MODEL-CHECKING IN-LINED REFERENCE MONITORS

by

Meera Sridhar

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Kevin W. Hamlen, Chair

Dr. Gopal Gupta

Dr. Bhavani Thuraisingham

Dr. I-Ling Yen

Om Saraswathi Namasthubyam

Vardey Kaamarupini

Vidhyarambham Karishyami

Siddhir Bhavathu Mey Sada

MODEL-CHECKING IN-LINED REFERENCE MONITORS

by

MEERA SRIDHAR, BS, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

August 2014

ACKNOWLEDGMENTS

The author expresses her sincere gratitude for being given the opportunity to conduct her doctoral studies under the supervision of her advisor, Dr. Kevin Hamlen. The author is deeply obliged to Dr. Kevin Hamlen for providing superior technical guidance throughout her studies, for being a constant source of inspiration, for setting a fine standard for meticulous science and technical writing, and for emphasizing the importance of conducting research that is backed by a combination of sound theory and strong practical applicability. The author is further indebted to him for providing her tremendous freedom in the pursuit of research, and for providing financial support for innumerable research endeavors, both of which she will always cherish. The author also thanks Dr. Hamlen for his indispensable contributions to this dissertation.

The author is obliged to her exemplary educators throughout her schooling for instilling in her a deep passion for mathematics and computer science, and producing the academician she is today. She would especially like to thank Mr. Horkan, Dr. Harry Bennett, Mr. William Marzen, Dr. Helmut Vogel, Dr. Tom Bohman, Dr. Peter Lee, Dr. Edmund Clarke, Dr. Michael Erdmann, Dr. Ramaswamy Chandrasekaran, and Dr. Neeraj Mittal. Special thanks go to Dr. Jeannette Wing at Carnegie Mellon University for introducing the author to computer science research and guiding her through undergraduate and Master's theses in model-checking.

The author is extremely grateful to her doctoral committee members, Dr. Gopal Gupta, Dr. Bhavani Thuraisingham, and Dr. I-Ling Yen, for their time, effort, and invaluable guidance throughout the writing of this dissertation. The author also thanks her colleagues and friends Dr. Feliks Kluzniak, Dr. Venkat Venkatakrishnan at University of Illinois, Chicago,

and Dr. Matthew Might at the University of Utah for numerous discussions and immensely valuable mentorship.

The author is indebted to her co-authors Brian DeVries, Scott Moore, Dr. Gopal Gupta, Dr. Micah Jones, Dr. Richard Wartell, Dr. Phu Phung, Maliheh Monshizadeh, Dr. Venkat Venkatakrishnan, and Dhiraj Karamchandani for their vital contributions to this dissertation. The author would also like to thank Peleus Uhley at Adobe, Inc. for several beneficial discussions, grant support letters, and for providing real-world SWF files of interest for testing and certification.

The research reported in this dissertation was supported in part by the U.S. Air Force Office of Scientific Research (AFOSR) under grant #FA9550-08-1-0044, the National Science Foundation (NSF) under grants #1065216 and #1054629, and by the Office of Naval Research (ONR) under grant #N00014-14-1-0030. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the AFOSR, NSF or ONR.

Staff at the Computer Science Department at The University of Texas at Dallas, especially Stacy Morrison and Rhonda Walls, deserve much appreciation for helping with countless administrative tasks throughout.

The author's doctoral studies would not have been possible without the priceless companionship of her friends. She is extremely indebted to her long-time friends Dr. Mehmet Tozal, Dr. Sunitha Ramanujam, Dolapo Kukoyi, Neslihan Deniz, Chinyelu Mozie, Shalini Patras, Anna Schlimme, and Charlese Galbraith, who celebrated with her during joyful moments, and helped her in times of need. Special token of appreciation goes to friends Dr. Yan Zhou, Jared Butler, Dr. Aravind Natarajan, Dr. Mustafa Canim, James Garrity, Jian Huang and Binal Kamani for providing assistance during key moments.

The author will be eternally grateful to her family's support for making this dissertation a possibility. The author is much appreciative of her younger brother, Narasimhan, for keeping her motivated throughout her doctoral journey, and just being a wonderfully kind, humorous, and witty sibling. She expresses her deep appreciation for her mother, who relentlessly supported her throughout her doctoral studies, and whose love and care transcended the physical distance to ensure continual pursuit of the final goal. The author is greatly indebted to her father for his patience, for his guidance, and for his tireless ardor for academics. She thanks both her parents for believing in her dreams absolutely and completely.

Most importantly, the author thanks her God Krishna for His divine blessings in enabling her to start and complete this dissertation.

May 2014

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

MODEL-CHECKING IN-LINED REFERENCE MONITORS

Publication No. _____

Meera Sridhar, PhD
The University of Texas at Dallas, 2014

Supervising Professor: Dr. Kevin W. Hamlen

The formidable growth of the cyber-threat landscape today is accompanied by an imperative need for providing high-assurance software solutions. In the last decade, binary instrumentation via *In-lined Reference Monitoring* (IRMs) has been firmly established as a powerful and versatile technology, providing superior security enforcement for many platforms. IRM frameworks rewrite untrusted binary code, inserting runtime checks to produce safe, self-monitoring code; IRMs are equipped with the ability to enforce a rich set of history-based policies, without requiring access to source code. These immense capabilities, however, come with a price—the power of most real-world IRM infrastructures is usually accompanied by both enormity and complexity, making them prone to implementation errors. In most scenarios this is highly undesirable; in a mission-critical system, this is unacceptable.

Independent certification of two important properties of the IRM, namely soundness and transparency, is extremely important for minimizing the error-space and providing formal assurances for the IRM. Soundness demands that the instrumented code satisfy the policy, whereas transparency demands that the behavior of policy-compliant code is preserved over the instrumentation process. These *Certifying In-lined Reference Monitors* combine the

power and flexibility of runtime monitoring with the strong formal guarantees of static analysis, creating automated, machine-verifiable, high-assurance systems.

This dissertation demonstrates how the powerful software paradigm of model-checking can be utilized to produce simple, elegant verification algorithms for the special domain of IRM code, providing case-by-case verification of the instrumented code. The dissertation discusses the challenges and subsequent success of developing model-checking IRM frameworks for various platforms. An important result discussed is a new, more powerful class of web security technologies that can be deployed and used immediately, without the need to modify existing web browsers, servers, or web design platforms. Implementations demonstrating security enforcement on a variety of fairly large-scale, real-world Java and ActionScript bytecode applications are also discussed. Theoretical formalizations and proof methodologies are employed to provide strong formal guarantees of the certifying in-lined reference monitoring systems.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION[1]

The last few decades have seen a relentless battle between software security attackers and defenders. Part of the challenge that security defenders face is the formidable, constantly-morphing attack space, complex source- and binary-level language features for writing software, combined with lack of complete information concerning the software they are trying to protect, such as source-level information.

For example, consider the web advertisement industry today. Web advertisements (ads) are an indispensable source of revenue for most webpage publishers today. Many security-sensitive websites, including websites for banking, health-care, and cloud-centered data management include web ads. Most web ads are derived from untrusted third-party ad-servers, creating a serious threat landscape, including threats to confidentiality of private client data (such as stealing cookies, hijacking sessions, etc.), integrity of the hosting webpage and user-owned content, and availability of the hosting site services.

Security apprehensions have been further exacerbated due to the recent trend of web environments, including ads, becoming aggressively heterogeneous (e.g., composed of *mash-ups* that mix mobile code from many mutually distrusting sources). An extremely popular combination of technologies for creating web ads is mixed JavaScript-ActionScript/Flash content. The reason for this popularity arises from unique features available through each platform, such as click-tracking and context customization in JavaScript, and sophisticated multimedia features from ActionScript/Flash. However, the complexity and idiosyncrasies

[1]This chapter includes previously published (Sridhar and Hamlen, 2011) joint work with Kevin W. Hamlen.

of each language, environment, and their interactive capabilities increase the attack surface significantly.

Traditional access control mechanisms built into modern web browsers do not suffice to protect users from these threats without also disrupting non-malicious advertisement functionality. For example, browsers can be configured to block all scripts or deny applets all access to page content, but this breaks the behavior of advertisement-loading mechanisms that consult page content when selecting relevant advertisements. Online advertising is a lucrative market; for many publishers it is an economic necessity. In fact, many webpage publishers today refuse access to content if ad-blockers are in place in the client-side browser. Therefore, such draconian measures are not reasonable in practice. More flexible and precise enforcement technologies are required to satisfy the security needs of webpage publishers and clients, as well as the financial needs of advertisers.

In the web advertisement domain, the challenge is to provide solutions that balance security and crucial revenue; other security scenarios, such as mission critical software, require providing solutions that balance environment-specific policy specification with iron-clad safety guarantees. Stepping back, the important question that begs to be answered is how to develop tools and technologies to aid security defenders that provide fine-grained, flexible policy enforcement, that are efficient, robust, and yet provide strong assurances for policy-adherence. This dissertation addresses several of these challenges.

1.1 In-lined Reference Monitoring

In-lined reference monitoring (cf., Yee et al., 2009; Chen and Roşu, 2005; Ligatti et al., 2005b; Schneider, 2000) has become a well-established software security enforcement mechanism in the last decade. In-lined Reference Monitors (IRMs) enforce safety policies by injecting run-time security guards directly into untrusted binaries. The guards test whether an impending operation constitutes a policy violation, taking corrective action to prevent the violation.

The result is *self-monitoring* code that can be executed safely without external monitoring. The approach is motivated by improved efficiency (since IRMs require fewer context switches than external monitors), deployment flexibility (since in-lining avoids modifying the VM or OS), and precision (since IRMs can monitor internal program operations not readily visible to an external monitor).

IRM's dynamically observe security-relevant events exhibited by the untrusted code they monitor, and maintain persistent internal state between these observations, enabling them to accept or reject based on the *history* of events observed. This allows them to enforce powerful security policies, such as safety policies, that are not precisely enforceable by any purely static analysis (Hamlen et al., 2006b). Additionally, IRM's afford code consumers the flexibility of specifying or modifying the security policy after receiving the code, whereas purely static analyses typically require the security policy to be known by the code producer.

Figure 1.1 presents psuedocode for a very simple IRM that prohibits more than three pop-up window opens. While the original and rewritten code is presented in source-style for clarity, the IRM is implemented in bytecode in actuality. In this example, `Popup.open()` is the security-relevant operation, the `count` variable maintains the history of security-relevant events, and `System.exit()` is the chosen intervention; the IRM implementation can choose other desired interventions.

```
                          L1: if (count ≤ 3)
L2: Popup.open();         L2:     Popup.open();
                          L3: else
                          L4:     System.exit();
                          L5: count++;
```

Figure 1.1. Original bytecode (left) that has been rewritten (right) with an IRM that prohibits more than three pop-up window opens.

Most modern IRM systems are implemented using some form of *aspect-oriented programming* (AOP) (Kiczales et al., 1997) (e.g., Viega et al., 2001; Shah and Hill, 2003; Dantas and

Walker, 2006; Dantas et al., 2008; Hamlen and Jones, 2008). *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 (Schneider, 2000; Hamlen et al., 2006b) and some liveness policies (Ligatti et al., 2005b). 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 (Chen and Roşu, 2005; Hamlen and Jones, 2008), .NET (Hamlen et al., 2006a), or ActionScript (Sridhar and Hamlen, 2010b), or by applying a sandboxing mechanism such as program shepherding (Kiriansky et al., 2002) or software fault isolation (Yee et al., 2009; McCamant and Morrisett, 2006; Erlingsson et al., 2006; Wartell et al., 2012; Sun et al., 2013).

1.2 Certifying In-lined Reference Monitors

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 (Jones and Hamlen, 2010). 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. For example, in the problem domain of web ad security (cf., Li and Wang, 2010; Sridhar and Hamlen, 2010a,b), 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 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 e, and rewritten code instance 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.

The problem of certifying IRMs 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., Holzmann, 2003), this dissertation shows that in the context of an IRM system, it is possible to create extremely small, efficient model-checkers that are powerful enough to verify large classes of IRMs formally (DeVries et al., 2009; Sridhar and Hamlen, 2010b). 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) (Necula and Lee, 1996; Necula, 1997) 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.

IRM certification has been implemented successfully in the past through the use of type-checking (Hamlen et al., 2006a) and contracts (Aktug and Naliuka, 2008). This dissertation extends the state-of-the-art in IRM certification in the following respects:

My Thesis. The main contribution of this dissertation is a technology for achieving IRM certification using *model-checking*. Model-checking is an extremely powerful software verification paradigm that fuels certification of properties more complex than those typically expressible by type-systems and more semantically flexible and abstract than those typically encoded by contracts. Challenges and subsequent success of developing model-checking IRM frameworks for various platforms are discussed. Certification is achieved on a case-by-case basis of instrumented code, allowing the IRM system itself to be completely untrusted. Certification is demonstrated for both soundness and transparency properties.

An important result discussed is a new, more powerful class of web security technologies that can be deployed and used immediately, without the need to modify existing web browsers, servers, or web design platforms. Implementations demonstrating security enforcement on a variety of fairly large-scale, real-world Java and ActionScript bytecode applications

are also discussed. Solid theoretical formalizations and proof methodologies are employed to provide strong formal guarantees of the certifier.

1.3 A Certifying IRM Example

Figure 1.2 shows an example of a certifying in-lined reference monitoring framework we developed for ActionScript Bytecode used in Adobe's Flash, Adobe Integrated Runtime (AIR), and related technologies. Chapters 2 and 7 motivate ActionScript security enforcement in detail. Chapters 3, 4, and 5 develop the *Soundness Verifier*. Chapter 6 develops the *Invariant Generator* and *Transparency Verifier*.

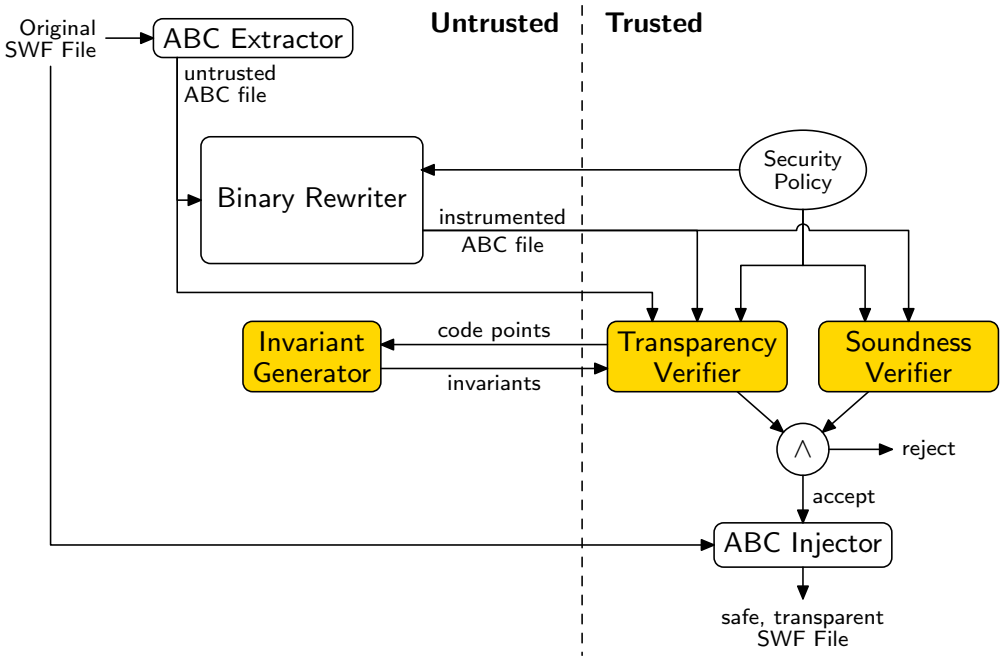


Figure 1.2. A certifying ActionScript IRM architecture

Our framework includes a collection of rewriters that automatically transform untrusted ActionScript bytecode into self-monitoring ActionScript bytecode. The untrusted code is obtained from *ShockWave Flash* (SWF) binary archives, which package ActionScript code with related data such as images and sound. Once the raw bytecode is extracted, a Definite Clause Grammar (DCG) (Shapiro and Sterling, 1994) parser converts it to an annotated

abstract syntax tree (AST) for easy analysis and manipulation. The rewriter then rewrites the bytecode according to the security policy, adopting the typical strategy of injecting guard code around potentially security-relevant bytecode instructions.

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.

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

1.4 Policy Specification for Effective Certification

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., Chen and Roşu, 2005; Erlingsson, 2004; Kim et al., 2004; Aktug and Naliuka, 2008; Bauer et al., 2005; Evans and Twynman, 1999). 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 (Aktug and Naliuka, 2008), 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 $\mathtt{e}'$ is checked against such a contract R by a process that essentially applies R to the original code $\mathtt{e}$ and tests whether $R(\mathtt{e}) = \mathtt{e}'$.

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 $R(\mathtt{e})$. (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, which 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 (Hamlen et al., 2006a). 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* (Rondon et al., 2010) and *dependent types* (Coquand and Huet, 1988; Leroy, 2011) 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* (Alpern and Schneider, 1986). 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 (Hamlen and Jones, 2008).

This dissertation shows that these specification languages are 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.

1.5 Certifier Soundness and Completeness

An important consideration for IRM system developers is a qualifier for policy-adherence assurance and computability limitations of IRM certification (Hamlen et al., 2006b). Specif-

ically, the interplay between the static analysis (the certifier) and the dynamic analysis (the rewriter) poses interesting mathematical relationships.

In this section, we present a mathematical framework for discussing these relationships. We start by outlining 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. We conclude the section by discussing treatment provided by past IRM works on soundness and completeness.

In order to examine what it means for an IRM to correctly enforce a policy, it is important to first define mathematically what an IRM is and what it does. An IRM rewriter R can be conceptualized as a total, computable function $R : \mathcal{M} \to \mathcal{M}$ from programs to programs. Rewriter R 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 (Erlingsson and Schneider, 1999) 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 IRMs (Hamlen et al., 2006a; Aktug and Naliuka, 2008; McCamant and Morrisett, 2006; Yee et al., 2009; Sridhar and Hamlen, 2010b) 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 (Aktug and Naliuka, 2008) 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 (McCamant and Morrisett, 2006) and NaCl (Yee et al., 2009) 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 (Hamlen et al., 2006a) 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.

The model-checking approaches to IRM certification that are contributed by this dissertation (DeVries et al., 2009; Sridhar and Hamlen, 2010b; Hamlen et al., 2012; Sridhar et al., 2014) use security automata (Alpern and Schneider, 1986) 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.

1.6 ActionScript Security

Many of the IRM implementations presented in this dissertation secure Adobe Flash binary programs. Adobe Flash applets (Shockwave Flash programs) provide web developers a superior platform for creating rich, dynamic web content such as web advertisements, online games, streaming media and interactive webpage animations, resulting in a soaring popularity of the technology on the web. Additionally, most of this content can also be made available to the desktop using the Adobe Integrated Runtime (AIR) cross-platform environment. The following statistics (Adobe Systems Inc., 2013c; W3Techs, 2013) demonstrate the

pervasive impact of Flash. More than 20,000 apps in mobile app stores such as Apple App Store and Google Play are created using Flash. A staggering revenue of over US$70 million per month is generated by the top nine Flash enabled games in China. Flash is used in 24 of the top 25 Facebook games. Flash is the choice of technology of more than three million developers for creating interactive and animated web environments. Flash is used by 16.9% of all websites.

The popularity of Flash combined with the complexity of its features has made it extremely attractive to attackers. The 2013 Cisco Annual Threat Report marks Adobe Flash as the third highest in the top content types for malware distribution (Cisco Systems, Inc., 2013). The 2013 Symantec Internet Security Threat Report mentions that of all plug-in vulnerabilities between 2010 and 2012, Adobe Flash Player constitutes 18%, 20%, and 22% in the years 2010, 2011, and 2012 respectively. Flash-powered attacks have successfully penetrated some of the most security-hardened facilities in the world, such as the famous 2011 penetration of RSA (Mikko, 2011), and the massive Luckycat campaign that targeted an entire spectrum of important U.S. industries such as aerospace, energy, engineering, shipping, and military research, as well as top-level international organizations such as Indian military research institutions, and groups in Japan and Tibet (Irinco, 2013).

One reason Flash security is so non-trivial is because of the feature-filled complexity of the ActionScript bytecode language (Adobe Systems Inc., 2013b), which Flash uses internally. Like other ECMAScript languages, ActionScript includes language features such as an object model, function calls, class inheritance, compile time and run-time type checking, packages, namespaces, regular expressions, and direct access to security-relevant system resources (Adobe Systems Inc., 2007). However, unlike JavaScript, ActionScript programs are disseminated as compiled binary Flash files (.swf files) that pack images, sounds, text, and bytecode in a webpage-embeddable form, which is then seamlessly JIT-compiled and/or interpreted by the Adobe Flash Player browser plug-in when the page is viewed. This trans-

parent purveyance of powerful binary content from Flash authors, through page publishers, to end-users educes many security threats.

Security apprehensions have been further exacerbated due to the recent trend of web environments becoming aggressively heterogeneous (e.g., composed of mash-ups that mix mobile code from many mutually distrusting sources), which expand the attack vulnerability surface area. Additionally, Flash's deployment as plug-in VM tends to widen threat windows due to patch lag. Newly discovered VM vulnerabilities are typically resolved by patches released by Adobe, but many consumers are apathetic or inattentive in downloading and installing the patches; in many large companies, older versions of the Flash plug-ins may be required for compatibility with critical business systems, creating reluctance in updating to the latest version. This lag in patch deployment rate has resulted in many home and large organization systems prone to Flash-attacks (Symantec Corporation, 2013). Consequently, effective intrusion detection of Flash-based attacks must consider a large array of past AVM versions and configurations. Due to the lack of a consistent, systematic solution to the problem, many security advisories suggest disabling Flash altogether as a fool-proof protection strategy (Sophos, 2013); unfortunately, this strategy is antithetical to the revenue models of many businesses.

Despite significant causes for concern, attention given by the formal research community has been disproportionately low compared to the gravity of the Flash security problem. For example, between 2008 and 2013 only 1.4% of publications in the top six non-cryptography security venues (ranked by Google Scholar h5-index) concerned Flash, and only one venue (the *IEEE Symposium on Security & Privacy*) devoted a large fraction of web security research (42%) to Flash-related threats (see Chapter 7 for a more detailed description of the scientific research methodology that yielded these statistics).

The in-depth discussion of the ActionScript language and features throughout this dissertation serves a dual purpose: (1) ActionScript provides a rich, complex, prevalent platform

for demonstrating many of the tools and techniques we developed for building certifying IRMs. (2) Due to the same richness, complexity and popularity, there is an urgent need to fill a void in formal study of the language, and tools for addressing the innumerable security concerns the platform faces today. This dissertation makes substantial progress towards developing robust security enforcement for ActionScript, building several formal analyses techniques for the language. Moreover, our tools and methodology subsequently contribute to achieving flexible, expressive security for mixed-content web advertisements, composed of ActionScript/JavaScript mash-ups.

1.7 Roadmap

The rest of the dissertation proceeds as follows. Chapter 2 explores security issues in cross-language web advertisement security, and proposes an IRM solution. Chapter 3 and Chapter 4 present preliminary ideas en route to using model-checking for IRM soundness certification: Chapter 3 presents a light-weight ActionScript bytecode verifier constructed using co-logic programming. Chapter 4 employs the ideas from Chapter 3 to create a prototype model-checking IRM framework for ActionScript bytecode, where the model-checking algorithm centers around a novel approach for using a security automaton to perform abstract interpretation.

Chapter 5 presents a full-scale certifying IRM framework for the SPoX Java IRM system, based on the algorithm developed in Chapter 4. Chapter 6 discusses IRM transparency certification—the chapter presents the first automated transparency verifier for IRMs.

Chapter 7 focuses on contemporary issues in ActionScript security, presenting a survey of ActionScript attack and vulnerability classes, and language/VM features that facilitate them. Chapter 8 presents related work, and Chapter 9 discusses conclusions and future work.

CHAPTER 2

SECURING MIXED JAVASCRIPT/ACTIONSCRIPT MULTI-PARTY WEB CONTENT[1]

2.1 Overview

JavaScript (JS) and Adobe ActionScript (AS) (the language for authoring Flash applet content) are two widely used platforms for developing web content. According to recent surveys on w3techs.com, 92% of *all websites* use JS and 23% of them use AS, demonstrating the popularity of these platforms for web development.

Due to these two platforms' popularity, much third-party web today contains mixed JS-AS content. By mixed AS-JS content, we refer to untrusted code that is primarily a combination of AS and JS. Such code is extensively used in interactive advertisements, embedded third-party videos, and plugins for content-management systems such as WordPress and Joomla. The popularity of such content stems in part from interactive and multimedia features that are uniquely available through each platform. Mixed AS-JS content leverages the benefits of both platforms: the interactive features of JavaScript for click-tracking and context customization, and the multimedia features of Flash for improving user experience.

Hosting sites that include such third-party content must deal with the security and privacy issues that such inclusions introduce. The most important security concerns tend to be confidentiality of private client data (e.g., document cookies), integrity of the hosting site and user-owned content, and availability of the hosting site services (e.g., ads must not prevent or dissuade users from visiting the site).

[1]This chapter includes joint work (Phung et al., 2013) with Phu H. Phung, Maliheh Monshizadeh, Kevin W. Hamlen and V.N. Venkatakrishnan.

The prior literature includes extensive research on addressing such third-party inclusion issues in the web. However, most of the solutions focus on third-party JS content. These include transformation of untrusted code (e.g., Google Caja, 2007; Microsoft Live Labs, 2009), security reference monitors (e.g., Phung et al., 2009; Meyerovich and Livshits, Meyerovich and Livshits; Cutsem and Miller, 2010; Heiderich et al., 2011; Meyerovich et al., 2010; Yu et al., 2007), and safe subsets of JS (e.g. Facebook Developers, 2010; Maffeis and Taly, 2009; Maffeis et al., 2009b; Finifter et al., 2010; Taly et al., 2011; Politz et al., 2011). To a lesser extent, there have been efforts to address security issues in Flash/AS as well (Li and Wang, 2010). In spite of these recent efforts, the security of *mixed AS-JS content* is relatively less researched.

Meanwhile, the abuse of mixed AS-JS content technologies to carry out malicious campaigns is a significant rising threat for untrusted content currently in circulation (M86 Security, 2010). For example, a vulnerability uncovered in Gmail allowed attackers to steal sessions by exploiting Gmail's use of the AS-JS interface (Amit, 2010a). Another attack on WordPress (CVE-2012-3414) exploits a vulnerable AS-to-JS interface call. A more recent study showed that 64 of over 1000 top sites contain vulnerable Flash applications that are exposed to JS XSS attacks (Acker et al., 2012). (Our evaluation discusses other real-world attacks).

A deeper examination of these attacks reveals that any defense mechanism that aims to prevent attacks arising from AS-JS interactions must adopt a *holistic* view of the security-relevant actions happening on both platforms. Prior work developed for JavaScript or for Flash has not been designed with this holistic perspective, and therefore does not satisfactorily address security issues arising from mixed AS-JS content. The problem of preventing malicious behaviors that exploit the combined use of AS-JS technologies has therefore remained open.

Problem Scenario: To illustrate the security challenges outlined above, consider a page publisher P who supports her site via embedded advertisements purveyed by an ad network N. Publisher P trusts neither the ads (some of which may be malicious) nor N (which may fail to filter some malicious ads, and whose ad-loading code might contain exploitable vulnerabilities). To protect the integrity and reputation of her site and retain clientele, P wishes to protect her clients from this malicious content.

Unfortunately, P cannot assume that all her clients take all available steps to protect themselves from the dangers that malvertisements pose. For example, some clients probably use un-patched browsers with known vulnerabilities. Finally, some of the policies P must enforce are specific to P's site or page content. For example, on a page that uses pop-up windows for legitimate navigation, P may wish to disallow all ad-generated pop-ups, which could fool clients with phishing attacks that impersonate the legitimate pop-ups.[2] P wishes to protect her clients as much as possible given these realities.

To host ads, N requires P to copy some JS ad-loading code provided by N onto her published pages. When this code is served to clients and executed, it dynamically modifies the hosting page within the browser to display dynamically chosen ads (implemented in JS, Flash, or both) served either by N or by the advertisers directly. Since the ad-loading code requires dynamic read and write access to the hosting page, it must not be placed in a protected *iframe*, nor may it be enclosed in any page element that disallows scripting. Such measures effectively deactivate ads, depriving P of most or all ad revenue. Likewise, many ads make heavy use of Flash-JS interaction (e.g., for click-tracking, contextual ad customization, and multimedia); therefore the hosting page must not disable such interaction lest it block many legitimate ads, losing significant ad revenue.

[2]Client-side pop-up blockers are not a viable defense, since P cannot assume that all clients use them or that they are all configured to distinguish the legitimate pop-ups from the illegitimate ones.

Our Approach: FlashJaX provides publisher P a means to enforce custom security policies on untrusted third-party ad and ad network content without deactivating the critical functionalities, like scripting and JS-Flash interaction, required by most ads. To use FlashJaX, P adds a ⟨`script`⟩ tag near the top of her published pages, which dynamically loads the FlashJaX IRM on client browsers before any other scripts run. She also statically labels any trusted, protected page content (e.g., publisher-authored JS code or Flash objects) with the owning principal (expressed as a principal-identifying *html* class attribute). Unlabeled Flash and JS code is, by default, fully untrusted by FlashJaX. Finally, she may write page-specific policies (detailed in Section 2.4) that define the events and event-traces that each principal may exhibit. To secure untrusted Flash content, P also hosts or accesses a trusted ad-proxy service that dynamically installs the FlashJaX IRM into untrusted Flash ads served to clients.

At runtime, the FlashJaX IRM dynamically monitors all untrusted JS and Flash code executed on client machines to enforce P's policies. As an example of such monitoring, consider the pop-up prevention policy mentioned above, which prohibits ad principals from exhibiting pop-ups but permits trusted publisher code from doing so. Pop-ups are implemented via a limited collection of JS Document Object Model (DOM) and Flash runtime API services. FlashJaX monitors calls to these services by intercepting them with guard code that first checks the impending operation against the acting principal's policy. FlashJaX passes the call through to the browser's underlying JS/Flash VM only if the principal's policy permits it.

To track the current principal, FlashJaX enforces history-based policies that constrain dynamically generated code and the events it exhibits. For example, a Flash ad owned by principal A that dynamically generates JS code that creates a new script within a region of the page owned by principal B must be successfully monitored by FlashJaX and constrained by policy A, not B. Such dynamic script generation is extremely common; almost all real-world ads and ad networks perform many layers of dynamic script generation and *html* tree

manipulation as they execute. Therefore, monitoring and constraining history-based policies (i.e., those that constrain event histories rather than just individual events in isolation) over dynamically generated, cross-platform code is a critical challenge addressed by our framework.

The remainder of the chapter is organized as follows: Section 2.2 sketches some attack scenarios that motivate securing the AS-JS interface. Section 2.3 outlines the FlashJaX architecture and technical approach. Design and implementation details are described in Section 2.4, and a security analysis is summarized in Section 2.5. Section 2.6 evaluates the implementation in terms of effectiveness, compatibility, and performance, and Section 2.7 discusses the relevance of FlashJaX to web security, and Section 2.8 concludes.

2.2 AS-JS Interface Attacks

In this section we describe the background of the AS-JS interface, and detail several motivating attack scenarios that exploit this interface to effect damage.

2.2.1 AS-JS interfaces

AS-JS interaction is implemented by two methods in the Adobe Flash runtime's `ExternalInterface` class: *call* and *addCallback*. AS calls JS method $f(a_1, \ldots, a_n)$ by invoking method $call(f, a_1, \ldots, a_n)$, where f is a string that is *passed uncensored to the JS VM and evaluated as JS code* to obtain a JS function reference, and where arguments $a_1, \ldots, a_n$ are arguments passed as values. The evaluation of expression f as JS code at global scope is a root of many vulnerabilities in AS-JS cross-language scripts. To permit JS to call AS, the AS code may invoke $addCallback(n, c)$, which registers AS function closure c as callable by JS under the pseudonym n (a name that is added to the JS namespace of the *html* object that embeds the AS script). Closure c may return a value, which is marshaled and passed by value back to the JS caller. Together, these facilitate two-way communication between AS and JS.

Security for this interface is provided by the `allowScriptAccess` property of the `object` and `embed` tags of the embedding page. In particular, the *call* method requires the `allowScriptAccess` property to be set to one of three options: `always` (full access), `sameDomain` (same origin access), or `never` (none); same origin access is the default. Additionally, by default JS may only call an AS closure registered with *addCallback* if the caller and callee originate from the same domain. AS callees may adjust this restriction using the *allowDomain* method of the Flash runtime's `Security` class.

While useful in some settings, these security features are too coarse-grained to distinguish malicious from non-malicious behavior in many contexts. Disallowing all AS-JS interaction or limiting it to same origin access breaks a large percentage of legitimate advertisement scripts. Therefore many ads and publishers resort to allowing all access, inviting attacks.

In the following subsection, we introduce some of the attacks that exploit the AS-JS interface. Such attacks can only be prevented by enforcements that span both languages. While the examples focus on AS-JS interface communication, FlashJaX is also designed to prevent attacks that are launched purely from JS or AS. However, to highlight the novelty of our system over prior works that can only guard each platform in isolation, we focus our discussion here on attacks that involve the interface.

2.2.2 Attack scenarios

Attack #1: Circumvention of SOP

The AS and JS VMs both enforce *Same Origin Policies* (SOPs) that prohibit cross-domain interactions. However, AS and JS SOPs have slightly different semantics (Zalewski, 2011a) due to their differing computation models, and these can presently be exploited to circumvent SOP on either side.

For example, a malicious Flash ad can abuse the AS-JS interface to circumvent AS's SOP in order to contact a victim third-party site. To do so, the malicious Flash ad dynamically

crafts a malicious JS script and passes it to the JS VM via Flash's external interface. The malicious JS script accesses the browser's DOM API to create a new ⟨`script`⟩ node (e.g., using `appendChild` or `document.write`). This new node has a `src` attribute whose URL references the third-party victim site. The URL can additionally contain information passed from the AS applet to the third-party site. The new node is not subject to AS's SOP, so it successfully contacts the remote site and retrieves the result, which is communicated back to the AS side using the external interface. Using this method, an attacker can escape the AS's SOP to perform two-way communication with the victim third-party site, which can be exploited to launch click forgery, resource theft, or flooding attacks.

This malicious behavior cannot be recognized by single-platform detection tools on either the AS or JS side, since AS permits (and ads regularly use) AS-to-JS communication, and JS permits (and ads regularly use) dynamic script generation. A cross-platform solution is required to link these two steps together and detect the SOP violation.

Attack #2: Malicious Payload Injection From Flash

Heap-spraying attacks are a form of code injection that first allocates large regions of malicious payload code into a victim VM's heap, and then exploits a control-flow hijack vulnerability (e.g., buffer overflow) in an effort to branch to the injected payload. Address space randomization and other protections prevent attackers from reliably learning the addresses of these injected payloads, but if the payload is large enough and has enough entry points, a randomly corrupted control-transfer targets it with high probability.

Since some vulnerabilities are previously unknown (i.e., zero-day), signature-matching malware protections often attempt to detect the payload injector instead, because it is larger and easier to identify using monitoring mechanisms. However, malware authors have been frustrating these defense efforts by using cross-language heap-spraying attacks (Wolf, 2009). In this scenario, the attacker implements AS code that sprays the JS VM's address space. The exploit is then implemented separately in JS. Identification of such attacks requires

cross-platform solutions that can piece together the two separate halves of the attack implementation.

Attack #3: Cross-Principal Resource Abuse

Sponsored pages often embed ads from more than one ad network. This exposes the publisher and one ad network to attacks launched by the (possibly less trustworthy) ads hosted by another network if those ads can abuse AS-JS interaction to hijack shared DOM resources, or improperly access functions and services exposed by victim scripts.

Flash has facilities for controlling access to exposed functions. For instance, the `allowDomain`(⟨*domains*⟩) call permits JS code from ⟨*domains*⟩ to access AS functions. However, the coarse granularity of these facilities makes it extremely common for ad developers to use them imprudently, such as by supplying wildcard "`*`" for ⟨*domains*⟩, which permits universal access (Elrom, 2010). This imprudent setting makes the AS functions accessible in the JS global scope, and hence can be invoked by all untrusted JS code.

Hosting sites cannot effectively filter ads by the quality of their underlying implementations, so inevitably some vulnerable ads become embedded in the served pages on the client side, exposing the clients to attack. For example, a malicious JS advertisement, even if sandboxed in the JS domain, can call such exported functions. This affords the ad illegitimate access to DOM objects if the exposed AS functions access or manipulate those objects in the DOM. Prevention of this attack requires the ability to attribute principals to actions across the AS-JS interface.

The above scenarios illustrate the need for cross-language monitoring. It is quite clear that JS sandboxing methods alone cannot prevent the attacks in scenarios #1–3. These scenarios involve the AS-JS boundary, which is typically outside the scope of approaches aimed at sandboxing purely JS or AS code. The next section describes how FlashJaX's architecture prevents these malicious scenarios.

2.3 Architecture

2.3.1 System Introduction

FlashJaX affords publishers a fine-grained mechanism to safely embed untrusted JS and AS content in their web pages. To avoid modifying the client browser or VMs (which would introduce significant deployment barriers), we adopt an in-lined reference monitoring approach.

The FlashJaX IRM consists of JS and AS code introduced by the embedding page. The IRM code mediates security-relevant events exhibited on the client, permitting or denying them based on a provider-specified policy.

A naïve design implements separate IRMs for the JS and AS platforms; however, this approach has many drawbacks. To enforce policies involving a global event history, separate IRMs must ensure that their security states are synchronized at every decision point. This raises difficult race condition and TOCTTOU vulnerability challenges, and impairs performance.

To avoid this, FlashJaX centralizes security state-tracking to the JS half of the IRM, and implements an AS side that shifts the significant policy decisions to the JS side. This is efficient because most security-relevant AS events include AS-JS communication as a subcomponent; the IRM therefore couples its AS-JS communications atop these existing ones to avoid unnecessary context-switches.

Figure 2.1 summarizes the resulting architecture. The FlashJaX components (shaded) consist of a JS-side event mediator, an AS-side event mediator, and a JS-side policy engine. The first two components intercept events from untrusted JS and AS, respectively, whereas the last tracks event history and makes policy decisions.

Step 1 of the figure depicts the exhibition of a security-relevant event *op* by the untrusted JS or AS code, which is intercepted by the IRM. If the event occurs on the AS side, the AS

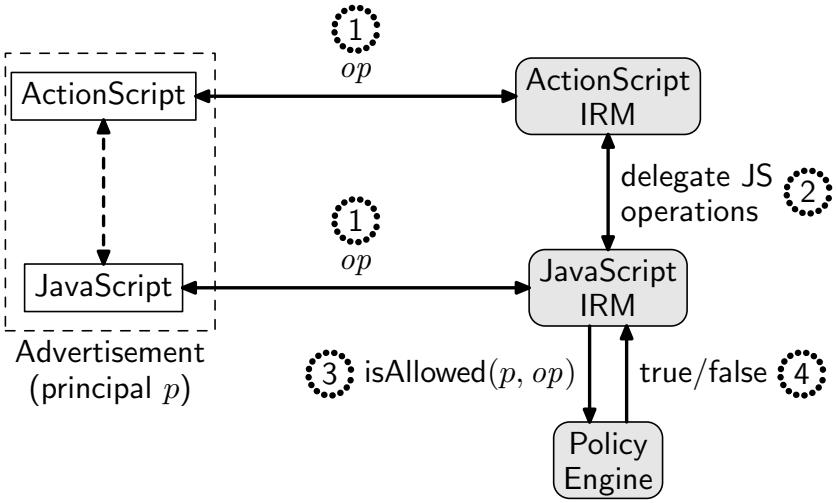


Figure 2.1. FlashJaX architecture. Trusted components are shaded; untrusted (monitored) components are unshaded.

IRM implementation consults the JS side in step 2. The JS-side IRM intercepts the event or AS-to-JS communication and consults a principal-specific policy in step 3. The policy engine updates the security state and yields a true/false answer in step 4, causing the operation to be permitted or suppressed.

As an example, an embedded AS ad might exhibit an *op* that spawns JS code that tries to overwrite the publisher's DOM. In a typical browser environment, there is nothing to prevent a malicious ad from successfully attacking its embedding page in this way. However, on a page equipped with FlashJaX, the ad's JS code is intercepted by the IRM and executed at a lower privilege level than the publisher. When the unprivileged write-operation is intercepted, the policy engine determines that the acting principal lacks write-access to publisher-owned content, and suppresses the operation.

2.3.2 Technical approach

The example above illustrates three essential capabilities of the IRM: It must (1) protect its own programming and other publisher-provided page content from harm, (2) guard access to

all security-relevant operations, and (3) attribute guarded events and page contents to acting/owning principals. We henceforth refer to these three capabilities as IRM *tamper-proofing* (i.e., *integrity enforcement*), *complete mediation*, and *principal-tracking*, respectively. In addition, to enforce multi-principal, history-based policies, the IRM must track both principal-specific and global security states. This section discusses our technical approach to achieving each of these goals.

FlashJaX implements a JS/AS cross-language IRM that constrains untrusted script access to the DOM API—functions and data properties that JS scripts access to manipulate the page and browser. AS code cannot access the DOM directly; instead, it submits strings to the JS VM via Flash's external interface, which are executed as JS code at global scope. The heart of our IRM is therefore a JS-side implementation that guards access to the DOM and tracks security state, while the AS half redirects external interface accesses to the JS half.

In addition to tamper-proofing and complete mediation, which are established challenges for any IRM implementation, our enforcement of multi-principal policies introduces significant challenges associated with principal-tracking. Accurate principal-tracking is challenging because modern ad scripts are highly dynamic, performing many layers of event-driven runtime code generation as they execute. Solutions that conservatively reject or lose principal information for dynamic code are therefore impractical because they break most ads.

We now describe each of these capabilities of FlashJaX at a high level. Section 2.4 discusses implementation details.

Tamper-proofing

Tamper-proofing ensures that the IRM's internals are unavailable to untrusted content. This is enforced differently at the JS and AS layers as described below.

At the JS layer, tamper-proofing is achieved by placing the majority of the IRM's implementation inside an anonymous JS function, as illustrated in Figure 2.2. Variable and

function declarations beginning with the **var** keyword are strictly local to the anonymous function's scope, and therefore cannot be directly accessed by JS code outside that scope unless the local scope explicitly exports global aliases to them. This enables the IRM to export and enforce a protected interface for its internal implementation.

```
(function(){                                          // begin local scope
  var principal = "bottom";                           // protected principal-tracking var
  getPrincipal = function(){                          // export global get-accessor
    return principal; };
  var wrap_window = function(w) {                     // wrap security-relevant op
    var o_open = w.open;
    w.open = function() {
      if (isAllowed(principal, "open", arguments))
        return wrap_window(o_open.apply(this, arguments));
      else return null; }
    return w; }
  wrap_window(window);
})();                                                 // close and execute local scope
```

Figure 2.2. A tamper-proof local scope.

A similar approach suffices to tamper-proof the AS half of the IRM. The majority of the in-lined AS code is implemented within a sealed, final, monitor class in a separate namespace. Type-safety and object encapsulation of AS prevent untrusted AS code from accessing the monitor class's private members.

JavaScript Mediation

FlashJaX mediates DOM API events by wrapping them with guard functions that consult the policy before forwarding the request to the DOM. To achieve complete mediation, the IRM assigns wrappers to all aliases of these security-relevant functions before any untrusted code runs. Aliases include static names and properties of dynamically created window objects (e.g., `frame` and `iframe`). Although some static aliases are browser-specific, all aliases of a given security-relevant operation can typically be captured by wrapping the properties of a single root object atop the JS prototype inheritance chain (Magazinius et al., 2010). The

wrapper assignment code is placed first on the page so that it is guaranteed to run prior to any untrusted code. Dynamically generated aliases are captured by mediating all DOM functions that can generate window objects, and wrapping any fresh aliases they introduce before returning control to untrusted code.

Data property accesses are guarded using JS *setters* and *getters*, which trigger specified handler code whenever an operation would otherwise read or write a given property.

ActionScript Mediation

The AS half of our IRM guards AS-to-JS control-flows by statically in-lining an external interface wrapper class into untrusted AS code at the binary level. FlashJaX's binary rewriter automatically, statically replaces all bytecode operations that access the external interface with ones that access the wrapper class instead. This affords the wrapper class complete mediation of all AS-to-JS flows.

Static identification of external interface references can be complicated by the fact that AS binaries frequently generate references to classes and their members dynamically (e.g., from strings). Malicious code can use such dynamic name generation to obfuscate references, concealing them from static analysis tools.

To avoid these complications, our rewriter therefore guards references to the external interface's *namespace* rather than its classes or members. The namespace part of an external interface reference is almost never generated dynamically. (The only AS mechanism for doing so requires a static reference, making it statically identifiable.) This approach greatly reduces the amount of in-lined code, improving performance and providing a natural resistance to reference obfuscation attacks.

Thus, all JS events invoked by AS are labeled with the originating principal and mediated by the JS IRM, so that the policy engine can apply the correct policy for each principal. For example, to block attack scenario #1 (Flash circumvention of SOP), the AS IRM labels

each AS-to-JS communication with the acting principal, allowing the JS IRM to enforce a whitelist policy that maps each principal to the domains it may access.

Principal Tracking and Event Attribution

Since there are multiple ads within a publisher page, each ad principal must be constrained by a distinct policy. We call this a *multi-principal policy*. Enforcing such policies is necessary to prevent scenarios such as cross-principal resource abuse (scenario #3 of Section 2.2).

To enforce multi-principal policies, FlashJaX deploys principal tracking and event attribution as follows. Whenever trusted code (e.g., the page publisher content) introduces untrusted code (e.g., by loading an ad), the untrusted code is launched using the IRM's `runAs` method, which defines and maintains the code's principal in a protected shadow stack. The stack stores a list of principal identifiers, one for each `runAs` frame on the JS VM's call stack. The IRM's `runAs` method is the only means by which the privilege level changes, and is strictly local to the IRM; untrusted code may not call it directly. The policy manager can read the shadow stack to identify the principal responsible for each event exhibited by the code, and thus apply a principal-specific policy.

Dynamic runtime code generation is a great challenge for principal tracking. FlashJaX addresses this challenge by catching all runtime code generation channels and wrapping them in new calls to `runAs`. We discuss this in more detail in Section 2.4.3.

Policy Engine

FlashJaX enforces safety properties, and must therefore track security state based on past events. Finite state automata (FSA) are a well-established formalism for doing so (Schneider, 2000; Ligatti et al., 2005a; Hamlen and Jones, 2008). FlashJaX therefore expresses policies as languages of permissible *traces*, where a trace is a sequence of principal-event pairs, and each event is a DOM operation parameterized by its argument values.

This event alphabet affords fine-grained control to the embedding page. For example, access permissions for page elements can restrict a principal's access to entire element subtrees, individual elements, or both. Our automata definitions are flexible enough to be used in any arbitrary DOM structure and expressive enough to allow specification of stateful constraints.

Security state is tracked on both a per-principal and a global basis. For example, a policy can specify that each principal may exhibit at most one pop-up window, and collectively they may exhibit at most three. This facilitates enforcement of policies that restrict individual principal behaviors as well as group behaviors.

Using this approach, FlashJaX is able to define and enforce not only policies for AS-JS cross-language interaction but also history-based and multi-principal policies. These policies are powerful enough to detect and prevent a wide range of attacks, including the attack scenarios described earlier. For example, FlashJaX can monitor the data sent by each AS principal to JS so that it can detect whether the cumulative transmission size and content suggests a possible cross-language heap spraying attack (scenario #2 of Section 2.2). A detailed exposition of policy specifications and their expressiveness is provided in Section 2.4.4.

2.4 Implementation

This section describes implementation details of FlashJaX that have been briefly introduced in the previous section.

2.4.1 JavaScript Wrappers

FlashJaX implements wrappers to mediate DOM API access. Figure 2.2 illustrates a wrapper that guards the `window.open` DOM function, which creates a pop-up window, by assigning `window.open` $= f$, where f is a function that creates the requested window if and only if the current principal's policy permits it. Thereafter, all calls to `window.open` call wrapper f instead.

Naïve JS wrapper implementations are known to be vulnerable to a variety of attacks, including *prototype poisoning* and *caller-chain abuse* (Magazinius et al., 2010; Meyerovich et al., 2010; Meyerovich and Livshits, Meyerovich and Livshits). FlashJaX therefore employs secure wrapper implementations advanced by prior works (Magazinius et al., 2010) to defend against these attacks. The safe wrappers (1) wrap all aliases of each security-relevant operation, (2) explicitly coerce untrusted inputs to expected types, and (3) only call securely stored copies of JS API methods (e.g., those of *Function* and *Array*), which are carefully protected from attacker corruption. FlashJaX augments these secure wrapper implementations to additionally mediate events from AS, and to precisely attribute events to the correct principal, even if the event-exhibiting code was generated dynamically.

In order to mediate data property accesses, FlashJaX leverages the `defineProperty` function (ECMA International, 2011, §15.2.3.6) to define setters and getters for a given property. All browsers compliant with the ECMAScript 5 standard[3] support this function. The getters and setters read and store values to protected, locally-scoped, principal-specific copies of each guarded property. The guarded properties are set *non-configurable* so that untrusted JS code cannot remove or change the guards. Global variables introduced by scripts are similarly protected from abuse by other scripts by adding non-configurable getters and setters to such variables during privilege-changes (i.e., within `runAs` from Section 2.3.2).

A special approach is required to adequately guard the DOM's `document.cookie` property, for which writes have the side-effect of creating or modifying browser cookies that may persist across sessions, and reads yield lists of previously written cookies (possibly some from prior sessions). FlashJaX employs two browser-dependent techniques to protect cookies: On browsers that support cloning of the `document` node (e.g., FireFox and IE), FlashJaX creates a local, protected copy of `document`, which the IRM's wrappers can henceforth exclusively

[3]Safari does not currently comply with this part of the standard, preventing protection of data properties on Safari. However, the rest of the DOM remains protected.

access to safely read and write cookies. On browsers that implement cookie facilities as browser-specific getters and setters of `document.cookie` (e.g., Opera), FlashJaX creates local, protected copies of these getters and setters to mediate access to them.

In both cases, FlashJaX introduces replacement getters and setters on the global `document.cookie` property to provide filtered, principal-specific views of the cookie store for each untrusted principal. (The trusted hosting origin's access is not filtered.) Each cookie's untrusted owning principal is added to the cookie's name when stored, and checked when retrieved. This confines each untrusted principal's cookie accesses to its own cookies.

One browser we tested (Chrome) currently admits neither approach due to a known browser bug,[4] preventing us from protecting cookies on that browser. However, once this bug is fixed, FlashJaX's cookie-protection is expected to be compatible with all major browsers.

2.4.2 ActionScript Rewriter

We implemented a binary rewriter that automatically in-lines wrappers around all AS-to-JS flows within AS bytecode applets. The in-lined wrapper class redirects all such flows to a JS method named `fromAS` exposed by the JS IRM. For example, a JS call originally of the form $f(a_1, \ldots, a_n)$ is translated by the wrapper into a JS call of the form $\texttt{fromAS}(id, s, f, a_1, \ldots, a_n)$, where id identifies the principal, s is a one-time secret (discussed below), f is a JS expression identifying the callee, and $a_1, \ldots, a_n$ are the arguments to f. The `fromAS` method then executes $f(a_1, \ldots, a_n)$ at privilege id.

Impersonation Attack & Defense. The `fromAS` function must protect itself from impersonation attacks in which a malicious JS principal calls it with a false id. JS callees cannot reliably identify their callers; incoming calls are essentially anonymous. Therefore, the `fromAS` implementation calls-back the AS applet from which each incoming AS call

[4] http://code.google.com/p/chromium/issues/detail?id=45277

claims to originate, asking it to confirm the call. The AS-side IRM confirms by validating secret s. Secret s is freshly chosen for each AS-to-JS call, exists only for the lifetime of the confirmation process (just a few AS/JS instructions), is temporarily stored on the AS side in a private field of the monitor class, and exists on the JS side only as a local argument to `fromAS`. This keeps it safe from interception by attackers during the limited window when it is valid.

2.4.3 Principal Tracking and Event Attribution

As mentioned earlier, FlashJaX tracks principals by maintaining a protected shadow stack, whose implementation is sketched in Figure 2.3.

```
var shadowStack = [ ]; // Implement a shadow stack as a list.

// Other code may read (but not write) the current principal.
thisPrincipal = function(){
  return (shadowStack.length < 1) ? "bottom" :
         shadowStack[shadowStack.length − 1];
}

// Execute closure f at a specified privilege level.
var runAs = function(principal,f) {
  shadowStack.push(principal);
  f.apply = js.Function.apply;
  var r = f.apply(this,
    js.Array.prototype.slice.call(arguments,2));
  shadowStack.pop();
  flush_write(principal);
  if (typeof r !== "undefined") return r;
}
```

Figure 2.3. Shadow stack code. Object `js` stores original native JS objects. Exception-handling is not shown.

Principal-tracking Algorithm. To execute an untrusted function f at privilege level p, the IRM invokes `runAs`(p, f), which pushes principal identifier p onto the shadow stack, runs f to completion, pops p off the shadow stack, and returns the result. Note that since f is a

closure with its own context, calling f within the lexical scope of the monitor does not give it access to anything in the IRM's local scope. Its scope continues to be whatever context it had when it was created.

As f executes, it may exhibit security-relevant events, which are intercepted by the IRM. The IRM's guards consult `thisPrincipal()` to determine the principal to whom each event should be attributed. Based on the result, a principal-specific policy is then consulted to determine whether to grant or deny each event.

The (trusted) embedding page may label static code f with a principal identifier p, causing the IRM to execute f at p's privilege level. Trusted (non-ad), static code is therefore typically labeled with identifier *top* ($\top$), which grants it full privileges. Untrusted, static code for ads is labeled with ad principals so that it executes with lesser privileges. Unlabeled code runs with *bottom* ($\bot$) privileges by default—i.e., the intersection of all privileges granted to all the principals in the system.

Dynamically Generated Code. As callee f runs, it may attempt to modify the page, such as by adding new elements with event-handlers containing code. The DOM provides several mechanisms for dynamic page modification (e.g., `Node.appendChild`), all of which are monitored by the FlashJaX IRM. No special monitoring is required for `eval`, since the code it generates inherits the context of the generating `eval`, preserving the contents of the shadow stack. Thus, FlashJaX handles all dynamically generated code. To illustrate the IRM's handling of such dynamic code, we here consider the most common and most general means of generation: `document.write`.

Operation `document.write`(s) pushes string s directly onto the head of the browser's input stream during page-loading. Browsers execute scripts as soon as they are parsed during the page-loading process, so these dynamic scripts run sometime after the generating script writes them but before the page is fully loaded. (The exact time of execution is

browser-specific.) Ads depend on this behavior, so it is important to support and preserve it.

To do so, FlashJaX intercepts and buffers strings passed to `document.write` (by storing them in the `write_buffer` variable as illustrated in Figure 2.4) without immediately committing them to the page. Once f completes, `runAs` calls the algorithm sketched in Figure 2.4 to parse these buffered strings, label the resulting HTML and JS code with the contributing principal's identifier, and commit it to the page. To avoid writing our own parser, we use a trick: Assigning to the `innerHTML` property in line 3 leverages the browser's built-in parser to convert the string into an HTML tree that is placed in the body of an ⟨`ins`⟩ node object.

Figure 2.4 replaces all code in the new content with closures that recursively call `runAs`, so they will run at the proper privilege level when triggered. For example, lines 11–13 replace event-handler `e.a` with such a closure. The JS closure semantics guarantee that when this closure is executed, `principal` has the value that was passed to this invocation of `flush_write` (which might differ from the code that calls or triggers the closure), and `oldHandler` executes at its original scope (not the IRM's protected scope). Thus, dynamically contributed code inherits the privileges of its creator.

Line 18 processes JS code contributed in the body of a dynamically generated ⟨`script`⟩ element. IRM subroutine `makeFunction` (implementation not shown) uses JS's `Function` constructor to convert its string argument into a closure that can be called by `runAs`. Closures created with `Function` always have global lexical scope, and therefore safely exclude the IRM's local scope. The new closure is executed immediately, since that is how most browsers treat dynamically contributed scripts.

In addition to the local ⟨`script`⟩ content handled by Figure 2.4, the full FlashJaX implementation also handles remote scripts (specified as a URL in a `src` attribute) by loading them through a proxy via `XMLHttpRequest` and processing the resulting string as a local script. This step is omitted from the listing for brevity.

```
1  var flush_write = function(principal){
2    var i = document.createElement("ins");
3    i.innerHTML = write_buffer[principal];
4    write_buffer[principal] = "";

6    foreach element e within i do {
7      // Enclose handlers in principal−preserving closures.
8      foreach attribute a of e do
9        if (typeof e.a == "function") {
10         var oldHandler = e.a;
11         e.a = function() {
12           var r = runAs(principal, oldHandler);
13           if (typeof r !== "undefined") return r; };
14       }

16     // Execute scripts at generating principal's privileges.
17     if (e is a ⟨script⟩ element) {
18       var newScript = makeFunction(e.textContent);
19       e.textContent = "";
20       runAs(principal, newScript);
21     }

23     // Wrap any fresh aliases of security−relevant functions.
24     if (e is a ⟨frame⟩ or ⟨iframe⟩ element) {
25       wrap_window(e.contentWindow);
26       wrap_document(e.contentWindow.document);
27     }
28   }

30   i.owner = principal;
31   document.lastChild.appendChild(i); // Append i to page.
32 }
```

Figure 2.4. Wrapping dynamically-generated code.

Finally, any dynamically generated window objects introduce fresh, unguarded aliases to security-relevant operations protected by the IRM. These are wrapped with suitable guards by lines 24–27.

2.4.4 Policy Definition and Enforcement

FlashJaX's policy engine is implemented in three layers of JS as depicted in Figure 2.5. The bottom layer provides two library classes, *FSM* and *GlobalFSM*, which are built on the FSMJS library (Jähnig, 2010). They provide tools for defining and accessing policy files within the policy engine.

The next layer defines global and per-principal policies. In a typical policy file, page publishers define security states, the initial state, forbidden states (i.e., those rejected by the security automaton) and the transition relation. There is typically one policy file per principal, plus a global policy file that constrains all untrusted principals and their interactions.

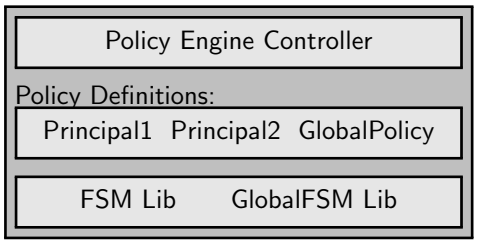


Figure 2.5. Policy enforcement system architecture

The third layer is the *Policy Engine Controller*, illustrated in Figure 2.6, which interfaces the policy engine to the monitor. Publishers assign policies by adding policy class instances to the `Ctrl.policies` array in the controller. At runtime, the controller calls `checkPolicy` to test whether the global FSM and acting principal's local FSM accept the impending event. If so, the controller updates the FSM states; otherwise it rejects.

With this design, FlashJaX's policy language accommodates history-based stateful policies on events triggered by various principals. These events include API calls or getter/setter

```
Ctrl.checkPolicy = function(principal, event, obj, flags){
  localFSM = Ctrl.policies[principal][FSM_CTRL];
  return localFSM.checkPolicy(event, obj, flags) &&
      globalFSM.checkPolicy(principal, event, obj, flags);
}
```

Figure 2.6. Policy engine controller

functions with various arguments such as DOM objects or global variables. Some expressive policy examples are illustrated below.

Formal Description. FlashJaX defines and enforces safety policies expressible as security automata (Alpern and Schneider, 1986) or edit automata (Ligatti et al., 2005a) that intervene by suppressing policy-violating events. (Other interventions are possible, but we have found suppression to be the most useful and practical for our policies.) Formally, a FlashJaX policy is a quadruplet $\langle P, E, S, G\rangle$ where P is the universe of principal identifiers, E is the universe of events, $S : P \rightarrow RE$ is a mapping from principals $p \in P$ to regular expressions over alphabet $\{p\} \times E$, and G is a regular expression over alphabet $P \times E$. Regular expression $S(p)$ specifies the language of permissible traces for principal p, and G specifies the language of globally permissible traces. The system-wide policy is therefore given by regular expression $\bigcap_{p\in P} S(p) \cap G$. Intuitively, the policy identifies the set $S(p)$ of event sequences that each individual principal p may exhibit, and an additional set G that all untrusted principals as a collective may exhibit.

Policy Example. Figure 2.7 shows a policy that prevents cross-platform heap-spraying attacks (scenario #2 of Section 2.2). Such attacks conceal themselves by implementing the spray entirely in AS so that it is not visible to JS analysis tools. The sprayed payload is then passed across the AS-JS boundary, allowing malicious JS code to branch to the payload via a JS-side exploit not visible to AS analysis tools.

The FSA in Figure 2.7 prevents such behavior by tracking the cumulative size of data passed from AS to JS by each untrusted principal p. When the cumulative transmission

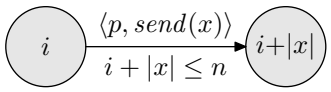


Figure 2.7. A local FSA for a policy preventing heap-sprays.

size reaches bound n, future transmissions are rejected. The policy therefore conservatively rejects applets that pass suspiciously large quantities of data from AS to JS. Our experience is that only malicious ads exhibit such behavior, but a more refined policy could additionally apply malware detection heuristics to the passed payloads to support non-malicious ads that pass large quantities of data to JS for legitimate purposes.

The FSA for this policy consists of $n + 1$ states, where n is the maximum cumulative transmission bound. (The number of states is not an implementation burden, since all $n+1$ can be expressed as a single integer whose values range from 0 to n.) For brevity, we draw the FSA using the notation of extended finite automata (XFAs) (Smith et al., 2008) in Figure 2.7 to visually depict the automaton.

In the same way, a global policy can also be defined to limit the total size of cumulative data transmissions by all principals by using $*$ (denoting any principal) on the edges. This blocks heap spraying through collusion.

Multi-principal Policies. FlashJaX's label-based attestation (Section 2.4.3) facilitates enforcement of some sophisticated write-protection policies, which can be leveraged to address cross-principal resource abuse (e.g., scenario #3 of Section 2.2). FlashJaX's label-based

Figure 2.8. Local FSAs for a policy that permits ads p_2 and p_3 to read data owned by ad network p_1 but not data owned by each other.

attestation (Section 2.4.3) facilitates enforcement of some sophisticated write-protection policies, which can be leveraged to address cross-principal resource abuse (e.g., scenario #3 of Section 2.2). Figure 2.8 shows an example with three principals: an ad network p_1, and two ads p_2 and p_3 served by the network. Event $read(e, p)$ denotes a read from element e labeled p. The label is assigned dynamically by the IRM's attestation mechanism. The FSA on the left allows p_2 to read p_1's data and its own data, but not p_3's data. Similarly, the FSA on the right prohibits p_3 from reading p_2. Thus, ads may consult the ad network but not each other. Wildcard $*$ is used to denote edge labels ranging over all principals and event arguments ranging over all values. Self-edges for events that do not affect the security state are not shown.

Other Policy Examples. Using this policy language, we designed and implemented policies that address several attack scenarios, including the three attack scenarios described in Section 2.2, which abuse AS-JS interactions. These are described below. As mentioned earlier, these attacks cannot be prevented by mechanisms in JS or AS alone. Other policies are discussed in Section 2.6.3.

To stop Flash circumvention of SOP (scenario #1), FlashJaX enforces a principal-based whitelist policy: *Each principal may only communicate with sites defined in a whitelist.* FlashJaX's principal-tracking and event attribution mechanisms attribute all JS code called from AS. Therefore, FlashJaX is able to identify whether the JS event originates from an AS principal, and apply an appropriate policy. The security policy enforces SOP by accepting communications with whitelisted sites and suppressing other communications.

To inhibit cross-language heap sprays (scenario #2), FlashJaX enforces a multi-principal, history-based, resource-bound policy: *The cumulative AS-JS data transmission by each principal may not exceed a per-principal bound defined by the policy, and the total transmission by all principals may not exceed a global bound defined by the policy.* The size of transmissions by each AS principal is tallied by the policy engine. If it exceeds the limit, FlashJaX destroys the violating Flash object by removing it from the page to prevent the attack.

To block cross-principal resource abuse (scenario #3), FlashJaX enforces principal-based access control policies: *Each principal may only access particular page elements.* The legitimate accesses for each principal are defined by a whitelist of DOM objects. The IRM monitors all DOM tree accesses and disallows accesses that originate from unauthorized principals.

2.5 Security Analysis

FlashJaX enforces *rewrite-enforceable safety policies* (Hamlen et al., 2006b)—i.e., trace properties that stipulate that some observable, decidable "bad thing" (possibly contingent upon the history of past events) must not happen. Security-relevant events consist of JS API calls and member accesses, parameterized by their arguments and a principal identifier. Prior work has shown that such policies can be formalized as aspect-oriented security automata (Hamlen and Jones, 2008). Principals are defined by the embedding page, which provides a trusted mapping from untrusted scripts to principal identifiers. Dynamically generated scripts inherit the identifier of the code that generates them.

The IRM's ability to enforce these policies is contingent upon its ability to (1) maintain IRM integrity (i.e., tamper-proofing), (2) completely mediate security-relevant events, and (3) accurately attribute events to principals. The enforcement strategy for each of these goals forms the foundation for enforcing the next, as illustrated in Figure 2.9. Each tier of security is described below.

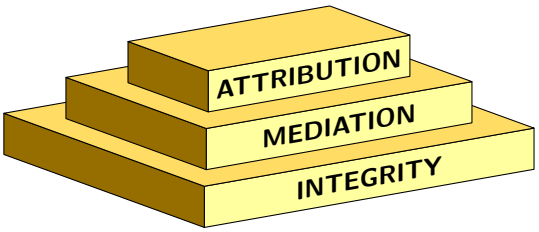


Figure 2.9. Three tiers of FlashJaX security.

IRM Integrity. IRM integrity follows from two core language features: lexical scoping on the JS side, and object encapsulation (type-safety) on the AS side. That is, on the JS side, all security-critical data and code are stored within the local scope of an anonymous JS function. This prevents any access to them except via closures exported from the anonymous function as global variables. This closure collection constitutes the protected interface to the IRM. Similarly, on the AS side, all security-critical data and code are stored as private members of a final, sealed AS class. Integrity of the AS portion of the IRM therefore follows from the type-safety and object encapsulation guarantees of the AS bytecode language.

Complete Mediation. Complete mediation of JS API calls is achieved by moving all security-relevant API method pointers inside the protected lexical scope before any untrusted code runs. For a given security-relevant API method, FlashJaX systematically explores and wraps all its aliases, including static names and dynamic aliases (Section 2.3.2). Furthermore, FlashJaX also wraps all channels generating JS code at runtime (Section 2.4.3). Thus, IRM integrity implies complete mediation of these events; once they are inside the local scope, they can only be accessed via the protected IRM interface.

Mediation of data property accesses is via non-configurable JS getters and setters, whose complete mediation is guaranteed by the JS VM (ECMA International, 2011). This setting makes it impossible for untrusted code to change or delete the properties of wrapped objects.[5] Complete mediation on the AS side is achieved by statically rewriting all references to the `flash.external`, `flash.net`, and `flash.utils` namespaces (except those within the trusted IRM class) before any AS code runs. This makes it impossible for any untrusted AS code to acquire a direct reference to any external interface member; all JS accesses must therefore use the AS IRM.

[5]In earlier versions of Mozilla browsers, deleting a wrapped object could result in silent restoration the original object (Phung et al., 2009), however this is no longer possible with the non-configurable feature in the current ECMAScript specification (ECMA International, 2011).

Accurate Event Attribution. This follows from complete mediation of security-relevant events, which include all page- and code-write operations. The IRM's write-mediation labels all dynamically written content with the authoring principal. JS code is labeled by dynamically replacing it with a closure that preserves the principal. Thus, when it runs, it inherits the privileges of its author.

For this labeling to succeed, the IRM must account for all possible locations where JS code can be dynamically submitted and stored. For example, if the JS `setTimeout` method is inadvertently omitted from the list of mediated methods, scripts could use it to escape the labeling mechanism and run unlabeled code. Since in practice the JS API has a broad, browser-specific, and ever evolving surface, we consider this to be the most attackable portion of our system. To make FlashJaX robust against such omissions, unlabeled code therefore always runs at the lowest privilege level (defined as the intersection of permissions granted to all principals in the system). Thus, a principal-tracking failure could lead to conservative rejection, but never a policy violation.

Correctness of the guard code that enforces each principal's policy is facilitated by our choice of an automaton-based policy formalism whose semantics, expressive power, and correct implementation are extremely well-established in the literature (cf., Schneider, 2000; Hamlen et al., 2006b; Ligatti et al., 2005a; Martinell, 2006; Hamlen and Jones, 2008). Our implementation leverages these solid design principles for high assurance.

2.6 Evaluation

2.6.1 Code and Experiment Settings

The core JS IRM is a 300-line static script atop the hosting page that wraps DOM functions before untrusted code runs. The wrappers consist of about 600 more lines that mediate security-relevant events, including dynamic writes, by consulting the policy engine. The

policy engine implements FSMs using an adaptation of the `fsmjs` library (Jähnig, 2010) (about 9K LOC). Each individual FSM-controller contributes less than 100 LOC.

Our AS binary rewriter is implemented as a small (< 1K SLOC) stand-alone Java application that uses no external libraries except the Java standard libraries. It injects the wrapper class (about 700 bytes of pre-compiled AS bytecode) and redirects all external interface references to the injected wrapper methods. Rewriting is fast; the median rewriting time was 0.62ms/K (averaged across 57 Flash ads on a 2.93 GHz, Intel quad core desktop running 64-bit JDK 1.7.0 atop Windows 7 SP1 with 4 GB ram), and rewriting increased the binary size of ads by just 1.24%.

The FlashJaX code and experiments described in this section are available at http://securemashups.net/flashjax. (The site does not collect any information regarding its visitors.)

2.6.2 Compatibility

Our compatibility experiments test whether FlashJaX preserves existing, policy-adherent content in JS, AS, and mixed AS-JS ads. We performed two sets of experiments to test a cross-section of ads from various sources:

1. In the first set of experiments, we deployed FlashJaX with ad scripts from four popular ad networks: Google AdSense, Yahoo! Network, Microsoft Media Network, and Clicksor. The first three of these were among the top 15 networks in U.S. market reach in April 2012, with market reach of 92.2%, 80.3%, and 76.9%, respectively (comScore, 2012).

   We ran these ads with and without FlashJaX to observe their rendering results. All render correctly with no visible distinctions introduced by monitoring. User interactions with the JS and Flash content also result in unmodified behaviors. This shows that our prototype can be deployed with real-world ad networks without loss of functionality.

2. In addition to the ad network code, we tested our AS binary rewriter on 57 Flash ads harvested from browsing sessions on popular browsers over several weeks, intended to reflect ads observed by typical users. Of the 57 ads, 32 interact with JS using Flash's external interface to perform tasks such as cookie manipulation, pop-up creation, click tracking, or information exchange with JS-side ad network support code. Our AS rewriting mechanism is able to successfully inject IRMs into all 57 samples.

2.6.3 Security

To evaluate FlashJaX's resilience against attack, FlashJaX was deployed and tested against several malicious JS and AS programs. These include several real-world attacks reported on CVE (http://cve.mitre.org/), the attack scenarios introduced throughout the paper, and other attacks related to wrapper implementation, confidentiality, integrity, and ad-specific attacks. Each experiment was conducted by first running the malicious code without FlashJaX to verify that the attack is successful. Then the same script was run with FlashJaX to test whether it was blocked. The attacks and defenses are categorized and described below, and summarized in Table 2.1.

Table 2.1. Attack scenarios and FlashJaX prevention

Attacks	**Policy applied**	**FlashJaX prevents?**
AS Circum- vention of SOP	Principal-specific whitelist	✓
Cross-language Heap-spraying	Principal-specific and history-based	✓
Cross-Principal Resource Abuse	Principal-specific access control	✓
Wrapper vulnerabilities	Wrapping all aliases	✓
Confidentiality and integrity	Principal-specific & fine- grained access control	✓
Ad-specific	Principal-specific & fine- grained access control	✓

Real-world attacks

We studied two recent real-world attacks reported on CVE, including CVE-2012-3414 ("XSS vulnerability in SWFUpload 2.2.0.1 and earlier") and CVE-2012-2904 ("XSS vulnerability in LongTail JW Player 5.9"):

CVE-2012-3414 is an attack reported as a vulnerability in Wordpress 3.3.2 installations. It allows reflected XSS via a Flash parameter derived from user input. The attacker can inject arbitrary JS code by passing it to the applet as a malicious URL string, resulting in execution of the injected code at the privileges of the hosting page. However, with FlashJaX added to the content produced by WordPress, the attack fails. The JS code injected by the attacker is labeled and executed with the lower privilege of the untrusted Flash, disallowing attacker access to protected JS functions and page content.

CVE-2012-2904 is a vulnerability in LongTail JW Player 5.9, which is active on over one million web sites. Exploits inject script text as a parameter of the Flash, provoking execution of the payload at the privileges of the hosting page. FlashJaX, however, successfully labels the injected code with the untrusted Flash principal, causing it to execute at the lower (untrusted) privilege level and denying it access to publisher-protected resources. Thus, all prohibited JS operations in the payload are suppressed by the monitor, foiling the attack.

Simulated Attacks

Attack scenarios. We implemented and validated policies that were discussed in Section 2.4.4. These policies address all three attack scenarios described in Section 2.2. These FlashJaX policies prevent these attacks.

Wrapper Attacks. We implemented wrapper attacks identified by prior works (Magazinius et al., 2010; Meyerovich et al., 2010; Meyerovich and Livshits, Meyerovich and Livshits), which defeat naïve JS wrapper implementations by abusing static aliases, dynamic aliases,

and caller-chains. When successful, the attacks pop up an unmediated alert box. All the attacks failed since FlashJaX wraps these channels.

Script injection. FlashJaX does not prohibit script injections; it downgrades them to an untrusted privilege level so that they cannot perform policy-violating actions. We tested all script injection channels, including remote script files, script code, event handlers via `document.write`, `eval`, and script inclusion via `appendChild` and `insertBefore`. The experiments show that our principal-tracking mechanism attributes correct privileges to all the dynamically-generated code. We note here again that scripts with an unidentified principal run with lowest privileges, and therefore never violate any principal's policy.

Confidentiality and integrity attacks. These attacks steal or modify sensitive data of the hosting page, such as cookies and protected content. To evaluate these attacks, we deployed a web email page with a fine-grained access control policy that prohibits ads from reading the contact list or changing the email content. Ads that try to do so are successfully blocked by FlashJaX in the experiment.

Cookie protection. FlashJaX does not prohibit cookie access, but each principal may only read and write its own cookies. Malicious code that attempts to steal cookies belonging to another principal was evaluated and found to be unsuccessful.

Ad-specific attacks. We tested numerous attacks specific to web ads, including *clickjacking*, *oversized/arbitrary ad positioning*, and *resource abuse*. Each is described below.

Clickjacking attacks create an invisible `iframe` that injects a remote page with an invisible click button (Hansen and Grossman, 2008). FlashJaX prevents this by enforcing a policy that disallows creating invisible `iframe`s.

Malicious ads often generate content that is larger than the maximum allowed by the ad network, or that is positioned inappropriately on the page (e.g., covering other content). These actions are prevented by FlashJaX by placing the ad in a fixed-sized ⟨`div`⟩ element

whose size it write-protects. The policy additionally forces the offset of any ad-generated content to 0×0 and write-protects the offset, preventing the ad from popping up misplaced or mis-sized dynamic content.

In addition, we enforced other fine-grained policies that disallow or limit calls, and that filter call arguments to a whitelist of API methods that are frequently targets of resource abuse attacks. These include pop-up creators like `window.alert` and `window.open`. FlashJaX correctly prevented these resource abuse attacks.

2.6.4 Performance

Macro-benchmarks. We evaluated the rendering overhead of ads in unmonitored and FlashJaX environments to understand the costs incurred by our monitor. The test machine is a 1.6 GHz AMD Athlon Neo MV-40 Processor laptop with 2 GB RAM running Chrome 19.0.1084.52m on Windows 7. The results are illustrated in Figure 2.10. We measure the total render time to load the page.

The rendering overhead imposed by FlashJaX varies widely based on the content from various ad networks. For Microsoft Media Network and Yahoo! Ads, the additional overhead is around 55%. However, for Google Adsense and Clicksor, we consistently observe rendering times that are actually faster with FlashJaX than without. We investigated this and found that Microsoft and Yahoo! generate Flash content, whereas Google and Clicksor generate `iframe`s that make heavy use of runtime-generated JS content. Our buffering of dynamic write operations (see Section 2.4.3) improves the performance of these dynamic writes, speeding rendering.

Micro-benchmarks. To evaluate the overhead imposed by the monitors, we performed a set of microbenchmark experiments that measure the overhead of five monitored JS operations called from AS. JS's `eval` method was selected to test JS property reading and writing (e.g., reading and writing `document.cookie`), since the AS-JS interface only provides for JS

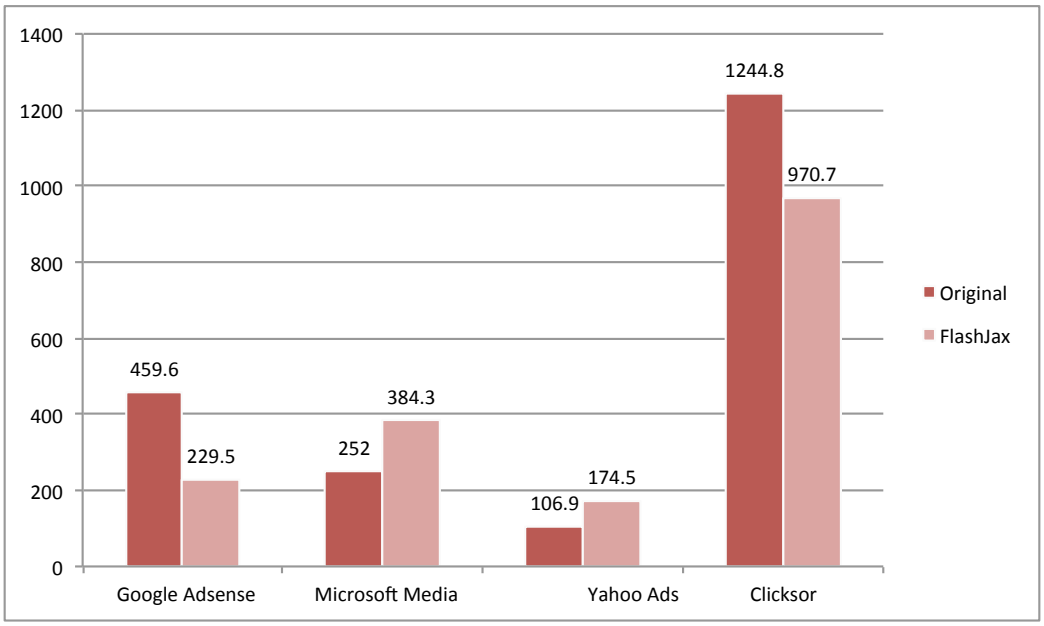


Figure 2.10. Rendering overheads (in ms) of unmonitored vs. FlashJaX-monitored ads.

method access. Each test ran a tight loop for 1000 iterations, and we averaged the results over five trials. On the original (unmodified) test files, the time was recorded in AS both before and after calling to JS. Then the same measurements were performed on IRM-injected Flash files both with and without the JS-side IRM. Table 2.2 shows the slow-down ratio of the rewritten flash without (column 2) and with (column 3) JS-side monitoring.

Table 2.2. FlashJaX micro-benchmarks measuring the ratio of the runtimes of FlashJaX-rewritten Flashes to originals without (column 2) and with (column 3) JS-side monitoring.

Operation	**Rewritten Flash**	**FlashJaX**
`document.appendChild`	3.52	3.59
`document.getElementById`	4.47	5.17
`toString()`	4.26	4.47
`eval("document.cookie='test'")`	3.67	5.91
`eval("document.cookie")`	3.07	6.33

The table shows a 3.07–4.47 times overhead for AS-JS boundary communications. This range compares favorably with similar microbenchmarks reported by related works (e.g.,

overheads of 1–324 times (Reis et al., 2006), or 0.09–19.54 times (Phung et al., 2009)). The inclusion of the JS IRM (column 3) results in a small additional overhead that is similar, except for the two eval tests. The overhead is higher for the eval tests because each iteration invokes the JS IRM twice (once for the `eval` and once for its content).

2.7 Discussion

In this section, we discuss the relevance of FlashJaX to the larger landscape of web application security.

Context and Relevance. A plethora of threats are faced by web applications today; the most common ones include script injection attacks, heap spraying, drive-by downloads, UI spoofing, and clickjacking. Extensive research in both server-side and browser-side defense techniques have been developed for mitigating these threats. The work presented in this paper exposes a relatively unexplored threat vector (compared to the threats mentioned above, which have been well explored).

We have also developed a systematic defense for this threat using the principled approach of IRMs. Our work can be seen as a defense that sits in conjunction with existing browser defenses, including those for JS (e.g., Meyerovich and Livshits, Meyerovich and Livshits), XSS attacks, and heap-spraying. FlashJaX strengthens those defenses by adding protection against a significant attack vector that these defenses do not address.

Other related browser plugins. Recent surveys indicate that Flash is the most commonly used plugin in browsers.[6] FlashJaX provides a systematic way to enforce security on Flash-JS content. Similar content can be authored in other plugins, such as Java and Silverlight. Our work could be extended to Silverlight, for instance, using similar IRM-based techniques that have been used for .NET binary rewriting (Hamlen et al., 2006a).

[6] http://www.statowl.com/plugin_overview.php

Deployment. Research efforts such as FlashJaX point out that the nature of attack surfaces will continue to evolve as browsers evolve to support new features. As a result, the nature of policies that security engineers want to enforce is continuously evolving as well, and there will always be a need to enforce policies that current browsers do not universally support. FlashJaX's approach to security through IRM enforcement allows for a principled defense mechanism that can be flexibly adapted to address future threats, while remaining compatible with existing browsers.

2.8 Conclusion

We present FlashJaX, a solution for enforcing security policies on third-party mixed JS/AS web content using an IRM approach. FlashJaX allows publishers to define and enforce fine-grained, multi-principal access policies on JS-AS third party content and runtime-generated code. Moreover, it can be easily deployed in practice without requiring browser modification. Experiments show that FlashJaX is effective in preventing attacks related to AS-JS communication, and its lightweight IRM approach exhibits low overhead for mediations. It is also compatible with advertisements from leading ad networks.

CHAPTER 3

ACTIONSCRIPT BYTECODE VERIFICATION USING CO-LOGIC PROGRAMMING[1]

3.1 Overview

The long-term objective of this dissertation is to augment in-lined referencing monitoring with powerful yet elegant certification algorithms. With regard to this, the three main goals of this chapter are: (1) experimenting with building light-weight model-checkers amenable to integration with in-lined reference monitoring frameworks; (2) experimenting with *co-logic programming* (Co-LP) (Simon et al., 2006; Gupta et al., 2007; Bansal, 2007) to provide and effective certifier development environment; and (3) contribute useful formal analysis techniques for studying the ActionScript language and its security implications, which is helpful for analyzing security vulnerabilities and threats in ActionScript bytecode binaries.

In this chapter, we report on preliminary work toward developing an ActionScript verifier, with an LTL model-checker at its core written using co-logic programming. Co-logic programming is a cutting-edge logic programming technology that combines tabled and coinductive logic programming and allows extremely elegant and succinct formalization of properties defined in terms of least and greatest fixed-points. Such properties lie at the heart of many model-checking analyses, such as those that regard potentially non-terminating loops.

Our verifier expresses security policies using Linear Temporal Logic (LTL) (Pnueli, 1977), which extends propositional logic with temporal operators. LTL underpins many modern software model checking technologies, and allows us to conveniently draw upon existing

[1]This chapter includes previously published (DeVries et al., 2009) joint work with Brian W. DeVries, Gopal Gupta, Kevin W. Hamlen and Scott Moore.

techniques from the field. With LTL, policy writers can specify security policies in a purely declarative and compositional manner; furthermore, formal analysis techniques, such as satisfiability checking (Rozier and Vardi, 2007), assist the policy writer in creating semantically valid policies.

Our work demonstrates how the verifier can be unusually small but powerful, minimizing the TCB of our framework. The declarative nature of co-logic programming helps keep the verifier code base concise and straightforward to encode semantic rules such as the encoding of ActionScript language semantics from the AVM2 Specification (Adobe Systems Inc., 2007), thus providing a certification technology ideal for integration with an IRM framework. We develop such integration in subsequent chapters.

Experiments demonstrate that our prototype can efficiently verify simple but interesting history-based policies for small ActionScript programs.

The chapter proceeds as follows. Section 3.2 discusses related work. Section 3.3 gives a detailed introduction to our system, including its high-level design and main features. Section 3.4 presents the details of our implementation. Section 3.5 discusses the results of a few preliminary experiments. Section 3.6 concludes.

3.2 Background and Related Work

Prolog is an attractive language for implementing program analysis tools, as program semantics and decision procedures can be efficiently written in a concise and declarative manner. At least two major extensions to the standard resolution semantics of Prolog have further enhanced its usefulness in this area in recent years: tabled logic programming and coinductive logic programming—collectively, co-logic programming.

Tabled logic programming (Chen and Warren, 1996; Tamaki and Sato, 1986) extends Prolog's standard inductive semantics with memoization and termination of cyclically infinite search paths. As a tabled recursive predicate executes, calls and intermediate solutions are

automatically stored in a table. If a variant (duplicate) call is encountered that would normally cause the computation to cyclically diverge, the call is replaced by a previously discovered solution, if it exists; otherwise, the call is suspended while other alternatives are tried. Any solutions found by these alternatives are stored in the table and can be substituted for the suspended calls as well. This substitution process can be repeated until no new solutions can be found, indicating that the tabled solutions correspond to the least fixed-point of the original call. Any suspended calls will fail at this point, as no new solutions can be found inductively.

Coinductive logic programming (Simon et al., 2006; Gupta et al., 2007; Bansal, 2007) allows Prolog to reason about cyclically infinite data structures and proof trees. When a recursive call to a coinductive predicate is made, the call stack is searched for a unifiable ancestor call; if found, the call is unified with its ancestor and succeeds. The solution obtained from the unification corresponds to the greatest fixed-point interpretation of the original call, meaning a coinductive predicate can generate a proof without encountering—or even specifying—a base case.

Because tabled logic programming and coinductive logic programming correspond to the least and greatest fixed-point semantics of proof derivations, respectively, they are useful for implementing program analyses. In particular, program loops are by nature cyclic structures for which both inductive and coinductive properties are important for establishing policy-adherence. While many interesting coinductive properties can be proved by the absence of an inductive counter-example, the use of coinductive logic programming allows constructive proofs of these properties to be computed. Co-LP, which subsumes both of these strategies, is therefore particularly well-suited because it allows least and greatest fixed-point analyses to be combined in a well-defined and semantically meaningful manner.

3.3 System Introduction

ActionScript source code is typically compiled to ActionScript Virtual Machine (AVM) bytecode (Adobe Systems Inc., 2007) and subsequently interpreted by the AVM. Our verifier models the AVM in Prolog and extends the semantics of each AVM instruction to include security-relevant state. It takes as input a program in ActionScript ByteCode (ABC) format and a security policy (expressed in LTL) and conservatively decides whether every program execution satisfies the policy.

We derive a particularly elegant implementation for our verifier by observing that an abstract interpreter with coinductive semantics facilitates the realization of a model checker. That is, where an interpreter loops on non-terminating programs, a model-checker based on co-LP succeeds (and terminates) when it revisits an abstract state that constitutes a valid loop invariant. This reduces much of the machinery normally required for model-checking to the relatively simple framework required for a standard abstract interpreter.

Both tabled and coinductive aspects of co-LP are useful in our approach. Using tabled LP, we implement model-checking as a search for a counter-example. Tabled LP semantics yield such a counterexample when this search succeeds. In the case of model-checking, the counter-example is a policy-violating sequence of AVM instructions that can be useful to a developer wishing to produce policy-adherent code. Coinductive LP yields a constructive proof of correctness when the search for a counter-example fails. Such a proof could be attached to the verified bytecode for use with a Proof-Carrying Code (PCC) system (Necula and Lee, 1996; Necula, 1997). While our current prototype does not yet produce a complete, machine-readable proof automatically, most of the machinery necessary for implementing this feature has already been completed as a natural result of our use of co-LP as the foundation for the verifier. We therefore expect to add this feature relatively painlessly in future work.

Policies are expressed as LTL formulas encoding the set of all permissible sequences of security-relevant events. To reflect the reality that a real process may terminate at any

point (e.g., due to practical issues like hardware failures or power outages), we model every program state as potentially terminating. This model limits our system to verifying safety policies; non-safety LTL properties are conservatively rejected. While LTL is therefore a more expressive language than necessary for our purposes, it is nonetheless convenient due to its familiarity in the model-checking community and the existence of many well-developed tools for writing and reasoning about LTL-specified policies.

```
1 event(callpropvoid(open,[File,_Mode]),open) :-
2    in_directory(File,`~/secret_files/').
3 event(callpropvoid(readByte,_Args),read).
4 event(callpropvoid(readUTF,_Args),read).
5 event(callpropvoid(writeByte,_Args),send).

7 ltl_policy(g(impl(open,g(impl(read,not(f(send)))))))).
```

Figure 3.1. A policy prohibiting network-sends after file-reads from a restricted directory.

As an example, Figure 3.1 specifies a policy that prohibits network-send operations after a file in a specific directory has been opened. This policy often appears in the IRM literature as a canonical example of a history-based policy that enforces data confidentiality. The `event` predicate defines three security-relevant events: `open`, `read`, and `send`. Method calls to `writeByte` (regardless of arguments) constitute `send` events. Method calls to `readByte` or `readUTF` constitute `read` events, demonstrating that events can abstract across multiple actions. The `open` event is defined as a call to the `open` method, where the first argument (the name of the file to open) satisfies the predicate `in_directory`. Here, the predicate `in_directory` checks whether the file name given resides within the prohibited directory.

The policy on line 7 encodes the LTL formula $\mathcal{G}(\texttt{open} \rightarrow \mathcal{G}(\texttt{read} \rightarrow \neg\mathcal{F}\texttt{send}))$. Informally, it stipulates that no execution may contain the sequence (`open` ... `read` ... `send`). This specification has been simplified for expository purposes, but a discussion of how to elaborate the specification for a real application is available in (Jones and Hamlen, 2009).

To keep the verifier implementation tractable, we make several important assumptions about untrusted programs that simplify the analysis. First, we assume that the compiled

ABC code is syntactically valid. This can be verified separately by the AVM bytecode verifier. We also assume that AVM programs do not perform runtime code generation, since our analysis is limited to the statically observable code. Since runtime code generation is only available to ABC programs via a limited set of system calls, this assumption can be enforced in policy specifications by prohibiting calls to those methods.

Our dataflow analysis is conservative in the sense that it abstracts away certain details of the heap and program variables. This can cause some policy-adherent code to be conservatively rejected (though it never results in policy-violating code being accepted). However, we argue that this limitation can be overcome by an IRM system that inserts additional security guards that ease the verification process. The security guards effectively reduce the state space by explicitly ruling out unrealizable control flows and their associated data flows.

Our early prototype does not yet implement an interprocedural analysis. This limits our current experiments to simple programs whose security-relevant behavior does not involve method calls. In Chapters 4 and 5, we discuss certifiers that perform interprocedural and intermodular analysis.

3.4 Implementation

There are three major components to our verifier: a parser for ActionScript ByteCode, an encoding of AVM semantics, and an LTL model-checker. Each component's implementation is described throughout the remainder of the section.

3.4.1 ABC File Parser

Since typical code consumers do not have access to the original ActionScript source code, we verify compiled ActionScript Bytecode (ABC) files directly. Our ABC parser is a Definite Clause Grammar (DCG) that transforms a file in ABC binary format (Adobe Systems Inc.,

2007) to an annotated abstract syntax tree (AST). For more information about DCGs and their Prolog implementations, the reader is invited to consult (Shapiro and Sterling, 1994).

3.4.2 AVM 2 Semantics

The AVM2 semantics module is a declarative implementation of an ActionScript VM derived from the AVM2 small-step semantics (Adobe Systems Inc., 2007). Using these semantics, we translate an ABC program into a transition system for use with the model-checker. The *transition system* (Baier and Katoen, 2008) consists of a set of *states* that model the VM state at each point in program execution, and a *transition relation* that relates each state to the states reachable from the current instruction. We encode the transition relation as a Prolog predicate that is invoked by the model-checker as it explores the state space.

To avoid state space explosion, we use an abstract interpretation rather than a concrete interpretation. A concrete data model is inadequate because it requires every possible AVM memory configuration to be represented by a unique state in the transition system, and searching the resulting state space becomes intractable. In contrast, an abstract interpretation allows a collection of possible variable values to be represented by a single abstract state. The precision of this abstraction determines the power of our analysis. For instance, modeling all program values as unknown results in a simple control-flow analysis. While this would ensure that every policy-violating program is rejected, it would conservatively reject many policy-adherent programs, severely limiting the usefulness of the verifier. In particular, since IRMs enforce security policies by tracking security state in injected program variables, a control-flow analysis cannot verify that IRM-inserted guards that consult these variables suffice to prevent policy violations.

To avoid this limitation, we use concrete values where they can be statically determined and use the abstract state $\top$ otherwise. Our AVM2 semantics are therefore extended with transitions for $\top$. For instance, the `ifeq` instruction semantics in Figure 3.2 model the

possibility of executing either branch when the outcome of the conditional expression cannot be determined statically; otherwise only the statically inferred branch is interpreted. Note that since model-checking explores each execution trace independently, the set of statically inferable values is much larger than would be available in a more traditional static analysis. The more powerful data-flow analysis provided by this abstraction captures relationships between data values and event sequences that the program might exhibit at runtime. As a result, the verifier can track the values of IRM security state variables.

```
1 % trans/3 encodes the transition relation
2 % EE is the current execution environment: the scope
3 % stack, the operand stack, the register file, and
4 % the current instruction.
5 % NewEE is the resulting execution environment
6 trans(ifeq,state(H,[EE|EEs]),state(H,[NewEE|EEs])) :-
7     EE =.. [env, SS, [V1,V2|Os], RF, Instr],
8     equal(V1, V2, TruthValue),
9     next_instr(TruthValue,Instr,NewInstr),
10    NewEE =.. [env, SS, Os, RF, NewInstr].

12 % equal/3 returns true or false if equality between
13 % the arguments can be determined, otherwise top (tt)
14 equal(int(V1), int(V1), bool(true)).
15 equal(int(V1), int(V2), bool(false)) :- V1 =\= V2.
16 equal(V1, V2, tt) :- V1 \= int(_X); V2 \= int(_Y).

18 % next_instr/3 gives the next instruction to be
19 % executed, based on the result of the comparison.
20 % If the result is top, both branches are searched.
21 next_instr(bool(true), Instr, NewInstr) :-
22     get_property(Instr, jumplabel, NewInstr).
23 next_instr(bool(false), Instr, NewInstr) :-
24     get_property(Instr, next, NewInstr).
25 next_instr(tt, Instr, NewInstr) :-
26     get_property(Instr, jumplabel, NewInstr).
27 next_instr(tt, Instr, NewInstr) :-
28     get_property(Instr, next, NewInstr).
```

Figure 3.2. The `ifeq` transition relation clause

3.4.3 Model Checker

The LTL model checker module takes as input a security policy specified as an LTL formula and the transition system supplied by the AVM2 semantics. Recall that LTL extends propositional logic with temporal operators; the logical propositions correspond to the individual security-relevant events, and the temporal operators specify the valid event sequences. Given two LTL formulae a and b, the formula $\mathcal{X}a$ mandates that a holds in the *next* state, $\mathcal{F}a$ mandates a holds in some *future* state, $\mathcal{G}a$ mandates that a holds *globally* (i.e., in the current and all subsequent states), $a\,\mathcal{U}b$ mandates that b holds in some future state and a holds in every state *until* that point, and $a\mathcal{R}b$ mandates that b holds in the current and all subsequent states until this requirement is *released* by a state where a and b both hold.

The expansion rules for the LTL operators (Baier and Katoen, 2008) provide a declarative semantics for the interpretation of LTL formulae by constraining the current state and immediate successor states. These rules can be encoded directly in Prolog to obtain an interpreter for LTL formulae, as seen in Figure 3.3. To check whether a path through the transition system satisfies a given LTL formula, the model checker recursively invokes the expansion rules, first checking the requirements placed on the current state, then making a transition to the next state on the path and repeating the process.

Using these expansion rules, we can cleanly separate the temporal operators into two categories: $\mathcal{X}$, $\mathcal{F}$, and $\mathcal{U}$ are inductive, as their expansion rules provide base cases for deciding whether a path satisfies the operator, while $\mathcal{G}$ and $\mathcal{R}$ are coinductive, as they are satisfied by a cyclically infinite execution path. Based on this distinction, our implementation relies on tabling and coinduction to reason about loops encountered when traversing the transition system—without these extensions, loops would cause the model checker to diverge in an attempt to construct the proof for a formula. Tabling terminates the proof search with a failure if an inductive formula does not hold for a loop, while coinduction terminates the search with the infinite-length proof when a coinductive formula holds.

```
% verify/2 takes a state and an existentially
% quantified LTL formula and checks
% whether the formula holds for that state.
%
% Atomic Propositions are labeled by `ap'.
%
% holds/2 is true when the atomic proposition holds
% in the current state
%
% ftype/2 is a mapping from top-level temporal
% operators to their interpretation semantics
%
% The clause for `a and b' should ensure that `a' and
% `b' hold on the same execution path. For simplicity
% of presentation, we omit this check here.

verify(State, F) :- ftype(F, inductive),
  verify_inductive(State, F).
verify(State, F) :- ftype(F, coinductive),
  verify_coinductive(State, F).

:- tabled verify_inductive/2.
verify_inductive(S, ap(AP)) :- holds(S,AP). % p
% Logical operators
verify_inductive(S, not(ap(AP))) :- % not(p)
  \+ holds(S, AP).
verify_inductive(S, or(A,B)) :- % a or b
  verify(S, A) ; verify(S, B).
verify_inductive(S, and(A,B)) :- % a and b
  verify(S, A), verify(S, B).
% Inductive temporal operators
verify_inductive(S, x(A)) :- % X(a)
  trans(S, S1), verify(S1, A).
verify_inductive(S, f(A)) :- % F(a)
  verify(S, A); verify(S, x(f(A))).
verify_inductive(S, u(A,B)) :- % a U b
  verify(S, B);
        verify_inductive(S, and(A, x(u(A,B)))).

:- coinductive verify_coinductive/2.
% Coinductive temporal operators
verify_coinductive(S, g(A)) :- % G(a)
  verify(S, and(A, x(g(A)))).
verify_coinductive(S, r(A,B)) :- % a R b
  verify(S, and(A,B)).
        % {a and b both occur, releasing b}
verify_coinductive(S, r(A,B)) :-
  verify(S, and(B, x(r(A,B)))).
        % {a does not hold, so b is not released}
```

Figure 3.3. A simple Co-LP LTL model checker

3.5 Experiments

As a preliminary test of our prototype, we verified three small test programs against the policy specified in Figure 3.1, which disallows network-send operations after a file has been read from a certain restricted directory. The three test programs—`Unsafe.as`, `Safe.as`, and `Loop.as`—were each compiled from ActionScript source code. The first, `Unsafe.as`, exhibits policy-violating behavior when executed. The second, `Safe.as`, was obtained by instrumenting `Unsafe.as` with runtime security guards similar to those that an IRM system would typically insert as part of the binary-rewriting process. This involved adding an additional program variable that tracks the current security state at runtime, along with instructions that test and update the variable as security-relevant events occur. The third program, `Loop.as`, enclosed the security-relevant operations of the second program in an infinite loop, allowing us to test our loop analysis. The source code of all three programs can be seen in Figure 3.4. Code inserted to generate `Safe.as` is marked in the listing with `*`, and the additional code in `Loop.as` is marked with `#`.

In this program the value of `flag` cannot be inferred statically, so our analysis conservatively assumes that it can take on any integer value. Thus, the outcome of the test `(flag > 0)` in line 12 is not statically known, leading to a possible policy violation at the network-send operation in line 18 if no runtime security guards were present. Lines 14 and 17, however, update and test (respectively) a new program variable `security` that tracks the current security state. In this case the state is 1 if a file has previously been read and 0 otherwise. Thus, testing `(security != 1)` in line 17 before each network-send event suffices to prevent a policy violation.

We ran each of these tests 10 times on an Intel Pentium Core 2 Duo machine with 4GB of RAM running Ubuntu Intrepid and Yap Prolog v5.1.4 (LIACC/Universidade do Porto and COPPE Sistemas/UFRJ, 2009). The median runtimes are reported in Table 3.5.

```
 1 public function Test(flag:int) {
 2   var socket:Socket = new Socket();
 3   var file:File = new File("secret.txt");
 4   var fileStream:FileStream = new FileStream();
 5 * var security:int = 0;

 7   fileStream.open(file, FileMode.READ);

 9   socket.connect("example.com", 1234);

11 # while (true) {
12     if (flag > 0) {
13       fileStream.readByte();
14 * security = 1;
15     }

17 * if (security != 1) {
18       socket.writeByte(0);
19 * }
20 # }

22   socket.close();
23   fileStream.close();
24 }
```

Figure 3.4. Source code of the test programs

`Unsafe.as` was correctly identified as policy-violating, while `Safe.as` and `Loop.as` were correctly identified as policy-adherent.

`Unsafe.as`	0.093s
`Safe.as`	0.110s
`Loop.as`	0.101s

Figure 3.5. Experimental Results

3.6 Conclusion

In this chapter, we describe preliminary work toward developing a security policy verifier for Adobe ActionScript bytecode programs. Our verifier consists of an interpreter for ActionScript bytecode and an LTL model-checker, both written in Prolog extended with tabling

and coinduction. Experiments demonstrate that our prototype can efficiently verify simple but interesting history-based policies for small ActionScript programs.

Verifiers are typically part of a secure system's trusted computing base. It is therefore important that the verifier itself be amenable to formal verification. The declarative nature of co-LP Prolog yields several significant advantages in this regard. First, our verifier code base is very concise—the parser is 2 kSLOC while the AVM2 semantics and the model-checker are each 1 kSLOC. Second, our experiences indicate that it is straightforward to encode semantic rules, such as those from the AVM2 Specification (Adobe Systems Inc., 2007), as a relation between program states (see Figure 3.2). Finally, implementing the expansion rules for LTL in co-LP avoids a great deal of tedious and error-prone implementation work by relying upon the well-defined termination semantics of tabling and coinduction in Prolog (see Figure 3.3).

Like all static analysis techniques, our verifier conservatively rejects some policy-adherent programs. Subesequent chapters (Chapters 4 and 5) extend this analysis to reduce this conservative rejection rate. This involves introducing richer abstractions to model data dependencies and data flows. We also use constraint logic programming to elegantly implement these abstractions while minimizing the state space that must be explored to verify useful security properties.

Additionally, these subsequent chapters provide a means by which a code producer or enforcement mechanism can supply hints to the verifier. These hints greatly increase the efficiency of the verifier by pre-computing the set of possible variable values at particular points in the program. These precomputed values need not be trusted since the verifier can ignore hints that are inconsistent with its analysis.

In future work, we plan to generate explicit policy-adherence proofs. This involves enhancing current implementations of coinductive Prolog (Gupta et al., 2007) to support enumeration of all coinductive proofs of a goal. After completing these efforts, generating these proofs should require minimal changes to our system.

There is a large body of existing work on optimizing LTL formulae used in model checking (e.g., Daniele et al., 1999; Etessami and Holzmann, 2000; Sebastiani and Tonetta, 2003). We also intend to explore the use of other temporal logics, especially Computation Tree Logic (CTL) and the μ-calculus (Jr. et al., 1999), for the specification of security policies. The method mentioned in (Dillon and Ramakrishna, 1996) suggests a means of modularizing the temporal logic engine out of the model-checker, allowing the same model-checking system to be used for multiple temporal logics by changing which temporal logic engine is used. Examining how to use several of these engines at once seems a promising direction for our tool as well.

CHAPTER 4

A PROTOTYPE MODEL-CHECKING IRM SYSTEM FOR ACTIONSCRIPT BYTECODE[1]

4.1 Overview

Certifying IRM systems (Hamlen et al., 2006a; Aktug and Naliuka, 2008) verify that IRMs generated by a binary rewriter are policy-adherent[2]. In Chapter 3, we presented a general model-checking system for ActionScript bytecode implemented using co-logic programming (DeVries et al., 2009), (Simon et al., 2006). This chapter extends that work by introducing new formalisms specific to the verification of safety policies enforced by IRMs.

Past work has implemented IRM certifiers using type-checking (Hamlen et al., 2006a) and contracts (Aktug and Naliuka, 2008). Model-checking is an extremely powerful software verification paradigm that is useful for verifying properties that are more complex than those typically expressible by type-systems (Hamlen et al., 2006a) and more semantically flexible and abstract than those typically encoded by contracts (Aktug and Naliuka, 2008). Yet, prior to the work presented in this dissertation, model-checking had not been applied to verify IRMs. In this chapter we describe and implement a technique for doing so. The work's main contributions are as follows:

- We present the design and implementation of a prototype IRM model-checking framework for ActionScript bytecode.

[1]This chapter includes previously published (Sridhar and Hamlen, 2010b,a) joint work with Kevin W. Hamlen.

[2]Throughout this chapter, for simplicity, the terminology *IRM certification* refers to *IRM soundness certification*. *Transparency certification* of IRMs is discussed in Chapter 6.

- Our design centers around a novel approach for constructing a *state abstraction lattice* from a security automaton (Alpern and Schneider, 1986), for precise yet tractable abstract interpretation of IRM code.

- Rigorous proofs of soundness and convergence are formulated for our system using Cousot's abstract interpretation framework (Cousot and Cousot, 1977).

- The feasibility of our technique is demonstrated by enforcing a URL anti-redirection policy for ActionScript bytecode programs.

The chapter proceeds as follows. Section 4.2 discusses related work. Section 4.3 gives an overview of our IRM framework, including an operational semantics and the abstract interpretation algorithm. Section 4.4 provides a formal soundness proof for our algorithm and a proof of fixed point convergence for the abstract machine. Section 4.5 discusses the details of our implementation of the system for ActionScript bytecode. Section 4.6 concludes.

4.2 Related Work

To our knowledge, ConSpec (Aktug and Naliuka, 2008) and Mobile (Hamlen et al., 2006a) are the only IRM systems to yet implement automatic certification. The ConSpec verifier performs a static analysis to verify that pre-specified guard code appears at each security-relevant code point; the guard code itself is trusted. Mobile implements a more general certification algorithm by type-checking the resulting Mobile code. While type-checking has the advantage of being light-weight, it comes at the expense of limited computational power. For instance, Mobile cannot enforce certain dataflow-sensitive security policies since its type-checking algorithm is strictly control-flow based. While the security policies described by these systems are declarative and therefore amenable to a more general verifier, both use a verifier tailored to a specific rewriting strategy.

4.3 System Introduction

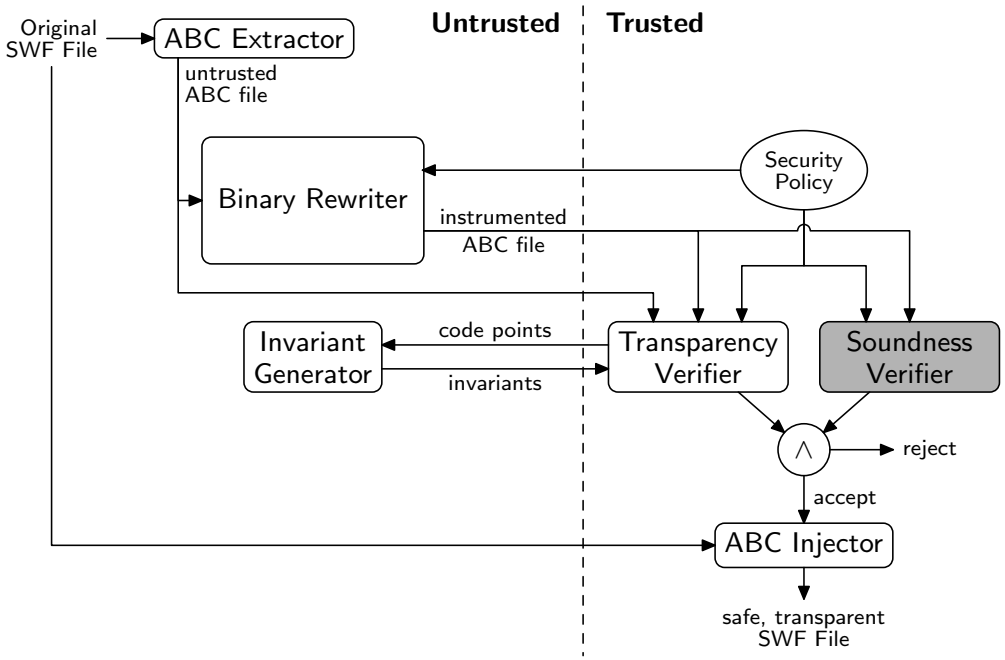


Figure 4.1. Certifying ActionScript IRM architecture

Figure 4.1 is re-presented from Chapter 1, and depicts the core of our IRM framework. The rewriter implementation is discussed in Section 4.5; the remainder of this section discusses the soundness verifier. The transparency veriifer and invariant generator are discussed in Chapter 6.

4.3.1 Verifier Overview

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

A standard challenge in implementing such an abstract interpreter is to choose an expressive yet tractable language of state abstractions for the abstract machine to consider. A

highly expressive state abstraction language allows very precise reasoning about untrusted code, but might cause the iteration process to converge slowly or not at all, making verification infeasible in practice. In contrast, a less expressive language affords faster convergence, but might result in conservative rejection of many policy-adherent programs due to information lost by the coarseness of the abstraction.

In what follows, we describe a state abstraction that is suitably precise to facilitate verification of typical IRMs, yet suitably sparse to facilitate effective convergence. Section 4.4 proves these soundness and convergence properties formally. To motivate our choice of abstractions, we begin with a discussion of an important implementation strategy for IRMs—*reified security state*.

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

When security policies are expressed as security automata (Alpern and Schneider, 1986), this reification strategy can be generalized as an integer variable that tracks the current state of the automaton. Security automata encode security policies as Büchi automata that accept the language of policy-satisfying event sequences. Formally, a deterministic security automaton $A = (Q, \Sigma, q_0, \delta)$ can be expressed as a set of states Q, an alphabet of security-relevant events Σ, a start state $q_0 \in Q$, and a transition relation $\delta : Q \times \Sigma \to Q$. For the purpose of this paper, we assume that Q is finite.[3] The automaton accepts all finite or infinite

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

sequences for which δ has transitions. Security automata therefore accept policies that are prefix-closed. That is, to prove that infinite executions of an untrusted program satisfy such a policy, it suffices to prove that every finite execution prefix satisfies the policy. We therefore define the set of finite prefixes $\mathcal{P}$ of the security policy denoted by a deterministic security automaton as follows.

Definition 2 (Security Policy)**.** *Let $A = (Q, \Sigma, q_0, \delta)$ be a deterministic security automaton. The* security policy *$\mathcal{P}_A$ for automaton A is defined by $\mathcal{P}_A = Res_A(Q)$, where notation $Res_A(q)$ denotes the* residual *(Denis et al., 2001) of state q in automaton A—that is, the set of finite sequences that cause the automaton to arrive in state q—and we lift Res_A to sets of states via $Res_A(Q) = \cup_{q \in Q} Res_A(q)$. When automaton A is unambiguous, we will omit subscript A, writing $\mathcal{P} = Res(Q)$.*

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

Definition 3 (Abstract Traces)**.** *The language SS of* abstract traces *is $SS = \{(Res(Q_0), \tau) \mid Q_0 \subseteq Q, \tau \in \Sigma^*, |\tau| \leq k\} \cup \{\top_{SS}\}$ where $\top_{SS} = \Sigma^*$. Abstract traces are ordered by subset relation $\subseteq$, forming the lattice $(SS, \subseteq)$.*

Intuitively, Definition 3 captures the idea that an IRM verifier must track abstract security states as two components: a union of residuals $Res(Q_0)$ and a finite sequence τ of literal events. Set $Res(Q_0)$ encodes the set of possible security states that the untrusted program might have been in when the reified state variable was last updated by the IRM to reflect the current security state. The actual current security state of the program can potentially be out of sync with the reified state value at any given program point because IRMs typically

cannot update the state value in the same operation that exhibits a security-relevant event. Thus, trace τ models the sequence of events that have been exhibited since the last update of the state value. In general, an IRM may delay updates to its reified state variables for performance reasons until after numerous security-relevant events have occurred. Dynamic tests of reified state variables therefore reveal information about an earlier security state that existed before τ occurred, rather than the current security state. This distinction is critical for accurately reasoning about real IRM code.

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

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

Definition 4 (Abstract Values)**.** *Define* $VS = \mathbb{Z} \cup \{\top_{VS}\}$ *to be the set of* abstract program values*, and define* value order *relation* $\leq_{VS}$ *by* $(n \leq_{VS} n)$ *and* $(n \leq_{VS} \top_{VS})$ $\forall n \in VS$*. Observe that* $(VS, \leq_{VS})$ *forms a height-2 lattice.*

4.3.2 Concrete Machine

The abstract states described above abstract the behavior of a concrete machine that models the actual behavior of ActionScript bytecode programs as interpreted by the ActionScript

$$
\begin{array}{lr}
\chi ::= \langle L : i, \sigma, \nu, m, \tau \rangle & \text{(CONFIGURATIONS)} \\
L & \text{(CODE LABELS)} \\
i ::= \textbf{ifle}\ L \mid \textbf{getlocal}\ n \mid \textbf{setlocal}\ n \mid \textbf{jmp}\ L \mid & \text{(INSTRUCTIONS)} \\
\quad \textbf{event}\ e \mid \textbf{setstate}\ n \mid \textbf{ifstate}\ n\ L & \\
\sigma ::= \cdot \mid v :: \sigma & \text{(CONCRETE STACKS)} \\
v \in \mathbb{Z} & \text{(CONCRETE VALUES)} \\
\nu : \mathbb{Z} \to v & \text{(CONCRETE STORES)} \\
m \in \mathbb{Z} & \text{(CONCRETE REIFIED STATE)} \\
e \in \Sigma & \text{(EVENTS)} \\
\tau \in \Sigma^* & \text{(CONCRETE TRACES)} \\
\chi_0 = \langle L_0 : p(L_0), \cdot, \nu_0, 0, \epsilon \rangle & \text{(INITIAL CONFIGURATIONS)} \\
\nu_0 = \mathbb{Z} \times \{0\} & \text{(INITIAL STORES)} \\
P ::= (L, p, s) & \text{(PROGRAMS)} \\
p : L \to i & \text{(INSTRUCTION LABELS)} \\
s : L \to L & \text{(LABEL SUCCESSORS)}
\end{array}
$$

Figure 4.2. Concrete machine configurations and programs

virtual machine. We define the concrete machine to be a tuple $(\mathcal{C}, \chi_0, \mapsto)$, where $\mathcal{C}$ is the set of concrete *configurations*, χ_0 is the initial configuration, and $\mapsto$ is the transition relation in the concrete domain. Figure 4.2 defines a configuration $\chi = \langle L : i, \sigma, \nu, m, \tau \rangle$ as a *labeled instruction* $L : i$, an *operand stack* σ, a *local variable store* ν, a reified security state value m, and a trace τ of security-relevant events that have been exhibited so far during the current run. A *program* $P = (L, p, s)$ consists of a program entrypoint label L, a mapping p from code labels to program instructions, and a label successor function s that defines the destinations of non-branching instructions.

To simplify the discussion, we here consider only a core language of ActionScript bytecode instructions. Instructions **ifle** L and **jmp** n implement conditional and unconditional jumps, respectively, and instructions **getlocal** n and **setlocal** n read and set local variable values, respectively. Instruction **event** e models a security-relevant operation that exhibits event e.

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

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

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

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

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

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

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

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

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

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

Figure 4.3. Small-step operational semantics for the concrete machine

Figure 4.3 provides a complete small-step operational semantics for the concrete machine. Observe that in Rule (CEvent), policy-violating events cause the concrete machine to enter a stuck state. Thus, security violations are modeled in the concrete domain as stuck states. The concrete semantics have no explicit operation for normal program termination; we model termination as an infinite stutter state. The soundness proof in Sect. 4.4 shows that any

program that is accepted by the abstract machine will never enter a stuck state during any concrete run; thus, verification is sufficient to prevent policy violations.

4.3.3 Abstract Machine

$$\hat{\chi} ::= \bot \mid \langle L : i, \hat{\sigma}, \hat{\nu}, m, (Res(q_m), \bar{\tau})\rangle \mid \langle L : i, \hat{\sigma}, \hat{\nu}, \top_{VS}, \hat{\tau}\rangle \quad \text{(ABSTRACT CONFIGS)}$$
$$\hat{\sigma} ::= \cdot \mid \hat{v} :: \hat{\sigma} \quad \text{(EVALUATION STACKS)}$$
$$\hat{v} \in VS \quad \text{(ABSTRACT VALUES)}$$
$$\hat{\nu} : \mathbb{Z} \to \hat{v} \quad \text{(ABSTRACT STORES)}$$
$$\hat{m} \in \mathbb{Z} \cup \top_{VS} \quad \text{(ABSTRACT REIFIED STATE)}$$
$$\bar{\tau} \in \cup_{n \leq k} \Sigma^n \quad \text{(BOUNDED TRACES)}$$
$$\hat{\tau} \in SS \quad \text{(ABSTRACT TRACES)}$$

Figure 4.4. Abstract machine configurations

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

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

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

Figure 4.5. State-ordering relation $\leq_{\hat{\chi}}$

We define our abstract machine as a tuple $(\mathcal{A}, \chi_0, \rightsquigarrow)$, where $\mathcal{A}$ is the set of configurations of the abstract machine, χ_0 is the same initial configuration as the concrete machine, and $\rightsquigarrow$ is the transition relation in the abstract domain. Abstract configurations are formally defined in Figure 4.4. Figure 4.5 lifts the $\leq_{VS}$ relation to operand stacks and stores to form a lattice $(\mathcal{A}, \leq_{\hat{\chi}})$ of abstract states. That is, stacks (stores) are related if their sizes (domains) are identical and their corresponding members are related.

The small-step operational semantics of the abstract machine are given in Figure 4.6. When the abstract machine can infer concrete values for operands, as in Rule (AIFLE-

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

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

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

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

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

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

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

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

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

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

Figure 4.6. Small-step operational semantics for the abstract machine

Pos), it performs a transition resembling the corresponding concrete transition. However, when operand values are unknown, as in Rule (AIfleTop), the abstract machine non-deterministically explores all possible control flows resulting from the operation.

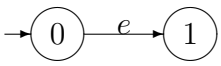
The premises of rules (AEvent), (ASetstate), and (AIfstateNeg) appeal to a model-checker that decides subset relations for abstract states according to Definition 3. Thus, the abstract machine enters a stuck state when it encounters a potential policy violation (see Rule (AEvent)). Abstract stuck states correspond to rejection by the verifier.

Rule (ASetstate) requires that acceptable programs must maintain a reified security state that is consistent with the actual security state of the program during any given concrete execution. This allows the (AIfstatePos) and (AIfstateNeg) rules of the abstract machine to infer useful security information in the positive and negative branches of program

operations that dynamically test this state. The verifier can therefore reason that dynamic security guards implemented by an IRM suffice to prevent runtime policy violations.

4.3.4 An Abstract Interpretation Example

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



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

Figure 4.7. An abstract interpretation example

To illustrate this, we here walk the abstract interpreter through the simple example program shown in the first column of Figure 4.7, enforcing the policy $\epsilon + e$ whose security automaton is depicted at the top of the figure. Abstract states inferred on *first entry* to each code point are written to the left of the $\sqcup$ in the second column. (All but the reified state value 0 and trace ϵ are omitted from each configuration since they are irrelevant to this particular example.) Abstract states inferred on *second entry* are written after the $\sqcup$, and the resulting join of these states is written in the third column. In this example a fixed point is reached after two iterations.

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

setstate 1 updates the reified state, and operation **jmp** L_1 returns to the original code point.

The join of these two states yields a new configuration in which the reified state is unknown ($\top$), so on the second iteration the abstract machine non-deterministically transitions to both L_2 and L_3. However, both transitions infer useful security state information based on the results of the dynamic test. Transition (AIFSTATEPOS) to label L_3 refines the abstract trace from $\epsilon + e$ to $Res(q_0) = \epsilon$, and transition (AIFSTATENEG) to label L_2 refines it to $\epsilon + e - Res(q_0) = e$. These refinements allow the verifier to conclude that all abstract states are policy-satisfying. In particular, the dynamic state test at L_1 suffices to prevent policy violations at L_3.

4.4 Analysis

4.4.1 Soundness

The abstract machine defined in Section 4.3.3 is *sound* with respect to the concrete machine defined in Section 4.3.2 in the sense that each inferred abstract state $\hat{\chi}$ conservatively approximates all concrete states χ that can arise at the same program point during an execution of the concrete machine on the same program. This further implies that if the abstract machine does not enter a stuck state for a given program, nor does the concrete machine. Since concrete stuck states model security violations, this implies that a verifier consistent with the abstract machine will reject all policy-violating programs.

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

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

Figure 4.8. Soundness relation $\sim$

We define the soundness of state abstractions in terms of a *soundness relation* (Cousot and Cousot, 1992) written $\sim \subseteq \mathcal{C} \times \mathcal{A}$ that is defined in Figure 4.8. Following the approach of (Chang et al., 2006), soundness of the operational semantics given in Figs. 4.3 and 4.6 is then proved via progress and preservation lemmas. The preservation lemma proves that a bisimulation of the abstract and concrete machines preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the concrete machine does not enter a stuck state. Together, these two lemmas dovetail to form an induction over arbitrary length execution sequences, proving that programs accepted by the verifier will not commit policy violations.

In the next part of this section, we sketch interesting cases of the progress and preservation proofs, and subsequently delineate the proof of soundness. We have recently completed fully machine-verified proofs for all cases of the progress and preservation lemmas, and the proof of soundness using the Coq proof assistant (INRIA, 2014).

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

Proof. Let $\chi = \langle L : i, \sigma, \nu, m, \tau \rangle \in \mathcal{C}$, $\hat{\chi} = \langle L : i, \hat{\sigma}, \hat{\nu}, \hat{m}, \hat{\tau} \rangle \in \mathcal{A}$, and $\hat{\chi}' \in \mathcal{A}$ be given, and assume $\chi \sim \hat{\chi}$ and $\hat{\chi} \rightsquigarrow \hat{\chi}'$ both hold. Proof is by a case distinction on the derivation of $\hat{\chi} \rightsquigarrow \hat{\chi}'$. The one interesting case is that for Rule (AEvent), since the corresponding (CEvent) rule in the concrete semantics is the only one with a non-trivial premise. For brevity, we show only that case below.

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

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

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

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

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

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

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

Case (CIfstateNeg)**:** From Rule (CIfstateNeg) in the concrete semantics, we have $i =$ **ifstate** n L_2 and $\chi' = \langle s(L_1) : p(s(L_1)), \sigma, \nu, m, \tau \rangle$, where $n \neq m$. If $\hat{m} \neq \top$ then

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

□

Theorem 1 (Soundness)**.** *If the abstract machine does not enter a stuck state from the initial state χ_0, then for any concrete state $\chi \in \mathcal{C}$ reachable from the initial state χ_0, the concrete machine can make progress. If state χ is a security-relevant event then this progress is derived by the rule for the* `event` *instruction in the concrete operational semantics, whose premise guarantees that the event does not cause a policy violation. Thus, any program accepted by the abstract machine does not commit a policy violation when executed.*

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

4.4.2 Convergence

In practice, effective verification depends upon obtaining a fixed point for the abstract machine semantics in reasonable time for any given untrusted program. The convergence rate of the algorithm described in Sect. 4.3.4 depends in part on the height of the lattice of abstract states. This height dictates the number of iterations required to reach a fixed point in the worst case. All components of the language of abstract states defined in Figure 4.4 have height at most 2, except for the lattice *SS* of abstract traces. Lattice *SS* is finite whenever security automaton A is finite; therefore convergence is guaranteed in finite time. In the

proof that follows we prove the stronger result that lattice SS has non-exponential height—in particular, it has height that is quadratic in the size of the security automaton. As with the soundness proof, we have recently completed the following fully machine-verified proof of convergence using the Coq proof assistant (INRIA, 2014).

Theorem 2. *Let $A = (Q, \Sigma, \delta)$ be a deterministic, finite security automaton. Lattice $(SS, \subseteq)$ from Definition 3 has height $O(|Q|^2 + k|Q|)$.*

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

$$m(L) = (|Q| + 1)|suf(L)| - |Pre(L)|$$

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

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

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

$suf(L_2) = x$, then we have the following:

$$L_1 \subsetneq L_2$$

$$Res(Pre(L_1))x \subsetneq Res(Pre(L_2))x$$

$$Res(Pre(L_1)) \subsetneq Res(Pre(L_2))$$

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

4.5 Implementation

The binary rewriter and verifier are both implemented in Prolog, while the ABC Extractor and Injector are implemented in C. The rewriting framework uses a Definite Clause Grammar (DCG) (Shapiro and Sterling, 1994) parser that converts extracted ActionScript bytecode to an annotated abstract syntax tree (AST) for easy analysis and manipulation. We implemented this parser in Prolog so that the same code functions as a code generator due to the reversible nature of Prolog predicates (DeVries et al., 2009).

We used our implementation to enforce and certify three different policies on a collection of real-world Flash applets and AIR applications. Experimental results are shown in Figure 4.9. All tests were performed on an Intel Pentium Core 2 Duo machine running Yap Prolog v5.1.4.

Program Tested	Policy Enforced	Size Before	Size After	Rewriting Time	Verification Time
`countdownBadge`	`redir`	1.80 KB	1.95 KB	1.429s	0.532s
`NavToURL`	`redir`	0.93 KB	1.03 KB	0.863s	0.233s
`fiona`	`redir`	58.9 KB	59.3 KB	15.876s	0.891s
`calder`	`redir`	58.2 KB	58.6 KB	16.328s	0.880s
`posty`	`postok`	112.0 KB	113.0 KB	54.170s	2.443s
`fedex`	`flimit`	77.3 KB	78.0 KB	39.648s	1.729s

Figure 4.9. Experimental results

The `redir` policy prohibits malicious URL-redirections by ABC ad applets. Redirections are implemented at the bytecode level by `navigateToURL` system calls. The policy requires that method `check_url(s)` must be called to validate destination `s` before any redirection to `s` may occur. Method `check_url` has a trusted implementation provided by the ad distributor and/or web host, and may incorporate dynamic information such as ad hit counts or webpage context. Our IRM enforces this policy by injecting calls to `check_url` into untrusted applets. For better runtime efficiency, it positions some of these calls early in the program's execution (to pre-validate certain URL's) and injects runtime security state variables that avoid potentially expensive duplicate calls by tracking the history of past calls.

Policy `postok` sanitizes strings entered into message box widgets. This can be helpful in preventing cross-site scripting attacks, privacy violations, and buffer-overflow exploits that affect older versions of the ActionScript VM. We enforced the policy on the `Posty` AIR application, which allows users to post messages to social networking sites such as `Twitter`, `Jaiku`, `Tumblr`, and `Friendfeed`.

Policy `flimit` enforces a resource bound that disallows the creation of more than n files on the user's machine. We enforced this policy on the `FedEx Desktop` AIR application, which continuously monitors a user's shipment status and sends tracking information directly to his or her desktop. The IRM implements the policy by injecting a counter into the untrusted code that tracks file creations.

4.6 Conclusion

This chapter discusses preliminary work on certifying IRMs through model-checking. Our technique derives a state abstraction lattice from a security automaton to facilitate precise abstract interpretation of IRM code. Formal proofs of soundness and convergence guarantee reliability and tractability of the verification process. We demonstrate the feasibility of our technique by enforcing a URL anti-redirection policy for ActionScript bytecode programs.

We also demonstrate the elegance of using Prolog for implementing a certifying IRM sysetm. Using Prolog resulted in faster development and simpler implementation due to code reusability from reversible predicates and succinct program specifications from declarative programming. This resulted in a smaller trusted computing base for the overall system.

While our algorithm successfully verifies an important class of IRM implementations involving reified security state, it does not support all IRM rewriting strategies. Reified security state that is per-object (Hamlen et al., 2006a) instead of global, or that is updated by the IRM before the actual security state changes at runtime rather than after, are two examples of IRM strategies not supported by our model. In the next chapter, Chapter 5, we generalize our approach to cover these cases.

One area of future work remaining is augmenting our system with support for recursion and mutual recursion, which is currently not handled by our implementation.

CHAPTER 5

FULL-SCALE CERTIFICATION FOR THE SPOX JAVA IRM SYSTEM[1]

5.1 Overview

Aspect-oriented Programming (AOP) has been recognized as a natural and elegant fit for developing IRM systems. It additionally enjoys an extensive support system and tool base, thus gaining significant popularity in the IRM world. This rise in prominence is paralleled with a strong desire by the IRM community to to enhance AOP-based IRMs with formal verification. To provide exceptionally high assurance guarantees, recent work has sought to separately machine-verify the self-monitoring code that the AOP IRMs produce (Hamlen et al., 2006a; Aktug et al., 2008; Sridhar and Hamlen, 2010b, 2011). For example, the S^{3}MS project uses a contract-based verifier (Aktug et al., 2008) to avoid trusting the the much larger in-liner, usually over 900K lines of Java code if one includes the underlying AspectJ system (Kiczales et al., 2001), that generates the IRMs.

However, TCB-minimization of large AOP-IRM systems has been frustrated by the inevitable inclusion of significant, trusted code within the AOP-style policy specifications themselves. Verifiers for these systems can prove that the IRM system has correctly in-lined the policy-prescribed advice code but not that this advice actually enforces the desired policy. Past case studies have demonstrated that such advice is extremely difficult to write correctly, especially when the policy is intended to apply to large classes of untrusted programs rather than individual applications (Jones and Hamlen, 2010). Moreover, in many domains, such as web ad security, policy specifications change rapidly as new attacks and vulnerabilities

[1]This chapter includes previously published (Hamlen et al., 2012, 2011) joint work with Kevin W. Hamlen and Micah Jones.

are discovered (cf., Li and Wang, 2010; Sridhar and Hamlen, 2010a,b). Thus, the considerable effort that might be devoted to formally verifying one particular aspect implementation quickly becomes obsolete when the aspect is revised in response to a new threat.

To address this open challenge, we present Chekov✓: the first IRM-certification framework that verifies full, AOP-style IRMs against purely declarative policy specifications without trusting the code that implements the IRM[2]. Chekov✓ uses light-weight model-checking and abstract interpretation to verify untrusted (but verifiably type-safe) Java bytecode binaries against trusted policy specifications that lack advice. Policies declaratively specify how security-relevant program operations affect an abstract system security state. Unlike contracts, which denote code transformations, policies in our system therefore denote pure code properties. Such properties can be enforced by untrusted aspects that dynamically detect impending policy violations and take corrective action. The woven aspects are verified (along with the rest of the self-monitoring code) against the trusted policy specification prior to its execution.

Chekov✓ was inspired by our prior work on model-checking IRMs (Sridhar and Hamlen, 2010b,a; DeVries et al., 2009) (see Chapters 3 and 4), but includes numerous substantial theoretic and pragmatic leaps beyond those earlier works. These include:

- support for a full-scale Java IRM framework (the SPoX IRM system Hamlen and Jones, 2008; Jones and Hamlen, 2009) that includes stateful (history-based) policies, event detection by pointcut-matching, and IRM implementations that combine (untrusted) before- and after-advice insertions;

- a novel approach to dynamic pointcut verification using Constraint Logic Programming (CLP) (Jaffar and Maher, 1994); and

[2]Throughout this chapter, for simplicity, the terminology *IRM certification* refers to *IRM soundness certification*. *Transparency certification* of IRMs is discussed in Chapter 6.

- proofs of correctness based on Cousot's abstract interpretation framework (Cousot and Cousot, 1977) that link the denotational semantics of SPoX policies to the operational semantics of the abstract interpreter.

The chapter proceeds as follows. Section 5.2 discusses related work. Section 5.3 begins with a discussion of the SpoX policy language and rewriter. Section 5.4 presents a high-level description of the verification algorithm. Section 5.6 presents in-depth case-studies of several security policy classes that we enforced on numerous real-world applications, and discusses challenges faced in implementing and verifying these policies. Section 5.5 discusses the formal model. Section 5.7 concludes.

5.2 Related Work

Machine-certification of IRMs was first proposed as type-checking (Walker, 2000)—an idea that was later extended and implemented in the Mobile system (Hamlen et al., 2006a). Mobile transforms Microsoft .NET bytecode binaries into safe binaries with typing annotations in an effect-based type system. The annotations constitute a proof of safety that a type-checker can separately verify to prove that the transformed code is safe. Type-based IRM certification is efficient and elegant but does not currently support dynamic pointcut matching. It has therefore not been applied to AOP-style IRMs to our knowledge.

ConSpec (Aktug and Naliuka, 2008; Aktug et al., 2008) adopts a security-by-contract approach to AOP IRM certification. Its certifier performs a static analysis that verifies that contract-specified guard code appears at each security-relevant code point. While certification-via-contract facilitates natural expression of policies as AOP programs, it has the disadvantage of including the potentially complex advice code in the TCB.

Our prior work (Sridhar and Hamlen, 2010b) (presented in Chapter 4) is the first to adopt a model-checking approach to verify such IRMs without trusted guard code. The prototype

IRM certifier in (Sridhar and Hamlen, 2010b) supports reified security state, but it does not support dynamic pointcuts and its support for advice is limited to a very constrained form of before-advice. It therefore does not support real-world IRM systems or their policies, which regularly employ dynamic pointcuts and after-advice.

In contrast, the verifier presented in this work targets SPoX (Hamlen and Jones, 2008; Jones and Hamlen, 2009), a fully featured, purely declarative AOP IRM system for Java bytecode. SPoX policies are advice-free; any advice that implements the IRM remains untrusted and must therefore undergo verification. Policy specifications consist of pointcuts and declarative specifications of how pointcut-matching events affect the security state. The abstract security state-changes specified by SPoX policies are significantly higher-level and simpler than the arbitrary advice code admitted by non-declarative AOP languages. Thus, SPoX policies are a significant TCB reduction over AOP contracts that implement them.

5.3 Policy Language and Rewriter

5.3.1 SPoX Background

SPoX (Security Policy XML) is a purely declarative, aspect-oriented policy specification language (Hamlen and Jones, 2008). A SPoX specification denotes a security automaton (Alpern and Schneider, 1986)—a finite- or infinite-state machine that accepts all and only those *event sequences* that satisfy the security policy.

Security-relevant program *events* are specified in SPoX by pointcut expressions similar to those found in other aspect-oriented languages. In source-level AOP languages, pointcuts identify code join points at which advice code is to be inserted. SPoX derives its pointcut language from AspectJ, allowing policy writers to develop policies that regard static and dynamic method calls and their arguments, object pointers, and lexical contexts, among other properties.

In order to remain fully declarative, SPoX omits explicit, imperative advice. Instead, policies declaratively specify how security-relevant events change the current security automaton state. Rewriters then synthesize their own advice in order to enforce the prescribed policy. The use of declarative state-transitions instead of imperative advice facilitates formal, automated reasoning about policies without the need to reason about arbitrary code (Jones and Hamlen, 2010). State-transitions can be specified in terms of information gleaned from the current join point, such as method argument values, the call stack, and the current lexical scope. This allows advice typically encoded imperatively in most other aspect-oriented security languages to be declaratively encoded in SPoX policies. Typically this results in a natural translation from these other languages to SPoX, making SPoX an ideal target for our analysis.

The remainder of this section outlines SPoX syntax as background for the case studies in Section 5.6. A formal denotational semantics can be found in Section 5.5. We here use a simplified Lisp-style syntax for readability; the implementation uses an XML-based syntax for easy parsing, portability, and extensibility.

A SPoX policy specification (*pol* in Figure 5.1) is a list of security automaton edge declarations. Each edge declaration consists of three parts:

- Pointcut expressions (Figure 5.2) identify sets of related security-relevant events that programs might exhibit at runtime. These label the edges of the security automaton.

- *Security-state variable* declarations (*sd* in Figure 5.1) abstract the security state of an arbitrary program. The security state is defined by the set of all program state variables and their integer[3] values. These label the automaton nodes.

[3]Binary operator `/` in Figure 5.1 denotes integer division.

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

Figure 5.1. SPoX policy syntax

- *Security-state transitions* (e in Figure 5.1) describe how events cause the security automaton's state to change at runtime. These define the transition relation for the automaton.

An example policy is given in Figure 5.3. Edges are specified by `edge` structures, each of which defines a set of edges in the security automaton. Each `edge` structure consists of a pointcut expression (Lines 5 and 9) and at least one `nodes` declaration (Lines 6 and 10). The pointcut expression defines a common label for the edges in the set, while each `nodes` declaration imposes a transition pre-condition and post-condition for a particular state variable. The pre-condition constrains the set of source states to which the edge applies, and the post-condition describes how the state changes when an event satisfying the pointcut expression and all pre-conditions is exhibited. Events that satisfy none of the outgoing edge

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

Figure 5.2. SPoX pointcut syntax

```
1 (state name="s")
2
3 (forall "i" from 0 to 9
4   (edge name="count"
5     (call "Mail.send")
6     (nodes "s" i,i+1)))
7
8 (edge name="10emails"
9   (call "Mail.send")
10  (nodes "s" 10,#))
```

Figure 5.3. A policy permitting at most 10 email-send events

labels of the current security state leave the security state unchanged. Policy-violations are identified with the reserved post-condition "`#`".

Multiple, similar edges can be introduced with a single `edge` structure by enclosing them within `forall` structures, such as the one in Line 3. These introduce iteration variables (e.g., `i`) that range over the integer lattice points of closed intervals. Thus, Figure 5.3 allows state variable `s` to range from 0 to 10, while an 11th send event triggers a policy violation. Such a policy could be useful for preventing spam.

A syntax for a subset of the SPoX pointcut language is given in Figure 5.2. SPoX pointcut expressions consist of all pointcuts available in AspectJ (The AspectJ Team, 2003) except for those that are specific to AspectJ's advice language.[4] This includes all regular expression operators available in AspectJ for specifying class and member names. Since SPoX policies are applied to Java bytecode binaries rather than to source code, the meaning of each pointcut expression is reflected down to the bytecode level. For example, the `target` pointcut matches any Java bytecode instruction whose *this* argument references an object of class c.

Instead of AspectJ's `if` pointcut (which evaluates an arbitrary, possibly effectful, Java boolean expression), SPoX provides a collection of effect-free *value predicates* that permit

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

dynamic tests of argument values at join points. These are accessed via the `argval` predicate and include object nullity tests, integer equality and inequality tests, and string regular expression matching. Regular expression tests of non-string objects are evaluated by obtaining the `toString` representation of the object at runtime. (The call to the `toString` method itself is a potentially effectful operation and is treated as a matchable join point by the SPoX enforcement implementation. However, subsequent use of the returned string within injected security guard code is non-effectful.)

5.3.2 Rewriter

The SPoX rewriter takes as input a Java binary archive (JAR) and a SPoX policy, and outputs a new application in-lined with an IRM that enforces the policy. The high-level in-lining approach is essentially the same as the other IRM systems discussed in Section 5.2. Each method body is unpacked, parsed, and scanned for potential security-relevant instructions—i.e., those that match the statically decidable portions of one or more pointcut expressions in the policy specification. Sequences of guard instructions are then in-lined around these potentially dangerous instructions to detect and preclude policy-violations at runtime. The runtime guards evaluate the statically undecidable portions of the pointcut expressions in order to decide whether the impending event is actually security-relevant. For example, to evaluate the pointcut `(argval 1 (intgt 2))`, the rewriter might guard method calls of the form $m(x)$ with the test $x > 2$.

In-lined guard code must also track event histories if the policy is stateful. To do so, the rewriter reifies abstract security state variables (e.g., `s` in Figure 5.3) into the untrusted code as program variables. The guard code then tracks the abstract security state by consulting and updating the corresponding reified state. To protect reified state from tampering, the variables are typically added as private fields of new classes with safe accessor methods. This prevents the surrounding original bytecode from corrupting the reified state and thereby effecting a policy violation.

	$(A{=}S \wedge A{=}T)$	**0.1**
`1 if (Policy.s >= 0 && Policy.s <= 9)`		
	$(A{=}S \wedge A{=}T \wedge S{\geq}0 \wedge S{\leq}9)$	**1.1**
`2  Policy.temp_s := Policy.s+1;`		
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1)$	**2.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9))$	**2.2**
`3 if (Policy.s == 10)`		
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1 \wedge S{=}10)$	**3.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9) \wedge S{=}10)$	**3.2**
`4  call System.exit(1);`		
	$(A{=}S \wedge A{=}T' \wedge S{\geq}0 \wedge S{\leq}9 \wedge T{=}S{+}1 \wedge S{\neq}10)$	**4.1**
	$(A{=}S \wedge A{=}T \wedge (S{<}0 \vee S{>}9) \wedge S{\neq}10)$	**4.2**
`5 Policy.s := Policy.temp_s;`		
	$(A{=}S' \wedge A{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T)$	**5.1**
	$(A{=}S' \wedge A{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T)$	**5.2**
`6 call Mail.send();`		
	$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}I \wedge I{\geq}0 \wedge I{\leq}9 \wedge A{=}I{+}1)$	**6.1**
	$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}10 \wedge A{=}\#)$	**6.2**
	$(A'{=}S' \wedge A'{=}T' \wedge S'{\geq}0 \wedge S'{\leq}9 \wedge T{=}S'{+}1 \wedge S'{\neq}10 \wedge S{=}T \wedge (A'{<}0 \vee A'{>}9) \wedge A'{\neq}10 \wedge A{=}A')$	**6.3**
	$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}I \wedge I{\geq}0 \wedge I{\leq}9 \wedge A{=}I{+}1)$	**6.4**
	$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge A'{=}10 \wedge A{=}\#)$	**6.5**
	$(A'{=}S' \wedge A'{=}T \wedge (S'{<}0 \vee S'{>}9) \wedge S'{\neq}10 \wedge S{=}T \wedge (A'{<}0 \vee A'{>}9) \wedge A'{\neq}10 \wedge A{=}A')$	**6.6**

Figure 5.4. An abstract interpretation of instrumented pseudocode

The left column of Figure 5.4 gives pseudocode for an IRM that enforces the policy in Figure 5.3. For each call to method `Mail.send`, the IRM tests two possible preconditions: one where $0 \leq \texttt{s} \leq 9$ and another where $\texttt{s} = 10$. In the first case, it increments `s`; in the second, it aborts the process.

Observe that in this example security state `s` has been reified as two separate fields of class `Policy`—`s` and `temp_s`. This reflects a reality that any given policy has a variety of IRM implementations, many of which contain unexpected quirks that address non-obvious, low-level enforcement details. In this case the double reification is part of a mechanism for resolving potential join point conflicts in the source policy (Jones and Hamlen, 2010). A certifier must tolerate such variations in order to be generally applicable to many IRMs and not just one rewriting strategy.

5.4 Verifier

Our verifier takes as input (1) a SPoX security policy, (2) an instrumented, type-safe Java bytecode program, and (3) some optional, untrusted hints from the rewriter (detailed shortly). It either accepts the program as provably policy-satisfying or rejects it as potentially policy-violating. Type-safety is checked by the JVM, allowing our verifier to safely assume that all bytecode operations obey standard Java memory-safety and well-formedness. This keeps tractable the task of reliably identifying security relevant operations and field accesses.

The main verifier engine uses abstract interpretation to non-deterministically explore all control-flow paths of the untrusted code, inferring an abstract program state at each code point. A model-checker then proves that each abstract state is policy-adherent, thereby verifying that no execution of the code enters a policy-violating program state. Policy-violations are modeled as *stuck states* in the operational semantics of the verifier—that is, abstract interpretation cannot continue when the current abstract state fails the model-checking step. This results in conservative rejection of the untrusted code. The verifier is

expressed as a bisimulation of the program and the security automaton. Abstract states in the analysis conservatively approximate not only the possible contents of memory (e.g., stack and heap contents) but also the possible security states of the system at each code point.

The heart of the verification algorithm involves inferring and verifying relationships between the abstract program state and the abstract security state. When policies are stateful, this involves verifying relationships between the abstract security state and the corresponding reified security state(s). These relationships are complicated by the fact that although the reified state often precisely encodes the actual security state, there are also extended periods during which the reified and abstract security states are not synchronized at runtime. For example, guard code may preemptively update the reified state to reflect a future security state that will only be reached after subsequent security-relevant events, or it may retroactively update the reified state only after numerous operations that change the security state have occurred. These two scenarios correspond to the insertion of before- and after-advice in AOP IRM implementations. The verification algorithm must be powerful enough to automatically track these relationships and verify that guard code implemented by the IRM suffices to prevent policy violations.

To aid the verifier in this task, we modified the SPoX rewriter to export two forms of untrusted hints along with the rewritten code: (1) a relation $\sim$ that associates policy-specified security state variables s with their reifications r, and (2) marks that identify code regions where related abstract and reified states might not be *synchronized* according to the following definition:

Definition 5 (Synchronization Point)**.** *A* synchronization point *(SYNC) is an abstract program state with constraints ζ such that proposition $\zeta \wedge \left(\bigvee_{r \sim s}(r \neq s)\right)$ is unsatisfiable.*

Chekov✓ uses these hints (without trusting them) to guide the verification process and to avoid state-space explosions that might lead to conservative rejection of safe code. In par-

ticular, it verifies that all non-marked instructions are *SYNC*-preserving, and each outgoing control-flow from a marked region is *SYNC*-restoring. This modularizes the verification task by allowing separate verification of marked regions, and controls state-space explosions by reducing the abstract state to *SYNC* throughout the majority of binary code which is not security-relevant. Providing incorrect hints causes Chekov✓ to reject (e.g., when it discovers that an unmarked code point is potentially security-relevant) or converge more slowly (e.g., when security-irrelevant regions are marked and therefore undergo unnecessary extra analysis), but it never leads to unsound certification of unsafe code.

A Verification Example. Figure 5.4 demonstrates a verification example step-by-step. The pseudocode constitutes a marked region in the target program, and the verifier requires that the abstract interpreter is in the *SYNC* state immediately before and after. At each code point, the verifier infers an abstract program state that includes one or more conjunctions of constraints on the abstract and reified security state variables. These constraints track the relationships between the reified and abstract security state. Here, variable A represents the abstract state variable `s` from the policy in Figure 5.3. Reifications `Policy.s` and `Policy.temp_s` are written as S and T, respectively, with $S \sim A$ and $T \sim A$. Thus, state *SYNC* is given by constraint expression $(A = S \wedge A = T)$ in this example.

The analysis begins in the *SYNC* state, as shown in constraint list 0.1. Line 1 is a conditional, and thus spawns two new constraint lists, one for each branch. The positive branch (1.1) incorporates the conditional expression $(S \geq 0 \wedge S \leq 9)$ in Line 2, whereas the negative branch (2.2) incorporates the negation of the same conditional. The assignment in Line 2 is modeled by alpha-converting T to T' and conjoining constraint $S = T + 1$; this yields constraint list 2.1.[5]

[5]The + here denotes two's-complement addition to handle arithmetic overflows.

Unsatisfiable constraint lists are opportunistically pruned to reduce the state space. For example, list 3.1 shows the result of applying the conditional of Line 3 to 2.1. Conditionals 1 and 3 are mutually exclusive, resulting in contradictory expressions $S \leq 9$ and $S = 10$; therefore, 3.1 is dropped. Similarly, 3.2 is dropped because no control-flows exit Line 4.

To interpret a security-relevant event such as the one in Line 6, the verifier simulates the traversal of all edges in the security automaton. In typical policies, any given instruction fails to match a majority of the pointcut labels in the policy, so most are immediately dropped. The remaining edges are simulated by conjoining each edge's pre-conditions to the current constraint list and modeling the edge's post-condition as a direct assignment to A. For example, edge `count` in Figure 5.3 imposes pre-condition $(0 \leq I \leq 9) \wedge (A = I)$, and its post-condition can be modeled as assignment $A := I + 1$. Applying these to list 5.1 yields list 6.1. Likewise, 6.2 is the result of applying edge `10emails` to 5.1, and 6.4 and 6.5 are the results of applying the two edges (respectively) to 5.2.

Constraints 6.3 and 6.6 model the possibility that no explicit edge matches, and therefore the security state remains unchanged. They are obtained by conjoining the negations of all of the edge pre-conditions to states 5.1 and 5.2, respectively. Thus, security-relevant events have a multiplicative effect on the state space, expanding n abstract states into at worst $n(m + 1)$ states, where m is the number of potential pointcut matches.

If any constraint list is satisfiable and contains the expression $A = \texttt{\#}$, the verifier cannot disprove the possibility of a policy violation and therefore conservatively rejects. Constraints 6.2 and 6.5 both contain this expression, but they are unsatisfiable, proving that a violation cannot occur. Observe that the IRM guard at Line 3 is critical for proving the safety of this code because it introduces constraint $S' \neq 10$ that makes these two lists unsatisfiable. If Lines 3–4 were not included, the verifier would reject at this point because constraints 6.2 and 6.5 are satisfiable with $A = \texttt{\#}$ without clause $S' \neq 10$.

At all control-flows from marked to unmarked regions, the verifier requires a constraint list that implies $SYNC$. In this example, constraints 6.1 and 6.6 are the only remaining lists that

are satisfiable, and conjoining them with the negation of $SYNC$ expression $(A = S) \wedge (A = T)$ yields an unsatisfiable list. Thus, this code is accepted as policy-adherent.

	$(A{=}S \wedge A{=}T)$ **0.1**
`1 x = 1;`	
	$(A{=}S \wedge A{=}T \wedge X{=}1)$ **1.1**
`2 if (Policy.s == 0 && x > 2)`	
	$(A{=}S \wedge A{=}T \wedge X{=}1 \wedge S{=}0 \wedge X{>}2)$ **2.1**
`3 System.exit()`	
	$(A{=}S \wedge A{=}T \wedge X{=}1 \wedge (S{\neq}0 \vee X{\leq}2)$ **3.1**
`4 call secure_method(x);`	
	$(A'{=}S \wedge A'{=}T \wedge X{=}1 \wedge (S{\neq}0 \vee X{\leq}2) \wedge A'{=}0 \wedge X{>}2 \wedge A{=}\#)$ **4.1**
	$(A'{=}S \wedge A'{=}T \wedge X{=}1 \wedge (S{\neq}0 \vee X{\leq}2) \wedge A'{\neq}0 \vee X{\leq}2) \wedge A'{=}A)$ **4.2**

$A{=}$ abstract security state
$S{=}$ reified security state `Policy.s`
$T{=}$ reified security state `Policy.temp_s`

Figure 5.5. An example verification with dynamically decidable pointcuts

Dynamically Decided Pointcuts. Verification of events corresponding to statically undecidable pointcuts (such as `argval`) requires analysis of dynamic checks inserted by the rewriter, which consider the contents of the stack and local variables at runtime.

An example is shown in Figure 5.5, which enforces a policy that prohibits calls to method `secure_method` with arguments greater than 2. Verifying this IRM requires the inclusion of abstract state variable X in constraint lists to model the value of local program variable `x`. The abstract interpreter therefore tracks all numerically typed stack and local variables, and incorporates Java bytecode conditional expressions that test them into constraint lists.

Numeric comparisons are translated directly into constraint expressions; for example, the instruction `if(x>2)` introduces clause $X > 2$ for the positive branch and $X \leq 2$ for the negative branch. Non-numeric dynamic pointcuts (e.g., `streq` pointcut expressions) are modeled by reducing them to equivalent integer encodings. For example, to support dynamic string regexp-matching, Chekov✓ introduces a boolean-valued variable X_{re} for each string-typed program variable `x` and policy regexp re. Program operations that test `x` against

re introduce constraint $X_{re} = 1$ in their positive branches and $X_{re} = 0$ in their negative branches. An in-depth verification example involving dynamically decidable pointcuts is provided in the companion technical report (Hamlen et al., 2011).

Limitations. Our verifier supports most forms of Java reflection, but in order to safely track write-accesses to reified security state fields, the verifier requires such fields to be static, private class members, and it conservatively rejects programs that contain reflective field-write operations within classes that contain reified state. Thus, in order to pass verification, rewriters must implement reified state fields within classes that do not perform write-reflection. This is standard practice for most IRM systems including SPoX, so did not limit any of our tests. Instrumented programs may detect and respond to the presence of the IRM through read-reflection, but not in a way that violates the policy.

Our system supports IRMs that maintain a global invariant whose preservation across the majority of the rewritten code suffices to prove safety for small sections of security-relevant code, followed by restoration of the invariant. Our experience with existing IRM systems indicates that most IRMs do maintain such an invariant ($SYNC$) as a way to avoid reasoning about large portions of security-irrelevant code in the original binary. However, IRMs that maintain no such invariant, or that maintain an invariant inexpressible in our constraint language, cannot be verified by our system. For example, an IRM that stores object security states in a hash table cannot be certified because our constraint language is not sufficiently powerful to express collision properties of hash functions and prove that a correct mapping from security-relevant objects to their security states is maintained by the IRM.

To keep the rewriter's annotation burden small, our certifier also uses this same invariant as a loop-invariant for all cycles in the control-flow graph. This includes recursive cycles in the call graph as well as control-flow cycles within method bodies. Most IRM frameworks do not introduce such loops to non-synchronized regions. However, this limitation could become

problematic for frameworks wishing to implement code-motion optimizations that separate security-relevant operations from their guards by an intervening loop boundary. Allowing the rewriter to suggest different invariants for different loops would lift the limitation, but taking advantage of this capability would require the development of rewriters that infer and express suitable loop invariants for the IRMs they produce. To our knowledge, no existing IRM systems yet do this.

While our certifier is provably convergent (since it arrives at a fixpoint for every loop through enforcing *SYNC* on loop back-edges), it can experience state-space explosions that are exponential in the size of each contiguous, unsynchronized code region. Typical IRMs limit such regions to relatively small, separate code blocks scattered throughout the rewritten code; therefore, we have not observed this to be a significant limitation in practice. However, such state-space explosions could be controlled without conservative rejection by applying the same solution above. That is, rewriters could suggest state abstractions for arbitrary code points, allowing the certifier to forget information that is unnecessary for proving safety and that leads to a state-space explosion. Again, the challenge here is developing rewriters that can actually generate such abstractions.

Our current implementation and theoretical analysis are for purely serial programs; concurrency support is reserved for future work. Analysis, enforcement, and certification of multithreaded IRMs is an ongoing subject of current research with several interesting open problems (cf., Dam et al., 2009).

5.5 System Formal Model

The certifier in our certifying IRM framework forms the centerpiece of the trusted computing base of the system, allowing the monitor and monitor-prod-ucing tools to remain untrusted. An unsound certifier (i.e., one that fails to reject some policy-violating programs) can lead to

system compromise and potential damage. It is therefore important to establish exceptionally high assurance for the certification algorithm and its implementation.

In this section we address the former requirement by formalizing the certification algorithm as the operational semantics of an abstract machine. For brevity, we here limit our attention to a core subset of Java bytecode that is representative of important features of the full language.[6] We additionally formalize the JVM as the operational semantics of a corresponding concrete machine over the same core subset. These two semantics together facilitate a proof of soundness in Section 5.5.6. The proof establishes that executing any program accepted by the certifier never results in a policy violation at runtime.

5.5.1 Java Bytecode Core Subset

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

Figure 5.6. Core subset of Java bytecode

Figure 5.6 lists the subset of Java bytecode that we consider. Instruction **ifle** L implements conditional jumps, instruction **jmp** n implements unconditional jumps, and instructions **getlocal** n and **setlocal** n read and set local variable values, respectively. Instruction **event**$_y$ n models a security-relevant operation that exhibits event n and pops y arguments off the operand stack. While the real Java bytecode instruction set does not include **event**$_y$, in practice it is implemented as a fixed instruction sequence that performs a security-relevant operation (e.g., a system call).

[6]The implementation supports the full Java bytecode language (see Section 5.6).

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

Figure 5.7. Core subset of SPoX

Figure 5.7 defines a core subset of SPoX for the Java bytecode language in Figure 5.6. Without loss of generality, it assumes all pointcuts are expressed in disjunctive normal form.

5.5.2 Concrete Machine

$\chi ::= \langle L : i, \rho, \sigma \rangle$	(CONFIGURATIONS)
L	(CODE LABELS)
i	(JAVA BYTECODE INSTRUCTIONS)
$\Sigma : (r \uplus a \uplus \ell) \rightharpoonup \mathbb{Z}$	(CONCRETE STORE MAPPINGS)
$\sigma \in \Sigma$	(CONCRETE STORES)
$\rho ::= \cdot \mid x :: \rho$	(CONCRETE STACK)
$x \in \mathbb{Z}$	(CONCRETE PROGRAM VALUES)
χ_0	(INITIAL CONFIGURATIONS)
$P ::= (L, p, s)$	(PROGRAMS)
$p : L \longrightarrow i$	(INSTRUCTION LABELS)
$s : L \longrightarrow L$	(LABEL SUCCESSORS)

Figure 5.8. Concrete machine configurations and programs

We start out by formalizing the JVM as the operational semantics of a concrete machine over our core Java bytecode subset. Following the framework established in (Sridhar and Hamlen, 2010b), Figure 5.8 defines the concrete machine as a tuple $(\mathcal{C}, \chi_0, \mapsto)$, where $\mathcal{C}$ is the

set of concrete configurations, χ_0 is the initial configuration, and $\mapsto$ is the transition relation in the concrete domain. A *concrete configuration* $\chi ::= \langle L{:}i, \rho, \sigma \rangle$ is a triple consisting of a labeled bytecode instruction $L{:}i$, a concrete operand stack ρ, and a concrete store σ. The store σ maps heap and the local variables ℓ, abstract security state variables a, and reified security state variables r to their integer values. A security automaton state is σ restricted to the abstract state variables, denoted $\sigma|_a$.

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

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

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

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

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

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

Figure 5.9. Concrete small-step operational semantics

Figure 5.9 provides the small-step operational semantics of the concrete machine. Policy-violating events fail to satisfy the premise of Rule (CEvent); therefore the concrete semantics model policy-violations as stuck states. The concrete semantics have no explicit operation for normal program termination; we model termination as an infinite stutter state. The soundness proof in Section 5.5.6 shows that any program that is accepted by the abstract machine will never enter a stuck state during any concrete run; thus, verified programs do not exhibit policy violations when executed.

5.5.3 SPoX Concrete Denotational Semantics

A SPoX security policy denotes a security automaton whose alphabet is the universe *JP* of all *join points*. We refer to such an automaton as an *aspect-oriented security automaton*. A join

$o \in Obj$	(OBJECTS)
$v ::= o \mid \mathsf{null}$	(VALUES)
$jp ::= \langle\rangle \mid \langle k, x^*, jp\rangle$	(JOIN POINTS)
$k ::= \texttt{call}\ c.md \mid \texttt{get}\ c.fd \mid \texttt{set}\ c.fd$	(JOIN KINDS)

Figure 5.10. Join points

point, defined in Figure 5.10, is a recursive structure that abstracts the control stack (Wand et al., 2004). Join point $\langle k, v^*, jp\rangle$ consists of static information k found at the site of the current program instruction, dynamic information v^* including any arguments consumed by the instruction, and recursive join point jp modeling the rest of the control stack. The empty control stack is modeled by the empty join point $\langle\rangle$.

The denotational semantics in Figure 5.11 transform a SPoX policy into an aspect-oriented security automaton, which accepts or rejects (possibly infinite) sequences of join points. We use $\uplus$ for disjoint union, Υ for the class of all countable sets, 2^A for the power set of A, $\sqsubseteq$ and $\sqcup$ for the partial order relation and join operation (respectively) over the lattice of partial functions, and $\perp$ for the partial function whose domain is empty. For partial functions f and g we write $f[g] = \{(x, f(x)) \mid x \in Dom(f) \backslash Dom(g)\} \sqcup g$ to denote the replacement of assignments in f with those in g.

Security automata are modeled in the literature (Schneider, 2000) as tuples (Q, Q_0, E, δ) consisting of a set Q of states, a set $Q_0 \subseteq Q$ of start states, an alphabet E of events, and a transition function $\delta : (Q \times E) \to 2^Q$. Security automata are non-deterministic; the automaton accepts an event sequence if and only if there exists an accepting path for the sequence. In the case of aspect-oriented security automata, Q is the set of partial functions from security-state variables to values, $Q_0 = \{q_0\}$ is the initial state that assigns 0 to all security-state variables, $E = JP$ is the universe of join points, and δ is defined by the set of edge declarations in the policy (discussed below).

$q \in Q = a \to \mathbb{Z}$ (SECURITY STATES)

$S \in SM = SV \rightharpoonup \mathbb{Z}$ (STATE-VARIABLE MAPS)

$\psi \in \Psi = \hat{v} \rightharpoonup \mathbb{Z}$ (META-VARIABLE MAPS)

$\mu \in \Psi \times \Sigma$ (ABSTRACT-CONCRETE MAP PAIRS)

$\mathcal{P} : pol \to \big(\Upsilon \times 2^Q \times \Upsilon \times ((Q \times JP) \to 2^Q)\big)$ (POLICY DENOTATIONS)

$\mathcal{ES} : edg \to \Psi \to 2^{(JP \to \{Suceed, Fail\}) \times SM \times SM}$ (EDGESET DENOTATIONS)

$\mathcal{PC} : pcd \to JP \to \{Succ, Fail\}$ (POINTCUT DENOTATIONS)

$\mathcal{EP} : s \to \Psi \to (SM \times SM)$ (ENDPOINT CONSTRAINTS)

$\mathcal{E} : e \longrightarrow (\Psi \times \Sigma) \longrightarrow \mathbb{Z}$ (EXPRESSION DENOTATIONS)

$\mathcal{P}[\![edg_1 \ldots edg_n]\!] = (Q, \{q_0\}, JP, \delta)$
where $q_0 = SV \times \{0\}$
and $\delta(q, jp) = \{q[S'] \mid (f, S, S') \in \cup_{1 \le i \le n} \mathcal{ES}[\![edg_i]\!]\bot, S \sqsubseteq q, f(jp) = Succ\}$

$\mathcal{ES}[\![\texttt{(forall } \hat{v} \texttt{ from } a_1 \texttt{ to } a_2 \texttt{ } edg\texttt{)}]\!]\psi =$
$\cup_{\mathcal{A}[\![a_1]\!]\psi \le j \le \mathcal{A}[\![a_2]\!]\psi} \mathcal{ES}[\![edg]\!](\psi[j/\hat{v}])$

$\mathcal{ES}[\![\texttt{(edge } pcd \texttt{ } ep_1 \ldots ep_n\texttt{)}]\!]\psi =$
$\big\{(\mathcal{PC}[\![pcd]\!], \sqcup_{1 \le j \le n} S_j, \sqcup_{1 \le j \le n} S'_j)\big\}$
where $\forall j \in \mathbb{N} \,.\, (1 \le j \le n) \Rightarrow ((S_j, S'_j) = \mathcal{EP}[\![ep_j]\!]\psi)$

$\mathcal{PC}[\![pcd]\!]jp = match\text{-}pcd(pcd)jp$

$\mathcal{EP}[\![\texttt{(nodes "}sv\texttt{" } a_1\texttt{,}a_2\texttt{)}]\!]\psi =$
$\big(\{(sv, \mathcal{E}[\![a_1]\!](\psi, \bot))\}, \{(sv, \mathcal{E}[\![a_2]\!](\psi, \bot))\}\big)$

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

$\mathcal{E}[\![x]\!](\psi, \sigma) = \sigma(x) \qquad (x \in r \uplus a \uplus \ell)$

$\mathcal{E}[\![\hat{v}]\!](\psi, \sigma) = \psi(\hat{v})$

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

$\mathcal{E}[\![e_1 - e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu - \mathcal{E}[\![e_2]\!]\mu$

$\mathcal{E}[\![e_1 \cdot e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu \cdot \mathcal{E}[\![e_2]\!]\mu$

$\mathcal{E}[\![e_1 / e_2]\!]\mu = \mathcal{E}[\![e_1]\!]\mu / \mathcal{E}[\![e_2]\!]\mu$

Figure 5.11. Denotational semantics for SPoX

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

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

Figure 5.12. Matching pointcuts to join points

Each edge declaration in a SPoX policy defines a set of source states and the destination state to which each of these source states is mapped when a join point occurs that *matches* the edge's pointcut designator. The denotational semantics in Figure 5.11 defines this matching process in terms of the *match-pcd* function from the operational semantics of AspectJ (Wand et al., 2004). We adapt a subset of pointcut matching rules from this definition to SPoX syntax in Figure 5.12.

5.5.4 Abstract Machine

In order to statically detect and prevent policy violations, we model the verifier as an abstract machine. The abstract machine is defined as a triple $(\mathcal{A}, \hat{\chi}_0, \rightsquigarrow)$, where $\mathcal{A}$ is the set of configurations of the abstract machine, $\hat{\chi}_0 \in \mathcal{A}$ is an initial configuration, and $\rightsquigarrow$ is the

$$\hat{\chi} ::= \bot \mid \langle L{:}i, \zeta, \hat{\rho}, \hat{\sigma} \rangle \qquad \text{(ABSTRACT CONFIGS)}$$

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

$$t ::= T \mid F \mid e_1 \leq e_2 \qquad \text{(PREDICATES)}$$

$$\hat{\rho} ::= \cdot \mid e :: \hat{\rho} \qquad \text{(ABSTRACT STACK)}$$

$$\hat{\Sigma} : (r \uplus \ell) \longrightarrow e \qquad \text{(ABSTRACT STORE MAPPINGS)}$$

$$\hat{\sigma} \in \hat{\Sigma} \qquad \text{(ABSTRACT STORES)}$$

$$\hat{\chi}_0 \qquad \text{(INITIAL ABSTRACT CONFIG)}$$

Figure 5.13. Abstract machine configurations

transition relation in the abstract domain. Figure 5.13 defines *abstract configurations* $\hat{\chi}$ to be either $\bot$ (denoting an unreachable state) or a tuple $\langle L{:}i, \zeta, \hat{\rho}, \hat{\sigma} \rangle$, where $L{:}i$ is a labeled instruction, ζ is a constraint list, and $\hat{\rho}$ and $\hat{\sigma}$ model the abstract operand stack and abstract store, respectively. The domains of $\hat{\rho}$ and $\hat{\sigma}$ consist of symbolic expressions instead of integer values.

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

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

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

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

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

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

Figure 5.14. Abstract small-step operational semantics

The small-step operational semantics of the abstract machine are given in Figure 5.14. Rules (AIFLEPOS), (AIFLENEG), and (AEVENT) are non-deterministic—the abstract machine

non-deterministically explores both branches of conditional jumps and all possible security automaton transitions for security-relevant events.

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

$$\begin{aligned}
&\hat{\mathcal{P}}[\![edg_1 \ldots edg_n]\!]\widehat{jp} = \bigcup_{i=1}^{n} \widehat{\mathcal{ES}}[\![edg_i]\!]\widehat{jp} \\
&\widehat{\mathcal{ES}}[\![\texttt{forall}\ \hat{v}{=}e_1..e_2\ edg]\!]\widehat{jp} = \{(\hat{v} \geq e_1) \wedge (\hat{v} \leq e_2) \wedge \zeta \mid \zeta \in \widehat{\mathcal{ES}}[\![edg]\!]\widehat{jp}\} \\
&\widehat{\mathcal{ES}}[\![\texttt{edge}\ pcd\ ep_1 \ldots ep_n]\!]\widehat{jp} = \{\zeta \wedge (\wedge_{i=1}^{n} \widehat{\mathcal{EP}}[\![ep_i]\!]) \mid \zeta \in \widehat{\mathcal{PCD}}[\![pcd]\!]\widehat{jp}\} \\
&\widehat{\mathcal{PCD}}[\![\texttt{or}\ pcc_1 \ldots pcc_n]\!]\widehat{jp} = \{\widehat{\mathcal{PCC}}[\![pcc_i]\!]\widehat{jp} \mid 1 \leq i \leq n\} \\
&\widehat{\mathcal{PCC}}[\![\texttt{and}\ pct_1 \ldots pct_n]\!]\widehat{jp} = \bigwedge_{i=1}^{n} \widehat{\mathcal{PCC}}[\![pct_i]\!]jp \\
&\widehat{\mathcal{PCC}}[\![\texttt{not}\ pca]\!]\widehat{jp} = \neg(\widehat{\mathcal{PCC}}[\![pca]\!]\widehat{jp}) \\
&\widehat{\mathcal{PCC}}[\![\texttt{event}_y\ n]\!]\langle \texttt{event}_z\ m, e^*, \widehat{jp}\rangle = (n{=}m) \\
&\widehat{\mathcal{PCC}}[\![\texttt{arg}\ n\ (\texttt{intleq}\ m)]\!]\langle k, e_1{::}\cdots{::}e_n{::}e^*, \widehat{jp}\rangle = (e_n \leq m) \\
&\widehat{\mathcal{EP}}[\![\texttt{nodes}\ a\ e_1\ e_2]\!] = (a_0 = e_1) \wedge (a = e_2)
\end{aligned}$$

Figure 5.15. Abstract Denotational Semantics

Rule (AEvent) is the model-checking step. Its premise appeals to an *abstract denotational semantics* $\hat{\mathcal{P}}$ for SPoX, defined in Figure 5.15, to infer possible security automaton transitions for policy-satisfying events. Policy-violating events (for which there is no transition in the automaton) therefore correspond to stuck states in the abstract semantics.

In Figure 5.15, $\widehat{jp} \in \widehat{JP}$ denotes an *abstract join point*—a join point (see Figure 5.10) whose stack consists of symbolic expressions instead of values. Valuation function $\hat{\mathcal{P}}$ accepts as input a policy and an abstract join point that models the current abstract program state.

It returns a set of constraint lists, one list for each possible new abstract program state. If ζ_1 is the original constraint list, then the new set of constraint lists is

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

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

5.5.5 Abstract Interpretation

$$\mathcal{T} : e \longrightarrow 2^{\Psi \times \Sigma}$$
$$\mathcal{T}[\![T]\!] = \Psi \times \Sigma$$
$$\mathcal{T}[\![F]\!] = \emptyset$$
$$\mathcal{T}[\![e_1 \leq e_2]\!] = \{\mu \in \Psi \times \Sigma \mid \mathcal{E}[\![e_1]\!]\mu \leq \mathcal{E}[\![e_2]\!]\mu\}$$
$$\mathcal{C} : \zeta \longrightarrow 2^{\Psi \times \Sigma}$$
$$\mathcal{C}[\![\bigwedge_{i=1\ldots n} t_i]\!] = \bigcap_{i=1\ldots n} \mathcal{T}[\![t_i]\!]$$
$$\frac{\mathcal{E}[\![e_1]\!](\mathcal{C}[\![\zeta_1]\!]) \subseteq \mathcal{E}[\![e_2]\!](\mathcal{C}[\![\zeta_2]\!])}{(\zeta_1, e_1) \preceq_e (\zeta_2, e_2)}$$
$$\frac{\mathcal{C}[\![\zeta_1]\!] \subseteq \mathcal{C}[\![\zeta_2]\!]}{(\zeta_1, \cdot) \preceq_\rho (\zeta_2, \cdot)}$$
$$\frac{(\zeta_1, e_1) \preceq_e (\zeta_2, e_2) \qquad (\zeta_1, \hat{\rho}_1) \preceq_\rho (\zeta_2, \hat{\rho}_2)}{(\zeta_1, e_1 :: \hat{\rho}_1) \preceq_\rho (\zeta_2, e_2 :: \hat{\rho}_2)}$$
$$\frac{\hat{\sigma}_1^{\leftarrow} = \hat{\sigma}_2^{\leftarrow} \qquad \forall x \in \hat{\sigma}^{\leftarrow}.\mathcal{E}[\![\hat{\sigma}_1(x)]\!](\mathcal{C}[\![\zeta_1]\!]) \subseteq \mathcal{E}[\![\hat{\sigma}_2(x)]\!](\mathcal{C}[\![\zeta_2]\!])}{(\zeta_1, \hat{\sigma}_1) \preceq_\sigma (\zeta_2, \hat{\sigma}_2)}$$
$$\frac{\mathcal{C}[\![\zeta_1]\!] \subseteq \mathcal{C}[\![\zeta_2]\!] \qquad (\zeta_1, \hat{\rho}_1) \preceq_\rho (\zeta_2, \hat{\rho}_2) \qquad (\zeta_1, \hat{\sigma}_1) \preceq_\sigma (\zeta_2, \hat{\sigma}_2)}{\hat{\chi}_1 = \langle L : i, \zeta_1, \hat{\rho}_1, \hat{\sigma}_1 \rangle \leq_{\hat{\chi}} \hat{\chi}_2 = \langle L : i, \zeta_2, \hat{\rho}_2, \hat{\sigma}_2 \rangle}$$

Figure 5.16. State-ordering relation $\leq_{\hat{\chi}}$

Abstract interpretation is implemented by applying the abstract machine to the untrusted, instrumented bytecode until a fixed point is reached. When multiple different abstract states are inferred for the same code point, the state space is pruned by computing the join of the abstract states. State lattice $(\mathcal{A}, \leq_{\hat{\chi}})$ is defined in Figure 5.16. This reduces the number of control-flows that an implementation of the abstract machine must explore.

5.5.6 Soundness

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

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

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

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

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

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

Figure 5.17. Soundness relation $\sim$

The abstract machine (defined in Section 5.5.4) is sound with respect to the concrete machine (defined in Section 5.5.2) in the sense that each inferred abstract state $\hat{\chi}$ conservatively approximates all concrete states χ that can arise at the same program point during an execution of the concrete machine on the same program. The soundness of state abstractions is formally captured in terms of a soundness relation (Cousot and Cousot, 1992) written $\sim \subseteq \mathcal{C} \times \mathcal{A}$, defined in Figure 5.17.

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

Theorem 3 (SPoX Soundness)**.** *If* $\mathcal{P}[\![pol]\!] = (\ldots, \delta)$ *and* $(\psi, \sigma), \widehat{jp} \models jp$ *holds, then* $\sigma' \in \delta(\sigma|_a, jp)$ *if and only if there exist* $\zeta' \in \hat{\mathcal{P}}[\![\theta(pol)]\!]\widehat{jp}$ *and* $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta']\!]$ *such that* $\psi''(a_0) = \sigma(a)$ *and* $\sigma''(a) = \sigma'(a)$.

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

Lemma 3. *If* $\mathcal{EP}[\![ep]\!]\psi = (\sigma, \sigma')$ *then there exists* $(\psi'', \sigma') \in \mathcal{C}[\![\widehat{\mathcal{EP}}[\![ep]\!]]\!]$ *such that* $\psi'' \sqsubseteq \psi[a_0 = \sigma(a)]$ *and* $\psi''(a_0) = \sigma(a)$.

Lemma 4. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *then* $\mathcal{PC}[\![or \ldots]\!]jp = Succ$ *if and only if there exists* $\zeta \in \widehat{\mathcal{PCD}}[\![or \ldots]\!]\widehat{jp}$ *such that* $(\psi'', \bot) \in \mathcal{C}[\![\zeta]\!]$ *and* $\psi'' \sqsubseteq \psi$.

Lemma 5. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *then* $\mathcal{PC}[\![pcc]\!]jp = Succ$ *if and only if* $(\psi'', \bot) = \mathcal{C}[\![\widehat{\mathcal{PCC}}[\![pcc]\!]\widehat{jp}]\!]$ *where* $\psi'' \sqsubseteq \psi$.

Lemma 6. *If* $(\psi, \sigma), \widehat{jp} \models jp$ *and* $f(jp) = Succ$ *then* $(f, \sigma|_a, \sigma') \in \mathcal{ES}[\![edg]\!]\psi$ *if and only if there exists* $\zeta' \in \widehat{\mathcal{ES}}[\![edg]\!]\widehat{jp}$ *such that* $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta']\!]$, $\sigma''(a) = \sigma'(a)$, *and* $\psi''(a) = \sigma(a)$.

Proof. Proofs of Lemmas 3–6 follow from a straightforward expansion of the definitions in Figures 5.11 and 5.7. □

Soundness of the abstract machine with respect to the concrete machine is proved via preservation and progress lemmas for a bisimulation of the abstract and concrete machines. The preservation lemma proves that the bisimulation preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the abstract machine anticipates all policy violations of the concrete machine. Together, these two lemmas dovetail to form an induction over arbitrary length execution sequences, proving that programs accepted by the verifier will not violate the policy.

Lemma 7 (Progress)**.** *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim \hat{\chi}$, if there exists $\hat{\chi}' = \langle L_{\hat{\chi}'} : i_{\hat{\chi}'}, \zeta', \hat{\rho}', \hat{\sigma}' \rangle \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ and $\mathcal{C}[\![\zeta']\!] \neq \emptyset$, then there exists $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$.*

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

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

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

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

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

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

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

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

We will prove $\sigma' \in \delta(\sigma|_a, jp)$ using Theorem 3. The premises of the derivation of $\chi \sim \hat{\chi}$ suffice to derive $\mu, \widehat{jp} \models jp$ by Rule (Jp-Sound). Denotation $\mathcal{C}[\![\zeta']\!]$ is non-empty by assumption; therefore we may choose $(\psi_0, \sigma_0) \in \mathcal{C}[\![\zeta']\!]$ and define $\psi'' = \psi_0[a_0 := \sigma(a)]$ and $\sigma'' = \sigma_0$. Observe that the definition of ζ' in terms of ζ_2 proves that $(\psi'', \sigma'') \in \mathcal{C}[\![\zeta_2]\!]$. Furthermore, since σ' is heretofore unconstrained, we may define $\sigma'(a) = \sigma''(a)$. Theorem 3 therefore proves that $\sigma' \in \delta(\sigma|_a, jp)$. □

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

Lemma 8. *For any expression e_0, mappings (ψ, σ), variables $\ell \in \sigma^{\leftarrow}$ and $\hat{v} \notin \sigma^{\leftarrow}$, and value x, $\mathcal{E}[\![e_0]\!](\psi, \sigma) = \mathcal{E}[\![e_0[\hat{v}/\ell]]\!](\psi[\hat{v} := \sigma(\ell)], \sigma[\ell := x])$.*

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

Lemma 9 (Preservation)**.** *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim \hat{\chi}$, for every $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$ there exists $\hat{\chi}' \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ and $\chi' \sim \hat{\chi}'$.*

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

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

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

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

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

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

Case (CIFLENEG)**:** Similar to (CIFLEPOS), omitted.

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

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

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

(B) $\mu, \hat{\sigma}(n)::\hat{\rho} \models \sigma(\ell)::\rho$ can be derived with Rule (STK-SOUND) by putting together $\mathcal{E}[\![\hat{\sigma}(\ell)]\!]\mu = \sigma(\ell)$ (from Rule (STR-SOUND)) and (2) above.

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

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

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

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

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

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

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

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

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

- If $y = \ell$ then $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]](\ell)]\!] = \mathcal{E}[\![e[\hat{v}/\ell]]\!]$. Applying Lemma 8 with $e_0 = e$ yields $\mathcal{E}[\![e[\hat{v}/\ell]]\!]\mu' = \mathcal{E}[\![e]\!]\mu$. By (2) above, and Rule (Stk-Sound), $\mathcal{E}[\![e]\!]\mu = x = \sigma[\ell := x](\ell)$.
- If $y \neq \ell$ then $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell][\ell := e[\hat{v}/\ell]](y)]\!] = \mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell](y)]\!]$. Applying Lemma 8 with $e_0 = \hat{\sigma}(y)$ yields $\mathcal{E}[\![\hat{\sigma}[\hat{v}/\ell](y)]\!]\mu' = \mathcal{E}[\![\hat{\sigma}(y)]\!]\mu$. By Rule (Str-Sound) and (3) above, $\mathcal{E}[\![\hat{\sigma}(y)]\!]\mu = \sigma(y) = \sigma[\ell := x](y)$.

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

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

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

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

(A) $\mu' \in \mathcal{C}[\![\zeta_1[\theta(a)/a] \wedge \zeta'[\theta(a)/a_0]]\!]$ is provable in two steps:

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

(B) $\mu', \hat{\rho}_r \models \rho_r$ is derivable by an induction on the height of stack $\hat{\rho}_r$ (which is equal to the height of stack ρ_r by (2) above). The base case of the induction follows trivially from Rule (EmpStk-Sound). The inductive case is derivable from Rule (Stk-Sound) provided that $\mathcal{E}[\![e]\!](\psi \uplus \psi''[\theta(a)/a_0], \sigma[a := \sigma'(a)]) = \mathcal{E}[\![e]\!](\psi, \sigma)$. To prove this, observe that e mentions neither a_0 (because by the definition of $\hat{\mathcal{P}}$, a_0 is a reserved meta-variable name that is not available to programs) nor a (because the abstract state is not directly readable by programs, and therefore cannot leak to the stack). A formal proof of both follows from an inspection of the rules in Figure 5.14.

(C) $\mu', \hat{\sigma} \models \sigma[a := \sigma'(a)]$ is derivable by Rule (Str-Sound) by deriving its two premises:

- $\sigma[a := \sigma'(a)]^{\leftarrow} = \hat{\sigma}^{\leftarrow}$ follows trivially from $a \in \sigma^{\leftarrow}$.
- $\forall x \in \sigma[a := \sigma'(a)]^{\leftarrow} . \mathcal{E}[\![\hat{\sigma}(x)]\!]\mu' = \sigma[a := \sigma'(a)](x)$ follows from (3) above, whose derivation includes a derivation of premise $\forall x \in \sigma^{\leftarrow} . \mathcal{E}[\![\hat{\sigma}(x)]\!]\mu = \sigma(x)$.

□

Theorem 4 (Soundness)**.** *Every program accepted by the abstract machine does not commit a policy violation when executed.*

Proof. By definition of abstract machine acceptance, starting from initial state $\hat{\chi}_0$ the abstract machine continually makes progress. By a trivial induction over the set of finite prefixes of this abstract transition chain, the progress and preservation lemmas prove that the concrete machine also continually makes progress from initial state χ_0. Every security-relevant event in this concrete transition chain therefore satisfies Rule (CEvent) of Figure 5.9, whose premise guarantees that the event does not violate the policy. □

5.6 Implementation and Case Studies

Our prototype verifier implementation consists of 5200 lines of Prolog and 9100 lines of Java. The Prolog code runs under 32-bit SWI-Prolog 5.10.4, which communicates with Java via the JPL interface. The Java side parses SPoX policies and Java bytecode, and compares bytecode instructions to the policy to recognize security-relevant events. The Prolog code forms the core of the verifier, and handles control-flow analysis, model-checking, and linear constraint analysis using CLP. Model-checking is only applied to code that the rewriter has marked as security-relevant. Unmarked code is subjected to a linear scan that ensures that it lacks security-relevant instructions and reified security state modifications.

We have used our prototype implementation to rewrite and then successfully verify several Java applications, discussed throughout the remainder of the section. Statistics are

Table 5.1. SPoX IRM Certifier Experimental Results

Program	Policy	File Sizes (KB) old /	new/	libs	# Classes old /	libs	Rewrite Time (s)	# Evts.	Total Verif. Time (s)	Model Check Time (s)
EJE	NoExecSaves	439/	439/	0	147/	0	6.1	1	202.8	16.3
RText		1264/	1266/	835	448/	680	52.1	7	2797.5	54.5
JSesh		1923/	1924/	20878	863/	1849	57.8	1	5488.1	196.0
vrenamer	NoExecRename	924/	927/	0	583/	0	50.1	9	1956.8	41.0
jconsole	NoUnsafeDel	35/	36/	0	33/	0	0.6	2	115.7	15.1
jWeather	NoSndsAftrRds	288/	294/	0	186/	0	12.3	46	308.2	156.7
YTDownload		279/	281/	0	148/	0	17.8	20	219.0	53.6
jfilecrypt	NoGui	303/	303/	0	164/	0	9.7	1	642.2	2.8
jknight	OnlySSH	166/	166/	4753	146/	2675	4.5	1	650.1	3.0
Multivalent	EncrpytPDF	1115/	1116/	0	559/	0	129.9	7	3567.0	26.9
tn5250j	PortRestrict	646/	646/	0	416/	0	85.4	2	2598.2	23.6
jrdesktop	SafePort	343/	343/	0	163/	0	8.3	5	483.0	17.8
JVMail	TenMails	24/	25/	0	21/	0	1.6	2	35.1	8.0
JackMail		165/	166/	369	30/	269	2.5	1	626.7	8.9
Jeti	CapLgnAttmpts	484/	484/	0	422/	0	15.3	1	524.3	8.8
ChangeDB	CapMembers	82/	83/	404	63/	286	4.3	2	995.3	12.0
projtimer	CapFileCreat	34/	34/	0	25/	0	15.3	1	56.2	6.1
xnap	NoFreeRide	1250/	1251/	0	878/	0	24.8	4	1496.2	56.4
Phex		4586/	4586/	3799	1353/	830	69.4	2	5947.0	172.7
Webgoat	NoSqlXss	429/	431/	6338	159/	3579	16.7	2	10876.0	120.0
OpenMRS	NoSQLInject	1781/	1783/	24279	932/	17185	78.7	6	2897.0	37.3
SQuirreL	SafeSQL	1788/	1789/	1003	1328/	626	140.2	1	3352.1	37.3
JVMail	LogEncrypt	25/	26/	0	22/	0	1.8	6	71.3	43.2
jvs-vfs	CheckDeletion	277/	277/	0	127/	0	4.4	2	193.9	6.3
sshwebproxy	EncryptPayload	36/	37/	389	19/	16	1.1	5	66.7	7.0

summarized in Table 5.1. All tests were performed on a Dell Studio XPS notebook computer running Windows 7 64-bit with an Intel i7-Q720M quad core processor, a Samsung PM800 solid state drive, and 4 GB of memory. A more detailed description of each application can be found in (Hamlen et al., 2011).

In Table 5.1, file sizes are expressed in three parts: the original size of the main program before rewriting, the size after rewriting, and the size of system libraries that needed to be verified (but not rewritten). Verification of system library code is required to verify the safety of control-flows that pass through them. Likewise, each cell in the classes column has two parts: the number of classes in the main program and the number of classes in the libraries.

Six of the rewritten applications listed in Table 5.1 (`vrenamer`, `jWeather`, `jrdesktop`, `Phex`, `Webgoat`, and `SQuirreL`) were initially rejected by our verifier due to a subtle security flaw that our verifier uncovered in the SPoX rewriter. For each of those cases, a bytecode

analysis revealed that the original code contained a form of generic exception handler that can potentially hijack control-flows within IRM guard code. This could cause the abstract and reified security state to become desynchronized, breaking soundness. We corrected this by manually editing the rewritten bytecode to exclude guard code from the scope of the outer exception handler. This resulted in successful verification. Our fix could be automated by in-lining inner exception handlers for guard code to protect them from interception by an outer handler.

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

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

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

Figure 5.18. `NoExecSaves` policy

Filename guards. Figure 5.18 shows a generalized SPoX policy that prevents file-creation operations from specifying a file name with an executable extension. This could be used to prevent malware propagation.

The regular expression in the `streq` predicate on Line 4 matches any string that ends in "`.exe`", "`.bat`", or a number of other disallowed file extensions. There is a very large number

of file extensions that are considered to be executable on Windows. For our implementation, we included every extension listed at (FileInfo.com, 2011).

This policy was enforced on three applications: `EJE`, a Java code editor; `RText`, a text editor; and `JSesh`, a heiroglyphics editor for use by archaeologists. After rewriting, each program halted when we tried to save a file with a prohibited extension.

Another policy that prevents deletion of policy-specified file directories (not shown) was enforced on `jconsole`. The policy monitors directory-removal system API calls for arguments that match a regular expression specifying names of protected directories.

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

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

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

Figure 5.19. `NoSendsAfterReads` policy

Event ordering. Figure 5.19 encodes a canonical information flow policy in the IRM literature that prohibits all network-send operations after a sensitive file has been read. Specifically, this policy prevents calls to `Socket.getOutput-Stream` after any `java.io.File` method call whose first parameter accesses the `windows` directory.

We enforced this policy on `jWeatherWatch`, a weather widget application, and `YouTube Downloader`, which downloads videos from `YouTube`. Neither program violated the policy, so no change in behavior occurred. However, both programs access many files and sockets, so SPoX instrumented both programs with a large number of security checks.

For `multivalent`, a document browsing utility, we enforced a policy that disallows saving a PDF document until a call has first been made to its built-in encryption method. The two-state security automaton for this policy is similar to the one in the figure.

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

Figure 5.20. `NoGui` policy

Pop-up protection. The `NoGui` policy in Figure 5.20 prevents applications from opening windows on the user's desktop. We enforced the `NoGui` policy on `jfilecrypt`, a file encrypt/decrypt application. Similar policies can be used to prohibit access to other system API methods and place constraints upon their arguments.

Port restriction. Policies such as the one in Figure 5.21 limit which remote network ports an application may access. This particular policy, which we enforced on the Telnet client `tn5250j`, restricts the port to the range from 20 to 29, inclusive. Attempting to open a connection on any port outside that range causes a policy violation.

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

Figure 5.21. `SafePort` policy

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

Resource bounds. In Section 5.3.1, we described a policy which prohibits an email client from sending more than 10 emails in a given execution, as seen in Figure 5.3. We enforced this policy on the email clients `JVMail` and `JackMail`.

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

No freeriding. Figure 5.22 specifies a more complex counting policy that prohibits freeriding in the file-sharing clients `xnap` and `Phex`. State variable `s` tracks the difference between the number of downloads and the number of uploads that the application has completed. That is, downloads increment `s`, while uploads decrement it. If the number of downloads exceeds 2 greater than the number of uploads, a policy violation occurs. This forces the software to share as much as it receives.

```
1 (state name="s")
2 (forall "i" from -10000 to 1
3   (edge name="download"
4     (nodes "s" i,i+1)
5     (call "Download.download")))
6 (forall "i" from -9999 to 2
7   (edge name="upload"
8     (nodes "s" i,i-1)
9     (call "Upload.upload")))
10 (edge name="too_many_downloads"
11   (nodes "s" 2,#)
12   (call "Download.download"))
```

Figure 5.22. `NoFreeRide` policy

```
1 (state name="s")
2 (edge name="SQL_Injection_occurred"
3   (nodes "s" 0,#)
4   (and (call "Login.login")
5        (not (argval 1 (streq "[a-zA-Z0-9]*")))))
6 (edge name="XSS_injection_occurred"
7   (nodes "s" 0,#)
8   (and (call "Employee.new")
9        (not (and (argval 2 (streq "[A-Za-z_0-9,.\-\s]*"))
10                 (argval 3 (streq "[A-Za-z_0-9,.\-\s]*"))
11                 ...
12                 (argval 16 (streq "[A-Za-z_0-9,.\-\s]*"))))))
```

Figure 5.23. `NoSqlXss` policy

Malicious SQL and XSS protection. SPoX's use of string regular expressions facilitates natural specifications of policies that protect against SQL injection and cross-site scripting attacks. One such policy is given in Figure 5.23. This figure is a simplified form of a policy that we enforced on `Webgoat`, an educational web application that is designed to be vulnerable to such attacks. The policy uses whitelisting to exclude all input characters except for those listed by the regular expressions (alphabetical, numeric, etc.).

The `XSS_injection_occurred` edge starting on Line 6 includes a large number of dynamic `argval` pointcuts—12 in the actual policy. Nevertheless, verification time remained roughly linear in the size of the rewritten code because the verifier was able to significantly prune the

search space by combining redundant constraints and control-flows during model-checking and abstract interpretation.

A similar policy was used to prevent SQL injection in `OpenMRS`, a web-based medical database system. Injection was prevented for the patient search feature. The library portion of this application is extremely large but contains no security-relevant events. Thus, the separate, non-stateful verification approach for unmarked code regions was critical for avoiding state-space explosions in this case.

We also enforced a blacklisting policy (not shown) on the database access client `SQuirreL`, preventing SQL commands which drop, alter, or rename tables or databases. Specifically, the policy identified all SQL commands matching the regular expression

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

as policy violations.

Ensuring advice execution. Most other aspectual policy languages, for example, JavaMOP (Chen and Roşu, 2005), allow explicit prescription of advice code that implements IRM guards and interventions. Such advice is typically intended to enforce some higher-level security property, though it can be difficult to prove that it enforces the actual policy that was intended. SPoX excludes trusted advice for this reason; however, untrusted advice that is not part of the policy specification may nevertheless be in-lined by rewriters to enforce SPoX policies. Chekov✓ can then be applied to verify that this advice actually executes under policy-prescribed circumstances to enforce the policy. This promotes separation of concerns, reducing the TCB so that it does not include the advice.

We simulated the use of advice by manually inserting calls to specific encrypt and log methods prior to email-send events in `JVMail`. Our intent was for each email to be encrypted, then logged, then sent, and in that order. A simplified SPoX specification for the policy is

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

Figure 5.24. `LogEncrypt` policy

given in Figure 5.24. After inserting the advice, we applied the ordering policy using the rewriter, and then used the verifier to prove that the rewritten `JVMail` application satisfies the policy.

A similar policy was applied to the Java Virtual File System (`jvs-vfs`), which prohibits deletion of files without first executing advice code that consults the user.

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

5.7 Conclusion

In this chapter, we developed Chekov✓—the first automated, model-checking-based certifier for an aspect-oriented, real-world IRM system (Hamlen and Jones, 2008). Chekov✓ uses a flexible and semantic static code analysis, and supports difficult features such as reified security state, event detection by pointcut-matching, combinations of untrusted before- and after-advice, and pointcuts that are not statically decidable. Strong formal guarantees are provided through proofs of soundness and convergence based on Cousot's abstract interpretation framework. Since Chekov✓ performs independent certification of instrumented binaries, it is flexible enough to accommodate a variety of IRM instrumentation systems, as long as they provide (untrusted) hints about reified state variables and locations of security-relevant events. Such hints are easy for typical rewriter implementations to provide, since they typically correspond to in-lined state variables and guard code, respectively.

Our focus was on presenting main design features of the verification algorithm, and an extensive practical study using a prototype implementation of the tool. Experiments revealed

at least one security vulnerability in the SPoX IRM system, indicating that automated verification is important and necessary for high assurance in these frameworks.

In future work we intend to turn our development toward improving efficiency and memory management of the tool. Much of the overhead we observed in experiments was traceable to engineering details, such as expensive context-switches between the separate parser, abstract interpreter, and model-checking modules. These tended to eclipse more interesting overheads related to the abstract interpretation and model-checking algorithms. We also intend to examine more powerful rewriter-supplied hints that express richer invariants. Such advances will provide greater flexibility for alternative IRM implementations of stateful policies.

CHAPTER 6

CERTIFYING IRM TRANSPARENCY PROPERTIES[1]

6.1 Overview

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., Yee et al., 2009; Chen and Roşu, 2005; Ligatti et al., 2005b; Schneider, 2000; Chudnov and Naumann, 2010; Erlingsson and Schneider, 1999; Hamlen et al., 2006a; Aktug and Naliuka, 2008; Li and Wang, 2010; Dam et al., 2009; Evans and Twynman, 1999; Kim et al., 2004; Bauer et al., 2005; Abadi et al., 2009, 2005; Erlingsson et al., 2006; Yu et al., 2007; Dantas and Walker, 2006; Fredrikson et al., 2012; Davis et al., 2012). 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, Figure 6.1 shows the implementation of a simple IRM in ActionScript (AS) pseudo-code. The original bytecode on the left has been instrumented 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

[1]This chapter includes previously published (Sridhar et al., 2013a,b, 2014) joint work with Richard Wartell and Kevin W. Hamlen, adapted, with permission from Elsevier.

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

Figure 6.1. Original bytecode (left) that has been rewritten (right) with an IRM that prohibits more than 100 URL navigations

more complex, but we use this simple example as a running illustration for clarity. The × marks are referenced in Section 6.4.6.)

Correct IRMs must satisfy two requirements: soundness and transparency (Hamlen et al., 2006b; Ligatti et al., 2005b). 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., Aktug and Naliuka, 2008; Bauer et al., 2005; Erlingsson, 2004; Evans and Twynman, 1999; Kim et al., 2004; Yu et al., 2007; Dantas and Walker, 2006; Hamlen and Jones, 2008), which typically leverages concepts from aspect-oriented programming (AOP) (Kiczales et al., 1997) to abstractly identify security-relevant program operations. For example, the SPoX IRM system (Hamlen and Jones, 2008) expresses safety policies encoded as aspect-oriented security automata.

Several past works have developed powerful technologies for formally machine-verifying the soundness of IRMs (Sridhar and Hamlen, 2010a; Hamlen et al., 2012; Sridhar and Hamlen, 2010b, 2011; Aktug and Naliuka, 2008; Aktug et al., 2008; Hamlen et al., 2006a; Blech et al., 2012). 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 (cf., Sridhar and Hamlen, 2011; Khoury

and Tawbi, 2012). 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 (Internet Advertising Bureau, 2013). 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 work 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 (Hamlen et al., 2012; Sridhar and Hamlen, 2010b) (presented in Chapters 4 and 5) 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 (Shapiro and Sterling, 1994) and Constraint Logic Programming (CLP) (Jaffar and Maher, 1994) 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 (Cousot and Cousot, 1992).

- The feasibility of our technique is demonstrated through a prototype implementation that targets the full AS bytecode language (please see Chapters 7 and 2 for detailed motivation for choosing ActionScript) .

The rest of the chapter is organized as follows. We begin with a summary of prior work that influences our system design in Section 6.2. Section 6.1 presents an overview of our transparency verifier, and Section 6.4 details the verification and symbolic interpretation algorithms. Section 6.5 presents implementation and results. Section 6.6 concludes.

6.2 Background and Related Work

The IRM technology has been well-established to be both powerful and adoptable: numerous IRM frameworks have been developed for Java (Chen and Roşu, 2005; Ligatti et al., 2005b; Aktug and Naliuka, 2008; Dam et al., 2009; Evans and Twynman, 1999; Kim et al., 2004; Bauer et al., 2005; Hamlen and Jones, 2008), JavaScript (Yu et al., 2007; Fredrikson et al.,

2012), .NET (Hamlen et al., 2006a), AS (Li and Wang, 2010; Hamlen and Jones, 2008), Android (Davis et al., 2012), and x86/64 native code (Yee et al., 2009; Abadi et al., 2009, 2005; Erlingsson et al., 2006) architectures. Our experiments target SPoX-IRMs (Hamlen and Jones, 2008), 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 Figure 6.1, 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.

In contrast to IRM soundness certification (see Chapters 4 and 5), 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 (Hamlen et al., 2006b; Ligatti et al., 2005b). 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 (Khoury and Tawbi, 2012). For an in-depth discussion of related work on IRM transparency, please see Chapter 8.

6.3 System Introduction

Figure 6.2, re-presented from Chapter 1, 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. The main contributions of this chapter are the transparency-verifier and invariant generator; rewriting and soundness verification for AS bytecode are presented in Chapters 3 and 4.

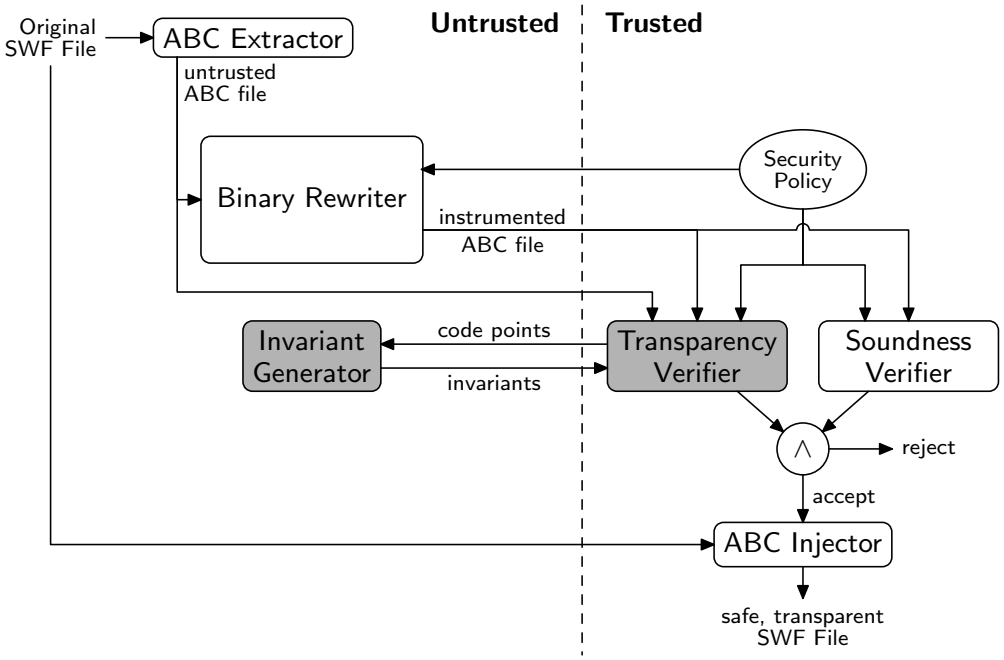


Figure 6.2. A certifying ActionScript IRM architecture

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 (Alpern and Schneider, 1986; Schneider, 2000) 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 Figure 6.1 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 (Alpern and Schneider, 1986; Schneider, 2000).

6.3.1 Defining IRM Transparency

Past work defines IRM transparency and policies in terms of traces (Hamlen et al., 2006b; Ligatti et al., 2005b):

Definition 6 (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 Figure 6.1 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 7 (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 8 (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 9 (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 Figure 6.1 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.

6.3.2 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 inter-procedural; calls flow into the bodies of callees. (Recursive and mutually recursive callees require a loop invariant, discussed in Section 6.3.3, 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 5 (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. 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 5. 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 Figure 6.1 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.

6.3.3 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 (Sridhar and Hamlen, 2010b; Hamlen et al., 2012).

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 5. 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 (Gupta, 1992) $\hat{\Gamma}$ (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 5.

6.3.4 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.

For verifying abstract parallel simulation states suggested by the untrusted invariant-generator, pruning policy-violating flows, and checking 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) (Jaffar and Maher, 1994), 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.

6.3.5 Limitations

We demonstrate experimentally (see Section 6.5) that generation of adequate invariants is tractable for typical IRMs that enforce safety properties (Schneider, 2000); 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.

6.4 Formal Approach

6.4.1 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 Figure 6.3. The implementation described in Section 6.5 supports the full AS bytecode language (subject to the limitations in Section 6.3.5).

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

Figure 6.3. Non-standard core language instructions

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 Figure 6.4. 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.

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

Figure 6.4. Semantics of the **obsevent** pseudo-instruction

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 Section 6.2.

6.4.2 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 6.5 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 (Sridhar and Hamlen, 2010b). 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, Figure 6.6 extends the concrete and abstract program states described above to

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

Figure 6.5. Concrete and abstract program states

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.

$$\zeta ::= \bigwedge_{i=1..n} t_i \quad (n \geq 1) \qquad \text{(Constraints)}$$
$$t ::= T \mid F \mid e_1 \leq e_2 \mid \hat{\tau}_1 = \hat{\tau}_2 \qquad \text{(Clauses)}$$
$$\Gamma = \langle \chi_{\mathcal{O}}, \chi_{\mathcal{R}} \rangle \qquad \text{(Concrete Interpreter States)}$$
$$\hat{\Gamma} = \langle \hat{\chi}_{\mathcal{O}}, \hat{\chi}_{\mathcal{R}}, \zeta \rangle \qquad \text{(Symbolic Interpreter States)}$$
$$\langle \mathcal{C}, \langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle, \mapsto_P^n \rangle \qquad \text{(Concrete Interpreter)}$$
$$\langle \mathcal{A}, \langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle, \rightsquigarrow_P^n \rangle \qquad \text{(Symbolic Interpreter)}$$

Figure 6.6. Concrete and abstract parallel-simulation machines

The concrete machine semantics are modeled after the AS VM 2 semantics (Adobe Systems Inc., 2007); Figure 6.7 shows the semantics of the special instructions of Figure 6.3. 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.

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

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

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

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

Figure 6.7. Concrete small-step operational semantics

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 6.8 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.

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

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

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

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

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

Figure 6.8. Abstract small-step operational semantics

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.

Algorithm 1 Verification

Input: $Cache = \{\}$, $Horizon = \{\hat{\Gamma}_0\}$ // *explored and unexplored states, respectively*
Output: *Accept* or *Reject*
1: **while** $Horizon \neq \emptyset$ **do** // *while reachable, unverified states remain*
2: $\hat{\Gamma} \leftarrow choose(Horizon)$ // *choose unexplored abstract state*
3: $S_{\hat{\Gamma}} \leftarrow VerificationSingleCodePoint(\hat{\Gamma})$ // *reduce state to subgoals*
4: **if** $S_{\hat{\Gamma}} = Reject$ **then return** *Reject*
5: $Cache \leftarrow Cache \cup \{\hat{\Gamma}\}$ // *mark state as explored*
6: $Horizon \leftarrow (Horizon \cup S_{\hat{\Gamma}}) \backslash Cache$ // *mark subgoals as unexplored*
7: **end while**
8: **return** *Accept*

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: $(\hat{\Gamma}_H, D_{\hat{\Gamma}}, n) \leftarrow InvariantGen(\hat{\Gamma})$ // *query untrusted invariant-generator*
2: $SatValue \leftarrow ModelCheck(\hat{\Gamma}, \hat{\Gamma}_H)$ // *verify that* $\hat{\Gamma}_H$ *abstracts* $\hat{\Gamma}$
3: **if** $SatValue = Reject$ **then return** *Reject*
4: $S_{\hat{\Gamma}} \leftarrow AbsIn^n(\{\hat{\Gamma}_H\}, D_{\hat{\Gamma}})$ // *interpret from* $\hat{\Gamma}_H$ *to get subgoals* $S_{\hat{\Gamma}}$
5: **if** $labels(S_{\hat{\Gamma}}) \not\subseteq D_{\hat{\Gamma}}$ **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: $SatValue \leftarrow ModelCheck(\hat{\Gamma}', \langle \hat{\chi}'_1, \hat{\chi}'_2, \zeta' \wedge (\hat{\tau}'_1 = \hat{\tau}'_2) \rangle)$
10: **if** $SatValue = Reject$ **then return** *Reject* // *verify goal* $\hat{\Gamma}'$ *is transparent*
11: **end for**
12: **return** $S_{\hat{\Gamma}}$ // *return subgoals*

6.4.3 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 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 (Gupta, 1992). 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 Figure 6.8). Next, it performs symbolic interpretation for n steps from $\hat{\Gamma}$ to confirm post-dominance of $D_{\hat{\Gamma}}$. Function $AbsIn^n(S, E)$ 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.

6.4.4 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

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: $\zeta_U \leftarrow Unify(\hat{\Gamma}, \hat{\Gamma}')$ *// check structural compatibility*
2: **if** $\zeta_U = Fail$ **then return** *Reject*
3: $SatValue \leftarrow CLP(\zeta \wedge \neg\zeta' \wedge \zeta_U)$ *// verify unsatisfiability of implication negation*
4: **if** $SatValue = False$ **then**
5: **return** *Accept*
6: **else**
7: **return** *Reject*
8: **end if**

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.

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 $\zeta \Rightarrow \zeta'$.

6.4.5 Invariant Generation

Recall from Section 6.4.3 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 (Hamlen and Jones, 2008), 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 (Hamlen et al., 2012). 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.

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**

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 Section 6.3.5, IRMs implementing non-trivial loops outside of interventions may cause this step to conservatively fail.

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

6.4.6 A Verification Example

To illustrate transparency verification, we revisit the pseudo-bytecode listing in Figure 6.1. Recall that the figure depicts an IRM that prohibits more than 100 calls to security-relevant method `NavigateToURL`. Lines with an $\times$ are those in the *Marked* set described in Section 6.4.5.

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 ($\times$) 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.

6.4.7 Proof of Verifier Correctness

Theorem 5 reduces correctness of the transparency verification algorithm to soundness of the abstract interpreter and model-checker. That is, if the verifier's abstract bisimulation of the two programs (including the interpretation rules that perform model-checking) soundly abstracts the programs' actual, concrete executions, and if the verifier accepts, then the set S described in Theorem 5 exists, and we conclude (from Theorem 5) that the IRM is transparent.

This notion of soundness can be formally defined in terms of a denotational semantics $\mathcal{D}$ of abstract states given in Figure 6.9. *Soundness relation* $\sim_{\mathbf{T}} \subseteq \mathcal{C} \times \mathcal{A}$ is then defined by

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

Following the approach of (Chang et al., 2006), soundness of the abstract interpreter is proved via two lemmas that establish preservation and progress (respectively) for a bisimulation of the abstract and concrete machines. The preservation lemma proves that the bisimulation preserves the soundness relation, while the progress lemma proves that as long as the soundness relation is preserved, the abstract interpreter covers all realizable flows. Together, these two lemmas dovetail to form an induction over arbitrary length execution

sequences. Both lemmas are proved by induction over the respective operational semantics (Figs. 6.7 and 6.8). Soundness of the model-checker follows from soundness of the Prolog CLP engine (Börger and Salamone, 1995).

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

Lemma 10 (Progress)**.** *For all $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim_{\mathbf{T}} \hat{\chi}$, if there exists $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$, then there exists $\hat{\chi}' = \langle L_{\hat{\chi}'} : i_{\hat{\chi}'}, \hat{\rho}', \hat{\sigma}', \hat{\tau}' \rangle \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$.*

Proof. The only non-trivial case is the one for **assert**. In that case, the second premise of rule CAssert (Figure 6.7) proves that the asserted event does not violate the policy. This suffices to prove that the denotation of $\hat{\mathfrak{a}}$ in the premise of rule AAssert (Figure 6.8) is well-defined, and therefore the abstract machine takes a corresponding step. □

Lemma 11 (Preservation)**.** *For every $\chi \in \mathcal{C}$ and $\hat{\chi} \in \mathcal{A}$ such that $\chi \sim_{\mathbf{T}} \hat{\chi}$, for every $\chi' \in \mathcal{C}$ such that $\chi \mapsto \chi'$ there exists $\hat{\chi}' \in \mathcal{A}$ such that $\hat{\chi} \rightsquigarrow \hat{\chi}'$ and $\chi' \sim_{\mathbf{T}} \hat{\chi}'$.*

Proof. The proof first eliminates the Abstraction rule from consideration through a structural cut elimination argument (cf., Pfenning, 1995). Two interesting cases remain: that for **api** instructions, and that for **assert** instructions. The case for **assert** is similar to Proof 6.4.7. Preservation of **api** instructions follows from relating the denotation of the **rval** expression in the premise of rule AAPI (Figure 6.8) to the system-modeling function *sys* in the premise of rule CAPI (Figure 6.7). This follows from the definition of $\mathcal{E}[\![\mathbf{rval}_m(e_1 :: \cdots :: e_n)]\!]$ in Figure 6.9. □

Theorem 6 (Soundness)**.** *Starting from the initial abstract interpreter state, $\langle \hat{\chi}_{\mathcal{O}_0}, \hat{\chi}_{\mathcal{R}_0}, \zeta_0 \rangle$, if the abstract interpreter does not reject, then, for each concrete interpreter state in each realizable flow starting from the initial concrete interpreter state $\langle \chi_{\mathcal{O}_0}, \chi_{\mathcal{R}_0} \rangle$, there exists an abstract interpreter state that soundly abstracts that concrete state.*

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

6.5 Implementation and Results

Our implementation of the transparency verification algorithm detailed in Section 6.4 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) (Jaffar et al., 1992) 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 (Sridhar et al., 2013a).

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 Section 6.4.5. 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6.9. Denotational semantics for verifier states

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 6.1. 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

Table 6.1. FlashTrack Experimental Results

Program	**Policy**	**File Size** (KB) old	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

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.

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 (FireEye, 2009). 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 6.1 resulted in no behavioral changes, as confirmed by the verifier.

6.6 Conclusion

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 present 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 Chen and Roşu, 2005) in addition to IRMs synthesized purely automatically. This requires an IRM development environment that includes program-proof co-development, such as Coq (INRIA, 2014). Such research will facilitate easier, more reliable development of customized IRMs with machine-checkable proofs of soundness and transparency.

CHAPTER 7

FLASH IN THE DARK: STUDYING THE LANDSCAPE OF ACTIONSCRIPT WEB SECURITY TRENDS AND THREATS[1]

7.1 Overview

Scattered, ad hoc information about Flash security abounds in the literature, especially in the form of news stories and "best practices" tips for Flash programmers. A systematic study of known attacks and attack classes, their potential impact, the landscape of the attack surface, and known strategies for mitigation, is badly needed for organizing this scattered information and helping both researchers and practitioners learn from past mistakes to build stronger defenses for this pervasive web technology.

Towards this goal, this chapter presents two main contributions:

1. We present a detailed taxonomy of fifteen Flash-relevant vulnerabilities and attacks. Our categorization provides a more fine-grained, informative classification specifically tuned to the *Flash* attack surface compared to the cross-section of Mitre's Common Weakness Enumeration (CWE) classification system used by the National Vulnerability Database (NVD) (National Institute of Standards and Technology (NIST), 2013) for scoring Common Vulnerability and Exposures articles (CVEs) (MITRE Corporation, 2013b).

2. Secondly, for each category, we highlight ActionScript language and Flash architecture features that make them particularly susceptible to that attack/vulnerability. We also

[1]This chapter includes joint work (Sridhar et al., 2014) with Dhiraj V. Karamchandani and Kevin W. Hamlen.

present a compilation of pertinent resources such as academic and news articles, examination of attack-type variants, high-impact real world incidents, and representative CVEs.

As security researchers, we were met with several challenges in conducting this survey on the Flash attack space. First and foremost was the challenge of sifting through and classifying a massive volume of completely disorganized information on Flash security such as thousands of news articles (including new articles that appear daily), numerous research publications, scattered information on past Flash attacks, and dispersed material on various components of the Flash ecosystem, including the Flash browser plug-in, VM, development and analysis tools, and the ActionScript language. The difficulty of taming information volume was further heightened due to the innumerable versions of various Flash software components such as the Player and the ActionScript language, each of which exhibits a multitude of features and weaknesses. CVE articles of Flash-relevant attacks tend to be too terse and coarse-grained to glean any useful technical details of an attack for educational purposes. Therefore, with our analysis, we aim to provide researchers, web developers, and security analysts a substantive sense of the Flash vulnerability and attack space, in a *consolidated* form, crucial for developing better Flash security practices and defenses, and we hope that this attempt will fuel security research towards the betterment of Flash security.

Past and future enforcement mechanisms for these attacks and vulnerabilities are beyond the scope of this work; however various prior works explore these topics (e.g., Phung et al., 2013; Li and Wang, 2010; Acker et al., 2012; Overveldt et al., 2012).

The rest of the chapter is organized as follows. Section 7.2 presents our taxonomy of Flash vulnerabilities and attacks. Section 7.3 discusses the results of our survey of the distribution of Flash-relevant scientific research. Section 7.4 concludes.

7.2 A Taxonomy of Vulnerabilities and Attacks

We begin our survey of Flash security with an in-depth taxonomy of prominent Flash-powered vulnerabilities and attacks. The taxonomy is inspired by our detailed study of 520 Flash-related CVEs and annual threat reports of major security organizations (e.g., Symantec, Cisco, Kaspersky) published over the past five years. High-impact, real-world attacks and the role of platform features unique to Flash in these attacks are highlighted.

7.2.1 Flash-based Phishing

In *phishing*, an attacker lures an unsuspecting user by masquerading as a trustworthy entity in order to steal important information such as usernames, passwords and credit card details. This is typically achieved by using various social engineering techniques to redirect the victim to a legitimate-appearing malicious site designed by the attacker.

Flash-based phishing often takes advantage of Flash's advanced animation features to create spoof sites capable of evading automated anti-phishing services. Typical automated anti-phishing services scan webpage text to identify certain suspect phrases (e.g., bank names). However, if the phishing is Flash-based, these tools are typically unable to detect these phrases or understand the scam, since they are not text-based. In fact, Flash-based features in the phishing site are even transparent to much more powerful tools such as spiders (search engines) (Nambiar, 2009).

Another common Flash-based phishing technique is to use Flash-based web advertisements for phishing; attackers abuse specific ActionScript 2 and ActionScript 3 language features only available through Flash. These features include *Flash Shared Objects*, which allow Flash applets to store client-side information similar to HTTP cookies. Shared Objects are useful for computing attack timestamps, can store up to 100KB per host domain, are persistent across sessions, and can work cross-browser (Chatterji, 2008). The features also include methods `MovieClip.getURL()` and `flash.net.navigateToURL()` (to perform actual

redirects), and `LoadVars.load()` (to make HTTP requests to the attacker's web domain, in order to keep track of the malicious redirects, or to disable any specific redirects if he or she chooses) (Ford et al., 2009). Malicious advertisements are discussed further in Section 7.2.15.

Real-world Example:

RSA SecurID Breach. One of the most shocking Flash-based phishing attacks in history was the attack on the website of RSA Security LLC (an American computer and network security company) in 2011 (Mikko, 2011; Keizer, 2011; Mills, 2011; Clark, 2011; Anthony, 2011). The attack was allegedly conducted by a nation-state, targeting Lockheed-Martin and Northrop-Grumman to steal military secrets (Mikko, 2011). These companies were using RSA's two-factor authentication product, SecurID, for network authentication.

In the attack, two phishing emails were sent to four EMC (RSA's parent company) employees. The emails carried a malicious Excel spreadsheet attachment with the subject line "2011 Recruitment plan.xls". The attachment used a zero-day exploit targeting vulnerability in the `authplay.dll` component in Flash player, creating a backdoor on the victim's machine . The attackers spoofed the emails as if they originated from a web master at `Beyond.com`, a job search and recruiting site. The email body had a deceptively innocuous simple line: "I forward this file to you for review. Please open and view it." The Excel attachment had just an "X" in its first cell. The attack used the Poison Ivy Remote Administration Tool (RAT) (F-Secure, 2013) (Trojan backdoor) on the compromised computers, using which the attackers were able to harvest users' credentials to access other RSA network machines, and copy senstive information and transfer data to their own servers (Keizer, 2011; Mills, 2011; Clark, 2011; Anthony, 2011). The speculation is that one of the credentials stolen was the unique numbers for the SecurID tokens (Mills, 2011).

The severity of the ramifications is demonstrated by the fact that RSA's only choice was to replace their SecurID tokens for their customers worldwide (Mikko, 2011) (CVE-2011-0609).

7.2.2 Flash-based Pharming and DNS Rebinding

Flash-Based Pharming

Pharming is a more sophisticated version of phishing in which an attacker redirects an unsuspecting user to an unintended website, either by changing the `hosts` file on the victim's computer, or by exploiting a vulnerability in the DNS server software.

Several ActionScript 2 and ActionScript 3 methods facilitate pharming, such as: i) `getURL()` (AS2) and `flash.net.navigateToURL()` (AS3), which can not only be used to navigate to a website, but also to directly execute JavaScript; ii) `loadMovie()` (AS2); and iii) `flash.net.navigateToURL()` in conjuction with `flash.net.URLRequest` objects (AS3) (fukami and Fuhrmannek, 2008).

DNS Rebinding

DNS rebinding allows an attacker to use a victim's browser environment (typically JavaScript or Flash) to connect to internal IP addresses in the victim's network (Striegel, 2007). This in turn can be used to leverage the victim machine for stealing information, spamming, distributed denial-of-service, and other attacks on the victim's internal network. While the *Same-origin Policy* (please see Section 7.2.4) restricts communication between objects from differing origins, the DNS rebinding attacker is able to bypass this by dynamically switching the target IP address to a host name that he or she controls (Striegel, 2007).

Real-world Example:

Massive DNS Poisoning Attack in Mexico. One of the first *drive-by-pharming* attacks in the wild occurred in Mexico, and exploited a vulnerability in *2wire* modems. The attack was conducted using spam email messages that fooled victims into believing that they received an electronic postcard from `Gusanito.com`, a popular e-card website. When the victims clicked on the link to view the cards, they were directed to a spoofed `Gusanito` page. This

spoofed page had a malicious SWF file that modified the *2wire* modem `localhost` table. Subsequently, the malicious Flash controls involved redirecting users to a fraudulent site whenever they attempt to access pages related to `Banamex.com`, a banking site (Oliveria, 2008).

7.2.3 Flash-based Drive-by-Download and Drive-by-Cache

In a Flash-based *drive-by-download* attack, the attacker compromises a website by injecting a malicious Flash binary into the site. The Flash binary loads a malicious payload (also called *shellcode*) into the address space of the browser. The code is usually a series of commands that directs the browser process to retrieve malware (usually from a different domain), write it to disk and subsequently execute it. This attack is extremely dangerous because not only the usual user warning for download is bypassed, even simply reading a webpage or viewing a document results in the malware being downloaded quietly in the background (Ducklin, 2013).

Drive-by-cache is a variation of drive-by-download attack, in which the malware is already present in the browser's cache directory and is executed, unlike in a drive-by-download attack, where the malware is downloaded and written to disk. Drive-by-cache makes infection harder to detect than drive-by-download—in a drive-by-download attack, the malware (which is often times the downloader) has to pass through the personal firewall and web filter in order to obtain the malicious payload. This makes the malware susceptible to detection. In drive-by-cache, the payload is pre-downloaded into the browser cache for easy access (Darryl, 2011a).

Drive-by-download attackers thrive on users visiting the malicious website. A typical method used is to manipulate search engines to list the site high in rankings to try and ensure that visitors will visit it (Ministry of Justice, United Kingdom, 2013). Often heap spraying (see Section 7.2.8) is used in conjunction to inject the shellcode.

Drive-by-download attacks also exploit other vulnerabilities, such as integer-overflow vulnerabilities. For example. several drive-by-download attacks have been conducted exploiting a vulnerability discovered in the `DefineSceneAndFrameLabelData` tag parsing routine in the Flash Player (Dowd, 2008). The vulnerability was caused by the routine reading an unsigned 32-bit integer and subsequently validating it using a signed comparison operator (CVE-2007-0071) (Ford et al., 2009).

Real-world Examples:

Drive-by-download Attacks on Windows and Apple Users. Drive-by-downloads were conducted on Windows (CVE-2013-0633) and Apple Macintosh (CVE-2013-0634) users, and targeted vulnerabilities through spear phishing email messages (Ducklin, 2013; Romang, 2013). The victims were from several industries, including aerospace (specifically Boeing) (Romang, 2013).

In the Windows attack, users were lured into opening a Microsoft Word document delivered as email attachments that contained malicious Flash files. As reported, one of the attachments used the 2013 IEEE Aerospace Conference schedule, and another was related to the US online payroll system company, ADP, to exploit the vulnerability in CVE-2013-0633 (Romang, 2013). The exploit targeted the ActiveX version of Flash Player on Windows (Ducklin, 2013). One of the malicious payloads (executable) was signed with a fake certificate from a South Korean company called MGAME. This certificate has been used several times in the past as part of targeted attacks (Blasco, 2013). In the Mac attack, the vulnerability in CVE-2013-0634 was exploited by tricking an Apple OS X user to open a webpage, which contained a malicious Flash file hosted on websites, targeting Flash Player in Firefox or Safari on the Macintosh platform (Ducklin, 2013; Romang, 2013).

Drive-by-cache attack on the UK Human Rights website. In April 2011, the UK Human Rights website (Ministry of Justice, United Kingdom, 2013) was hit by a Flash drive-by cache attack, in which a Flash zero-day vulnerability (CVE-2011-0611) was exploited to

infect multiple pages of the website, and install a malware which allowed the attackers to connect back to a malicious IP in Hong Kong. At the time of attack, there was no patch available for the zero-day vulnerability. The following fact demonstrates the difficulty of detection of the malicious software—VirusTotal detection was 0 out of 42 for the zero-day exploit, and 1 out of 42 for the malicious payload (Huang, 2011).

7.2.4 Same-Origin Policy Abuse

Same-origin policy allows major interactions and scripting between pages originating from the same site (determined using a combination of protocol, host and port number), and restricts inter-communication between unrelated sites. Many variations of same-origin policy exist, including ones for DOM access, XMLHttpRequest, cookies, Java, JavaScript, and Flash (Zalewski, 2011b).

The security context for Flash applets is derived from the their originating URL, and not from their embedding site. This is achieved by comparing protocol, host name and port of requestor and requested resource; for a Flash applet from a specific origin, universal access is granted to local disk contents at that origin. Flash applets can request permission for outside-domain resources using a `crossdomain.xml` policy file or the `Security.allowDomain()` directive in the `flash.system` package. For example, consider `foo.swf` in domain `X` and `bar.swf` in domain `Y`. In order for `bar.swf` to access `foo.swf`, `Y` must be added to `crossdomain.xml` policy file at `X` or `Security.allowDomain("Y")` statement must be added in `foo.swf`. Note that these methods give all Flash files in domain `Y` access to `foo.swf` (Zalewski, 2011b).

Lax development practices and subtle differences in same-origin policies have led to myriad security problems, discussed below. Despite Flash's above mentioned methods for controlling access to exposed functions (viz., `flash.system.Security.allowDomain()` and `flash.system.Security.allowInsecureDomain()`), many ad developers commonly use these features unwisely, for example by specifying wildcard ("*") that permits universal access (Elrom, 2010; Phung et al., 2013). Often, this is the case because it is difficult for developers to

determine which domains are needed by the library at the time of development. However, using the wildcard in this manner is highly imprudent because many sites that have a "*" access policy use cookies for authentication and maintain private information for logged-in users (Jang et al., 2011).

Malicious Flash applets can exploit subtle differences between Flash and JavaScript same-origin policy to bypass the Flash same-origin policy and deliver malicious JavaScript code to a third-party victim site via the `flash.external.ExternalInterface` class. Subsequently, attackers can successfully implement two-way communication with the victim third-party site, becoming fully capable of conducting attacks such as click forgery, resource theft, or flooding attacks upon victim sites (Phung et al., 2013). Please see Section 7.2.5 for more details.

More recently, attacks through malvertisements (malicious advertisements) have gained momentum (see Section 7.2.15 for more details). Most webpages today contain web ads, an important source of revenue for publishers; many contain ads derived from multiple ad networks. Malvertisements can place both the publisher and the ad network at risk by abusing Flash-JavaScript interaction to extend its privileges to DOM objects and call exposed functions from a more trustworthy ad on the same page (Phung et al., 2013).

In addition to these attack vectors, Flash's same-origin policy contains various other potentially risky leniencies. Examples include the ability to make cookie-bearing cross-domain `HTTP GET` and `POST` requests via the browser stack, through the `URLRequest` API; the ability for embedding webpages to allow various permissions via the `<OBJECT>` or `<EMBED>` parameters, such as: load external files and navigate the current browser window using `allowNetworking` attribute; interact with on-page JavaScript context `allowScriptAccess` attribute; run in full-screen mode `allowFullScreen` attribute (Zalewski, 2011b).

Case-study: Client-side Flash Proxies

An excellent example of security issues rising from subtle differences in same-origin policy between Flash and JavaScript is the concept of client-side Flash proxies (Johns and Lekies, 2011). While Flash allows cross-domain HTTP requests through the `crossdomain.xml` policy file, cross-domain HTTP requests in JavaScript are achieved via the new Cross-origin Resource Sharing (CORS) feature (enable-cors.org, 2013). CORS uses HTTP response headers to allow or deny requests unlike `crossdomain.xml` (Johns and Lekies, 2011).

While many newer browsers support CORS (e.g., mobile browsers that do not have plug-ins or browsers where plug-ins have been disabled due to security reasons), many legacy browsers do not. Therefore, developers have to create a CORS and a non-CORS version of cross-domain HTTP requests. Many developers currently use Flash proxies to aid in this process. Flash proxies are Flash applets that include a small JavaScript library that interfaces with the JavaScript on the hosting page, handling HTTP requests to cross-domain targets and responses back to the calling script (Johns and Lekies, 2011).

If the Flash applet provides a public JavaScript interface, scripts running in the context of the embedding page can abuse this interface to perform functions executed under the cross-domain origin of the Flash applet (possibly higher privileges). Note that, for this type of abuse, the exported methods for external domains have to be reported using the `allowDomain()` directive by the Flash applet. However, as mentioned above, many developers whitelist all domains using the wildcard mechanism `allowDomain("*")`, rendering such attacks feasible (Johns and Lekies, 2011).

Cross-site request forgery, session hijacking, and leakage of sensitive information attacks can be conducted via an abuse of Flash proxies and gaps in same-origin-policy to obtain trust through transitivity (Johns and Lekies, 2011). Additionally, Flash proxies and gaps in same-origin-policy can also be abused to make requests from malicious domain `A` to `B`, even without `B` providing a `crossdomain.xml` policy file (Johns and Lekies, 2011).

7.2.5 Attacks using `ExternalInterface`, `URLRequest`, and `navigateToURL`

`ExternalInterface` Abuse

Flash, through the `ExternalInterface` class in ActionScript 2 and ActionScript 3, provides two methods to interact with external containers, such as JavaScript in the embedding page: `call()` and `addCallback()`. ActionScript method `call(s, ...)` invokes JavaScript function `s` (which is passed as a string to the JavaScript VM and evaluated as JavaScript code at global scope to obtain a JavaScript function reference). ActionScript method `addCallback(s, f)` makes ActionScript function `f` callable from JavaScript under pseudonym `s` (a fresh JavaScript property name). The ActionScript method can return a value, and JavaScript receives it immediately as the return value of the call. Hence, the methods `call()` and `addCallback()` allow two-way communication between ActionScript and JavaScript (Phung et al., 2013).

The cross-language communication is extremely useful for developing rich web applications. For example, some of the uses include tracking clicks on web advertisements to gather revenue, interactive communication between the embedding HTML page and the Flash movie, including HTML buttons to start/stop the movie, random access to different chapters in the Flash movie, sending usage reporting of user interactions with the movie to Google Analytics, and data transportation between Flex chart and HTML data table (Lance, 2009).

Security for the ActionScript-JavaScript interface is provided by the `allowScriptAccess` property in the `<OBJECT>` and `<EMBED>` tags of the HTML page. In particular, The `call()` method requires the `allowScriptAccess` property to be set to one of three options: `always` (full access), `sameDomain` (same origin access), or `never` (none); same origin access is the default. For the `addCallback()` method, the default setting is that the HTML page can communicate with the ActionScript only if it originates from the same domain. To override the default, one must use the `allowDomain()` method in the `flash.system.Security` class.

Since `ExternalInterface.call` behaves very similarly to JavaScript's `eval`, it can be abused to corrupt the DOM and develop attack back channels similar to BeEF (Alcorn, 2011). `ExternalInterface.call` has also been featured in attacks that use a combination of ActionScript and JavaScript to perform cross-domain code-injection (Howard, 2012) (CVE-2011-0611).

Real-world Examples:

Cross-site scritping (XSS) and cross-site request forgery (CSRF) vulnerabilities were found in two prevalent applications SWFUpload (swfupload, 2013) and Plupload (Moxiecode Systems AB, 2013). The applications have Flash at their core, and allow developers customization of user-interface upload features such as multiple file selection, upload progress, and client-side file size checking for incorporation into sophisticated web publishing software such as Wordpress (WordPress.org, 2013).

XSS in SWFUpload. ActionScript code for SWFupload uses callbacks as the first parameter to `ExternalInterface.call()`, which in turn executes JavaScript in the current page. The value of `movieName` derived from input by the user and direct loading of the applet by passing parameters in the URL result in the XSS attack. Sites where the applet is hosted on the same domain as that of the main website are vulnerable to this kind of attack (Poole, 2012) (CVE-2012-3414,CVE-2013-2205).

CSRF in Plupload. An attacker was able to make a request to the domain where a Plupload applet was hosted, and was able to read the full response; the applet was embedded on a page using JavaScript. This was facilitated by Flash's same-origin policy. As a result, CSRF tokens and other sensitive information were disclosed on Wordpress installations. Plupload v1.5.4 was released with the CSRF issue patched—the issue had been a whitelisting of all domains by default through `Security.allowDomain('*')` (Poole, 2012) (CVE-2012-3415).

`URLRequest` and `navigateToURL` Abuse

Flash applications extensively use URL redirection (*viz.* `navigateToURL()` in the ActionScript 3 runtime and `getURL()` in ActionScript 2) and HTTP requests (via `URLRequest`) to direct user clicks to advertiser web sites, or load external resources. However, these same methods can be abused to perform highly dangerous attacks (Petkov, 2008). The `URLRequest` class allows a Flash applet to create HTTP requests using `GET` or `POST` methods; the `navigateToURL()` method takes two parameters: a `URLRequest` object containing all the information needed to perform the HTTP request, and an optional `String` that determines which frame in the current browser to open the new webpage specified in the `URLRequest`. ActionScript allows developers to pass URL values `navigateToURL()` obtained from external sources such as `FlashVars` (see Section 7.2.14), creating a vulnerability that attacks can easily manipulate to perform cross-site scripting (Adobe Systems Inc., 2013a).

Real-world Example:

Reconfiguring Home Router. A proof-of-concept example has been shown describing how these two methods can be used in conjunction to effortlessly reconfigure a well-known home router, *BT Home Hub*, distributed by a leading British telecommunications company (Petkov, 2008). The attack uses these ActionScript methods to request Universal Plug and Play (UPnP) functionality via the Simple Object Access Protocol (SOAP) (CVE-2008-1654).

7.2.6 Flash-based Cross-Site Scripting (XSS)

A *Cross-Site Scripting (XSS)* attack involves the injection of a malicious, client-side script into a vulnerable website that can be executed at the privileges of the victim page. When an unsupecting user visits the victim page, the script can exploit the user's trust to perform malicious activity.

In a classic XSS involving Flash, an attack can be conducted by passing in a malicious script through *global flash variables* (see Section 7.2.14) (Chatterji, 2008). In a *Cross-Site Flashing (XSF)* attack (MITRE Corporation, 2013a), the attacker-injected malware is a malicious *Flash applet*. Subsequently, when the applet runs on the client browser's Flash plug-in, it compromises the plug-in and allows the attacker to abuse native Flash functionality in the client browser, creating arbitrary code execution possibilities. Figure 7.2.6 presents a list of several ActionScript classes, methods and variables that pose severe risks for XSS. It is vital for developers to conduct adequate validation and sanitization of user input leaking into any of these methods to defend against XSS and XSF attacks (Rapid7 Inc., 2013; Jagdale, 2009; The OWASP Foundation, 2013).

ActionScript 2	ActionScript 3
`getURL()`	`flash.net.URLLoader.load()`
`MovieClip.loadVariables()`	`flash.net.URLStream.load()`
`TextField.htmlText`	`flash.text.htmlText`
`loadMovie()`	`flash.external.ExternalInterface.call()`
`loadMovieNum()`	`flash.external.ExternalInterface.addCallback()`
`LocalConnection.connect()`	`flash.net.LocalConnection`
`NetStream.play()`	`flash.net.NetStream.play()`
`SharedObject.getLocal()`	`flash.net.SharedObject.getLocal()`
`SharedObject.getRemote()`	`flash.net.SharedObject.getRemote()`
`XML.load()`	
`XML.sendAndLoad()`	
`Sound.loadSound()`	
`LoadVars.sendAndLoad()`	
`FScrollPane.loadScrollContent`	

Figure 7.1. Potentially dangerous ActionScript classes, methods, and properties

Real-world Example:

Flash-based XSS in Yahoo! Mail. In June 2013, a Flash XSS vulnerability was discovered (Rad, 2013) in the `IO Utility` of the Yahoo! User Interface library (Yahoo! Inc., 2013). The utility contained a Flash applet, `io.swf` which used user inputs as parameters in an `ExternalInterface.call()` without validation, rendering malicious JavaScript execution feasible in the `io.swf` container. The applet `io.swf` was hosted in the Yahoo! Mail main domain, creating an appalling vulnerability in Yahoo! Mail; users logged into Yahoo! Mail were able to access the applet at `http://us-mg5.mail.yahoo.com/neo/ued/assets/flash/io.swf`,

enabling attacks such as read access to other Yahoo Mail! users' inbox by sending a cleverly crafted URL to them (Rad, 2013).

Please see the *`SWFUpload` example* (Section 7.2.5) and the *`Gmail` example* (Section 7.2.14) for more examples of Flash-based XSS.

7.2.7 Flash-based Cross-Site Request Forgery (CSRF)

In a *Cross-Site Request Forgery (CSRF)* attack a malicious website conducts an attack on a trusted website employing a user's browser (Zeller and Felten, 2008). While CSRF attacks are often confused with the more well-known XSS, the two are strategically quite different; with CSRF, the attack is based on the exploitation of naïve web servers, which accept client requests without validation. In XSS, the trust of the user is targeted, whereas in CSRF, the cross-origin trust of the user's browser is targeted.

Flash-based CSRF attacks take advantage of several clever abuses of the language. For example, ActionScript can be used to craft spoofed HTTP headers to bypass HTTP `referer` header checking, thereby defeating a mechanism used to prevent CSRF attacks (Chatterji, 2008).

Additionally, Flash proves handy for CSRF distribution, because of the following features: (i) Flash applet's same-origin policy is determined from the origin of the Flash, not the embedding page; (ii) malicious CSRF code can be easily obfuscated and placed inside a Flash applet (see Section 7.2.10); (iii) Flash shared objects (see Section 7.2.1) enable manipulation of date and time of attack easily, and maintain hack status; and (iv) the facts that stolen data can be retrieved back to the Flash applet, and cross-domain POST can be used in place of GET, facilitate theft of large-sized data (guya, 2008).

Real-world Example:

CSRF vulnerability in IBM Tivoli Endpoint Manager Software Usage Analysis (SUA) application. The application used Flash's *Action Message Format* (AMF) (format used to

send messages between a Flash applet and a remote service) to serialize messages between web clients and the SUA server. A CSRF attack was feasible by attackers creating malicious AMF messages and deceiving an authenticated SUA user into visiting an attacker-controlled website (CVE-2013-0452) (IBM Corporation, 2013).

Another example of CSRF includes the *`Plupload` example* (Section 7.2.5).

7.2.8 Flash-based Heap Spraying

In a *Heap Spraying* attack, the attacker repetitively writes, or *sprays*, premeditated byte sequences into a large section of the victim program's heap. The shellcode is duplicated, and augmented with long sequences of *NOP (No Operation) sleds*, to provide an increased jump target to maximize the probability of success. A second exploit is required to point control flow to jump to the sprayed code.

Flash-based heap spraying often employs the `flash.utils.ByteArray` class to conduct heap spraying attacks. The `ByteArray` class, originally meant for facilitating developer interaction with binary data, unfortunately facilitates heap spraying as well, due to this very same characteristic of ease of byte-level access, including byte-level access to chunks of data, read and write access to arbitrary bytes, and read and write access to binary representation of integers, floating point numbers, and strings. Additionally, the implementation of the `ByteArray` class in the ActionScript 3 VM uses a contiguous block of memory and is expanded dynamically for storing array contents (Overveldt et al., 2012).

In the attack, two `ByteArray` instances are used—one for the shellcode, and the other as the heap spray target. The shellcode-loaded `ByteArray` is repeatedly copied into the target `ByteArray`, thereby spraying the latter with the desired malicious payload (Overveldt et al., 2012).

Real-world Examples:

Watering Hole Attack on the Council on Foreign Relations Website. On December 27, 2012, the Council on Foreign Relations (CFR) website (Council on Foreign Relations, 2013) was compromised, and subsequently was used as a medium to serve malware to its visitors (Kindlund, 2013). The final stages of the exploit used a Flash applet, `today.swf` to conduct a heap spray attack against users using Internet Explorer version 8 (CVE-2012-4792) (Kindlund, 2013, 2012).

Flash Heap Sprays without JavaScript support. While most heap sprays involving Flash borrow help from JavaScript, several instances of purely Flash-based heap spray attacks exist. One example involves a Microsoft Office Word document containing an embedded uncompressed malicious Flash file with heap spraying code. The document was a news article on iPhone batteries (CVE-2012-1535) (Blasco, 2012).

7.2.9 Flash-based JIT Spraying

Just-In-Time (JIT) spraying attacks abuse JIT compilers to defeat code control-flow protections, such as those based on *Address Space Layout Randomization* (ASLR) and *Data Execution Prevention* (DEP). ASLR reduces the reliability of attacker payloads by randomizing the locations of binary code sections in victim processes. This frustrates attackers' ability to predict valid code pointer values, and therefore invalidates many payloads containing such pointers. DEP restricts write- and execute-access to most code and data bytes, respectively, impeding malicious code-injections. However, JIT compilers typically open loopholes in both defenses by dynamically allocating writable, executable data sections for JIT-compiled code at discoverable locations.

The Adobe Flash player proves a prime target for JIT spraying as the ActionScript 3 Virtual Machine (AVM2) uses JIT-compiler enhancements to speed up execution (Adobe Systems Inc., 2007). Additionally, the Flash player implements a vast number of features

that all unfortunately aid in conducting a JIT spray attack, including a large GUI library, a JIT 3D shader language, embeddable PDF support, multiple audio and video embedding and streaming options, and of course the scripting VM (Blazakis, 2010b).

JIT spraying was first introduced, using Flash, in BlackHat D.C. 2010 (CVE-2010-1297) (Seltzer, 2010; Blazakis, 2010a,b).

Real-world Example:

Evolution of Flash-based JIT spraying, and Adobe's Continuous Reactions. Starting from the BlackHat D.C. demo by Blazakis, it has been a back-and-forth war between JIT spraying developers and Adobe (Serna, 2013). Since the first demo, Adobe has introduced various features in the Flash compiler to mitigate JIT spray vulnerabilities, including constant folding, and introduction of NOP-like instructions that break the continuity of shellcode.

An extremely sophisticated example of JIT Spraying (mitigated by Adobe in Flash version 11.8) uses *ROP (Shacham, 2007) info leak gadgets* and heap spraying to defeat prior Adobe mitigations (such as the introduction of random NOP-like instructions). The attack exploits a vulnerability in Windows 7/Internet Explorer 9 (CVE-2012-4787). Adobe's mitigation to this attack implemented a technique called *constant blinding*—XORing the value of a user-supplied integer, used later in an assignment or function argument, with a random cookie generated at runtime (Serna, 2013).

7.2.10 Obfuscation

Binary code obfuscation techniques are widely used by legitimate Flash developers to hinder reverse-engineering of Flash applets and protect intellectual property. Consequently, Commercial Off-The-Shelf (COTS) Flash obfuscators, such as SWFEncrypt (Amayeta Corporation, 2013), Kindi secureSWF (Kindi Software, 2013), DoSWF (DoSWF, 2013), and DCoM SWF Protector (DCOMSOFT, 2013), are found aplenty in the market. Unfortu-

nately, the same techniques and tools are often exploited by attackers to evade intrusion detection systems.

Malicious obfuscation techniques include name substitution, improper use of keywords, removal of debugging- and meta-information, introduction of redundant and cyclic control flows, and inclusion of unrealizable or illegal code. Well-executed obfuscation makes manual or COTS tool decompilation extremely challenging; typically, decompilation attempts using these techiques lead to invalid or unintelligible source code.

Flash-based malicious obfuscation takes advantage of various ActionScript Virtual Machine features and methods. For example, one obfuscation technique employs a combination of the `Loader.loadBytes()` method and the `DefineBinaryData` SWF tag (Adobe Systems Inc., 2007). `Loader.loadBytes()` allows dynamic loading of Flash applets; the `DefineBinaryData` tag allows arbitrary binary data to be embedded into the tagged section of a SWF file; the data becomes available to ActionScript through a `ByteArray` instance at runtime, which can be used as input to `loadBytes()` for evaluating a new Flash file. This gives attackers the potential to create a series of encrypted malicious Flash files, embedded within one another (Overveldt et al., 2012). Identifying the embedded exploits by a simple examination of the external Flash file is extremely challenging (Ford et al., 2009).

Several other SWF tags (Adobe Systems Inc., 2007) and ActionScript classes present similar powerful obfuscation-aiding mechanisms, including the `DoAction`, `ShowFrame` tags (ActionScript 2), and `SymbolClass`, `DefineBits`, `DoABC` tags (ActionScript 3) (Kovac, 2011a,b). Obfuscation techniques that adopt any of these features turn out to be deviously powerful because they house arbitrary dynamic code generation capabilities within operations that are widely used for legitimate purposes.

Additionally, ActionScript allows string identifiers of built-in ActionScript variables and methods to be stored in obfuscated form, and de-obfuscated at runtime when needed. Nearly all Flash instructions represent object member names as string values at the binary level,

making it acutely difficult to robustly determine which methods are called by even a standard, non-obfuscated Flash program. The ubiquity of obfuscation only makes this frightful situation worse.

Real-world Example:

Peeling Obfuscation Like An Onion. Several instances of real-world Flash malware use the attacker-favorite obfuscation technique of wrapping a series of malicious Flash files one into another. One such real attack example involves wrapping an obfuscated Flash 8 exploit (CVE-2007-0071) into multiple layers of a Flash 9 file (Ford et al., 2009).

An Avast! Blog article describes in detail an interesting real-world sample. The malware uses the `DefineBinaryData` and `SymbolClass` tags to load one obfuscated Flash into a byte array, and subsequently use the latter in a `DoABC` tagged code section, creating a layered Flash exploit. The main point of the article was to demonstrate the immense ease by which Flash files can be obfuscated, making obfuscators increasingly rampant in the Flash malware world (Kovac, 2011a).

7.2.11 Type Confusion Exploitation

In a *type confusion* attack, the attacker abuses a vulnerability created by a discrepancy in data type representation (Dowd et al., 2009). Type confusion attacks are particularly insidious, since they can bypass DEP and ASLR without any kind of heap or JIT spraying (Overveldt et al., 2012). This kind of vulnerability is often found in software components that bind more than one language (Dowd et al., 2009). For example, in Flash, type confusion vulnerabilities have appeared in the binding layer between ActionScript and native code. Improper error-checking by compilers while converting between fundamental and user-defined types can also cause type confusion vulnerabilities (Dowd et al., 2009).

Both the ActionScript 2 and ActionScript 3 virtual machines have been targeted for type-confusion vulnerability exploitations. The ActionScript 3 virtual machine uses data

types known as *atoms* and *type-tags* to support runtime type detection when a variable's type is not specified at the source level. Additionally, native code resulting from the JIT compilation uses native data types; therefore, when a native method is called, the result is wrapped into a type-tag for use by the VM (Overveldt et al., 2012).

This kind of *type-tag wrapping* has led to type confusion vulnerabilities. For example, in one attack, the identifier of a class `A` is changed to the same name as another class `B` in the bytecode, resulting in type confusion'. This results in calls to the `B`'s methods actually calling native code implementations of class `A`. Upon return from the native code method, the wrapped type-tag of the result depends on the types defined in `B`. The mismatch between `A`'s native code methods being called, and `B`'s return types being used creates an exploitable vulnerability, which can be used for various attacks such as leaking objects' memory addresses, reading arbitrary memory addresses, and gaining control of execution (CVE-2010-3654) (Overveldt et al., 2012).

FlashDetect (Overveldt et al., 2012) presents a very interesting technique for bypassing DEP in Flash. The technique involves discovering the address of the `VirtualProtect` function (used for changing the protection on a region of committed pages in the virtual address space of a calling process) in the Flash player DLL, through an ActionScript object (Overveldt et al., 2012).

Real-world Example:

Massive E-mail Attachment Exploits Targeting Type Confusion Vulnerabilities. Attacks were conducted exploiting type confusion vulnerability in several versions of the Flash Player. The vulnerability was exploitable upon supplying a corrupt response to *ActionScript Message Format0* `error` field, giving attackers the ability to execute arbitrary code with user privileges (Rapid7 Inc., 2012).

Several attacks were conducted—each consisted of sending victims custom crafted emails with malicious attachments. The attachments were `.doc` files that contained references to

a malicious Flash file on a remote server. The .doc files also contained a hidden malicious payload in encrypted form. The Flash file, when downloaded and played using a local vulnerable Flash Player, sprays the heap with shellcode and triggers CVE exploit. When executed, the shellcode finds the encrypted malicious payload in the original document, decrypts and executes it (Symantec Corporation, 2012).

The emails were targeted at several members of the U.S. defense industry, and contacted servers hosted in China, Korea, and the United States to acquire the necessary data to complete the exploitation (Symantec Corporation, 2012). Most recently, an emailed called "World Uyghur Congress Invitation.doc" was sent, targeting the World Uyghur Assembly (Parkour, 2012).

7.2.12 Vulnerabilities in Flash Parser and Analysis Tools

Flash application parsers, runtime analysis and decompilation tools have also been targets of attacks (fukami and Fuhrmannek, 2008).

Several attacks have been conducted due to lack of validation of ActionScript 2 *jumps*, thereby allowing code execution to jump to non-code locations in the Flash file. The ActionScript architecture confines bytecode to specific *tagged* sections in the binary (.swf) file, such as in `DoAction` or `DoInitAction` tags (Adobe Systems Inc., 2012b). The Flash VM does not verify that the jump location exists within the original tagged section, malware can therefore jump outside the section to execute bytecode elsewhere in the file. Many Flash disassemblers and decompilers such as Flasm (Kogan, 2007) and Flare (Kogan, 2005) only examine tagged sections designated for bytecode, and therefore typically miss malware that has jumped outside. In fact, much malware has used the fact that most Flash analysis tools do not examine tagged sections not designated for bytecode to hide malicious executable code (Ford et al., 2009).

Ford et al. (Ford et al., 2009) additionally point out how tag validation is a problem in itself—the Flash VM does not validate data inserted into tags. Furthermore, the Flash

VM also quietly ignores invalid tag types; invalid tag types can be created and filled with ActionScript bytecode, which can be used for attacks such as above (Ford et al., 2009).

Real-world Examples:

Improper Parsing of Various Entities. Flash Player's sloppy parsing has led to innumerable attacks, some examples are highlighted below.

Iranian Oil and Nuclear Situation Used As Bait for Defense Employees. In March 2012, targets were sent emails with an "Iran's Oil and Nuclear Situation.doc" attachment. The attachment consisted of an embedded malicious Flash applet, which when run plays a malformed MP4 file. An MP4 parsing error in the Flash Player (CVE vulnerability CVE-2012-0754) while parsing caused memory corruption and subsequently the downloading and installing of a Trojan, identified by many anti-virus products as "Graftor" or "Yayih.A". The targets of the attack were suspected to be members of the U.S. defense industry (Constantin, 2012).

Parsing TrueType Font. The Flash Player did not perform necessary validation while parsing a TrueType font. While parsing, the Flash Player was supposed to calculate the size of data to be copied based on a specific field; however, due to the lack of proper validation, an integer overflow vulnerability was created, which exposed the VM to the execution of arbitrary malicious code (Adobe Systems Inc., 2012a).

Proof-of-Concept Parsing Error Exploit. Improper validation of integer value by the parser causes vulnerability (CVE-2009-1869) that is exploited by a cleverly executed proof-of-concept heap-spray attack on Windows XP SP3 with IE7. The source code is available on Google Code (Hay, 2009).

7.2.13 JavaScript and HTML Script Injection

Most JavaScript code injection is conducted through the `ExternalInterface` class that Adobe provides for ActionScript-JavaScript communication. For more information about JavaScript code injection, please see Section 7.2.5.

Several security risks are posed by a combination of two ActionScript-HTML interactive features. The first is the ability of the Flash VM to interpret HTML tags such as the *anchor* (`<a>,</a>`) and *image* (`<img>`) tags. The second is the ability of HTML code to invoke public and static ActionScript methods through a special protocol for URLs in HTML text fields. In ActionScript 2, this is achieved through the `asfunction` protocol, which takes two arguments `function` and `parameter`, where `function` is a string identifier for an ActionScript 2 function, and `parameter` is the parameter to the former (Paola, 2007). In ActionScript 3, this is achieved through the `flash.events.TextEvent` class, by listening for click events from HTML, using the `TextEvent.LINK` property for transferring information to ActionScript, and adding an event handler in ActionScript.

Using these features, HTML code can perform cross-scripting to JavaScript through ActionScript, call an ActionScript method directly, call a particular SWF file's public functions, or call native static ActionScript directives such as `flash.system.Security.allowDomain`. While these features provide powerful convenience in ActionScript-HTML interactions, they obviously pose several security risks. For example, with the `Security.allowDomain`, it is easy to allow access to a malicious domain (Paola, 2007).

The HTML image tag allows the `src` attribute to take files with `.jpg` and `.swf` extensions. This of course presents an easy cross-site scripting vulnerability for Flash Player versions 7 or less, or if the `AllowScriptAccess` attribute is used imprudently. Additionally, if a `.swf` extension is added to a malicious JavaScript code in the `src` attribute, such as `<img src='javascript: alert(foo); //.swf'>`, the Flash plug-in will go ahead and run the Flash binary (Paola, 2007; Fukami, 2007).

Real-world Example:

Cross-Platform Attack Using Malicious JavaScript Injection and Flash. Various forums such as "windows7forums.com" and "www.macrumors.com" were attacked by a combination of malicious JavaScript and Flash scripts, originating from a malicious website "www.priceofinsurance.com". The attack was a cross-platform attack launched from multiple sites, with the JavaScript hosted on a distribution server "www.googlefreehosting.com". The JavaScript triggered the Flash file `4.swf`, which was used for data collection, and perhaps for click fraud and privacy violation (Fara, 2013).

7.2.14 Flash Parameter Injection

Users can pass values to a Flash applet from its embedding container (typically an HTML environment) into *global variables* inside the applet. In ActionScript 2, global variables are pre-pended by keywords `_root`, `_global` or `_level0`. In ActionScript 3, these are deprecated, and replaced by a single global variable `root`. Arguments to a Flash movie using ActionScript 3 can be passed into the `root.loaderInfo.parameter` object, which will contain name-value pairs of the parameters passed in from HTML. In *flash parameter injection*, an attacker abuses this facility to take control of other objects within the Flash applet, as well as full control over the embedding page's DOM model.

Three popular ways to pass values to Flash applets include:

1. *Passing arguments using direct reference.* This method references the Flash file directly and passes the arguments through the URI (this is the same as HTTP parameters using the GET method). For example, in `http://URL/myMovie.swf?a=5&b=hello`, the global variable `a` receives the value `5` and `b` receives the value `hello`. When this method is used the Flash file is not embedded in the original HTML page, but is instead embedded in a second "dummy" HTML page that is created automatically.

2. *Embedded URI.* This method passes the arguments in the URI of the embedded object in the original HTML page. For example, consider the HTML code in Figure 2.

```
1 <body>
2   <object>
3     type = "application/x−shockwave−flash"
4     data= "myMovie.swf?a=5&b=hello"
5     width = "600" height="345">
6   </object>
7 </body>
```

Figure 7.2. HTML code with injection of Flash parameters using the Embedded URI method

3. *Using the* `FlashVars` *parameter in* `<object>` *and* `<embed>` . Variables passed through `FlashVars` will go into the `_root` level of the Flash movie. For example, in the `Object` tag, `<PARAM NAME=FlashVars value="foo=bar">` will assign `bar` to `foo` on the `_level0` timeline. All variables passed in through `FlashVars` have to be strings.

An uninitialized global variable usually has whatever value was in its memory location before it was declared. However, uninitialized global variables are assumed to be `FlashVars`—variables that can be declared and passed into the Flash applet for use from the embedding container such as the embedding HTML webpage, through the `<object>` and `<embed>` tags. Irrespective of ActionScript's version, `FlashVars` can be easily abused since there are many ways to pass them into the Flash applet (Paola, 2007).

Potential unsafe operations include: (1) location of the Flash movie is retrieved through a URL parameter: `http://host/index.cqi?movie=movie.swf?globalVar=e-v-i-l`; (2) a victim is lured into clicking on a potentially malicious link such as `http://host/index.cqi?language English%26globalVar=e-v-i-l`, which happens when global flash variables are received from HTML parameters without sanitization; (3) global variable is injected into the Flash movie embedded inside the DOM: `http://host/index.htm#&globalVar=e-v-i-l` (Paola, 2007).

Real-world Example:

XSS in Gmail Based Services through Flash Parameter Injection. Users of the staple applications `Gmail` and `Google Apps` became vulnerable to full account hijacking through a Flash-based XSS vulnerability (Amit, 2010b). Internally, `Gmail` used a Flash applet called `uploaderapi2.swf` for file uploads; the applet used two user-inputs, (`apiInit` and `apiId`), as parameters to an `ExternalInterface.call()`. Proof-of-concept script injection attack was conducted before Google patched the vulnerability: the attacker was able to execute arbitrary JavaScript code in the `mail.google.com` (the `Gmail` domain) by setting `apiInit` to `eval` and `apiId` to some desired malicious code, and then enticing a user to click on a malicious link with these variables set: `https://mail.google.com/mail/uploader/uploader-api2.swf?apiInit=eval&apiId=alert(document.cookie)`.

The malicious JavaScript ran in the context of active Gmail sessions; attackers were able to fully impersonate their victims and steal information from their accounts (Amit, 2010b).

An interesting point to note is that this attack can be executed transparently in browsers such as Firefox and Google Chrome—since Flash is executed on the client side, the values of `apiInit` and `apiId` (the malicious payload) can be hidden from the server by adding the "#" sign before the query part of the URL: `https://mail.google.com/mail/uploader/uploader-api2.swf#?apiInit=eval&apiId=alert(document.cookie)`. The receiving server sees a request without parameters: `https://mail.google.com/mail/uploader/uploaderapi2.swf`, therefore concluding it to be benign; `Gmail` loads `uploaderapi2.swf` without any parameters. However, at the client side, a successful exploitation occurs since the Flash player refers to the whole URL, including the attack payload, which comes after the "#" sign (Paola, 2007; Amit, 2010b).

7.2.15 Flash-based Malvertisements

Web advertisements are an important source of revenue for webpage publishers. However, recently they are gaining traction as a tenacious vehicle for various malicious activities such as

stealing personal and banking details, corrupting data and webpages, and spreading viruses and spyware. According to the Symantec annual *Internet Security Threat Report*, malicious advertising, or *malvertising* may be the primary reason why drive-by-web attacks increased by one-third from 2011 to 2012 (Symantec Corporation, 2013). According to Cisco's *Annual Security Report*, malvertisements comprise of 16% of total web malware, mainly because a single online advertisement is typically used to fuel revenue for many webpages (Cisco Systems, Inc., 2013). According to a 2010 report by the Internet security company Dasient, there were a staggering 1.3 million malvertisements viewed daily (Danchev, 2010).

The user's trust is paramount to a malvertisement; therefore, many malvertisers target popular sites such as The New York Times, the London Stock Exchange pages, and top social networking sites such as Facebook. Malvertisers also wait until the ad has been well-circulated before triggering malicious activity, since alarming users at the start of the malicious campaign would defeat their purpose.

Websites usually contain embedded resource containers for an advertisement, with a reference to the advertisement, which is typically hosted by a third-party ad network. When a user visits the site, the webpage loads and communicates with the advertising third-party network requesting a relevant ad. For a Flash-based ad, after checking that the user's browser is Flash-enabled, the network sends relevant code back to the user's browser that needs to be inserted for the webpage to display the ad. The code in turn downloads the actual ad Flash binary file from the ad network (Ford et al., 2009).

Flash provides malvertisers a more sophisticated, powerful, and flexible platform than JavaScript/HTML for numerous reasons. Firstly, Flash allows the encoding of detailed ActionScript instructions for expressing the business/domain logic within the ad itself. Secondly, ActionScript tends to be more difficult to examine than JavaScript, providing attackers more flexibility in their attack code. ActionScript also allows malvertisements to judge time and location targets—-for example, they can defer malice until they have been deployed successfully on the ad network and target certain geographic areas using the ActionScript `Date`

class. Additionally, *Flash Shared Objects* (see Section 7.2.1) contain a timestamp attribute that can be used by malvertisements to determine whether the malicious activity has been performed on a particular machine within a particular time frame, avoiding a redundant attack to evade suspicion. The `LoadVars.load()` method can be used to send HTTP requests, and navigation methods such as `MovieClip.getUrl()` can be used to redirect to particular new sites. Finally, Flash-based malvertisements have access to a plethora of COTS obfuscators such as SWF Encrypt (Amayeta Corporation, 2013) to evade detection and defeat casual SWF decompiler tools (Ford et al., 2009; Zeltser, 2011a,b).

An interesting point to note here is that Flash-based malvertisements borrow heavily from several vulnerability and attack types that this paper presents from Section 7.2.1 to Section 7.2.14. For example, Flash-based malvertisements often use the obfuscation technique of layered-embedding of malicious Flash files as discussed in Section 7.2.10, and `getUrl()` and `navigateToURL()` methods for malicious redirection to conduct phishing, pharming and drive-by-download attacks (Section 7.2.1—Section 7.2.3). Malvertisements also use the Flash platform for `ExternalInterface` attacks (Section 7.2.5) and as a vehicle for heap or JIT spraying (Section 7.2.8, Section 7.2.9).

Real-world Examples:

DDoS Attacks on Stop Malvertising Site. In July 2011, a DDoS attack was conducted on the *Stop Malvertising Site* using a series of cleverly crafted mini Flash files. The files came from three different domains, `data-ero-advertising.com`, `flatfee.ero-advertising.com`, and `www.ero-advertising.com`. Analysis of several of the malicious Flash advertisements showed embedded `Sprite` and `MovieClip` classes with Flash malware (Kimberly, 2011).

Malvertising Attack on American Idol Website. Another interesting attack was on the American Idol fan page just prior to the competition finale of 2011. The attack consisted of an abuse of the `ExternalInterface` class to perform malicious redirection to the malicious ad network (Darryl, 2011b).

7.3 Scientic Research Survey Methodology

To better understand the scientific community's responsiveness to Flash security threats, we surveyed publications in the six highest-impact, security-themed, computer science venues, excluding venues that focus mainly on cryptography. The top six such venues ranked by Google's h5 index as of November 2013 are *ACM Symposiym on Information, Computer and Communications Security (CCS)*, *USENIX Security Symposium*, *IEEE Symposium on Security and Privacy (S&P)*, *IEEE Transactions on Dependable and Secure Computing (TDSC)*, *ACM Transactions on Information and System Security (TISSEC)*, and *European Conference on Research in Computer Security (ESORICS)*.

We read the abstracts and introductions of all publications in these venues between 2008–2012, and all publications in these venues in 2013 published till date, manually identifying those papers that are web-related, and conservatively classifying all works that make more than anecdotal reference to Flash or ActionScript as Flash-targeting.

Figure 7.3 illustrates the results. Overall, 9.6% (100/1045) of surveyed publications are devoted to web security. Of these, only $15/100 = 15\%$ papers target Flash (CCS: Magazinius et al., 2013; Acar et al., 2013; Heiderich et al., 2011, IEEE: Kolbitsch et al., 2012; Nikiforakis et al., 2013; Wang et al., 2012; Invernizzi and Comparetti, 2012; Mayer and Mitchell, 2012; Weinberg et al., 2011; Levchenko et al., 2011; Thomas et al., 2011; Chen et al., 2010; Bau et al., 2010, and USENIX Johns et al., 2013; Huang et al., 2012). IEEE S&P has the greatest percentage, devoting $10/24 = 41.7\%$ of web security publications to Flash. The remaining five venues collectively devoted only $5/76 = 6.6\%$ of web security publications to Flash. This indicates that in general the scientific community's attendance to Flash security issues has been disproportionately small relative to the role of Flash in real-world attacks.

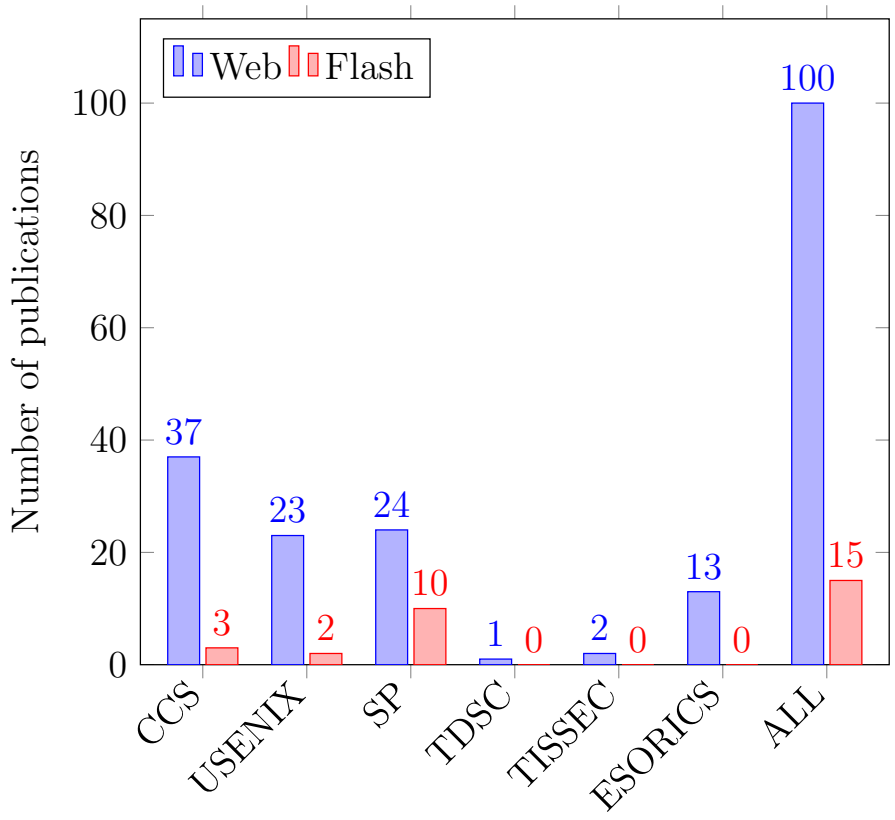


Figure 7.3. Flash presence in the top six security publication venues in 2008–2013.

7.4 Conclusion

Adobe's Flash platform has undoubtedly become a pervasive technology with a spectrum of rich features. The same flexibility and power, however, lead to a vast range of security issues. Despite the gravity of the problem, little formal study has been done on systematizing this large body of knowledge. In order to fill this void and stimulate future research, we present a systematic study of Flash security threats and trends, including an in-depth taxonomy of fifteen major Flash vulnerability and attack categories, and an examination of what makes Flash security challenges unique. The results of these analyses provide researchers, web developers, and security analysts a better sense of this important attack space, and identify the need for stronger security practices and defenses for protecting users of these technologies.

CHAPTER 8

RELATED WORK

8.1 In-lined Reference Monitoring

IRMs were first formalized in the development of the PoET/PSLang/SASI systems (Erlingsson and Schneider, 1999; Schneider, 2000), which instrument Java bytecode and GNU assembly code. Subsequently, numerous IRM frameworks have been developed for Java (Chen and Roşu, 2005; Ligatti et al., 2005b; Aktug and Naliuka, 2008; Dam et al., 2009; Evans and Twynman, 1999; Kim et al., 2004; Bauer et al., 2005; Hamlen and Jones, 2008), JS (Yu et al., 2007; Fredrikson et al., 2012), .NET (Hamlen et al., 2006a), AS (Li and Wang, 2010; Hamlen and Jones, 2008), Android (Davis et al., 2012), and x86/64 native code (Yee et al., 2009; Abadi et al., 2009, 2005; Erlingsson et al., 2006) architectures.

Most IRMs today express security policies in an AOP or AOP-like language with pointcut expressions for identifying security-relevant binary program operations, and guard code fragments (advice) that specify actions for detecting and prohibiting impending policy violations.

ConSpec (Aktug and Naliuka, 2008) restricts IRM-injected code to effect-free operations, which allows a static analysis to verify that a rewritten program does not violate the intended policy. The Java-MOP system (Chen and Roşu, 2005) allows policy-writers to choose from a sizable collection of formal policy specification languages, including LTL. Mobile (Hamlen et al., 2006a) is an In-lined Reference Monitoring system for the Microsoft .NET framework. It rewrites .NET CLI programs to satisfy a declarative policy specification by transforming the program into a well-typed Mobile program. Finally, SPoX (Hamlen and Jones, 2008) rewrites Java VM bytecode programs to satisfy declarative, Aspect-Oriented security policies.

8.2 IRM Soundness Certification

To our knowledge, ConSpec and Mobile are the only IRM systems to yet implement automatic certification. While the security policies described by these systems are declarative and therefore amenable to a more general verifier, both use a verifier tailored to a specific rewriting strategy.

8.2.1 Type-based Certification

Machine-certification of IRMs was first proposed as type-checking (Walker, 2000)—an idea that was later extended and implemented in the Mobile system (Hamlen et al., 2006a). Mobile transforms Microsoft .NET bytecode binaries into safe binaries with typing annotations in an effect-based type system. The annotations constitute a proof of safety that a type-checker can separately verify to prove that the transformed code is safe. While type-checking has the advantage of being light-weight and elegant, it comes at the expense of limited computational power. For instance, Mobile cannot enforce security policies based on data-flow; instead, it is limited to control-flow based policies. Therefore, since it does not currently support dynamic pointcut matching, it has not been applied to AOP-style IRMs to our knowledge.

8.2.2 Contract-based Certification

ConSpec (Aktug and Naliuka, 2008; Aktug et al., 2008) adopts a security-by-contract approach to AOP IRM certification. Its certifier performs a static analysis that verifies that contract-specified guard code appears at each security-relevant code point. While certification using contracts facilitates natural expression of policies as AOP programs, it has the disadvantage of including the potentially complex advice code in the TCB.

8.2.3 Per-rewriter Certification

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 (Blech et al., 2012). 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 (Chen and Roşu, 2005). Moreover, proofs of rewriter soundness are inapplicable to architectures where code-recipients must verify IRMs produced by an untrusted third party (Jones and Hamlen, 2011). For these reasons, automated IRM verification has become the dominant approach.

8.3 IRM Transparency and Program Equivalence

In contrast to IRM soundness certification, 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 (Hamlen et al., 2006b; Ligatti et al., 2005b). 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 (Khoury and Tawbi, 2012).

Chudnov and Naumann provide the first formal IRM transparency proof (Chudnov and Naumann, 2010). 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.

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., Tarditi et al., 1996), 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. (Jia et al., 2010) 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 e and e' are semantically equivalent in a context Δ of assumptions about the equivalence of terms.

8.3.1 Compiler Verification

Program equivalence-checking has been studied in the context of *translation validation*, which verifies behavior-preservation of compiler optimization phases (Pnueli et al., 1998; Necula, 2000; Program-Transformation.Org, 2013; Leroy, 2009). 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 (Zaks and Pnueli, 2008). By changing the property being checked, one can potentially verify software security properties, such as information flow policies (Barthe et al., 2004).

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.

8.3.2 Secure Protocol Analysis

Observational equivalence of abstract processes is often formally verified to ensure data confidentiality of communication protocols (Blanchet et al., 2008; Delaune et al., 2007; Cheval et al., 2011; Tiu and Dawson, 2010). In this context, observational equivalence implies that a secret exchanged by the protocol is not divulged to an attacker. This differs from our work in at least two significant respects: (1) Observational equivalence of IRMs is less strict because our goal is program feature preservation, not privacy. Thus, some visible behavior changes (e.g., timing changes) are acceptable as long as they do not impair desired program functionality. (2) We decide observational equivalence of real binary programs expressed in a real-world language (ActionScript), rather than abstract process descriptions.

8.3.3 Semantic Equivalence for Revision Tracking

Differential symbolic execution (Person et al., 2008) involves characterizing revision changes to software systems. Its goal is code documentation and comprehension rather than preclusion of observable behavior differences at the binary level.

8.4 General Model-checking

Related research on general model-checking is vast, but to our knowledge no past work has applied model-checking to the IRM verification problem. A majority of model-checking research has focused on detecting deadlock and assertion violation properties of source code.

For example, Java PathFinder (JPF) (Kisser et al., 2003) and Moonwalker (Ruys and de Brugh, 2007) verify properties of Java and .NET source programs, respectively. Both provide built-in support for deadlock detection and unhandled exceptions, but not LTL model-checking.

Other model-checking research (Ramakrishna et al., 1997; Basu and Smolka, 2007) has targeted abstract languages, such as the π-calculus (Milner, 1999) or Calculus of Communicating Systems (CCS). While important, these systems do not address certain significant practical issues, such as state space explosion, that typically arise when model-checking real software systems.

Model-checking of binary code is useful in situations where the code consumer may not have access to source code. For example, CodeSurfer/x86 and WPDS++ have been used to extract and check models for x86 binary programs (Balakrishnan et al., 2005).

8.5 ActionScript Security

The ActionScript VM includes standard object-level encapsulation as well as a sandboxing model. While useful, these protections are limited to enforcing a restricted class of low-level, coarse-grained security policies.

8.5.1 Survey Papers on ActionScript Security

Ford et al. (Ford et al., 2009) discuss many interesting attacks and vulnerabilities in the Flash attack space, such as obfuscation techniques, malvertisements, parser and decompilation tool issues. However, their survey is about four years ago, in which there have been innumerable changes to the Flash architecture and ActionScript language (Ford et al., 2009).

A more recent work earlier this year analyzed security threat reports as reported by the United States Computer Emergency Readiness Team (US-CERT). In their work, they

analyze the several company-related vulnerabilities and threats, in which Adobe was ranked third with 14% amongst top seven software giants (Baker et al., 2013).

In FlashDetect (Overveldt et al., 2012), the authors analyze language and architecture features that aid in Flash-based malware. The paper specifically covers obfuscation, heap spraying, JIT spraying, and the usage of ActionScript 3 as an exploit facilitator. However, their study is limited to this scope.

8.6 Securing Mixed Web Content

8.6.1 Behavioral Sandboxing

Our FlashJaX framework adopts a reference monitor approach, which monitors the behavior of web pages to detect and prevent attacks. There are a number of such methods in the recent literature (Agten et al., 2012; Cao et al., 2012; Phung and Desmet, 2012; Taly et al., 2011; Cutsem and Miller, 2010; Heiderich et al., 2011; Meyerovich et al., 2010; Phung et al., 2009). These works explore many subtle scenarios that arise when considering security issues in JS. For instance, JSand (Agten et al., 2012) isolates untrusted JS by loading it into a sandbox environment that can only interact with a virtual DOM. Thus, the policy definition and enforcement are implemented in a virtual DOM implementation. In contrast, FlashJaX keeps track of principals for untrusted scripts within a shadow stack in order to enforce an appropriate policy at runtime. FlashJaX can therefore handle JS script actions from AS while JSand cannot, since the latter requires full source codes of untrusted scripts. Virtual Browser (VB) (Cao et al., 2012) mediates third-party JS accesses to the browser via a virtual browser expressed in JS. The implementation is a variant of a security reference monitor. Unlike FlashJaX, VB does not support multi-principal or fine-grained policies for multi-party web applications, and does not support Flash content.

Isolating third-party content into (often invisible) iframes and providing a mechanism for cross-domain communication is an alternative approach to constraining untrusted scripts.

Some examples include Adjail (Louw et al., 2010), Webjail (Acker et al., 2011), and Subspace (Jackson and Wang, 2007). This technique is unsuitable for Flash content for performance reasons—transporting Flash content through browser-supported communication channels is prohibitively slow.

Configurable Origin Policy (COP) (Cao et al., 2013) is a recent proposal that allows web developers to associate web pages with a security principal via a configurable ID in the browser, so that web applications having a common ID are treated as same-origin even when hosted from different domains, such as `gmail.com` vs. `docs.google.com`. This clean-slate approach is a promising one in the design space of browser security. In contrast to a clean-slate approach such as COP, FlashJaX follows a design that is compatible with today's browsers and Flash interpreters. In general, since these methods only focus on the JS side, they cannot prevent attacks exploiting JS-AS interactions.

Similarly, there are several protection methods focusing on privacy and behavioral targeting, for example, Privads (Guha et al., 2009), Adnostic (Toubiana et al., 2010), and RePriv (Fredrikson and Livshits, 2011), which address user privacy issues from behavioral targeting. These rely on specialized, in-browser systems that support contextual placement of ads while preventing behavioral profiling of users. In contrast, our work mainly focuses on a different, publisher-centric problem of protecting confidentiality and integrity of publisher and user-owned content. Our work is also aimed at providing compatibility with existing ad networks and browsers.

8.6.2 Restricting Content Languages

There have been a number of works in the area of JS analysis that restrict content from untrusted sources to provide security protections (Facebook Developers, 2010; Guarnieri and Livshits, 2009; Maffeis and Taly, 2009; Maffeis et al., 2009b; Finifter et al., 2010; Politz et al., 2011). These works focus on limiting the JS language features that untrusted scripts may

use. Only those language features that are statically deterministic and amenable to analysis are allowed. Since these methods restrict content at a language level, they do not impose the runtime penalty of reference monitors. In the cases of FBJS (Facebook Developers, 2010) and ADsafe (Crockford, 2007), untrusted scripts are confined to an access-controlled DOM interface, which incurs some overhead but affords additional control.

The disadvantage of a restricted JS subset is that ads authored by many advertisers are unlikely to conform to this subset, and will therefore require re-development. In contrast, FlashJaX neither imposes the burden of new languages nor places restrictions on JS language features used in ad scripts. The only effort required from a publisher that incorporates FlashJaX is to specify policies that reflect site security practices.

8.6.3 Code Transformation Approaches

Many recent works have transformed untrusted JS code to interpose runtime policy enforcement checks (Reis et al., 2006; Yu et al., 2007; Kikuchi et al., 2008; Google Caja, 2007; Microsoft Live Labs, 2009; Li et al., 2011). These works cover many diverse attack vectors by which third-party content may subvert the checks. Since these works are aimed at general JS security, they do not consider the security of the JS-AS interface and attacks that target this interface.

8.6.4 Browser-enforced Protection

A modified browser can be instructed to enforce security policies, as illustrated by BEEP (Jim et al., 2007), CoreScript (Yu et al., 2007), End-to-End Web Application Security (Erlingsson et al., 2007), Content Security Policies (Stamm et al., 2010), and ConScript (Meyerovich and Livshits, Meyerovich and Livshits). Other works, such as AdSentry (Dong et al., 2011), JCShadow (Patil et al., 2011), ESCUDO (Jayaraman et al., 2010), and Tahoma (Cox et al.,

2006), have taken this approach to prevent attacks by untrusted content. The main advantage of this approach is that it can enforce fine-grained policies with low overhead. However, the primary drawback is that today's browsers do not agree on a standard for publisher-browser collaboration, leaving a large void in the near-term for protecting users from malicious third-party content.

8.6.5 Safety of ActionScript content

Jang et al. (Jang et al., 2011) point out the pervasive nature of misconfigured AS content, particularly with reference to cross-domain policies. Ford et al. (Ford et al., 2009) describe a malware identification approach for Flash advertisements.

The work that is closest to ours is FIRM (Li and Wang, 2010), which uses an IRM approach for prevention of Flash-related attacks. FIRM is strictly limited to AS mediation, whereas FlashJaX tackles a much broader class, that of mixed AS-JS content. As a result, our monitor is able to address a much broader class of attack vectors that target JavaScript as well as ActionScript (as discussed in Section 2.2), especially those that exploit the interface boundary. Since FIRM focuses purely on AS-side monitoring, it adopts a less conservative threat model that assumes that some parts of the JS namespace can be read-protected from adversaries. This relaxed model admits a capability-based approach, which FIRM implements using secret tokens that are maintained by the reference monitor. In contrast, FlashJaX's threat model acknowledges that protection of secrets in a JS environment is hard. There are many different ways through which an attacker can get *read* access to the JS namespace (cf., Maffeis et al., 2009a) in order to gain access to secret tokens. We therefore conservatively assume that adversaries may have the ability to read the complete JS namespace, and therefore developed a more robust approach whose security is argued in Section 2.5.

CHAPTER 9

CONCLUSIONS[1]

Binary instrumentation via IRMs is now well-established as a powerful and versatile technology, providing a flexible, powerful, yet efficient security enforcement for many platforms. Independent certification of the instrumented code helps minimize and stabilize the trusted computing base, combining the power and flexibility of the runtime monitoring with strong formal guarantees of static analysis.

This dissertation builds several tools and methodologies for providing case-by-case certification of instrumented code via model-checking. Certification is achieved for both soundness and transparency properties, thus establishing both policy-adherence of the instrumented code, and behavior-preservation over the instrumentation process. The certification task is simplified by leveraging untrusted information from the rewriting process, rendering case-by-case certification significantly lighter-weight than certification of arbitrary code.

The dissertation presents the results of developing model-checking IRM frameworks for several platforms, with several compelling applications to important classes of challenging contemporary security issues, such as cross-domain web advertisement security. The technologies presented are backed by rigorous formalisms and proofs. In the following sections, we discuss in-depth conclusions and future-work pertaining to each major work presented in the dissertation, and conclude by presenting two interesting future research directions in IRM certification research.

[1]This chapter includes previously published (Sridhar and Hamlen, 2011) joint work with Kevin W. Hamlen.

9.1 FlashJaX: Securing Mixed JavaScript/ActionScript Multi-party Web Content

In Chapter 2, we presented FlashJaX, a solution for enforcing security policies on third-party mixed JS/AS web content using an IRM approach. FlashJaX allows publishers to define and enforce fine-grained, multi-principal access policies on JS-AS third party content and runtime-generated code. Moreover, it can be easily deployed in practice without requiring browser modification. Experiments show that FlashJaX is effective in preventing attacks related to AS-JS communication, and its lightweight IRM approach exhibits low overhead for mediations. It is also compatible with advertisements from leading ad networks.

9.2 ActionScript Bytecode Verification Using Co-logic Programming

In Chapter 3, we described preliminary work toward developing a security policy verifier for Adobe ActionScript bytecode programs. Our verifier consists of an interpreter for ActionScript bytecode and an LTL model-checker, both written in Prolog extended with tabling and coinduction. Experiments demonstrated that our prototype can efficiently verify simple but interesting history-based policies for small ActionScript programs.

Verifiers are typically part of a secure system's trusted computing base. It is therefore important that the verifier itself be amenable to formal verification. The declarative nature of co-LP Prolog yields several significant advantages in this regard. First, our verifier code base is very concise—the parser is 2 kSLOC while the AVM2 semantics and the model-checker are each 1 kSLOC. Second, our experiences indicate that it is straightforward to encode semantic rules, such as those from the AVM2 Specification (Adobe Systems Inc., 2007), as a relation between program states (see Figure 3.2). Finally, implementing the expansion rules for LTL in co-LP avoids a great deal of tedious and error-prone implementation work by relying upon the well-defined termination semantics of tabling and coinduction in Prolog (see Figure 3.3).

In future work, we plan to generate explicit policy-adherence proofs. This involves enhancing current implementations of coinductive Prolog (Gupta et al., 2007) to support enumeration of all coinductive proofs of a goal. After completing these efforts, generating these proofs should require minimal changes to our system.

There is a large body of existing work on optimizing LTL formulae used in model checking (e.g., Daniele et al., 1999; Etessami and Holzmann, 2000; Sebastiani and Tonetta, 2003). We also intend to explore the use of other temporal logics, especially Computation Tree Logic (CTL) and the μ-calculus (Jr. et al., 1999), for the specification of security policies. The method mentioned in (Dillon and Ramakrishna, 1996) suggests a means of modularizing the temporal logic engine out of the model-checker, allowing the same model-checking system to be used for multiple temporal logics by changing which temporal logic engine is used. Examining how to use several of these engines at once seems a promising direction for our tool as well.

9.3 A Prototype Model-Checking IRM System for ActionScript Bytecode

Chapter 4 discussed preliminary work on certifying IRMs through model-checking. Our technique derives a state abstraction lattice from a security automaton to facilitate precise abstract interpretation of IRM code. Formal proofs of soundness and convergence guarantee reliability and tractability of the verification process.

While the section covers proof sketches for the interesting cases of progress and preservation, and soundness and convergence, we have also recently completed fully machine-verified proofs for all of the above using the Coq proof assistant (INRIA, 2014).

We demonstrated the feasibility of our technique by enforcing a URL anti-redirection policy for ActionScript bytecode programs. We also demonstrated the elegance of using Prolog for implementing a certifying IRM system. Using Prolog resulted in faster development and simpler implementation due to code reusability from reversible predicates and

succinct program specifications from declarative programming. This resulted in a smaller trusted computing base for the overall system.

While our algorithm successfully verifies an important class of IRM implementations involving reified security state, it does not support all IRM rewriting strategies. Reified security state that is per-object (Hamlen et al., 2006a) instead of global, or that is updated by the IRM before the actual security state changes at runtime rather than after, are two examples of IRM strategies not supported by our model. Subsequently, Chapter 5 generalized our approach to cover these cases.

One area of future work remaining is augmenting our system with support for recursion and mutual recursion, which is currently not handled by our implementation.

9.4 Full-Scale Certification for the SPoX Java IRM System

In Chapter 5, we developed Chekov✓—the first automated, model-checking-based certifier for an aspect-oriented, real-world IRM system (Hamlen and Jones, 2008). Chekov✓ uses a flexible and semantic static code analysis, and supports difficult features such as reified security state, event detection by pointcut-matching, combinations of untrusted before- and after-advice, and pointcuts that are not statically decidable. Strong formal guarantees are provided through proofs of soundness and convergence based on Cousot's abstract interpretation framework. Since Chekov✓ performs independent certification of instrumented binaries, it is flexible enough to accommodate a variety of IRM instrumentation systems, as long as they provide (untrusted) hints about reified state variables and locations of security-relevant events. Such hints are easy for typical rewriter implementations to provide, since they typically correspond to in-lined state variables and guard code, respectively.

Our focus was on presenting main design features of the verification algorithm, and an extensive practical study using a prototype implementation of the tool. Experiments revealed

at least one security vulnerability in the SPoX IRM system, indicating that automated verification is important and necessary for high assurance in these frameworks.

In future work we intend to turn our development toward improving efficiency and memory management of the tool. Much of the overhead we observed in experiments was traceable to engineering details, such as expensive context-switches between the separate parser, abstract interpreter, and model-checking modules. These tended to eclipse more interesting overheads related to the abstract interpretation and model-checking algorithms. We also intend to examine more powerful rewriter-supplied hints that express richer invariants. Such advances will provide greater flexibility for alternative IRM implementations of stateful policies.

Finally, we also plan to complete fully machine-verified proofs for all the proofs in this chapter using the Coq proof assistant (INRIA, 2014).

9.5 Certifying IRM Transparency Properties

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, in Chapter 6, 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 (Chen and Roşu, 2005)) in addition to IRMs synthesized purely automatically. This requires an IRM development environment that includes program-proof co-development, such as Coq (INRIA, 2014). Such research will facilitate easier, more reliable development of customized IRMs with machine-checkable proofs of soundness and transparency.

Finally, we also plan to complete fully machine-verified proofs for all the proofs in this chapter using the Coq proof assistant (INRIA, 2014).

9.6 Flash in the Dark: Studying the Landscape of ActionScript Web Security Trends and Threats

Adobe's Flash platform has undoubtedly become a pervasive technology with a spectrum of rich features. The same flexibility and power, however, lead to a vast range of security issues. Despite the gravity of the problem, little formal study has been done on systematizing this large body of knowledge. In order to fill this void and stimulate future research, in Chapter 7, we presented a systematic study of Flash security threats and trends, including an in-depth taxonomy of fifteen major Flash vulnerability and attack categories, and an examination of what makes Flash security challenges unique. The results of these analyses provide researchers, web developers, and security analysts a better sense of this important attack space, and identify the need for stronger security practices and defenses for protecting users of these technologies.

9.7 Future Directions in IRM Certification

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. However, several practical problems are faced by developers of certifying IRM systems today. We present two of these here.

9.7.1 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 (International, 1999) (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 (Cova et al., 2010; Egele et al., 2009; Richards et al., 2010), 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 (Richards et al., 2010). 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 (Anderson and Giannini, 2005; Jang and Choe, 2009; Thiemann, 2005b; Anderson and Drossopoulou, 2003), 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 (Guha et al., 2009; Jensen et al., 2009).

The hybrid static-dynamic monitoring approach of certifying IRMs 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. (Christensen et al., 2003) 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 (Thiemann, 2005a) 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 (Earley, 1983). Minamide (Minamide, 2005) presents an analysis of string expressions based on a prior Java string analyzer (Christensen et al., 2003), but instead of transforming the extracted grammar into a regular grammar, they use trancuders to define a context-free grammar. Abstract parsing (Doh et al., 2009) 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.

Recall Definition 1, the definition of $\mathcal{P}$-verifiability, from Chapter 1. Based on Definition 1, 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 IRMs. 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.

9.7.2 Concurrency

A second major challenge area for certifying IRMs 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 IRMs 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 (cf., Bodden and Havelund, 2008).

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 dissertation, 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., Havelund, 2000; Artho et al., 2003; Bensalem and Havelund, 2005). There has also been some work done on dynamic detection of race conditions using aspects (cf., Bodden and Havelund, 2008). 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 IRMs, those that enforce purely non-temporal policies (e.g., McCamant and Morrisett, 2006; Yee et al., 2009) can safely ignore the concurrency issue because they need not maintain a *history* of security-relevant events. ConSpec (Aktug and Naliuka, 2008) leaves concurrency

to future work, while Mobile (Hamlen et al., 2006a) 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 IRMs 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 IRMs? 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 potentially 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 (Schneider, 2000). 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 (Milner, 1999). The other involves more practical tools such as aspect-oriented temporal assertions (Stolz and Bodden, 2006), tracematching (Allan et al., 2005), and their applications for race detection (Bodden and Havelund, 2008). Stolz et al. (Stolz and Bodden, 2006) present a runtime verification framework for Java programs, where properties can be specified in LTL over AspectJ pointcuts. The work on AspectJ tracematching (Allan et al., 2005) enhanced with Racer (Bodden and Havelund, 2008) 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., Bensalem and Havelund, 2005; Havelund, 2000; Artho et al., 2003). 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.

REFERENCES

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pp. 340–353.

Abadi, M., M. Budiu, Ú. Erlingsson, and J. Ligatti (2009). Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and Systems Security (TISSEC) 13*(1), 1–40.

Acar, G., M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel (2013). FPDetective: Dusting the web for fingerprinters. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 1129–1140.

Acker, S. V., N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens (2012). FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 12–13.

Acker, S. V., P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen (2011). WebJail: Least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pp. 307–316.

Adobe Systems Inc. (2007). ActionScript virtual machine 2 overview. http://www.adobe.com/devnet/actionscript/articles/avm2overview.pdf.

Adobe Systems Inc. (2012a). Adobe Flash Player TrueType font parsing integer overflow vulnerability. http://www.verisigninc.com/en_US/products-and-services/network-intelligence-availability/idefense/public-vulnerability-reports/articles/index.xhtml?id=1001.

Adobe Systems Inc. (2012b). Swf file format specification, version 19. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/swf/pdf/swf-file-format-spec.pdf.

Adobe Systems Inc. (2013a). ActionScript 3.0 reference for the Adobe Flash platform, package `flash.net`. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/package.html#navigateToURL().

Adobe Systems Inc. (2013b). ActionScript technology center. http://www.adobe.com/devnet/actionscript.html.

Adobe Systems Inc. (2013c). Adobe Flash runtimes statistics. http://www.adobe.com/products/flashruntimes/statistics.edu.html.

Agten, P., S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens (2012). JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pp. 1–10.

Aktug, I., M. Dam, and D. Gurov (2008). Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods (FM)*, pp. 262–277.

Aktug, I. and K. Naliuka (2008). ConSpec – a formal language for policy specification. *Science Computer Programming (SCP) 74*, 2–12.

Alcorn, W. (2011). Browser exploitation framework BeEF (2011) software. http://code.google.com/p/beef/.

Allan, C., P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble (2005). Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*.

Alpern, B. and F. B. Schneider (1986). Recognizing safety and liveness. *Distributed Computing 2*, 117–126.

Amayeta Corporation (2013). SWF ENCRYPT 7.0. http://www.amayeta.com/software/swfencrypt/,Amayeta.

Amit, Y. (2010a, Mar.). Cross-site scripting through Flash in Gmail based services. http://blog.watchfire.com/wfblog/2010/03/cross-site-scripting-through-flash-in-gmail-based-services.html.

Amit, Y. (2010b). Cross-site scripting through flash in Gmail based services. http://blog.watchfire.com/wfblog/2010/03/cross-site-scripting-through-flash-in-gmail-based-services.html.

Anderson, C. and S. Drossopoulou (2003). BabyJ: from object based to class based programming via types. *Electronic Notes in Theoretical Computer Science 82*(7), 53–81.

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

Anthony, S. (2011). Security firm RSA attacked using Excel-Flash one-two sucker punch. http://downloadsquad.switched.com/2011/04/06/security-firm-rsa-attacked-using-excel-flash-one-two-sucker-punc/.

Artho, C., K. Havelund, and A. Biere (2003). High-level data races. *Journal on Software Testing, Verification and Reliability (STVR) 13*(4), 207–227.

Baier, C. and J.-P. Katoen (2008). *Principles of Model Checking (Representation and Mind Series)*. The MIT Press.

Baker, Y. S., R. Agrawal, and S. Bhattacharya (2013). Analyzing security threats as reported by the united states computer emergency readiness team (US-CERT). In *Proceedings of the 11th IEEE Intelligence and Security Informatics Conference (ISI)*, pp. 10–12.

Balakrishnan, G., T. W. Reps, N. Kidd, A. Lal, J. Lim, D. Melski, R. Gruian, S. H. Yong, C.-H. Chen, and T. Teitelbaum (2005). Model checking x86 executables with CodeSurfer/x86 and WPDS++. In *Proceedings of the 17th International Conference on Computer-Aided Verification (CAV)*, pp. 158–163.

Bansal, A. (2007). *Next Generation Logic Programming Systems*. Ph. D. thesis, The University of Texas at Dallas, Dallas, Texas.

Barthe, G., P. R. D'Argenio, and T. Rezk (2004). Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSF)*, pp. 100–114.

Basu, S. and S. A. Smolka (2007). Model checking the Java metalocking algorithm. *ACM Transactions on Software Engineering and Methodology 16*(3).

Bau, J., E. Bursztein, D. Gupta, and J. Mitchell (2010). State of the art: Automated black-box web application vulnerability testing. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*, pp. 332–345.

Bauer, L., J. Ligatti, and D. Walker (2005). Composing security policies with Polymer. In *Proceedings of the 26th ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 305–314.

Bensalem, S. and K. Havelund (2005). Dynamic deadlock analysis of multi-threaded programs. In *Proceedings of the 1st Haifa International Conference on Hardware and Software Verification and Testing (HVC)*, pp. 208–223.

Blanchet, B., M. Abadi, and C. Fournet (2008). Automated verification of selected equivalences for security protocols. *Journal of Logic and Algebraic Programming 75*(1), 3–51.

Blasco, J. (2012). CVE-2012-1535: Adobe Flash being exploited in the wild. http://www.alienvault.com/open-threat-exchange/blog/cve-2012-1535-adobe-flash-being-exploited-in-the-wild.

Blasco, J. (2013). Adobe patches two vulnerabilities being exploited in the wild. http://www.alienvault.com/open-threat-exchange/blog/adobe-patches-two-vulnerabilities-being-exploited-in-the-wild.

Blazakis, D. (2010a). BHDC2010 - JITSpray demo #1. http://www.youtube.com/watch?v=HJuBpciJ3Ao.

Blazakis, D. (2010b). Interpreter exploitation: Pointer inference and JIT spraying.

Blech, J. O., Y. Falcone, and K. Becker (2012). Towards certified runtime verification. In *Proceedings of the 10th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering (SEFM)*, pp. 494–509.

Bodden, E. and K. Havelund (2008). Racer: Effective race detection using AspectJ. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 155–166.

Börger, E. and R. F. Salamone (1995). CLAM specification for provably correct compilation of CLP(R) programs. In E. Börger (Ed.), *Specification and Validation Methods*, pp. 96–130. Oxford University Press.

Cao, Y., Z. Li, V. Rastogi, Y. Chen, and X. Wen (2012). Virtual Browser: A virtualized browser to sandbox third-party JavaScripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 8–9.

Cao, Y., V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk (2013). Redefining web browser principals with a configurable origin policy. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12.

Chang, B.-Y. E., A. Chlipala, and G. C. Necula (2006). A framework for certified program analysis and its applications to mobile-code safety. In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pp. 174–189.

Chatterji, S. (2008). Flash Security and Advanced CSRF.

Chen, F. and G. Roşu (2005). Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 546–550.

Chen, S., R. Wang, X. Wang, and K. Zhang (2010). Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*, pp. 191–206.

Chen, W. and D. S. Warren (1996). Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43*, 43–1.

Cheval, V., H. Comon-Lundh, and S. Delaune (2011). Trace equivalence decision: Negative tests and non-determinism. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, pp. 321–330.

Christensen, A. S., A. Møller, and M. I. Schwartzbach (2003, June). Precise analysis of string expressions. In *Proceedings of the 10th International Static Analysis Symposium (SAS)*, Volume 2694 of *LNCS*. Springer-Verlag.

Chudnov, A. and D. A. Naumann (2010). Information flow monitor inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pp. 200–214.

Cisco Systems, Inc. (2013). 2013 Cisco annual security report.

Clark, J. (2011). RSA hack targeted Flash vulnerability. http://www.zdnet.com/rsa-hack-targeted-flash-vulnerability-4010022143/.

comScore (2012, April). comScore media metrix ranks top 50 U.S. web properties for April 2012. http://www.comscore.com/Press_Events/Press_Releases/2012/5/comScore_Media_Metrix_Ranks_Top_50_U.S._Web_Properties_for_April_2012.

Constantin, L. (2012). Iranian nuclear program used as lure in Flash-based targeted attacks. http://www.csoonline.com/article/701565/iranian-nuclear-program-used-as-lure-in-flash-based-targeted-attacks.

Coquand, T. and G. Huet (1988). The calculus of constructions. *Information and Computation 76*(2-3), 95–120.

Council on Foreign Relations (2013). Council on foreign relations. http://www.cfr.org/.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 234–252.

Cousot, P. and R. Cousot (1992). Abstract interpretation frameworks. *Journal Logic and Computation 2*(4), 511–547.

Cova, M., C. Kruegel, and G. Vigna (2010). Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*.

Cox, R. S., S. D. Gribble, H. M. Levy, and J. G. Hansen (2006). A safety-oriented platform for web applications. In *Proceedings of the 27th IEEE Symposium on Security & Privacy (S&P)*, pp. 350–364.

Crockford, D. (2007). ADsafe. http://www.adsafe.org.

Cutsem, T. V. and M. S. Miller (2010). Proxies: Design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th Dynamic Languages Symposium (DLS)*, pp. 59–72.

Dam, M., B. Jacobs, A. Lundblad, and F. Piessens (2009). Security monitor inlining for multithreated Java. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pp. 546–569.

Danchev, D. (2010). Research: 1.3 million malicious ads viewed daily. http://www.zdnet.com/blog/security/research-1-3-million-malicious-ads-viewed-daily/6466,ZDNet.

Daniele, M., F. Guinchiglia, and M. Y. Vardi (1999). Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV)*, Volume 1633 of *LNCS*, pp. 249–260. Springer-Verlag.

Dantas, D. S. and D. Walker (2006). Harmless advice. In *Proceedings of the 34th ACM Symposium on Principles Of Programming Languages (POPL)*, pp. 383–396.

Dantas, D. S., D. Walker, G. Washburn, and S. Weirich (2008). AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems 30*(3), 1–60.

Darryl (2011a). Drive-by-cache: Payload hunting. http://www.kahusecurity.com/2011/drive-by-cache-payload-hunting/.

Darryl (2011b). Flash used in Idol malvertisement. http://www.kahusecurity.com/2011/flash-used-in-idol-malvertisement/,KahuSecurity.

Davis, B., B. Sanders, A. Khodaverdian, and H. Chen (2012). I-ARM-Droid: A rewriting framework for in-app reference monitors for android applications. In *Proceedings of the IEEE Workshop on Mobile Security Technologies (MoST)*.

DCOMSOFT (2013). DCoM SWF protector. http://www.dcomsoft.com/.

Delaune, S., S. Kremer, and M. D. Ryan (2007). Symbolic bisimulation for the applied pi-calculus. In *Proceedings of the 27th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pp. 133–145.

Denis, F., A. Lemay, and A. Terlutte (2001). Residual finite state automata. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pp. 144–157.

DeVries, B. W., G. Gupta, K. W. Hamlen, S. Moore, and M. Sridhar (2009). ActionScript bytecode verification with co-logic programming. In *Proceedings of the 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 9–15.

Dillon, L. K. and Y. S. Ramakrishna (1996). Generating oracles from your favorite temporal logic specifications. *ACM SIGSOFT Software Engineering Notes 21*(6), 106–117.

Doh, K.-G., H. Kim, and D. A. Schmidt (2009). Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, pp. 256–272.

Dong, X., M. Tran, Z. Liang, and X. Jiang (2011). AdSentry: Comprehensive and flexible confinement of JavaScript-based advertisements. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pp. 297–306.

DoSWF (2013). DoSWF-Professional Flash SWF Encryptor. http://www.doswf.org/.

Dowd, M. (2008). Application-specific attacks: Leveraging the ActionScript virtual machine.

Dowd, M., R. Smith, and D. Dewey (2009). Attacking interoperability. In *Black Hat USA*. http://www.hustlelabs.com/stuff/bh2009_dowd_smith_dewey.pdf.

Ducklin, P. (2013). Adobe patches Flash - heads off in-the-wild attacks against Windows and Apple users. http://nakedsecurity.sophos.com/2013/02/08/adobe-patches-flash-heads-off-attacks-on-windows-and-apple/.

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

ECMA International (2011, June). *ECMAScript Language Specification (ECMA-262)* (5.1 ed.). Geneva, Switzerland: ECMA.

Egele, M., P. Wurzinger, C. Kruegel, and E. Kirda (2009). Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 88–106.

Elrom, E. (2010, July). Top security threats to Flash/Flex applications and how to avoid them. http://www.slideshare.net/eladnyc/top-security-threats-to-flashflex-applications-and-how-to-avoid-them-4873308.

enable-cors.org (2013). Enable cross-origin resource sharing. http://enable-cors.org/.

Erlingsson, Ú. (2004). *The In-lined Reference Monitor Approach to Security Policy Enforcement*. Ph. D. thesis, Cornell University, Ithaca, New York.

Erlingsson, U., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006). XFI: Software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 75–88.

Erlingsson, Ú., B. Livshits, and Y. Xie (2007). End-to-end web application security. In *Proceedings of the 15th IEEE Workshop on Hot Topics in Operating Systems (HotOS)*.

Erlingsson, Ú. and F. B. Schneider (1999). SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pp. 87–95.

Etessami, K. and G. J. Holzmann (2000). Optimizing büchi automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR)*, pp. 153–167.

Evans, D. and A. Twynman (1999). Flexible policy-directed code safety. In *Proceedings of the 20th IEEE Symposium on Security & Privacy (S&P)*, pp. 32–45.

F-Secure (2013). Backdoor:W32/PoisonIvy. http://www.f-secure.com/v-descs/backdoor_w32_poisonivy.shtml.

Facebook Developers (2010). JavaScript SDK. http://developers.facebook.com/docs/reference/javascript.

Fara, M. (2013). Cross-platform malicious code discovered in "in the wild". http://forum.bitdefender.com/lofiversion/index.php/t47561.html.

FileInfo.com (2011). Executable file types. www.fileinfo.com/filetypes/executable.

Finifter, M., J. Weinberger, and A. Barth (2010). Preventing capability leaks in secure JavaScript subsets. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*.

FireEye (2009). Heap spraying with ActionScript. http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html.

Ford, S., M. Cova, C. Kruegel, and G. Vigna (2009). Analyzing and detecting malicious Flash advertisements. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, pp. 363–372.

Fredrikson, M., R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, and V. Yegneswaran (2012). Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, pp. 548–563.

Fredrikson, M. and B. Livshits (2011). RePriv: Re-imagining content personalization and in-browser privacy. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 131–146.

Fukami (2007). Testing and exploiting flash applications.

fukami and B. Fuhrmannek (2008). SWF and the malware tragedy. In *Proceedings of the OWASP Application Security Conference*.

Google Caja (2007). Compiler for making third-party HTML, CSS, and JavaScript safe for embedding. http://code.google.com/p/google-caja.

Guarnieri, S. and B. Livshits (2009). GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the 18th USENIX Security Symposium*, pp. 151–168.

Guha, A., S. Krishnamurthi, and T. Jim (2009). Using static analysis for Ajax intrusion detection. In *Proceedings of the 18th International Conference on World Wide Web (WWW)*, pp. 561–570.

Guha, S., B. Cheng, A. Reznichenko, H. Haddadi, and P. Francis (2009, October). Privad: Rearchitecting online advertising for privacy. Technical Report MPI-SWS-2009-004, Max Planck Institute for Software Systems, Kaiserslautern-Saarbrücken, Germany.

Gupta, G., A. Bansal, R. Min, L. Simon, and A. Mallya (2007). Coinductive Logic Programming and Its Applications. In *Proceedings of the 24th International Conference on Logic Programming (ICLP)*, pp. 27–44.

Gupta, R. (1992). Generalized dominators and post-dominators. In *Proceedings of the 20th ACM Symposium on Principles Of Programming Languages (POPL)*, pp. 246–257.

guya (2008). Encapsulating CSRF attacks inside massively distributed Flash movies—real world example. http://blog.guya.net/2008/09/14/encapsulating-csrf-attacks-inside-massively-distributed-flash-movies- real-world-example/.

Hamlen, K. W. and M. Jones (2008). Aspect-oriented in-lined reference monitors. In *Proceedings of the 3rd ACM Workshop on Programming Languages and Analysis for Security*, pp. 11–20.

Hamlen, K. W., M. M. Jones, and M. Sridhar (2011, May). Chekov: Aspect-oriented runtime monitor certification via model-checking (extended version). Technical report, Department of Computer Science, University of Texas at Dallas.

Hamlen, K. W., M. M. Jones, and M. Sridhar (2012). Aspect-oriented runtime monitor certification. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 126–140.

Hamlen, K. W., G. Morrisett, and F. B. Schneider (2006a). Certified in-lined reference monitoring on .NET. In *Proceedings of the 1st ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 7–16.

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

Hansen, R. and J. Grossman (2008, September). Clickjacking. http://www.sectheory.com/clickjacking.htm.

Havelund, K. (2000). Using runtime analysis to guide model checking of Java programs. In *Proceedings of the 7th International Workshop on SPIN Model Checking and Software Verification (SPIN)*, pp. 245–264.

Hay, R. (2009). Exploitation of cve-2009-1869. http://roeehay.blogspot.com/2009/08/exploitation-of-cve-2009-1869.html.

Heiderich, M., T. Frosch, and T. Holz (2011). IceShield: Detection and mitigation of malicious websites with a frozen DOM. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 281–300.

Heiderich, M., T. Frosch, M. Jensen, and T. Holz (2011). Crouching tiger - hidden payload: security risks of scalable vectors graphics. In *Proceedings of the 18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 239–250.

Holzmann, G. (2003). *The SPIN model checker: primer and reference manual*. Addison-Wesley Professional.

Howard, F. (2012). Exploring the blackhole exploit kit. In *Sophos Technical Paper*.

Huang, L.-S., A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson (2012). Clickjacking: Attacks and defenses. In *Proceedings of the 21st USENIX Security Symposium*, pp. 22–22.

Huang, W. (2011). Newest Adobe Flash 0-day used in new drive-by-download variation: drive-by-cache, targets human rights website. http://blog.armorize.com/2011/04/newest-adobe-flash-0-day-used-in-new.html.

IBM Corporation (2013). Security bulletin: Tivoli Endpoint Manager for software use (CVE-2013-0452). http://www-01.ibm.com/support/docview.wss?uid=swg21631350.

INRIA (2014). The Coq proof assistant. coq.inria.fr.

International, E. (1999, December). ECMAScript language specification. ECMA Standard 262, 3rd Edition.

Internet Advertising Bureau (2013). Internet advertising revenue report FY 2012. http://www.iab.net/media/file/IAB_Internet_Advertising_Revenue_Report_FY_2012_rev.pdf.

Invernizzi, L. and P. M. Comparetti (2012). Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pp. 428–442.

Irinco, B. (2013). Luckycat leads to attacks against several industries. http://about-threats.trendmicro.com/us/webattack/111/Luckycat+Leads+to+Attacks+Against+Several+Industries,TrendMicroThreatEncyclopedia.

Jackson, C. and H. J. Wang (2007). Subspace: Secure cross-domain communication for web mashups. In *Proceedings of the 16th on International Conference on World Wide Web (WWW)*, pp. 611–620.

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *Journal of Logic Programming 19*, 503–581.

Jaffar, J., S. Michaylov, P. J. Stuckey, and R. H. C. Yap (1992, May). The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS) 14*(3), 339–395.

Jagdale, P. (2009). Blinded by Flash: Widespread security risks Flash developers don't see.

Jähnig, G. (2010). fsmjs: A finite state machine library in JavaScript. http://code.google.com/p/fsmjs.

Jang, D. and K.-M. Choe (2009). Points-to analysis for Javascript. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pp. 1930–1937.

Jang, D., A. Venkataraman, G. M. Sawka, and H. Shacham (2011). Analyzing the cross-domain policies of flash applications. In *Proceedings of the IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*.

Jayaraman, K., W. Du, B. Rajagopalan, and S. J. Chapin (2010). ESCUDO: A fine-grained protection model for web browsers. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 231–240.

Jensen, S. H., A. Møller, and P. Thiemann (2009). Type analysis for JavaScript. In *Proceedings of the 16th International Static Analysis Symposium (SAS)*, pp. 238–255.

Jia, L., J. Zhao, V. Sjöberg, and S. Weirich (2010). Dependent types and program equivalence. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 275–286.

Jim, T., N. Swamy, and M. Hicks (2007). Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th International Conference on the World Wide Web (WWW)*, pp. 601–610.

Johns, M. and S. Lekies (2011). Biting the hand that serves you: A closer look at client-side Flash proxies for cross-domain requests. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pp. 85–103.

Johns, M., S. Lekies, and B. Stock (2013). Eradicating DNS rebinding with the extended same-origin policy. In *Proceedings of the 22nd USENIX Security Symposium*.

Jones, M. and K. W. Hamlen (2009). Enforcing IRM security policies: Two case studies. In *Proceedings of the 7th IEEE Intelligence and Security Informatics Conference (ISI)*, pp. 214–216.

Jones, M. and K. W. Hamlen (2010). Disambiguating aspect-oriented policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 193–204.

Jones, M. and K. W. Hamlen (2011). A service-oriented approach to mobile code security. In *Proceedings of the 8th International Conference on Mobile Web Information Systems (MobiWIS)*, pp. 531–538.

Jr., E. M. C., O. Grumberg, and D. A. Peled (1999). *Model Checking*. Cambridge, Massachusetts: The MIT Press.

Keizer, G. (2011). RSA hackers exploited Flash zero-day bug. http://www.computerworld.com/s/article/9215444/RSA_hackers_exploited_Flash_zero_day_bug.

Khoury, R. and N. Tawbi (2012). Corrective enforcement: A new paradigm of security policy enforcement by monitors. *ACM Transactions on Information and Systems Security (TISSEC) 15*(2), 1–27.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold (2001). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pp. 327–353.

Kiczales, G., J. Lamping, A. Medhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pp. 220–242.

Kikuchi, H., D. Yu, A. Chander, H. Inamura, and I. Serikov (2008). JavaScript instrumentation in practice. In *Proceedings of the 6th Asian Symposium on Programming Languages And Systems (APLAS)*, pp. 326–341.

Kim, M., M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky (2004). Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design 24*(2), 129–155.

Kimberly (2011). DDoS attacks - a new twist in malvertisements. http://stopmalvertising.com/malvertisements/ddos-attacks-a-new-twist-in-malvertisements.html.

Kindi Software (2013). KINDI software secureSWF. http://www.kindi.com/.

Kindlund, D. (2012). CFR watering hole attack details. http://www.fireeye.com/blog/technical/malware-research/2012/12/council-foreign-relations-water-hole-attack-details.html,FireEyeBlog.

Kindlund, D. (2013). Holiday watering hole attack proves difficult to detect and defend against. *ISSA Journal*, 10–12. http://c.ymcdn.com/sites/www.issa.org/resource/resmgr/journalpdfs/feature0213.pdf.

Kiriansky, V., D. Bruening, and S. P. Amarasinghe (2002). Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*.

Kisser, W., K. Havelund, G. Brat, S. Park, and F. Lerda (2003). Model Checking Programs. *Automated Software Engineering Journal 10*(2).

Kogan, I. (2005). Flare: Actionscript decompiler. tool. http://www.nowrap.de/flare.html.

Kogan, I. (2007). Flasm: Command line assembler/disassembler of actionscript byte-code. tool. http://www.nowrap.de/flasm.html.

Kolbitsch, C., B. Livshits, B. Zorn, and C. Seifert (2012). ROZZLE: De-cloaking internet malware. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pp. 443–457.

Kovac, P. (2011a). Breaking through flash obfuscation. https://blog.avast.com/2011/09/09/breaking-through-flash-obfuscation/,Avast!Blog.

Kovac, P. (2011b). Flash malware that could fit a twitter message. http://blog.avast.com/2011/06/28/flash-malware-that-could-fit-a-twitter-message/,Avast!Blog.

Lance, B. (2009). Connecting JavaScript and Flash. http://www.slideshare.net/BeautifulInterfaces/connecting-flash-and-javascript-using-externalinterface-2452543.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM (CACM) 52*(7), 107–115.

Leroy, X. (2011). Safety first!: Technical perspective. *Communications of the ACM (CACM) 54*(12), 122.

Levchenko, K., A. Pitsillidis, N. Chachra, B. Enright, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage (2011). Click trajectories: End-to-end analysis of the spam value chain. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 431–446.

Li, Z. and X. F. Wang (2010). FIRM: capability-based inline mediation of Flash behaviors. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, pp. 181–190.

Li, Z., T. Yi, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa (2011). WebShield: Enabling various web defense techniques without client side modifications. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*.

LIACC/Universidade do Porto and COPPE Sistemas/UFRJ (2009). Yap prolog. http://www.dcc.fc.up.pt/~vsc/Yap/.

Ligatti, J., L. Bauer, and D. Walker (2005a). Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security 4*(1–2), 2–16.

Ligatti, J., L. Bauer, and D. Walker (2005b). Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, pp. 355–373.

Louw, M. T., K. T. Ganesh, and V. N. Venkatakrishnan (2010). AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX Security Symposium*.

M86 Security (2010, July). Security labs report: January–June 2010 recap. Technical report, M86 Security.

Maffeis, S., J. C. Mitchell, and A. Taly (2009a). Isolating JavaScript with filters, rewriting, and wrappers. In *Proceedings of the 14th European Conference on Research in Computer Security (ESORICS)*, pp. 505–522.

Maffeis, S., J. C. Mitchell, and A. Taly (2009b). Run-time enforcement of secure JavaScript subsets. In *Proceedings of the IEEE Workshop on Web 2.0 Security & Privacy (W2SP)*.

Maffeis, S. and A. Taly (2009). Language-based isolation of untrusted JavaScript. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pp. 77–91.

Magazinius, J., P. H. Phung, and D. Sands (2010). Safe wrappers and sane policies for self protecting JavaScript. In *Proceedings of the 15th Nordic Conference on Secure IT Systems (NordSec)*, pp. 239–255.

Magazinius, J., B. K. Rios, and A. Sabelfeld (2013). Polyglots: Crossing origins by crossing formats. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, pp. 753–764.

Martinell, F. (2006). Through modeling to synthesis of security automata. In *Proceedings of the 2nd International Workshop on Security and Trust Management (STM)*, pp. 31–46.

Mayer, J. R. and J. C. Mitchell (2012). Third-party web tracking: Policy and technology. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, pp. 413–427.

McCamant, S. and G. Morrisett (2006). Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium*.

Meyerovich, L. A., A. P. Felt, and M. S. Miller (2010). Object views: Fine-grained sharing in browsers. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pp. 721–730.

Meyerovich, L. A. and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (S&P)*.

Microsoft Live Labs (2009). Microsoft web sandbox. http://websandbox.livelabs.com.

Mikko (2011). How we found the file that was used to hack RSA. http://www.f-secure.com/weblog/archives/00002226.html.

Mills, E. (2011). Attack on RSA used zero-day Flash exploit in Excel. http://news.cnet.com/8301-27080_3-20051071-245.html.

Milner, R. (1999). *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press.

Minamide, Y. (2005). Static approximation of dynamically generated web pages. In *Proceedings of the 14th International Conference on World Wide Web (WWW)*, pp. 432–441.

Ministry of Justice, United Kingdom (2013). U.K. human rights website. http://www.justice.gov.uk/human-rights.

MITRE Corporation (2013a). CAPEC-178:cross-site flashing. http://capec.mitre.org/data/definitions/178.html.

MITRE Corporation (2013b). Common vulnerabilities and exposures. http://cve.mitre.org/.

Moxiecode Systems AB (2013). Plupload v2.0.0. http://www.plupload.com/.

Nambiar, S. N. (2009). Flash phishing. http://www.symantec.com/connect/blogs/flash-phishing.

National Institute of Standards and Technology (NIST) (2013). Cwe — common weakness enumeration. http://nvd.nist.gov/cwe.cfm.

Necula, G. C. (1997). Proof-Carrying Code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 106–119.

Necula, G. C. (2000). Translation validation for an optimizing compiler. In *Proceedings of the 21st ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 83–94.

Necula, G. C. and P. Lee (1996). Safe kernel extensions without run-time checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 229–243.

Nikiforakis, N., A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna (2013). Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, pp. 541–555.

Oliveria, P. (2008). Flash pharming attack. http://blog.trendmicro.com/trendlabs-security-intelligence/targeted-attack-in-mexico-part-2-yet-another-drive-by-pharming/.

Overveldt, T. V., C. Kruegel, and G. Vigna (2012). FlashDetect: ActionScript3 malware detection. In *Proceedings of the 15th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 274–293.

Paola, S. D. (2007). Testing Flash applications.

Parkour, M. (2012). May 3 - CVE-2012-0779 World Uyghur Congress Invitation.doc. http://contagiodump.blogspot.com.es/2012/05/may-3-cve-2012-0779-world-uyghur.html.

Patil, K., X. Dong, X. Li, Z. Liang, and X. Jiang (2011). Towards fine-grained access control in JavaScript contexts. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS)*, pp. 720–729.

Person, S., M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu (2008). Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 226–237.

Petkov, P. D. (2008). Hacking the interwebs. http://www.gnucitizen.org/blog/hacking-the-interwebs/.

Pfenning, F. (1995). Structural cut elimination. In *Proceedings of the 10th Annual Symposium on Logic in Computer Science (LICS)*, pp. 156–166.

Phung, P. H. and L. Desmet (2012). A two-tier sandbox architecture for untrusted JavaScript. In *Proceedings of the Workshop on JavaScript Tools (JSTools)*, pp. 1–10.

Phung, P. H., M. Monshizadeh, M. Sridhar, K. W. Hamlen, and V. Venkatakrishnan (2013). Between worlds: Securing mixed JavaScript/ActionScript multi-party web content. Submitted for publication.

Phung, P. H., D. Sands, and A. Chudnov (2009). Lightweight self-protecting JavaScript. In *Proceedings of the 4th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pp. 47–60.

Pnueli, A. (1977). The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 46–57.

Pnueli, A., M. Siegel, and E. Singerman (1998). Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 151–166.

Politz, J. G., S. A. Eliopoulos, A. Guha, and S. Krishnamurthi (2011). ADsafety: Type-based verification of JavaScript sandboxing. In *Proceedings of the 20th USENIX Security Symposium*.

Poole, N. (2012). XSS and CSRF via SWF applets (SWFUpload, Plupload). https://nealpoole.com/blog/2012/05/xss-and-csrf-via-swf-applets-swfupload-plupload/.

Program-Transformation.Org (2013). Program optimization. http://www.program-transformation.org/Transform/ProgramOptimization.

Rad, M. B. (2013). Flash based XSS in Yahoo Mail. http://miladbr.blogspot.com/2013/06/flash-based-xss-in-yahoo-mail.html.

Ramakrishna, Y. S., C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren (1997). Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*, pp. 143–154.

Rapid7 Inc. (2012). Adobe flash player object type confusion. http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_flash_rtmp.

Rapid7 Inc. (2013). Cross-site flashing. http://www.rapid7.com/db/vulnerabilities/spider-actionscript-cross-site-flashing.

Reis, C., J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir (2006). BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 61–74.

Richards, G., S. Lebresne, B. Burg, and J. Vitek (2010). An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*.

Romang, E. (2013). Posts tagged CVE-2013-0633; gong da/gondad exploit pack add Flash CVE-2013-0634 support.

Rondon, P., M. Kawaguchi, and R. Jhala (2010). Low-level liquid types. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Rozier, K. Y. and M. Y. Vardi (2007). LTL satisfiability checking. In *In 14th International SPIN Workshop, volume 4595 of LNCS*, pp. 149–167. Springer.

Ruys, T. C. and N. H. M. A. de Brugh (2007). MMC: The Mono model checker. *Electronic Notes in Theoretical Computer Science 190*(1), 149–160.

Schneider, F. B. (2000). Enforceable Security Policies. *ACM Transactions on Information and System Security 3*, 30–50.

Sebastiani, R. and S. Tonetta (2003). "more deterministic" vs. "smaller" büchi automata for efficient LTL model checking. In *Proceedings of the Correct Hardware Design and Verification Methods Conference (CHARME)*, pp. 126–140.

Seltzer, L. (2010). New JIT spray penetrates best Windows defenses. http://securitywatch.pcmag.com/apple/284124-new-jit-spray-penetrates-best-windows-defenses.

Serna, F. J. (2013, July). Flash JIT—spraying info leak gadgets.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, pp. 552–561.

Shah, V. and F. Hill (2003). An aspect-oriented security framework. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, Volume 2.

Shapiro, L. and E. Y. Sterling (1994). *The Art of PROLOG: Advanced Programming Techniques*. The MIT Press.

Simon, L., A. Mallya, A. Bansal, and G. Gupta (2006). Coinductive logic programming. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP)*.

Smith, R., C. Estan, and S. Jha (2008). XFA: Faster signature matching with extended automata. In *Proceedings of the 29th IEEE Symposium on Security & Privacy (S&P)*, pp. 187–201.

Sophos (2013). Sophos security threat report 2013. new platforms and changing threats.

Sridhar, M. and K. W. Hamlen (2010a). In-lined reference monitoring in Prolog. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pp. 149–151.

Sridhar, M. and K. W. Hamlen (2010b). Model-checking in-lined reference monitors. In *Proceedings of the 11th International Conference on Verification, Model-Checking and Abstract Interpretation (VMCAI)*, pp. 312–327.

Sridhar, M. and K. W. Hamlen (2011). Flexible in-lined reference monitor certification: Challenges and future directions. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, pp. 55–60.

Sridhar, M., D. V. Karamchandani, and K. W. Hamlen (2014). Flash in the dark: Surveying the landscape of ActionScript security trends and threats. Submitted for publication.

Sridhar, M., R. Wartell, and K. W. Hamlen (2013a). FlashTrack: A transparency checker tool for Flash applications. http://code.google.com/p/flashtrack.

Sridhar, M., R. Wartell, and K. W. Hamlen (2013b, February). Hippocratic binary instrumentation: First do no harm (extended version). Technical report, Department Of Computer Science, University of Texas at Dallas.

Sridhar, M., R. Wartell, and K. W. Hamlen (2014). Hippocratic binary instrumentation: First do no harm. *Science of Computer Programming (SCP)*. forthcoming.

Stamm, S., B. Sterne, and G. Markham (2010). Reining in the web with content security policy. In *Proceedings of the 19th International Conference on World Wide Web (WWW)*, pp. 921–930.

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

Striegel, J. (2007). DNS rebinding: how an attacker can use your web browser to bypass a firewall. http://makezine.com/2007/08/01/dns-rebinding-how-an-attacker/.

Sun, M., G. Tan, J. Siefers, B. Zeng, and G. Morrisett (2013). Bringing Java's wild native world under control. *ACM Transactions on Information and System Security 16*(3), 1–28.

swfupload (2013). swfupload: JavaScript & Flash upload library. https://code.google.com/p/swfupload/.

Symantec Corporation (2012). Targeted attacks using confusion (CVE-2012-0779). http://www.symantec.com/connect/blogs/targeted-attacks-using-confusion-cve-2012-0779.

Symantec Corporation (2013, April). Symantec internet security threat report. Technical Report Volume 18, Symantec Corporation.

Taly, A., Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra (2011). Automated analysis of security-critical JavaScript APIs. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 363–378.

Tamaki, H. and T. Sato (1986). OLD resolution with tabulation. In *Proceedings of the 3rd International Conference on Logic Programming (ICLP)*, pp. 84–98.

Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee (1996). TIL: a type-directed optimizing compiler for ML. In *Proceedings of the 17th ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*.

The AspectJ Team (2003). The AspectJ programming guide. www.eclipse.org/aspectj/doc/released/progguide/index.html.

The OWASP Foundation (2013). Testing for cross site flashing(OWASP-DV-004). https://www.owasp.org/index.php/Testing_for_Cross_site_flashing_%28OWASP-DV-004%29.

Thiemann, P. (2005a). Grammar-based analysis of string expressions. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*.

Thiemann, P. (2005b). Towards a type system for analyzing JavaScript programs. In *Proceedings of 14th European Symposium on Programming (ESOP)*.

Thomas, K., C. Grier, J. Ma, V. Paxson, and D. Song (2011). Design and evaluation of a real-time url spam filtering service. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 447–462.

Tiu, A. and J. E. Dawson (2010). Automating open bisimulation checking for the spi calculus. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pp. 307–321.

Toubiana, V., A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas (2010). Adnostic: Privacy preserving targeted advertising. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS)*.

Viega, J., J. Bloch, and P. Chandra (2001). Applying aspect-oriented programming to security. *Cutter IT Journal 14*(2), 31–39.

W3Techs (2013). Usage of Flash for websites. http://w3techs.com/technologies/details/cp-flash/all/all.

Walker, D. (2000). A type system for expressive security policies. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Wand, M., G. Kiczales, and C. Dutchyn (2004). A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems 26*(5), 890–910.

Wang, R., S. Chen, and X. Wang (2012). Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 33rd IEEE Symposium on Security & Privacy (S&P)*, pp. 365–379.

Wartell, R., V. Mohan, K. W. Hamlen, and Z. Lin (2012, December). Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, Orlando, Florida, pp. 299–308.

Weinberg, Z., E. Y. Chen, P. R. Jayaraman, and C. Jackson (2011). I still know what you visited last summer: Leaking browsing history via user interaction and side channel attacks. In *Proceedings of the 32nd IEEE Symposium on Security & Privacy (S&P)*, pp. 147–161.

Wolf, J. (2009, July). Heap spraying with ActionScript: Why turning off JavaScript won't help this time. FireEye Malware Intelligence Lab. http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html.

WordPress.org (2013). WordPress.org. http://wordpress.org/.

Yahoo! Inc. (2013). Yahoo! user interface library. http://yuilibrary.com/.

Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009). Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P)*, pp. 79–93.

Yu, D., A. Chander, N. Islam, and I. Serikov (2007). JavaScript instrumentation for browser security. In *Proceedings of the 35th ACM Symposium on Principles Of Programming Languages (POPL)*, pp. 237–249.

Zaks, A. and A. Pnueli (2008). CoVaC: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods (FM)*, pp. 35–51.

Zalewski, M. (2011a). Browser security handbook. http://code.google.com/p/browsersec/.

Zalewski, M. (2011b). Same-origin policy Flash. https://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy,Google.

Zeller, W. and E. W. Felten (2008). Cross-site request forgeries: Exploitation and prevention.

Zeltser, L. (2011a). Malvertising: The mechanics of malicious ads. http://blog.zeltser.com/post/6275464150/malvertising-mechanics-of-malicious-ads.

Zeltser, L. (2011b). Malvertising: The use of malicious ads to install malware. http://www.infosecisland.com/blogview/14371-Malvertising-The-Use-of-Malicious-Ads-to-Install-Malware.html.

VITA

Meera Sridhar was born in Chennai, India, to parents Kanamanapalli and Sarojini Sridhar. Meera's parents strongly valued education, and enrolled her in the best schools starting from kindergarten. Her elementary schooling in Birla Vidya Niketan and Vasant Valley in New Delhi, India, formed the beginnings of a wonderful academic journey.

When she was twelve, Meera's family moved to Manila, Philippines. There, she completed her middle and high schooling from the International School Manila, one of the most competitive schools in Asia, during which she participated in intense academic training, including training in accelerated mathematics, physics, and writing, and earned the prestigious International Baccalaureate Diploma.

Meera completed her undergraduate and Master's degrees in Computer Science from Carnegie Mellon University, where she graduated with University and CS Department College Honors. Carnegie Mellon introduced Meera to the dazzling world of top-notch computer science education and research, and set standards for those in her mind that she continues to use for measuring education and research excellence. Carnegie Mellon also enabled her to cross paths with leaders in the fields of programming languages and model-checking such as Dr. Peter Lee, Dr. Jeannette Wing, and Dr. Edmund Clarke, who, with their energy and enthusiasm for training a young student, inspired Meera to start performing computer science research in programming languages and model-checking very early in her undergraduate years. Her work led to a prestigious Carnegie Mellon CS Department Fellowship award for a fully funded Master's program, presented to select undergraduate students who show excellence in undergraduate research.

Meera embarked on her doctoral journey in the Fall of 2007 at The University of Texas at Dallas. There, with her advisor, Dr. Kevin Hamlen, she started her most enjoyable research experience yet, in language-based security and formal methods. Her doctoral career presented her with numerous fresh, exciting research opportunities and achievements, including several publications in highly-reputed venues, industry and cross-university collaborations, and internships at Adobe Advanced Technology Labs and Amazon Lab126. Her doctoral studies culminated in this dissertation, designing algorithms for model-checking in-lined reference monitors.