, computable, transparent rewriter $R : \mathcal{M} \to \mathcal{M}$ is said to be *sound* with respect to a policy $\mathcal{P}'$ if and only if $R(\mathcal{M}) \subseteq \mathcal{P}'$. Adding certification to an IRM framework removes a rewriter from the TCB, relieving us of the burden of proving rewriter soundness. Instead, we introduce a certifier that rejects any unsafe rewriter output on a case-by-case basis. Providing high assurance in this new framework requires proving certifier soundness.

A certifier decides some other property $\mathcal{P} \subseteq \mathcal{M}$. The certifier is *sound* with respect to policy $\mathcal{P}'$ if and only if $\mathcal{P} \subseteq \mathcal{P}'$. That is, sound certifiers accept only policy-adherent programs. Typically $\mathcal{P}$ is a strict subset of $\mathcal{P}'$. Programs in $\mathcal{P}' - \mathcal{P}$ are conservatively rejected; they are safe but unverifiable programs. *Complete* certifiers satisfy $\mathcal{P}' \subseteq \mathcal{P}$. Thus, certifier soundness and completeness together imply that $\mathcal{P}' = \mathcal{P}$. However, for this to be true, policy $\mathcal{P}'$ must be statically decidable (because $\mathcal{P}$ is decidable). Thus, for any non-trivial policy $\mathcal{P}'$, $\mathcal{P}$ is a strict subset and the certifier is sound but not complete.

While full certifier completeness is therefore impossible to achieve in the context of non-trivial policy languages, it is nevertheless important to obtain a certifier that is sufficiently complete to allow effective, certified rewriting of arbitrary binary code for a reasonably large class of security policies (e.g., the safety policies). We capture this idea a little more formally below.

**Definition 1** ($\mathcal{P}$-verifiable)**.** *Define $\mathcal{M}$ to be the universe of all programs and assume program property $\mathcal{P} \subseteq \mathcal{M}$ is statically decidable. A property $\mathcal{P}' \supseteq \mathcal{P}$ is* $\mathcal{P}$-verifiable *if there exists a total, computable, rewriter $R : \mathcal{M} \to \mathcal{M}$ that is transparent with respect to $\mathcal{P}'$ and that satisfies $R(\mathcal{P}') \subseteq \mathcal{P}$. That is, $R$ maps any program satisfying $\mathcal{P}'$ to a member of $\mathcal{P}$. A class of properties $\mathcal{C} \subseteq 2^{\mathcal{M}}$ is* $\mathcal{P}$-verifiable *if every member of $\mathcal{C}$ is $\mathcal{P}$-verifiable.*

Note that in the above definition, $\mathcal{P}$ is the property being verified and $\mathcal{P}'$ is the property being enforced. Informally, we refer to $\mathcal{P}'$ as $\mathcal{P}$-verifiable when there is a rewriter that enforces $\mathcal{P}'$ and whose output can be certified by a verifier that decides $\mathcal{P}$. Similarly, a class $\mathcal{C}$ of policies is $\mathcal{P}$-verifiable if a verifier for $\mathcal{P}$ suffices to certifiably enforce all policies in $\mathcal{C}$ with one or more rewriters (possibly different rewriters for different policies in $\mathcal{C}$ and different untrusted programs in $\mathcal{M}$). Definition 1 is useful because the suitability of an arbitrary code analysis $\mathcal{P}$ for purposes of IRM certification can be assessed by considering which policy classes are $\mathcal{P}$-verifiable. This provides a convenient and uniform means of connecting the IRM certification problem to related work.

Past work on IRM systems without certification typically argues rewriter soundness informally, due to the difficulty of formally verifying a full-scale rewriter implementation. For example, the SASI system [22] includes an informal argument that all control-flows that include potentially policy-violating operations in rewritten code are protected by a guard operation derived from partially evaluating a security automaton. In an AOP setting, the analogous proof involves showing correctness of the aspect-weaving algorithm. However, formally proving that each guard operation is adequate to preclude all policy-violations of the operations they protect in arbitrary untrusted code is much harder. It essentially means proving the correctness of the aspects that are woven—an undecidable problem in general.

Work on certifying IRM's [1, 28, 40, 49, 55] tries to make these arguments more rigorous by formally proving soundness on a case-by-case basis instead of proving soundness for the general rewriting mechanism. Conspec [1] shifts the burden of proving that guard instructions are adequate to the contract-writer. The contract specifies which guard instructions are required and where; it is trusted to encode an adequate implementation of the desired high-level policy. PittSFIeld [40] and NaCl [55] have an extremely rigorous proof of safety that rests on a machine-checkable ACL2 proof showing that its guard instructions are adequate, but only for a very specific, limited control-flow and memory-safety policy. Mobile [28] can verify far more general temporal properties, but is limited in which guards it can verify. It requires a specific dynamic state representation scheme with limited aliasing of security-relevant objects.

Model-checking approaches to IRM certification [16, 49] use security automata [3] for constructing a lattice for abstract interpretation. Policy violations are modeled as stuck states in the concrete small-step operational semantics, and the presented proof of certifier soundness involves establishing that the abstract machine is sound with respect to the concrete machine.

## 4. Runtime Code-Generation

The increasing ubiquity of runtime code generation constitutes one of the most significant outstanding challenges for effective, real-world IRM certification. A major class of examples are languages adhering to the *ECMA-262* standard [19] (e.g., JavaScript, ActionScript, etc.), which supports the built-in function `eval`. The `eval` function evaluates a (possibly dynamically generated) string argument as a program. Simply passing this runtime-generated code through a second round of rewriting is not feasible for the majority of IRM frameworks in which the rewriter is unavailable at runtime. For example, web ad frameworks typically assume that ad publishers or distributors perform the more significant rewriting task, whereas recipients simply certify.

While runtime code-generation is a consistent security concern in the broader language-based security literature [13, 20, 44], the majority of past IRM work assumes that such operations are sufficiently rare to justify conservative rejection of runtime-generated code, or sufficiently innocuous as to be safely ignored. Recent case

studies have contradicted these assumptions, showing that non-trivial, security-relevant use of `eval` is both widespread and a major source of cross-site scripting and similar attacks [44]. We therefore consider rewriting and certification of `eval` to be a significant but underdeveloped area of IRM research.

The main challenge from the verification perspective is that the string input to `eval` is typically constructed from several components, some of which are typically only available at runtime (e.g., user input). Static analysis of string inputs to `eval` is therefore widely recognized as challenging. Many static analyses either ignore it [4, 5, 31, 52], or supply a relatively inflexible dynamic monitoring mechanism that does not generalize to non-trivial generation of strings outside of a particular, fixed reference grammar [25, 32].

The hybrid static-dynamic monitoring approach of certifying IRM's seems potentially better suited to addressing this problem than purely static analyses, but the certifier must be sufficiently powerful to allow effective, flexible, yet provably sound rewriting for these domains. Several static analyses seem potentially promising in this regard. Christensen et al. [11] extract context-free grammars from Java programs and use a natural language processing algorithm to compute regular grammars that generate each string expression's language of possible values. Thiemann [53] presents a type system for analyzing string expressions, where type inferencing infers language inclusion constraints for each string expression. The constraints are then viewed as a context-free grammar with a nonterminal for each string-valued expression, and solved using algorithms based on Earley's parsing algorithm [18]. Minamide [42] presents an analysis of string expressions based on a prior Java string analyzer [11], but instead of transforming the extracted grammar into a regular grammar, they use trancuders to define a context-free grammar. Abstract parsing [17] strengthens the above by statically computing an abstract parse stack in an LR(k) grammar for each security-relevant string. The abstract parse stacks retain structural information about dynamically generated strings that can be checked by a tainting analysis or for inclusion in a reference grammar to detect attacks.

The suitability of each of these approaches as the basis for IRM certification can be posed as the following research question:

**Question 1.** *For each static string analysis that decides a property $\mathcal{P}$, is the class of safety policies $\mathcal{P}$-verifiable? That is, is there a total, computable, transparent rewriter function $R : \mathcal{M} \to \mathcal{M}$ such that for all safety policies $\mathcal{P}'$, $R(\mathcal{P}') \subseteq \mathcal{P}$?*

In order for the safety policies to be $\mathcal{P}$-verifiable, at any point where the static decision algorithm conservatively rejects, there must be a way to transform the code to include a statically verifiable dynamic check that conditionally preserves or prohibits the unverifiable behavior. Our intuition is that the existing work does not yet support sufficiently powerful dynamic checks to achieve this. A common inadequacy is the treatment of all conditional branching as non-determinism. There is no obvious way for a rewriter to generate meaningful guard instructions that convince such a certifier that the self-monitoring code is safe, since the content of the guard code is mostly ignored by the static analysis.

One possible solution is the development of a dependently-typed string analysis that can incorporate the test criteria of conditional branches into reconstructed types. The dependent types would expose information about program variables and the guard predicates that consult them to the string analysis in order to strengthen the resulting inferences. Such type-checking need not (and indeed cannot) be complete in order to be an effective means of certifying IRM's. It need only support a sufficiently powerful set of conditional tests that rewriters have useful options for inserted guards. That is, it need only decide a sufficiently powerful property $\mathcal{P}$ as to make safety properties $\mathcal{P}$-verifiable.

## 5. Concurrency

A second major challenge area for certifying IRM's is verifying the enforcement of safety policies in a multi-threaded environment. Consider the following code fragment that a standard IRM might use to enforce a resource-bound policy on a security-relevant resource. That is, a rewriter might enforce the resource-bound policy by replacing `use_resource()` operations with the fragment below.

```
if (count < limit) {
    use_resource();
    count := count + 1;
}
```

Here, `count` is a *state variable* introduced by the rewriter to track a conservative approximation of the security-relevant event history.

Clearly the above does not suffice to prevent policy-violations in the presence of concurrency. To do so, concurrency-aware IRM's must add some form of synchronization. One naïve approach is to surround the guard code above with lock-acquire and lock-release primitives so as to form a critical section. However, when the bounded resource is itself asynchronous (e.g., an asynchronous I/O operation) then this is unreasonably expensive. This situation is extremely common, so real-world IRM systems frequently implement a variety of complex, non-standard synchronization strategies in rewritten code (c.f. [9]).

Building a certifier for such IRM systems is a challenging task, and in this section, we explore some of the reasons for this challenge. Related work in the area of general verification of concurrent programs is vast and beyond the scope of this paper, but we here focus on work most directly related to verification of IRM-style, self-monitoring code—e.g., code that results from aspect-weaving.

There is a large body of related work on AOP dynamic detection of race conditions and deadlocks (e.g., [6, 8, 29]). There has also been some work done on dynamic detection of race conditions using aspects (c.f. [9]). Here, there is more hope for certifying that the instrumented code satisfies the security policy since there is a neat separation of concerns. Amongst certifying IRM's, those that enforce purely non-temporal policies (e.g., [40, 55]) can safely ignore the concurrency issue because they need not maintain a *history* of security-relevant events. ConSpec [1] leaves concurrency to future work, while Mobile [28] supports only one form of synchronization implemented as a trusted library. However, to our knowledge there has not been previous work that certifies general synchronization properties of IRM's without implicitly trusting the synchronization strategy by baking it into the trusted policy specification.

Building such a certifier would involve answering two major questions. First, is it possible to design a universal certifier that can machine-check programs instrumented with myriad different low-level synchronization primitives implemented by IRM's? Stated more formally:

**Question 2.** *When the language of a concurrent program domain $\mathcal{M}$ is augmented with a sufficiently versatile collection of low-level, trusted synchronization primitives, do the safety policies become $\mathcal{P}$-verifiable for some decidable property $\mathcal{P}$?*

The idea of "sufficiently versatile" is difficult to capture formally, but intuitively it encompasses at least two requirements: (1) The language must be flexible enough to afford rewriters a wide range of effective options for implementing certifiable synchronization strategies; these should include options for dynamically detecting and ameliorating concurrency bugs in a way that the certifier can identify as provably sound. (2) The language must allow for efficient synchronization of security-relevant events even when the events themselves may be asynchronous program operations.

The second major question involves specification of security policies. How should we specify security policies that involve po-

tentially asynchronous, security-relevant events? Consider a canonical sample IRM policy that enforces data confidentiality by prohibiting all network-send operations after the program has read from a confidential file [46]. In a concurrent setting, the temporal notion of "after" is ambiguous and requires a more precise definition. When network-sends and file-reads are non-atomic, possibly asynchronous operations, the policy must specify which interleavings are permissible. Fine-grained nuances must be expressible in order to formulate policies in a way that does not impose an undue performance overhead for self-monitoring code.

Related work in this area falls into two broad groups. One involves abstract concurrent policy specification languages such as the Pi-Calculus [41]. The other involves more practical tools such as aspect-oriented temporal assertions [50], tracematching [2], and their applications for race detection [9]. Stolz et al. [50] present a runtime verification framework for Java programs, where properties can be specified in LTL over AspectJ pointcuts. The work on AspectJ tracematching [2] enhanced with Racer [9] would allow for maintaining history in the presence of concurrent events.

Each group of work may provide important leads in answering Question 2: the abstract languages may provide a suitable framework for defining more formally the informal notion of "sufficient versatility", whereas the practical tools suggest useful concurrent IRM implementation strategies that future work should pair with corresponding static certification strategies.

Much past work on AOP for concurrent languages is devoted to automatic detection and avoidance of deadlocks, livelocks, and race conditions (e.g., [6, 8, 29]). While such flaws do not constitute violations of safety policies, they do constitute possible liveness policy violations. In addition, they break otherwise policy-adherent program behaviors, and therefore violate rewriter transparency. Reliable, static detection of such violations is therefore of critical interest to certifiers that prove transparency. We discuss this challenge in the next section.

## 6. Behavior-Preserving Runtime Monitoring

Almost all certifying IRM systems focus exclusively on proving soundness of rewritten code without considering the more difficult problem of proving transparency—i.e., that the behavior of policy-satisfying code is preserved. While transparency failures are often deemed less severe than policy violations, behavior-preservation is nonetheless a practical concern for many users. Indeed, in many mission-critical venues, non-transparency can be considered a denial-of-service. Formal, automated certification of transparency would therefore be a valuable contribution to the field.

Hamlen et al. [27] express transparency in terms of equivalence classes of programs under equivalence relation $\approx$, where $\approx$ denotes semantic equivalence. Transparency demands rewriter closure over relation $\approx$; however, relation $\approx$ is left abstract. Chudnov and Naumann [12] provide the first formal proof of transparency for a real IRM. Their system enforces information flow properties and therefore defines relation $\approx$ in terms program input-output pairs. In lieu of automated certification they manually prove general transparency of the rewriting algorithm.

Providing a definition of semantic equivalence that is applicable to more complex systems whose behavior is not precisely characterizable in terms of input-output behavior is difficult. For example, it is possible to encode unique behaviors as unique types (e.g., [51]), but the resulting equivalence relation is so strict as to preclude most IRM's that enforce history-based access-control policies. This highlights the need for an equivalence relation that successfully distinguishes code-transformations that affect only policy-violating program behaviors from those that potentially affect even policy-satisfying behaviors. The former should be accepted by a transparency-certifier, whereas the latter should not.

Jia et al. [33] recently introduced a dependently typed language that does not need decidable type-checking and therefore decidable program equivalence. The language has a type system that is parametrized by an abstract relation $\mathbf{isEq}(\Delta, \mathsf{e}, \mathsf{e}')$ that specifies program equivalence. The relation holds when $\mathsf{e}$ and $\mathsf{e}'$ are semantically equivalent in a context $\Delta$ of assumptions about the equivalence of terms. This suggests the following approach to certifying IRM transparency:

**Question 3.** *Is there a total, computable rewriter $R : \mathcal{M} \to \mathcal{M}$ that yields well-typed code satisfying $\mathbf{isEq}(\Delta_0, \mathsf{e}, R(\mathsf{e}))$?*

Formulation of such a rewriter would allow for a certification system that proves transparency of rewritten code through type-checking, rather than merely soundness. This would provide the first formal guarantees that rewriters for IRM systems do not corrupt the behavior of policy-adherent code.

## 7. Conclusion

Static certification of IRM systems is an emerging challenge that, if surmounted, offers to marry the power and flexibility of dynamic policy enforcement with the strong formal guarantees of purely static analysis. To formulate this challenge more rigorously, we introduced a definition of $\mathcal{P}$-verifiability that captures the notion that a purely static certifier of some conservative program property $\mathcal{P}$ suffices to allow provably secure enforcement of a more general policy $\mathcal{P}'$ by an IRM. We further argued that meaningful certification in these contexts requires IRM policy specification languages that express program properties in a purely declarative manner without overspecifying the policy's implementation.

These motivating discussions led to a presentation of three outstanding practical problems faced by developers of IRM systems today: IRM certification in the presence of runtime code generation (e.g., `eval`), certification of concurrent IRM code, and certification of behavior-preservation (transparency).

## References


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

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 345–364, 2005.

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

[4] C. Anderson and S. Drossopoulou. BabyJ: From object based to class based programming via types. In *Proc. Workshop on Object-Oriented Developments (WOOD)*, pages 53–81, 2003.

[5] C. Anderson and P. Giannini. Type checking for JavaScript. *Electronic Notes in Theoretical Computer Science*, 138(2):37–58, 2005.

[6] C. Artho, K. Havelund, and A. Biere. High-level data races. In *Proc. Int. Workshop on Validation and Verification of Software for Enterprise Information Systems (VVEIS)*, pages 82–93, 2003.

[7] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 305–314, 2005.

[8] S. Bensalem and K. Havelund. Dynamic deadlock analysis of multi-threaded programs. In *Haifa Verification Conf.*, pages 208–223, 2005.

[9] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *Proc. Int. Symp. on Software Testing and Analysis (ISSTA)*, pages 155–166, 2008.

[10] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proc. Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, 2005.

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proc. Int. Static Analysis Symp. (SAS)*, pages 1–18, 2003.

[12] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proc. IEEE Computer Security Foundations Symp. (CSF)*, 2010.

[13] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. Int. Conf. on World Wide Web (WWW)*, pages 281–290, 2010.

[14] D. S. Dantas and D. Walker. Harmless advice. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 383–396, 2006.

[15] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.

[16] B. W. DeVries, G. Gupta, K. W. Hamlen, S. Moore, and M. Sridhar. ActionScript bytecode verification with co-logic programming. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 9–15, 2009.

[17] K.-G. Doh, H. Kim, and D. A. Schmidt. Abstract parsing: Static analysis of dynamically generated string output using LR-parsing technology. In *Proc. Int. Static Analysis Symp. (SAS)*, pages 256–272, 2009.

[18] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 26(1):57–61, 1983.

[19] ECMA International. ECMAScript language specification, December 1999. ECMA Standard 262, 3rd Edition.

[20] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proc. Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 88–106, 2009.

[21] Ú. Erlingsson. *The In-lined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, NY, 2004.

[22] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. New Security Paradigms Workshop (NSPW)*, 1999.

[23] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Proc. Symp. on Operating Systems Design and Implementation (OSDI)*, pages 75–88, 2006.

[24] D. Evans and A. Twynman. Flexible policy-directed code safety. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 32–45, 1999.

[25] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. Int. Conf. on World Wide Web (WWW)*, pages 561–570, 2009.

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

[27] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

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

[29] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *Proc. Int. SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, 2000.

[30] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[31] D. Jang and K.-M. Choe. Points-to analysis for JavaScript. In *Proc. ACM Symp. on Applied Computing (SAC)*, pages 1930–1937, 2009.

[32] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. Int. Static Analysis Symp. (SAS)*, pages 238–255, 2009.

[33] L. Jia, J. Zhao, V. Sjöberg, and S. Weirich. Dependent types and program equivalence. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 275–286, 2010.

[34] M. Jones and K. W. Hamlen. Disambiguating aspect-oriented policies. In *Proc. Int. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 193–204, 2010.

[35] G. Kiczales, J. Lamping, A. Medhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. European Conf. on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[36] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance tool for Java programs. In *Proc. Int. Workshop on Runtime Verification (RV)*, 2001.

[37] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proc. USENIX Security Symp.*, pages 191–206, 2002.

[38] Z. Li and X. Wang. FIRM: Capability-based inline mediation of Flash behaviors. In *Proc. Annual Computer Security Applications Conf. (ACSAC)*, 2010. to appear.

[39] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proc. European Symp. on Research in Computer Security (ESORICS)*, pages 355–373, 2005.

[40] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Proc. USENIX Security Symp.*, 2006.

[41] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[42] Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. Int. Conf. on World Wide Web (WWW)*, pages 432–441, 2005.

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

[44] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–12, 2010.

[45] P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 131–144, 2010.

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

[47] V. Shah and F. Hill. An aspect-oriented security framework. In *Proc. DARPA Information Survivability Conf. and Exposition*, volume 2, pages 143–145, 2003.

[48] M. Sridhar and K. W. Hamlen. ActionScript in-lined reference monitoring in Prolog. In *Proc. Int. Symp. on Practical Aspects of Declarative Languages (PADL)*, pages 149–151, 2010.

[49] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proc. Int. Conf. on Verification, Model-Checking and Abstract Interpretation (VMCAI)*, pages 312–327, 2010.

[50] V. Stolz and E. Bodden. Temporal assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109–124, 2006.

[51] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 181–192, 1996.

[52] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. European Symp. on Programming (ESOP)*, pages 408–422, 2005.

[53] P. Thiemann. Grammar-based analysis of string expressions. In *Proc. ACM Int. Workshop on Types in Languages Design and Implementation (TLDI)*, pages 59–70, 2005.

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

[55] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 79–93, 2009.