DECLARATIVE ASPECT-ORIENTED SECURITY POLICIES FOR

IN-LINED REFERENCE MONITORS

by

Micah Jones

APPROVED BY SUPERVISORY COMMITTEE:

Dr. Kevin Hamlen, Chair

Dr. Farokh Bastani

Dr. Gopal Gupta

Dr. Bhavani Thuraisingham

Dedicated to my parents,

for their teaching, encouragement, and love.

DECLARATIVE ASPECT-ORIENTED SECURITY POLICIES FOR IN-LINED REFERENCE MONITORS

by

MICAH JONES, B.S., M.S.

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

December 2011

ACKNOWLEDGMENTS

The author is thoroughly grateful to have had the opportunity to work under his advisor, Kevin Hamlen, whose knowledge and creativity helped form the foundation of the research leading up to this dissertation. Moreover, his patience and clarity of teaching and direction proved invaluable to the author's education in the fine art of technical writing and research.

Much of this research could not have been accomplished without the help of the author's colleague, Meera Sridhar. Special mention should also go to Richard Wartell, Vishwath Mohan, Brian DeVries, and Scott Moore, as their work is inextricably linked to that of this dissertation.

This research was supported in part by U.S. AFOSR YIP award number FA9550-08-1-0044 and NSF Trustworthy Computing collaborative research grant 1065134/1065216.

The author extends his heartfelt thanks to his many excellent and inspiring undergraduate professors at Oklahoma Baptist University. Dale and Cindy Hanchey taught him computer programming, software development methods, and vital presentation skills, continually challenging him and never permitting laziness in his studies. John Nichols's passionate, kind-hearted teaching style accomplished the impossible task of giving the author a love of mathematics, along with an even greater love of teaching itself. Karen Youmans and John Powell instilled in the author an appreciation for literature and history, and helped convince him to attend graduate school.

Finally, and most importantly, the author thanks his Lord Jesus Christ, who is his greatest encourager, helper, teacher, and friend. For all things work for the good of those who love Him.

November 2011

DECLARATIVE ASPECT-ORIENTED SECURITY POLICIES FOR

IN-LINED REFERENCE MONITORS

Publication No. _____

Micah Jones, Ph.D.
The University of Texas at Dallas, 2011

Supervising Professor: Dr. Kevin Hamlen

Aspect-oriented in-lined reference monitor frameworks provide an elegant and increasingly popular means of enforcing security policies on untrusted code through automated rewriting. However, it is often difficult to prove that the resulting instrumented code is secure with respect to the intended policy, especially when that policy is described in terms of arbitrary imperative code insertions. The dissertation presents a fully declarative policy language, powerful enough to enforce real-world security policies yet simple enough for automated verification systems to correctly reason about them. It also presents an analysis tool that detects policy inconsistencies, a service-oriented framework for instrumenting untrusted code, and a full-scale abstract interpretation and model-checking system that verifies the safety of rewritten programs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

One of the greatest challenges facing today's technology industry is the problem of precisely and reliably securing untrusted code. In any real-world system, there are a number of program actions that are considered to be unacceptable. These actions may include, for example, dissemination of private information over a network, excessive use of system resources, or modification of sensitive data. As modern software grows more complex, prohibiting such actions requires increasingly sophisticated policy specification and enforcement mechanisms. This gives rise to the further, inextricably linked challenges of limiting the size and proving the correctness of the enforcing security code.

Modern software security enforcement systems include those that perform static analysis prior to a program's execution, and those that dynamically monitor its actions during execution. The Java Virtual Machine (JVM) performs static analysis of Java bytecode to ensure that it adheres to fundamental rules, such as limiting jump targets to instructions within the bounds of a method. Execution monitors are implemented in most operating systems; for example, Windows 7 detects actions that modify the system registry and requires user permission before such a modification is allowed to occur. Modern anti-virus software frequently performs both static and dynamic program analysis.

A powerful, highly flexible alternative enforcement approach may be found in *In-lined Reference Monitors* (IRMs) (Schneider 2000). An IRM system is a framework in which untrusted code is automatically rewritten prior to execution, such that the rewritten code internally enforces a specified security policy. The instrumentation process involves insertion of dynamic guard instructions that detect and prevent impending policy violations at runtime. The rewritten program is said to be *self-monitoring*, meaning that it can be safely

executed without the need for external security systems. Rewriting can be applied to either original source code or binary executables.

The instrumentation of an IRM security policy, in which security-relevant events are located and new code is injected around them, is essentially an instance of *Aspect-Oriented Programming* (AOP) (Kiczales et al. 1997). In AOP, *aspects* define code fragments called *advice*, as well as *pointcut* expressions that describe the code points at which to insert those fragments. During program compilation, an *aspect weaver* performs these code insertions to create a final executable. Aspects are commonly used to express cross-cutting concerns, such as generating log entries immediately before file operations.

Other work has recognized and leveraged the connections between IRMs and AOP (Chen and Roşu 2005), but in doing so it supports imperative advice that is difficult for policy-writers to fully and accurately reason about. Advice is normally written using a Turing-complete language (e.g., Java), and thus constitutes a substantial addition to the security framework's *Trusted Computing Base* (TCB). Aspect-weaving is an inherently complex process that can cause unexpected changes to the resulting program, which makes it difficult to determine whether a given aspectual policy actually enforces the higher-level security policy it was meant to implement.

Security policies that are amenable to formal analysis are especially important for certifying IRMs such as Mobile (Hamlen 2006), which secure untrusted code using a process similar to that in Figure 1.1. Such frameworks strictly separate trusted components (security policies and a verification tool) from untrusted components (the original and rewritten program binaries, and the rewriter itself). The trusted components must be as small as possible in order to minimize the TCB and facilitate proofs of correctness.

An ideal IRM framework should balance power with simplicity, allowing enforcement of a wide range of security policies without rendering it impractical to prove that the rewritten program is actually secure. The system must also place strict limitations on the size of the TCB, thus restricting the manner in which policies are specified and enforced. Such a task greatly benefits from separation of enforcement components, excluding the more complex

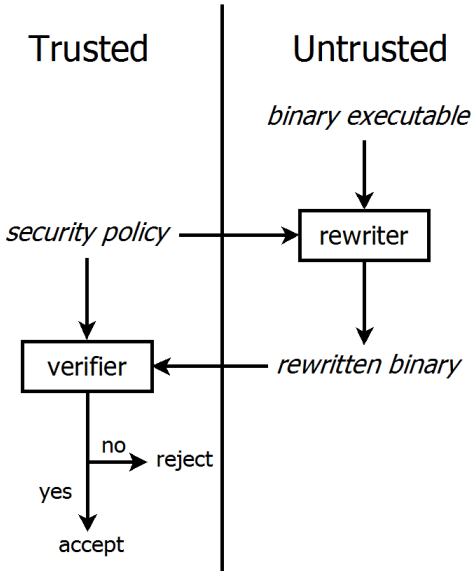

Figure 1.1. Flow of a certifying IRM framework.

ones from the TCB wherever possible. There is a particular need for a security policy specification language that is simple enough to be included in the TCB yet powerful enough to describe desirable policies.

My Thesis. This dissertation presents a purely declarative, aspect-oriented IRM enforcement system, consisting of a security policy specification language, a policy analysis tool, a rewriter, a rewriting web service, and a static verification system for rewritten code. It emphasizes the advantages of using a fully declarative policy language, capable of expressing complex security automata and describing relevant program events with pointcuts from AOP, yet amenable to mathematical analysis. It further discusses how programs may be instrumented with policies either by a local rewriter or via a web service. Finally, it shows how rewritten applications can be proved secure through model-checking and the use of linear constraints.

The dissertation is structured as follows. Chapter 3 describes the SPoX policy specification language, its denotational semantics, and a simple rewriting algorithm. Chapter 4 presents our rewriter implementation, and provides three case studies to show its usefulness and efficiency. Chapter 5 discusses an algorithm for detecting inconsistencies in SPoX policy specifications, and presents an implementation of our analysis tool. Chapter 6 presents a service-oriented implementation of our rewriter. Chapter 7 presents Chekov✓, a verifier that certifies a rewritten binary as safe with respect to a SPoX policy. Finally, Chapter 8 concludes with a summary and a discussion of future work.

CHAPTER 2
RELATED WORK

2.1 Certifying IRMs

The concept of automated program rewriting has existed for some time, but a formal theory of In-lined Reference Monitoring was not developed until the past decade (Schneider 2000). IRMs can be used to enforce *safety policies*, which prevent some "bad event" from happening. Furthermore, safety policies can be modeled as security automata that accept all and only those event sequences that do not have proscribed prefixes. Later work (Hamlen et al. 2006; Ligatti et al. 2005) explored the use of program rewriting to enforce broader classes of policies, including some types of liveness policies.

Prior research has shown that in-lined monitors are strictly more powerful than external execution monitors (Hamlen et al. 2006). This result assumes that neither type of monitor has access to any resource that the other does not. For example, in multithreaded Java applications where an external monitor can control the JVM scheduler and an in-lined one cannot, the external monitor may in fact be more powerful (Dam et al. 2009).

The SASI system (Erlingsson and Schneider 1999) was one of the earliest IRM implementations. SASI rewrites x86 machine code and Java bytecode programs to enforce safety policies expressed in PSLang (Erlingsson 2004). PSLang is a simple imperative language that can be used to identify security-relevant program instructions and provide the code that the rewriter should inject around each. As with most existing IRM systems, both the policy and the rewritten code are trusted.

SASI was followed by many other IRM systems, such as Naccio (Evans and Twynman 1999), Java-MaC (Kim et al. 2004), and Polymer (Bauer et al. 2005). Naccio enforces resource bound policies using both declarative and imperative policy specifications, with

separate components for defining system resources, safety policies, and architecture-specific security-relevant operations. A significant limitation of Naccio is that IRMs may only be instrumented in the source code of C programs, as opposed to executable binaries. Java-MaC rewrites Java bytecode to enforce policies written in MEDL/PEDL, a language that defines security-state variables, security-relevant events, and state changes effected by those events. This language lacks a powerful mechanism for specifying security-relevant events, which are limited to method calls and field accesses. Polymer focuses on enforcing composable security policies in Java using policy combinators and imperative code injections.

Certifying IRMs (Hamlen et al. 2006; Hamlen 2006; Aktug and Naliuka 2008; DeVries et al. 2009; Sridhar and Hamlen 2011) are designed to exclude rewritten code from the TCB, instead using an independent, automated theorem-prover to verify that the new program cannot violate the security policy. Mobile (Hamlen et al. 2006; Hamlen 2006) rewrites .NET CLI binaries according to declarative policy specifications, producing proofs of policy-adherence in the form of typing annotations in an effect-based type system (cf. (DeLine and Fähndrich 2004)). Those proofs are then verified by a trusted type-checker. Mobile is limited by both its policy language—which denotes types in Mobile's own type system—and by its requirement that security states are represented in a very specific manner.

ConSpec (Aktug and Naliuka 2008) (Contract Specification Language) simplifies PSLang such that its non-declarative components consist only of effect-free operations. This restriction helps to facilitate formal reasoning in a contract-based certification system (Dragoni et al. 2007). However, the fact remains that ConSpec policies include imperative code, limiting verification to ensuring that such code is placed correctly in the rewritten program. Independent analysis is required to ensure that the provided guard instructions accurately enforce the intended policy. Therefore, those instructions are actually included in the TCB.

The Chekov✓ verification system discussed in this dissertation (Hamlen et al. 2011) extends prior work on model-checking rewritten Adobe ActionScript bytecode (DeVries et al. 2009). The prior work is limited to simple programs, able to consider only a few kinds of security-relevant events (method calls) and specific forms of guard code. Chekov✓ greatly

improves upon it, and is able to prove safety for real-world programs instrumented with a variety of guard code insertions that enforce complex policies.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming was introduced by Kiczales et al. (Kiczales et al. 1997) as a means to elegantly express and implement *cross-cutting* concerns for existing source code. A canonical example of cross-cutting is the scenario in which calls to a logging module need to be placed before significant program operations (e.g., file operations). AOP languages such as AspectJ (Kiczales et al. 2001) perform code insertions of this sort through the use of aspects.

Researchers quickly discovered the close connection between AOP and IRM technologies (Viega et al. 2001; Shah and Hill 2003; Hamlen et al. 2006; Hamlen and Jones 2008). AOP's pointcuts could indicate security-relevant events, while advice could provide policy enforcement code. Pointcut use turned out to be straightforward, but the range of permissible types of advice became a subject of much debate. Early aspectual security policies (Viega et al. 2001) allowed all types of aspects and all types of enforcement code, even including *replacement advice*, which actually removes existing code and replaces it with instructions provided in the policy description.

Java-MOP (Chen and Roşu 2005) is a Java-based IRM system built upon AspectJ. In addition to its inclusion of full aspects (consisting of both pointcuts and advice), the Java-MOP policy specification language supports multiple logic engines, including CFGs, EREs, FSMs, and LTL. This makes the system very powerful and flexible, capable of expressing security policies in a number of ways. However, this power comes with such complexity that formal certification is rendered unfeasible. In particular, proving the correctness of arbitrary imperative advice is an undecidable problem in general.

The problem of advice with side effects is far-reaching, and is a fundamental barrier to certification of rewritten code. Recognition of this issue gave rise to the concept of *harmless*

advice (Dantas and Walker 2006): advice which does not affect the output of the code into which it is injected, aside from possibly causing slowed execution or premature termination. ConSpec's advice is limited to effect-free operations for this reason (Aktug and Naliuka 2008). However, even effect-free code is difficult to certify, especially in terms of the desired, higher level policy it is meant to enforce. For this reason, SPoX does not include advice at all, outside of declarative, mathematical state transition information (Hamlen and Jones 2008).

Even harmless advice can still affect policy-adherent program behavior, so there is much concern over how to preserve that behavior even after rewriting. For example, injected instructions consume CPU cycles, which might be noticeable to the end user depending on the situation. Behavior preservation by an IRM is called *transparency* (Sridhar and Hamlen 2011). Recent work has provided formal proofs of transparency for an IRM focused on the enforcement of information flow policies (Chudnov and Naumann 2010).

Ensuring aspectual consistency is important for both IRM policies and AOP programs in general (Douence et al. 2002; Katz and Katz 2010; Yu et al. 2007; Jones and Hamlen 2010). Conflicting advice for the same join point can cause results at runtime that are difficult to predict. For example, the order of advice execution may yield different results, as some advice executions may change the environment in which others run afterward (Katz and Katz 2010). This can result in nonsensical scenarios, such as one in which a program action A_1 is transformed into A_2 but A_2 must be transformed into A_1 (Yu et al. 2007). If two or more aspects are *strongly independent*, then they never conflict for any application; if they are *weakly independent*, then they do not conflict for some subset of applications (Douence et al. 2002). As SPoX advice has been simplified to include only automaton state transitions, it is possible to use a combination of pointcut intersection detection and linear constraint analysis to detect policy non-determinism and thus decide strong independence (Jones and Hamlen 2010).

2.3 Other Approaches

A substantial, well-established body of certifying IRM research (Erlingsson et al. 2006; McCamant and Morrisett 2006; Yee et al. 2009) has focused on enforcing specific control flow and memory safety policies. Microsoft's XFI system (Erlingsson et al. 2006) places a signature byte sequence around every possible jump target. That sequence's existence is dynamically confirmed by guard instructions before they permit jumps to the corresponding target address, thus enforcing control flow safety. PittSFIeld (McCamant and Morrisett 2006) isolates possible jump targets by partitioning binary instructions into chunks, and requiring every jump to point to the beginning of one of those chunks. The resulting limitations on a program's control flow effectively allow the insertion of guard instructions that cannot be circumvented (Hamlen et al. 2010). Native Client (NaCl) (Yee et al. 2009) uses sandboxing to allow native x86 code to run inside a web browser. It limits accessible memory space, prohibits certain instructions like system calls, and uses a chunk partitioning system similar to PittSFIeld's to enforce both control flow and memory safety.

An alternative to IRM certification is a Proof-Carrying Code (PCC) framework (Necula 1997), in which code-producers provide binary code to the consumer, along with a separate proof that the code is correct. Subsequently, the consumer must independently verify that the proof is valid. It is necessary that the untrusted code-producer be involved in the certification process, as constructing the proof usually requires information about the original source code and how it is compiled. Moreover, the proof can be enormous, often many times the size of the code it describes. In contrast, most certifying IRM frameworks operate entirely in the realm of binary executables, and there is no need for either the rewriter or the code-consumer to have access to the source code. The instrumentation process obviates the need for a full proof of safety, generating small, untrusted hints about the rewritten binary code. Those hints are sufficient for a trusted, automated verifier to independently check the code's safety.

Model-Carrying Code (MCC) (Sekar et al. 2003) attempts to bridge the gap between PCC and IRMs by requiring the code-producer to provide a model of program behavior,

which the code-consumer verifies and compares to the security policy. The contract-based verification system (Dragoni et al. 2007) of ConSpec (Aktug and Naliuka 2008) is actually a variant of MCC, where the contract is a model generated through static analysis of the target application. Enforcing new policies requires a new contract to be generated. Model-checking certifying IRMs (DeVries et al. 2009; Hamlen et al. 2011) are another form of MCC, but they avoid requiring assistance from the code-producer by generating their own (implicit) models automatically during the certification process.

Although it is often unrealistic to rely on the code-producer as part of the verification process, many programmers wish to have assistance in writing code that is safe to begin with. Recent work (Vanoverberghe and Piessens 2009) has focused on giving software developers static verification tools that test code for policy adherence and determine if in-lined guard instructions are necessary. The key difference between this system and traditional IRM frameworks is the detection of situations where guard instructions would cause unexpected side effects at runtime, thus allowing the creation of programs that are robust as well as secure.

2.4 Remaining Challenges

Multithreaded programs remain among the most significant obstacles current IRM research must overcome. It is often straightforward to lock security-relevant resources as necessary; avoiding deadlock and severe drops in program efficiency is not. However, these problems can often be avoided by writing race-free policies (Dam et al. 2009), for which threads in the target program do not interfere in a security-relevant manner. In fact, the cited paper's authors argue that non-race-free policies are inherently nonsensical, and are most likely the manifestation of a bug in the policy logic. Yet regardless of the policy, concurrency support in a certifying IRM framework necessitates some kind of race detection analysis, such as the aspect-oriented Racer system (Bodden and Havelund 2008).

Modern applications are increasingly networked, running simultaneously across several machines. For example, massively multiplayer online games may run thousands of client instances at the same time, all of them connected to one or more central servers. An individual client may even shut down and restart, keeping the same active session from the perspective of the user and the server. Security policies for such systems may encompass several parallel and/or successive executions of a program, requiring a more complex treatment of both the policy definition and inlining processes. One proposed policy language to handle these situations is 2D-LTL (Massacci et al. 2006), a *bi-dimensional* variant of LTL. Dimensionality here refers to the different possible kinds of executions described above.

Mobile code has become significantly more complex in recent years, and merits attention in IRM research. The S3MS project (Desmet et al. 2007; Vanoverberghe and Piessens 2008; Desmet et al. 2009) provides a framework for enforcing security policies on mobile applications. Its focus thus far is on code running under the .NET framework. It utilizes cryptographic signatures, IRMs, and PCC, and supported policy languages include ConSpec and 2D-LTL. Due in part to its strong connections to the ConSpec project, it is designed around security-by-contract, and thus inherits all associated strengths and weaknesses.

Untrusted binaries come in a variety of forms, requiring IRM systems that can support a variety of different languages and platforms. A significant part of current research is focused on high-level bytecode languages, namely Java (Kim et al. 2004; Bauer et al. 2005; Chen and Roşu 2005; Hamlen and Jones 2008) and .NET (Hamlen 2006; Vanoverberghe and Piessens 2008; Desmet et al. 2009). These frameworks are particularly helpful in that they provide strong built-in security assurances, such as restrictions on dynamic jumps and memory usage, as well as an easily parsed structure (Lindholm and Yellin 1999; ECMA 2002). Other systems target Adobe Flash and ActionScript (DeVries et al. 2009; Li and Wang 2010), JavaScript (Yu et al. 2007), and C source code (Evans and Twynman 1999; Viega et al. 2001; Shah and Hill 2003).

Many critical untrusted applications are x86 assembly programs, which are much more difficult to secure than high-level source code and bytecode applications. However, their sheer

prevalence requires IRM research to focus attention on them. SASI (Erlingsson and Schneider 1999) has been used to enforce a limited software fault isolation (SFI) policy on x86 code. Critically, PittSFIeld's chunk partitioning system (Erlingsson et al. 2006) makes it possible to prevent circumventions of IRM guard instructions, and is used by Native Client (Yee et al. 2009) to enforce control flow and memory safety policies. The key remaining problem for x86 IRMs is enforcement of a wider variety of security policies, which is the subject of current research (Hamlen et al. 2010).

CHAPTER 3
POLICY LANGUAGE[1]

3.1 Overview

SPoX (Security Policy XML) is a *purely declarative*, Aspect-Oriented security policy specification language for In-lined Reference Monitoring. In our analysis we define a formal denotational semantics for our language that merges the semantics of AOP languages and that of IRM's. The result is a language in which policies denote *Aspect-Oriented security automata*—security automata whose edge labels are encoded as pointcut expressions. Each policy in our language therefore encodes a property that can be said to be true or false of rewritten code apart from any original, unmodified code from which it may have been derived. The property modeled by a policy is the acceptance condition of the automaton it denotes.

The existence of a formal denotational semantics for the language is useful because it provides a means of formally proving that untrusted code satisfies a specified security policy. This provides the necessary theoretical foundation whereby a certifying In-lined Reference Monitoring system (cf., (Hamlen et al. 2011)) can generate a proof of policy-adherence for the code it produces. Code-recipients can use such proofs to independently verify that the code is safe to execute even when the code-producer is not trusted. A denotational semantics also facilitates the stronger objective of formally verifying that a program-rewriting system always produces policy-adherent code.

The purely declarative nature of our language means that policies define *what* security property to enforce without overspecifying *how* it is to be enforced. For any given policy

[1]This chapter includes previously published (Hamlen and Jones 2008; Jones and Hamlen 2009) joint work with Kevin Hamlen.

there will typically be many possible rewriting strategies that enforce it. This flexibility affords IRM implementations the freedom to choose an optimal rewriting algorithm based on architectural details, the results of program analyses, and other information that becomes available at rewriting time. This also makes our language suitable for separate certification as discussed in Chapter 7. That is, a separate verifier could examine rewritten code to determine whether it actually satisfies the security policy. We consider this to be important for building trustworthy IRM systems since it constitutes an extra level of redundancy for detecting and debugging incorrect policy specifications.

In making policies purely declarative we do not exclude the possibility that some rewriter implementations might also accept additional input from the policy-writer suggesting how the policy should be enforced. For example, policy-writers might suggest remedial actions (e.g., premature termination, roll-back, etc.) expressed as imperative code fragments to be executed in the event that a policy violation would have otherwise occurred. However, this information is not trusted and therefore remains separate from the policy. Hence, if the implementation of roll-back includes an operation that constitutes a violation of the security policy, this flaw in the rewriting algorithm can be detected and rejected by a verifier.

The process of writing a correct policy specification can be quite challenging, and we anticipate that widespread use of SPoX will require good visualization tools. Recent work (Patwardhan et al. 2010) allows analysis of a target program's bytecode with respect to a given policy through *security-aware* UML models. The tool provides views for comparisons of original and rewritten code, and uses control-flow diagrams to track all possible security states for each code block. Also, SPoX-denoted security automata readily lend themselves to visualization, so GUI-based policy viewers and editors are an interesting avenue for future work.

The chapter proceeds as follows. The SPoX language is described in Section 3.2, including its syntax and features. In Section 3.3, we provide analysis of the language, including its denotational semantics (Section 3.3.1) and enforcement strategies (Section 3.3.2).

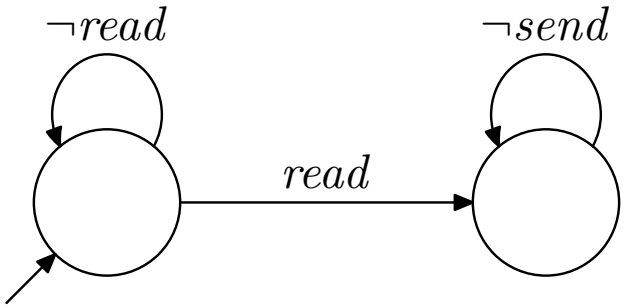

Figure 3.1. A security automaton prohibiting *send* after *read*

3.2 Language Syntax

SPoX is an XML-based security policy specification language suitable for enforcement by IRM's. A SPoX specification defines a *security automaton* (Schneider 2000)—a finite- or infinite-state machine that accepts all and only those *event sequences* that satisfy a security policy. For example, the security automaton in Figure 3.1 encodes the policy that prohibits any network *send* operation after a file *read* operation. Such a policy might be used to prevent untrusted code from leaking files over the network. Observe that the transitions of the automaton are labeled with predicates that denote sets of security-relevant events—program operations that change the security state of the program. Any program operation satisfying the label of an outgoing edge from the current state causes the automaton to transition to the destination state. If no outgoing edge label satisfies the next operation to be executed, that operation is a policy violation and the automaton rejects it.

A grammar for the core language of SPoX is given in Figure 3.2. SPoX specifications are lists of edge declarations, each consisting of three parts:

- *Pointcut expressions* identify sets of related security-relevant events that programs might exhibit at runtime. These serve as edge labels for the automaton.
- *Security-state variable* declarations abstract the security state of an arbitrary program. These serve as node labels for the automaton.

$n \in \mathbb{N}$	natural numbers	$c \in C$	class names
$sv \in SV$	state variables	$iv \in IV$	iteration variables
$id \in ID$	object identifiers	$ed \in ED$	edge names
$pcn \in PCN$	pointcut names		

pol ::=	$spc^* sd^* edg^*$	**policies**
spc ::=	`<pointcut name="`pcn`">`pcd`</pointcut>`	**stored pointcuts**
sd ::=	`<state name="`sv`">[`c`]</state>`	**state declarations**
edg ::=		**edges**
	`<edge name="`ed`" [after="true"]>`$pcd\ ep^*$`</edge>`	edgesets
	| `<forall var="`iv`" from="`e_1`" to="`e_2`">`edg^*`</forall>`	iteration
ep ::=	`<nodes [obj="`id`"] var="`sv`">` e_1`,`e_2 `</nodes>`	**edge endpoints**
e ::=	n | iv | e_1`+`e_2 | e_1`-`e_2	**arithmetic expressions**
	| e_1`*`e_2 | e_1`/`e_2 | `(`e`)`	

Figure 3.2. SPoX language syntax

$n \in \mathbb{N}$	natural numbers	$c \in C$	class names
$re \in RE$	regular expressions	$md \in MD$	method names
$fd \in FD$	field names	$id \in ID$	object identifiers
$bi \in BI$	bytecode instructions	$mo \in MO$	modifiers
$dt \in DT$	data types	$pcn \in PCN$	pointcut names

$pcd ::=$		**pointcuts**
	`<call>`$msig$`</call>`	method calls
	`<execution>`$msig$`</execution>`	method executions
	`<get>`mo^* $c.fd$`</get>` | `<set>`mo^* $c.fd$`</set>`	field accesses
	`<argval num="`n`" [obj="`id`"]>`vp`</argval>`	stack args (values)
	`<argtyp num="`n`">`dt`</argtyp>`	stack args (types)
	`<staticinitialization>`c`</staticinitialization>`	static constructors
	`<handler>`c`</handler>`	exception handlers
	`<this [obj="`id`"]>[`c`]</this>`	*this* pointer references
	`<target [obj="`id`"]>[`c`]</target>`	target object references
	`<within>`c`</within>` | `<withincode>`$msig$`</withincode>`	lexical context
	`<cflow>`pcd`</cflow>`	control flows
	`<instr>`bi`</instr>`	bytecode instructions
	`<pointcutid name="`pcn`" />`	predefined pointcuts
	`<and>`pcd^*`</and>` | `<or>`pcd^*`</or>` | `<not>`pcd`</not>`	boolean combinators
	`<true />` | `<false />`	constants
$msig ::=$	mo^* $[dt]$ $c.md$ `[throws` c`[` c`[..]]]`	**method signatures**
$vp ::=$	`<true />` | `<isnull />` | `<streq>`re`</streq>`	**value predicates**
	`<inteq>`n`</inteq>` | `<intne>`n`</intne>`	
	`<intle>`n`</intle>` | `<intge>`n`</intge>`	
	`<intlt>`n`</intlt>` | `<intgt>`n`</intgt>`	

Figure 3.3. SPoX pointcut syntax

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

In the following paragraphs we define the language of edge labels (pointcuts), state labels (security-state variables), and transition relations supported by SPoX, along with an informal description of their semantics. A more formal treatment of the denotational semantics of SPoX along with strategies for enforcing SPoX policies are given in Section 3.3.

Pointcuts. SPoX expresses security automaton edge labels as pointcut designator expressions in the style of Aspect Oriented Programming (AOP) (Kiczales et al. 1997). A pointcut designator defines a set of program-states that constitute security-relevant events, where a program-state consists of the complete runtime memory image of the program including the stack, code, and program-counter (i.e., the next instruction to be executed). For easy machine parsing, SPoX expresses pointcut expressions in an XML syntax. For example, the pointcut expression

```
<and>
  <call>File.renameTo</call>
  <not><argval num="1" obj="x"><isnull /></argval></not>
</and>
```

denotes the set of all program-states in which the next instruction to be executed is a call to the `renameTo` method of a Java `File` object, and the first argument being passed to the method is non-null.

The language of pointcut expressions in Figure 3.2 includes support for method calls, field accesses, inspection of stack arguments, lexical contexts, boolean operators, and the cflow temporal operator from AspectJ (Kiczales et al. 2001). (Informally, a program state satisfies `<cflow>`p`</cflow>` if its call stack contains a frame satisfying p.) SPoX includes all pointcuts from AspectJ that do not concern advice (e.g., `adviceexecution`). In addition to AspectJ pointcuts, SPoX has an `<instr>` tag that can be used to identify any individual Java

bytecode instruction as a security-relevant operation. SPoX also supports predicates that test nullity of stack arguments, integer comparisons, and string regular-expression matching.

Security states. A *security state* in a SPoX policy is a set of security-state variables and their integer values. These sets are dynamic in size; state transitions can add or remove state variables as well as change the value of existing state variables. Security-state variables come in two varieties:

- *Global security-state variables* are members of every security state; they cannot be added or removed by state transitions.
- *Instance security-state variables* describe the security state of an individual runtime object. Such variables are added to the security state by program operations that create a new instance of a security-relevant object, and they are removed from the security state by program operations that destroy a security-relevant object.

Instance security-state variables allow SPoX specifications to express policies that include per-object security properties. For example, a policy could require that each `File` object can be read at most ten times by defining an instance security-state variable associated with each `File` object and defining state transitions that increment the object's security-state variable each time that individual `File` object is read (up to ten times). Global security-state variables allow SPoX specifications to express policies that include instance-independent security properties. For example, a policy could require that at most ten `File` objects may be created during the lifetime of the program by defining a global security-state variable that gets incremented each time any `File` object is created (up to ten times).

Security-state variables are not program variables; they are meta-variables declared by the SPoX specification purely for the purpose of defining the structure of a security automaton that encodes the policy to be enforced. However, rewriters might implement the security policy by reifying these meta-variables into the untrusted code and tracking their values at runtime. For example, a rewriter might add a new global runtime variable for each

global security-state variable, and might add a new object field for each instance security-state variable. The rewriter could then add new runtime operations that update these new program variables whenever security-relevant operations occur, and might consult their values to determine whether an impending operation is a policy violation.

The start state of a SPoX security automaton is the state that assigns 0 to all global security-state variables and that has no instance security-state variables (since no objects yet exist at program start). Each security-relevant program operation changes the current security state by adding or removing a finite set of zero or more instance security-state variables (corresponding to the finite set of security-relevant objects the operation creates or destroys at runtime). Security-relevant operations change the values of existing security-state variables (described in more detail below). Thus, each security state consists of a countably infinite set of security-state variables and their values, and the total number of security states in the automaton is at most countably infinite.

Security-state Transitions. Each edge in a security automaton is modeled as a triple (q_0, p, q_1), where q_0 is a source state, q_1 is a destination state, and p is an edge label. In SPoX, edge labels are pointcut expressions and states are sets of security-state variables and their values. Since SPoX security automata have a potentially infinite number of states, we allow a single `<edge>` element to introduce a possibly infinite set of edges to the automaton. For example, the following SPoX fragment introduces an edge from every state in which global variable g has value 3 to a corresponding state in which g has value 4 (and all other security-state variables are unchanged):

<edge>p<nodes var="g">3,4</nodes></edge>

The effect is that program operations matching pointcut expression p will change the security state of variable g from 3 to 4.

To refer to instance security-state variables, pointcut expressions declare *object identifiers* associated with the security-relevant arguments of operations that match the expression. For

example, the pointcut expression given by

```
<and>
  <call>File.renameTo</call>
  <and><argval num="0" obj="x"><true /></argval>
       <argval num="1" obj="y"><true /></argval></and>
</and>
```

declares two identifiers x and y that refer (respectively) to the `File` object whose `renameTo` method is about to be invoked and the object that is being passed as its first argument. One could then write

```
<nodes obj="x" var="v">0,1</nodes>
<nodes obj="y" var="v">0,0</nodes>
```

to specify that when the v security-state variable of both objects is 0, then the v security-state variable of the object whose method was invoked should change to 1, but that of the other object should remain unchanged. Note that `<nodes>` elements in an `<edge>` element are conjunctive. That is, a transition is introduced for each pair of states that satisfy *all* `<nodes>` elements given.

The security automata corresponding to many realistic security policies have repetitive, redundant structure. For example, the security automaton that permits at most 1000 *read* operations consists of 1001 states with edges from one to the next, each labeled *read*. To allow policy-writers to elegantly specify such structure, SPoX introduces a third kind of variable for iteration. As an example, the following fragment introduces the 1000 transitions

described above:

```
<forall var="i" from="0" to="999">
  <edge>
        read
        <nodes var="g">i,i+1</nodes>
  </edge>
</forall>
```

Here, `i` is an *iteration variable* that ranges from 0 to 999. Security-state variables and iteration variables can appear in simple arithmetic expressions (constants, addition, subtraction, multiplication, and division) to define the source and destination states of the transitions.

3.2.1 Pointcut Language

In this section we describe the different kinds of pointcuts from Figure 3.3. In general, the SPoX pointcut language closely resembles that of AspectJ (Kiczales et al. 2001).

Method invocations are statically matched by `<call>` pointcuts. Any number of modifiers (`public`, `static`, etc.) may be included, as well as an optional return type to narrow down the signature. The regular expression wildcard character "`*`" may also be used to, for example, match all methods in a class, or all class names with a given prefix or suffix. So `<call>public java.lang.*.clone</call>` matches all calls to public methods called `clone` in any class under the `java.lang` package, but *not* under one of its subpackages. As in AspectJ, the character sequence "`..`" denotes any sequence of subpackages, and "`+`" includes any subclasses. So `<call>java..HashMap+.*</call>` matches calls to any method belonging to any class named `HashMap` such that the top-level containing package is `java`, as well as any other class that inherits from that `HashMap`.

Method entrypoints are statically matched by `<execution>`. This is different from `<call>`, which matches callsites. Thus, `<execution>` pointcuts will frequently produce less enforcement code than `<call>`s, simply because the rewriter need only insert that code at

the top of the method and not around every call to that method. However, rewriters usually can't modify system libraries, so `<execution>` should not be used to describe library methods.

Field accesses are statically matched by `<get>` and `<set>`. These work much the same as `<call>`, but for field gets and sets, respectively.

The runtime values of method and field access arguments are dynamically matched by `<argval>`. The index of the argument to be considered is given by the `num` attribute, where index 0 refers to the target object of a virtual method call or field access and any index $n \geq 1$ refers to the nth argument. If an unavailable index is given (e.g., 0 for a static call, 3 for a 2-argument method call, or 1 for a field get operation), `<argval>` will always fail to match.

In order to consider runtime values, `<argval>` uses value predicates. The default predicate `<true />` matches everything, and is often used when the policy writer only wishes to access a specific instance state variable via the `obj` attribute. Object nullity is matched using `<isnull />`. Integer comparisons are possible through a series of integer predicates; for example, `<intge>`n`</intge>` matches runtime integer values greater than or equal to n. Finally, `<streq>` matches strings against regular expressions. Note that `<streq>` actually considers the result of an object's `toString` output, and is therefore useful even for objects that are not of type `String`.

Argument types are statically matched by `<argtyp>`. Like `<argval>`, the `num` attribute refers to the argument index, where the target object is at index 0. Note that because `<argtyp>` is a statically decidable pointcut, it only matches against the type described in the bytecode signature itself. For example, if a method call signature says that the first argument is of type `Number`, but the runtime variable being passed is of the inheriting type `Integer`, then `<argtyp num="1">Number</argtyp>` will match, but `<argtyp num="1">Integer</argtyp>` will not. The `<argtyp>` pointcut is often used to specify a method signature, for which dynamic matching is undesirable.

Executions of static class initializers are statically matched by `<staticinitialization>`. This pointcut is equivalent to an `<execution>` that refers to a Java class's `<clinit>` method. Executions of the class `C`'s instance constructor method `<init>` are matched by `<execution> C.new</execution>`.

Exception handler executions are statically matched by `<handler>`. As with `argtyp`, this pointcut only matches the exception type described in the bytecode.

The "this" pointer, which refers to the instance of the class within which code is executing at runtime, is dynamically matched by `<this>`. The pointcut is most commonly used to access the "this" pointer's instance state variable through the `obj` attribute. It may also optionally check the pointer against a dynamically decidable type.

The target object of a method call or a field access is dynamically matched by `<target>`. As with `<this>`, the policy writer may use it to access the object's instance state variable or its runtime type.

The lexical context of a join point is statically matched by `<within>` and `<withincode>`. The former only checks the containing class, while the latter may also check the containing method.

Join points in the control flow of a method are dynamically matched by `<cflow>`. That is, `<cflow><call>C.m</call></cflow>` matches all join points whenever method `C.m` is at any point on the runtime call stack. A `<cflow>` pointcut in SPoX may be removed and converted into equivalent constructions, as discussed in Section 4.2.1.

Specific bytecode instructions are statically matched by `<instr>`. The wildcard "`*`" may be used to describe sets of instructions, so `<instr>if*</instr>` matches all conditional branches in Java bytecode that begin with `if`.

Stored pointcuts may be referenced anywhere inside other pointcuts by `<pointcutid>`. For example, `<and><pointcutid name="pc" /><within>C</within></and>` matches all join points in class `C` that also match the predefined pointcut `pc`.

$o \in Obj$	objects
$v ::= o \mid \mathsf{null}$	values
$jp ::= \langle\rangle \mid \langle k, v^*, jp\rangle$	join points
$k ::= \texttt{call}\ c.md \mid \texttt{get}\ c.fd \mid \texttt{set}\ c.fd$	join kinds

Figure 3.4. Join points

Finally, pointcuts may be combined using the boolean operators `<and>`, `<or>`, and `<not>`. Note that `<and>` and `<or>` are not strictly binary. For example, `<and><true /><true /><true /></and>` is a legal (if trivial) pointcut.

3.3 Analysis

3.3.1 Denotational Semantics

In this section we define a formal semantics for SPoX that unambiguously identifies what a policy specification means, and what it means for a program to satisfy a SPoX policy. We begin by defining *join points* in Figure 3.4. Following the operational semantics of AOP (Wand et al. 2004), a join point is a recursive structure that abstracts the control stack. Join point $\langle k, v^*, jp\rangle$ consists of static information k found at the site of the current program instruction, dynamic information v^* consisting of the arguments about to be 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 $\langle\rangle$.

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*. Such an automaton accepts or rejects (possibly infinite) sequences of join points. A formal denotational semantics is provided in Figure 3.5. 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

$$
\begin{array}{ll}
q \in Q = (SV \uplus (Obj \times SV)) \to \mathbb{N} & \text{security states} \\
S \in SM = (SV \uplus (ID \times SV)) \rightharpoonup \mathbb{N} & \text{state-variable maps} \\
I \in IM = IV \rightharpoonup \mathbb{N} & \text{iteration var maps} \\
b \in Bnd = ID \rightharpoonup Obj & \text{bindings} \\
r \in OBnd = Bnd \uplus \{Fail\} & \text{optional bindings} \\
\mathcal{P} : pol \to \big(\Upsilon \times 2^Q \times \Upsilon \times ((Q \times JP) \to 2^Q)\big) & \text{policy denotations} \\
\mathcal{ES} : edg \to IM \to 2^{(JP \to OBnd) \times SM \times SM} & \text{edgeset denotations} \\
\mathcal{PC} : pcd \to JP \to OBnd & \text{pointcut denotations} \\
\mathcal{EP} : s \to IM \to (SM \times SM) & \text{endpoint constraints} \\
\mathcal{E} : e \to IM \to \mathbb{N} & \text{arithmetic expressions}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{P}[\![edg_1 \ldots edg_n]\!] = (Q, \{q_0\}, JP, \delta) \\
\quad \text{where } q_0 = (SV \uplus (Obj \times SV)) \times \{0\} \\
\quad \text{and } \delta(q, jp) = \{q[S'[b]] \mid (f, S, S') \in \cup_{1 \le i \le n} \mathcal{ES}[\![edg_i]\!]\bot, \\
\qquad\qquad f(jp) = b,\ S[b] \sqsubseteq q\} \\
\mathcal{ES}[\![\texttt{<forall var="}iv\texttt{" from="}e_1\texttt{" to="}e_2\texttt{">}edg\texttt{</forall>}]\!]I = \\
\quad \cup_{\mathcal{E}[\![e_1]\!]I \le j \le \mathcal{E}[\![e_2]\!]I}\ \mathcal{ES}[\![edg]\!](I[j/iv]) \\
\mathcal{ES}[\![\texttt{<edge>}pcd\ ep_1 \ldots ep_n\texttt{</edge>}]\!]I = \\
\quad \big\{(\mathcal{PC}[\![pcd]\!],\ \sqcup_{1 \le j \le n} S_j,\ \sqcup_{1 \le j \le n} S'_j)\big\} \\
\quad \text{where } \forall j \in \mathbb{N}\,.\,(1 \le j \le n) \Rightarrow ((S_j, S'_j) = \mathcal{EP}[\![ep_j]\!]I) \\
\mathcal{PC}[\![pcd]\!]jp = match\text{-}pcd(pcd)jp \\
\mathcal{EP}[\![\texttt{<nodes var="}sv\texttt{">}e_1\texttt{,}e_2\texttt{</nodes>}]\!]I = \\
\quad \big(\{(sv, \mathcal{E}[\![e_1]\!]I)\}, \{(sv, \mathcal{E}[\![e_2]\!]I)\}\big) \\
\mathcal{EP}[\![\texttt{<nodes obj="}id\texttt{" var="}sv\texttt{">}e_1\texttt{,}e_2\texttt{</nodes>}]\!]I = \\
\quad \big(\{((id, sv), \mathcal{E}[\![e_1]\!]I)\}, \{((id, sv), \mathcal{E}[\![e_2]\!]I)\}\big)
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{E}[\![n]\!]I = n & \mathcal{E}[\![e_1\texttt{+}e_2]\!]I = \mathcal{E}[\![e_1]\!]I + \mathcal{E}[\![e_2]\!]I \\
\mathcal{E}[\![iv]\!]I = I(iv) & \mathcal{E}[\![e_1\texttt{-}e_2]\!]I = \mathcal{E}[\![e_1]\!]I - \mathcal{E}[\![e_2]\!]I \\
 & \mathcal{E}[\![e_1\texttt{*}e_2]\!]I = \mathcal{E}[\![e_1]\!]I \cdot \mathcal{E}[\![e_2]\!]I \\
 & \mathcal{E}[\![e_1\texttt{/}e_2]\!]I = \mathcal{E}[\![e_1]\!]I / \mathcal{E}[\![e_2]\!]I
\end{array}
$$

Figure 3.5. Denotational semantics for SPoX

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 $f[g] = \{(x, f(x)) \mid x \in f^{\leftarrow} \backslash g^{\leftarrow}\} \sqcup g$ to denote the replacement of assignments in f with those in g. When $S \in SM$ is a state-variable map and $b \in Bnd$ is a binding of object identifiers to objects, we let $S[b]$ denote the partial function that results from substituting b into the domain of S:

$$\begin{aligned} S[b] = &\{((b(id), sv), n) \mid ((id, sv), n) \in S\} \sqcup \\ &\{(sv, n) \mid (sv, n) \in S\} \end{aligned}$$

Security automata (Schneider 2000) are modeled in the literature 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).

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 process of matching a pointcut designator to a join point binds identifiers in the pointcut to runtime objects in the program state abstracted by the join point. Thus, pointcut designators denote mappings from join points to bindings. The denotational semantics in Figure 3.5 defines this matching process in terms of the *match-pcd* function from the operational semantics of AspectJ (Wand et al. 2004). We adapt this definition to SPoX syntax in Figure 3.6.

Defining what it means for a program to satisfy a policy requires an operational semantics that defines what it means to execute a program. For this purpose we adopt Flatt, Krishnamurthi and Felleisen's small-step operational semantics of ClassicJava, as defined in (Flatt et al. 1998) (with two minor changes introduced below). ClassicJava programs consist

$$match\text{-}pcd(\texttt{<call>}c.md\texttt{</call>})\langle \texttt{call } c.md, v^*, jp\rangle = \bot$$

$$match\text{-}pcd(\texttt{<get>}c.fd\texttt{</get>})\langle \texttt{get } c.fd, v^*, jp\rangle = \bot$$

$$match\text{-}pcd(\texttt{<set>}c.fd\texttt{</set>})\langle \texttt{set } c.fd, v^*, jp\rangle = \bot$$

$$match\text{-}pcd(\texttt{<argval num="}n\texttt{" obj="}id\texttt{">}vp\texttt{</argval>})\langle k, v_0 \cdots v_n \cdots, jp\rangle$$
$$= \{(id, v_n)\} \text{ if } vp = \texttt{<true />} \text{ or } (vp = \texttt{<isnull />} \text{ and } v_n = \texttt{null})$$

$$match\text{-}pcd(\texttt{<and>}pcd_1 pcd_2\texttt{</and>})jp =$$
$$match\text{-}pcd(pcd_1)jp \wedge match\text{-}pcd(pcd_2)jp$$

$$match\text{-}pcd(\texttt{<or>}pcd_1 pcd_2\texttt{</or>})jp =$$
$$match\text{-}pcd(pcd_1)jp \vee match\text{-}pcd(pcd_2)jp$$

$$match\text{-}pcd(\texttt{<not>}pcd\texttt{</not>})jp = \neg match\text{-}pcd(pcd)$$

$$match\text{-}pcd(\texttt{<cflow>}pcd\texttt{</cflow>})\langle k, v^*, jp\rangle =$$
$$match\text{-}pcd(pcd)\langle k, v^*, jp\rangle \vee match\text{-}pcd(\texttt{<cflow>}pcd\texttt{</cflow>})jp$$

$$match\text{-}pcd(pcd)jp = Fail \quad \text{otherwise}$$

$$b \vee r = b \qquad Fail \vee r = r$$

$$Fail \wedge r = Fail \qquad b \wedge Fail = Fail \qquad b \wedge b' = b \sqcup b'$$

$$\neg Fail = \bot \qquad \neg b = Fail$$

Figure 3.6. Matching pointcuts to join points

$$
\begin{array}{rll}
P & ::= & def^*(\textsf{let } input = v \textsf{ in } e) & \text{programs} \\
def & ::= & \textsf{class } c \{ \ldots \} \mid \ldots & \text{class defs} \\
e & ::= & v \mid se \mid mc \mid \textsf{ret}_{c.md(v^*)} e & \text{expressions} \\
se & ::= & \textsf{new } c \mid var & \text{simple expr} \\
 & & \mid (e : c).fd \mid (e : c).fd = e & \\
 & & \mid \textsf{view } c\ e \mid \textsf{let } var = e \textsf{ in } e & \\
mc & ::= & e.md(e^*) \mid (\textsf{super} \equiv \textsf{this} : c).md(e^*) & \text{method calls} \\
\mathsf{E} & ::= & [\,] \mid (\mathsf{E} : c).fd & \text{eval contexts} \\
 & & \mid (\mathsf{E} : c).fd = e \mid (v : c).fd = \mathsf{E} & \\
 & & \mid \mathsf{E}.md(e^*) \mid v.md(v^* \mathsf{E} e^*) & \\
 & & \mid (\textsf{super} \equiv v : c).md(v^* \mathsf{E} e^*) & \\
 & & \mid \textsf{view } c\ \mathsf{E} \mid \textsf{let } var = \mathsf{E} \textsf{ in } e &
\end{array}
$$

Figure 3.7. Syntax of ClassicJava with **ret** (CJR)

$$\frac{P \vdash_{CJ} \langle \mathbf{E}[se], S\rangle \hookrightarrow \langle \mathbf{E}[e'], S'\rangle}{P \vdash_{CJR} \langle \mathbf{E}[se], S\rangle \hookrightarrow \langle \mathbf{E}[e'], S'\rangle}$$

$$\frac{P \vdash_{CJ} \langle \mathbf{E}[o.md(v^*)], S\rangle \hookrightarrow \langle \mathbf{E}[e'], S'\rangle \quad S(o) = \langle c, F\rangle}{P \vdash_{CJR} \langle \mathbf{E}[o.md(v^*)], S\rangle \hookrightarrow \langle \mathbf{E}[\mathbf{ret}_{c.md(v^*)} e'], S'\rangle}$$

$$\frac{P \vdash_{CJ} \langle \mathbf{E}[(\mathbf{super} \equiv \mathbf{this}\!:\!c).md(v^*)], S\rangle \hookrightarrow \langle \mathbf{E}[e'], S'\rangle}{P \vdash_{CJR} \langle \mathbf{E}[(\mathbf{super} \equiv \mathbf{this}\!:\!c).md(v^*)], S\rangle \hookrightarrow \langle \mathbf{E}[\mathbf{ret}_{c.md(v^*)} e'], S'\rangle}$$

$$P \vdash_{CJR} \langle \mathbf{E}[\mathbf{ret}_{c.md(v^*)} v], S\rangle \hookrightarrow \langle \mathbf{E}[v], S\rangle$$

$$\frac{P \vdash_{CJR} \langle \mathbf{E}[e], S\rangle \hookrightarrow \langle \mathbf{E}[e'], S'\rangle}{P \vdash_{CJR} \langle \mathbf{E}[\mathbf{ret}_{c.md(v^*)} e], S\rangle \hookrightarrow \langle \mathbf{E}[\mathbf{ret}_{c.md(v^*)} e'], S'\rangle}$$

Figure 3.8. Operational semantics of CJR in terms of those for ClassicJava

of a sequence of class and interface declarations followed by an entrypoint expression that models the program's main method. Small-step judgment $P \vdash_{CJ} \langle e, S\rangle \hookrightarrow \langle e', S'\rangle$ asserts that in program P, expression e in store S evaluates to new expression e' and new store S'. Stores $S : Obj \to (c \times F)$ map objects to class-tagged field records. A partial syntax is provided in Figure 3.7 for the reader's convenience; for the full syntax and semantics the reader is invited to consult (Flatt et al. 1998).

The syntax in Figure 3.7 differs from that in (Flatt et al. 1998) in two important respects. First, to model program input we adopt the convention that the entrypoint expression must have the form (**let** *input* $= v$ **in** e), where *input* is a reserved variable name and value v is the input supplied to the program. Thus, in our treatment each program P actually denotes the equivalence class of ClassicJava programs obtained by substituting value v with any other value of equivalent type. Second, we introduce the expression $\mathbf{ret}_{c.md(v^*)}\ e$, which indicates that subexpression e is the (partially reduced) body of method md of class c that was called with values v^* as parameters. These **ret** expressions make method-returns explicit. They do not affect expression evaluation but they make it possible to recover the runtime call-stack from a partially reduced expression, which is necessary for matching expressions to pointcut designators in our analysis.

$$
\begin{aligned}
&J : (e \times jp) \to jp \\
&J(\langle \mathbf{E}[(o\!:\!c).fd], S\rangle, jp) = \langle \texttt{get}\ c.fd, S(o), jp\rangle \\
&J(\langle \mathbf{E}[(o\!:\!c).fd = v], S\rangle, jp) = \langle \texttt{set}\ c.fd, S(o), jp\rangle \\
&J(\langle \mathbf{E}[o.md(v^*)], S\rangle, jp) = \langle \texttt{call}\ c.md, o\, v^*, jp\rangle \\
&\qquad \text{where } S(o) = \langle c, F\rangle \\
&J(\langle \mathbf{E}[(\textbf{super} \equiv o\!:\!c).md(v^*)], S\rangle, jp) = \langle \texttt{call}\ c.md, o\, v^*, jp\rangle \\
&J(\langle \mathbf{E}[\textbf{ret}_{c.md(v^*)} e], S\rangle, jp) = J(\langle e, S\rangle, \langle \texttt{call}\ c.md, v^*, jp\rangle) \\
&J(\langle e, S\rangle, jp) = \langle\rangle \text{ for all other } e
\end{aligned}
$$

Figure 3.9. Mapping partially reduced CJR expressions to join points

Hereafter we refer to our modified language as CJR (ClassicJava with **ret**). Figure 3.8 defines the small-step operational semantics of CJR in terms of those for ClassicJava.

Some (but not all) CJR configurations $c = \langle e, S\rangle$ are join points. In the context of SPoX policies, we consider join points to be abstractions of security-relevant program states. Function $J(c, \langle\rangle)$ yields the join point that abstracts configuration c (or the empty join point $\langle\rangle$ if c is not security-relevant). We lift J to configuration sequences χ and to sets X of configuration sequences in the obvious ways: $J(c_1 c_2 \cdots) = J(c_1, \langle\rangle)J(c_2, \langle\rangle)\cdots$ and $J(X) = \{J(\chi) \mid \chi \in X\}$.

Armed with these definitions we are finally able to formally define what it means for a CJR program to satisfy a SPoX policy:

Definition 3.3.1 (Executions)**.** *Let* $P = def^*$ (**let** *input* $= v$ **in** e) *be a well-typed CJR program. An* execution χ *of* P *is a finite or countably infinite sequence of configurations* $\langle e_0, S_0\rangle\langle e_1, S_1\rangle\cdots$ *such that* $e_0 = e$, $S_0 = \{(input, v_0)\}$ *where* v_0 *has the same type*[2] *as* v, *and for all* $i < length(\chi) - 1$, $P \vdash \chi_i \hookrightarrow \chi_{i+1}$ *holds. Furthermore, if* χ *is finite then there*

[2]Formally, there exists a type τ such that ClassicJava typing judgments (Flatt et al. 1998) $P, \{\} \vdash_e v \Rightarrow v' : \tau$ and $P, \{\} \vdash_e v_0 \Rightarrow v_0' : \tau$ both hold.

exists no configuration $\langle e_n, S_n \rangle$ *satisfying* $P \vdash \chi_{length(\chi)-1} \hookrightarrow \langle e_n, S_n \rangle$. *We denote the set of all executions of* P *by* X_P.

Definition 3.3.2 (Policy-adherence)**.** *A CJR program* P *satisfies SPoX policy pol if and only if* $J(X_P) \subseteq L(\mathcal{P}[\![pol]\!])$ *holds, where* $L(A)$ *denotes the language accepted by security automaton* A.

The above asserts that program P satisfies policy *pol* if and only if every execution of P is accepted by the Aspect-oriented security automaton that *pol* denotes. Thus, executing P will never result in a security violation.

3.3.2 Policy Enforcement

Most (but not all) SPoX policies can be enforced by an IRM system through the insertion of dynamic security checks into the untrusted code. Dynamic checks are required in general because many SPoX policies are not statically decidable. In particular, SPoX policies that involve predicates on runtime values (e.g., those with `<argval>`) will typically not be statically decidable since the general problem of deciding whether an arbitrary runtime value will satisfy an arbitrary predicate is equivalent to the halting problem. However, we argue in this section that SPoX policies are dynamically decidable, and we sketch a simple algorithm for inserting runtime checks into untrusted code to detect policy violations before they occur. This algorithm is the basis for our prototype implementation of SPoX.

A means of *detecting* impending policy violations before they occur is not always sufficient for an In-lined Reference Monitor to *enforce* the policy, however. The IRM can still fail if the decision algorithm for detecting policy violations commits a security violation when executed as part of the untrusted code. For example, the SPoX policy `<and>`p`<not>`p`</not></and>` (where p is any pointcut) is unsatisfiable, rejecting all program states. Therefore, there is no code that a rewriter could insert that would not itself violate the security policy. Unsatisfiable policies are a trivial example of this problem but there are more realistic policies that present

similar difficulties. For example, the policy

```
<edge>
  <not><call>System.exit</call></not>
  <nodes var="g">0,#</nodes>
</edge>
```

rejects any program that aborts execution prematurely. This restriction effectively disallows the use of rewriting strategies that halt execution when policy violations are imminent, as the enforcement operation itself violates the policy.

In what follows we make the simplifying assumption that code inserted as part of the detection algorithm is not security-relevant. In practice, a certifying In-lined Reference Monitoring system can check this assumption by using the denotational semantics in Section 3.3.1 to verify code produced by the rewriter. Rewritten code that fails verification is rejected to prevent a security violation. Our prototype implementation discussed in Chapter 4 checks this conservatively by statically verifying that no rewriter-inserted operations satisfy any pointcut expression in the policy; thus, no inserted operations are security-relevant. More precise (but still conservative) verification algorithms are obviously possible (e.g., see (Hamlen et al. 2006; Aktug and Naliuka 2008)); we leave the development of such a system to future work. To simplify the discussion, we also limit our attention in this section to policies that model deterministic security automata. Non-deterministic automata could be modeled by tracking sets of states at runtime instead of individual states.

As outlined in Section 3.2, an IRM can track a program's security state at runtime by reifying security-state variables into the untrusted code. In particular, consider the following (non-optimized) rewriting procedure:

1. Inject a new `SecurityState` class into the untrusted code with a static field for each global security-state variable and an instance field for each instance security-state variable.

$G(edg_1 \ldots edg_n)jp =$
```
  guard: do {
    ES(edg_1)jp  ...  ES(edg_n)jp
    System.exit(1);
  } while (false);
```
$ES($`<forall var="`iv`" from="`e_1`" to="`e_2`">`edg`</forall>`$)jp =$
```
  if (e_1<=e_2)
    for (int iv=e_1; iv<=e_2; ++iv) { ES(edg)jp }
```
$ES($`<edge>`$pcd\ ep_1 \ldots ep_n$`</edge>`$)jp =$
```
  b = match-pcd(pcd)jp;
  if ((b!=Fail) && EP(ep_1) && ... && EP(ep_n) ) {
    EP'(ep_1)  ...  EP'(ep_n)
    break guard;
  }
```
$EP($`<nodes var="`sv`">`e_1`,`e_2`</nodes>`$) =$
```
  (SecurityState.sv == e_1)
```
$EP($`<nodes obj="`id`" var="`sv`">`e_1`,`e_2`</nodes>`$) =$
```
  (b(id).sv == e_1)
```
$EP'($`<nodes var="`sv`">`e_1`,`e_2`</nodes>`$) =$
```
  SecurityState.sv = e_2;
```
$EP'($`<nodes obj="`id`" var="`sv`">`e_1`,`e_2`</nodes>`$) =$
```
  b(id).sv = e_2;
```

Figure 3.10. Java pseudo-code for a rewriting algorithm for SPoX

2. Rewrite each instruction that manipulates an object of type $c \in C$ to instead manipulate an object pair of type $c \times$ `SecurityState`, where the first member of the pair is the original object and the second member models the object's security state. This expands each original instruction into a *chunk* of one or more rewritten instructions.

3. To each chunk, prepend an instruction sequence that first computes $jp = J(\langle e, S\rangle, \langle\rangle)$, where J is defined in Figure 3.9 and $\langle e, S\rangle$ is the current program state; followed by instruction sequence $G(pol)jp$, where pol is the policy and G is defined in Figure 3.10.

4. Finally, rewrite all static jumps in the original program to target the beginning of whichever chunk contains their destination addresses. (The only computed jumps in Java are method returns, which need not be rewritten.)

The rewriting procedure described above enforces a security policy by inserting a runtime security check before each program operation. When an impending security violation is detected, the program is prematurely terminated. This is only one method of rewriting untrusted code to enforce SPoX policies; clearly many other approaches also exist. For example, instead of premature termination, rewriters could implement other remedial actions such as event suppression, checkpointing with roll-back, or specific corrective operations specified by advice external to the policy.

The simple rewriting algorithm presented here does not produce particularly efficient code, but the code it produces can be significantly optimized through partial evaluation. For example, for many policies it can be statically determined that most instructions in the untrusted code are not security-relevant (e.g., they match no pointcut expression in the policy). For those instructions the code defined in Figure 3.10 partially evaluates to an empty instruction sequence. Thus, in practice a rewriter typically only inserts a few runtime security checks around the few program operations that might be security-relevant.

The code in Figure 3.10 can be optimized further by replacing the for-loops with a more efficient integer linear programming algorithm. In particular, each set of n nested `<forall>` elements in a SPoX policy that surround m `<nodes>` elements defines a rational polytope

$T \subseteq \mathbb{Q}^n$ with $2(n+m)$ linear constraints. To decide whether the current runtime security state matches the source state of any edge defined by this structure it suffices to decide whether feasible region T contains an integer lattice point. (See (Barvinok and Pommersheim 1999) for a summary of efficient algorithms for computing this.) In the common case where each `<forall>` and `<nodes>` element refers to at most one iteration variable, polytope T is a box, and therefore the problem can be trivially decided with $2n$ integer inequality tests and no loops.

The simple rewriting algorithm outlined in this section might fail when applied to Java code that is self-modifying or multi-threaded. Self-modifying code can be supported by adding the rewriter to the load path of the Java virtual machine, so that it can transform any modified code at runtime before it is first executed. Multi-threaded code can be supported by making each chunk produced by the rewriting algorithm atomic. Each of these solutions might have an adverse effect on performance and is worthy of further study.

CHAPTER 4
REWRITER IMPLEMENTATION[1]

4.1 Overview

We have implemented an application that rewrites Java bytecode programs in accordance with a SPoX policy specification by in-lining runtime security checks into the untrusted code. The application is written in Java, and uses Apache's BCEL API (Apache Software Foundation 2006) to read and write bytecode binaries. Discounting library code, the rewriter consists of about 13000 lines of Java source code.

The chapter proceeds as follows. Interesting components of our rewriter implementation are discussed in Section 4.2, including how it handles the parsing of SPoX policies (Section 4.2.1) and rewriting Java binaries (Section 4.2.2). Three realistic case studies are provided in Section 4.3, showing the practicality and efficiency of the rewriter and the self-monitoring code it outputs. Finally, Section 4.4 discusses limitations and potential for future work.

4.2 Implementation Discussion

4.2.1 Parsing SPoX

As SPoX is XML-based, it is easy to parse using commonly available libraries. Our implementation uses Java's built-in XML parsing libraries under the `javax.xml` packages. This provides all the benefits native to XML, including its support for importing of external XML files via `<xi:include>` tags. Because SPoX allows the use of predefined pointcuts, it is pos-

[1]This chapter includes previously published (Hamlen and Jones 2008; Jones and Hamlen 2009) joint work with Kevin Hamlen.

```
1 <state name="c" />
2 <forall var="i" from="0" to="MAXINT">
3   <edge name="cflowinc">
4     p
5     <nodes var="c">i,i+1</nodes>
6   </edge>
7   <edge name="cflowdec" after="true">
8     p
9     <nodes var="c">i+1,i</nodes>
10  </edge>
11 </forall>
```

Figure 4.1. SPoX fragment for <cflow>*p*</cflow>

sible to define large, complex pointcut libraries and import those into a smaller core policy file (see the Columba example in Section 4.3).

The implementation converts the policy XML tree into corresponding recursive Java class objects. We take advantage of Java's typing system in our data structures, so that pointcut objects such as `Call` and `ArgVal` are all subtypes of `Pointcut`.

Our management of the pointcut <cflow> is critical in the rewriter implementation as well as our work on policy disambiguation (Chapter 5) and certification (Chapter 7). Pointcuts of the form <cflow>*p*</cflow> match join points whose call stacks contain a frame matching pointcut p. During parsing, each such pointcut is replaced by the policy fragment seen in Figure 4.1.[2] Control flow operators in edges are then replaced with equivalent <nodes> elements that stipulate an equivalent condition (e.g., `c` ≥ 1). This translation of <cflow> pointcuts into other SPoX constructions greatly reduces the complexity of enforcing them at rewrite time, as injecting code that tracks the runtime call stack is nontrivial.

[2]In Figure 4.1, `c` is a thread-local state variable and is instantiated with a unique name for each cflow instance in the policy.

```
<state name="s" />

<forall var="i" from="0" to="9">
  <edge name="count">
    <call>Mail.send</call>
    <nodes var="s">i,i+1</nodes>
  </edge>
</forall>

<edge name="10emails">
  <call>Mail.send</call>
  <nodes var="s">10,#</nodes>
</edge>
```

Figure 4.2. A policy permitting at most 10 email-send events

4.2.2 Rewriting

The implementation linearly scans through every class and method in the given Java program, looking for code points that match any pointcut in the policy definition. Wherever a security-relevant join point is discovered, new code is placed before and/or after it to consult the reified security state and make changes depending on its current value. If a policy violation is determined to be imminent, the program is aborted using a call to `System.exit(1)`.

Consider the example policy in Figure 4.2, which prohibits an application from calling `Mail.send` more than 10 times. When given that policy, the rewriter will inject the code in Figure 4.3 prior to every call to `Mail.send` in the target program. Note that this example provides source code purely for simplicity; the rewriter translates enforcement instructions down to the bytecode level.

Observe that in this example security state `s` has been reified as two separate fields of class `Policy`—`s` and `temp_s`. The double reification is part of a mechanism for resolving potential interference in the enforcement code. Had there been no `temp_s` with which to store state changes prior to their finalization, the first alteration to the reified state variable

```
1 if (Policy.s >= 0 && Policy.s <= 9)
2    Policy.temp_s = Policy.s + 1;
3 if (Policy.s == 10)
4    System.exit(1);
5 Policy.s = Policy.temp_s;
6
7 Mail.send();
```

Figure 4.3. Enforcement code for the policy in Figure 4.2

could affect the result of later checks on the same variable (here, setting `Policy.s = 10` too early would trigger premature termination).

For dynamically decidable pointcuts, more complex code insertions are necessary. Consider the pointcut that guards against SQL injection attacks by comparing a text argument to a regular expression:

```
<and>
   <call>Database.query</call>
   <not><argval num="1"><streq>[a-zA-Z0-9]*</streq></argval></not>
</and>
```

In addition to checking state variable preconditions, guard code must also check the runtime value of the first argument in any call to `Database.query`. For string regular expression comparisons of this sort, we use the JVM library's own methods to perform the comparison. If the security-relevant call is of the form `Database.query(x)`, the conditional will be as follows:

```
if (regex.Pattern.matches("[a-zA-Z0-9]*",x.toString())) { ... }
```

Whereas global state variables may be reified in Java as static fields, reifications of instance state variables must be generated and mapped to every corresponding runtime object. There are many ways to accomplish this, some more complex than others. Our implementation handles instance states for non-library classes by directly reifying state variables as

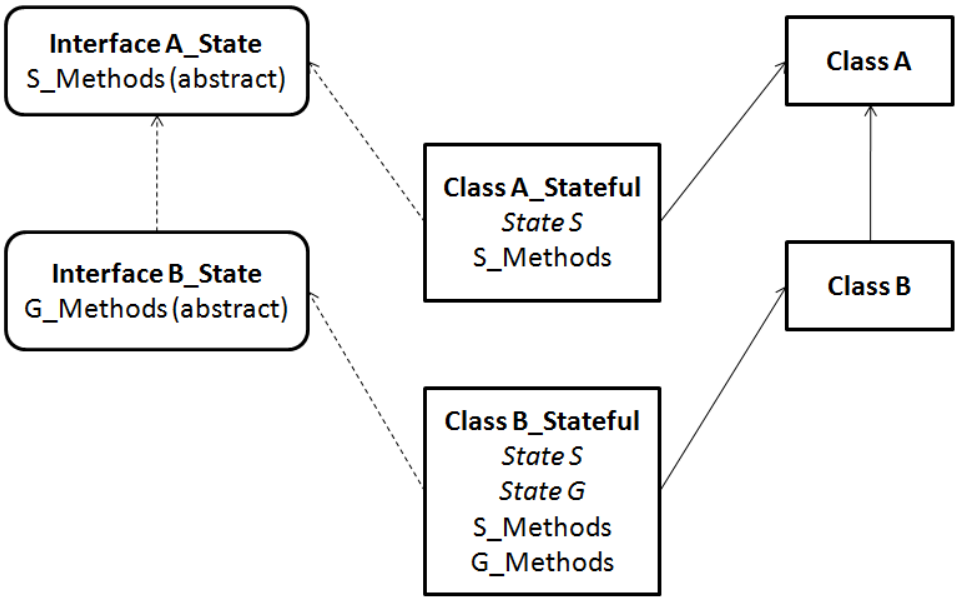

Figure 4.4. Class structure for reification of instance state variables `s` (tied to library class `A`) and `g` (tied to library class `B`, which inherits from `A`).

private instance fields in the class definitions themselves. Library classes, however, are not rewritten and are thus far more complex to manage.

Instance state variables for library classes are reified as instance state fields in new, inheriting classes. Wherever the original program created an instance of library class `A`, the rewritten program instead creates an instance of a new class `A_Stateful`, which inherits from `A` and contains all corresponding reified instance state fields. For any class `B` that is a descendant of `A`, another class `B_Stateful` is also created, which inherits from `B` and contains the reified state fields defined for `A_Stateful` along with any other instance states that correspond specifically to `B`. To maintain the ability to reference reified fields at any point in the class hierarchy (e.g., if `B_Stateful` is upcast to `A`, but we need to check `A`'s instance state variable `s`), we include interfaces for every `Stateful` class. These interfaces emulate the original class hierarchy and provide accessor methods for each reified state field (so `A` can always be cast to its interface `A_State` to get to state variable `s` whether it is in an

instance of `A_Stateful` or `B_Stateful`). Figure 4.4 shows the class structure for an example scenario.

4.3 Case Studies

Here we discuss three case studies. The first two show how SPoX can be used to enforce practical SPoX policies on an email client and a file-sharing client. The third uses a processor-intensive benchmarking program to show the effect guard instructions can have on runtime efficiency.

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.

4.3.1 Columba Email Client

Columba is an open-source Java email application. For this program, we enforced a policy that prevents creations or accesses of files whose names end in `.exe`. Such a policy is useful for inhibiting viruses propagation through email attachments.

Effectively enforcing this policy requires constraining calls to Java library methods that access the file system. The number of such methods is surprisingly vast; it includes methods that stream data, those that create and manipulate SQL databases (since those databases reside in files that could have a prohibited name), etc. Listing all such methods would be difficult (and error-prone) for the average administrator, so we developed a pointcut library that identifies these method calls. The following is an excerpt:

```
<pointcut name="fileMethods">
  <or>
    <call>java.io.File.new</call>
    <call>java.io.FileWriter.new</call>
    <call>java.io.FileReader.new</call>
```

```
    ...
  </or>
</pointcut>
```

SPoX's aspect-oriented design allows such libraries to be maintained modularly by a trusted expert. Policy-writers can then refer to these libraries to compose higher-level, application-specific policies.

To precisely enforce the policy, the specification must constrain the runtime arguments to these calls. For this we use SPoX's `argval` predicate:

```
<or>
  <and><pointcutid name="fileMethods" />
       <or><argval num="1"><streq>.*\.exe.*</streq></argval>
           <argval num="2"><streq>.*\.exe.*</streq></argval></or></and>
  <call>java.lang.Runtime.exec</call>
</or>
```

In addition to constraining the arguments of file-related methods, we also prohibited all use of Java's `java.lang.Runtime.exec` method. This is a standard addition to many of the policies we have enforced because this method can be used to execute an arbitrary (untrusted) external application.

The original Columba JAR file was approximately 2.8MB in size and rewriting added about 112K. Analysis found 403 security-relevant join points across 1702 class files. The rewriting process took approximately 84 seconds. The runtime performance of the rewritten code could not be measured formally because Columba requires user interaction when executed; however, we did not observe any noticeable performance overhead due to the inserted security checks. Likewise, no behavioral change to the application was observed except in the event of a policy violation—accessing a file with a `.exe` extension resulted in premature termination of the application as intended.

We did encounter one interesting side-effect after applying the policy: the spell checker for email composition no longer worked. After examining the disassembled bytecode of the

rewritten program, we found that this feature used `Runtime.exec` to launch an external spell check application, which was blocked by the policy. Had we wished to allow this program to run anyway, we could have modified the policy to whitelist certain application names as follows:

```
<and><call>java.lang.Runtime.exec</call>
     <not><argval num="1"><streq>filename</streq></argval></not></and>
```

where `filename` is a regular expression denoting legal file names. For modularity, the list of trusted executables could be maintained as a separate pointcut library.

The policy specification above defends against certain malware propagation attacks, but has several deficiencies that could be remedied by a more sophisticated specification. For example, one could easily extend the regular expressions to prohibit other dangerous file extensions. In addition, our `fileMethods` library was somewhat informally derived. A more rigorous examination of the Java runtime libraries would likely uncover additions to the library that would be necessary to rule out all possible violations.

4.3.2 XNap Peer-to-Peer Filesharing Client

XNap is an open-source file-sharing client implemented in Java. We enforced an anti-freeriding policy that requires the number of downloads to be at most two larger than the number of uploads during a given session. This is an interesting policy because it is both history-based and application-specific. In addition, the security state space is potentially large—one security state for each possible difference between the number of downloads and uploads.

To enforce the policy, we wrote the specification seen in Figure 4.5. The policy effectively creates a counter where additions to the download queue increment the value of state variable `s` and additions to the upload queue decrement it. A user is allowed to download two more files than he has uploaded; if he tries to download any more than that, a policy violation is triggered and the program halts.

```
<pointcut name="download">
  <and><call>xnap.util.DownloadQueue.add</call>
       <not><argtyp num="2">boolean</argtyp></not></and></pointcut>
<pointcut name="upload">
  <and><call>xnap.util.UploadQueue.add</call>
       <argtyp num="1">xnap.net.IUploadContainer</argtyp></and></pointcut>

<state name="s" />
<forall var="i" from="-10000" to="1">
  <edge name="download">
    <pointcutid name="download" />
    <nodes var="s">i,i+1</nodes></edge></forall>
<forall var="i" from="-9999" to="2">
  <edge name="upload">
    <pointcutid name="upload" />
    <nodes var="s">i,i-1</nodes></edge></forall>
<edge name="too_many_downloads">
  <pointcutid name="download" />
  <nodes var="s">2,#</nodes></edge>
```

Figure 4.5. SPoX policy to limit P2P freeriding

Since this is an application-specific policy, its formulation required some knowledge of the internal structure of the application; however, this was easily gleaned without any access to the application source code. We pinpointed the relevant methods for download and upload operations via a cursory examination of the bytecode disassembly.

The original program was 1290K in size, and rewriting actually *reduced* it by 68K. This is actually common in our tests, as unneeded metadata is stripped from rewritten bytecode. 7 join points were found across 878 class files, and rewriting took 58 seconds in total.

We tested the rewritten application with various combinations of downloads and uploads. As expected, XNap halted whenever we downloaded three more files than we had uploaded. Aside from this, there were no observable differences in program execution.

We are aware of at least two important deficiencies in this policy as we have defined it here. First, since the security state does not persist across application instances, users can exit out of the application and restart it after every two downloads to reset the counter and thereby increase downloads over uploads over time. Previous work has demonstrated how to implement persistent security state in an IRM to remedy such vulnerabilities (Aktug and Naliuka 2008). Second, our policy only tallies queued downloads and uploads but not completed ones. A malicious user could freeride by cancelling queued uploads before completion. To close this vulnerability, we could have added extra logic regarding download completions and cancellations.

4.3.3 SciMark Benchmarking Tool

To measure the runtime overhead introduced by the rewriter we applied a security policy to the SciMark benchmark suite and measured the performance of each benchmark before and after rewriting. SciMark includes five processor-intensive mathematical routines: Fast Fourier Transform (FFT), Jacobi Successive Over-Relaxation (JOR), Monte Carlo integration (MCI), sparse matrix multiplication (SMM), and dense LU matrix factorization (LU). The policy we applied prohibits a program from performing more than n floating point multiplication operations during its lifetime, where we set n to an unreachably high number to

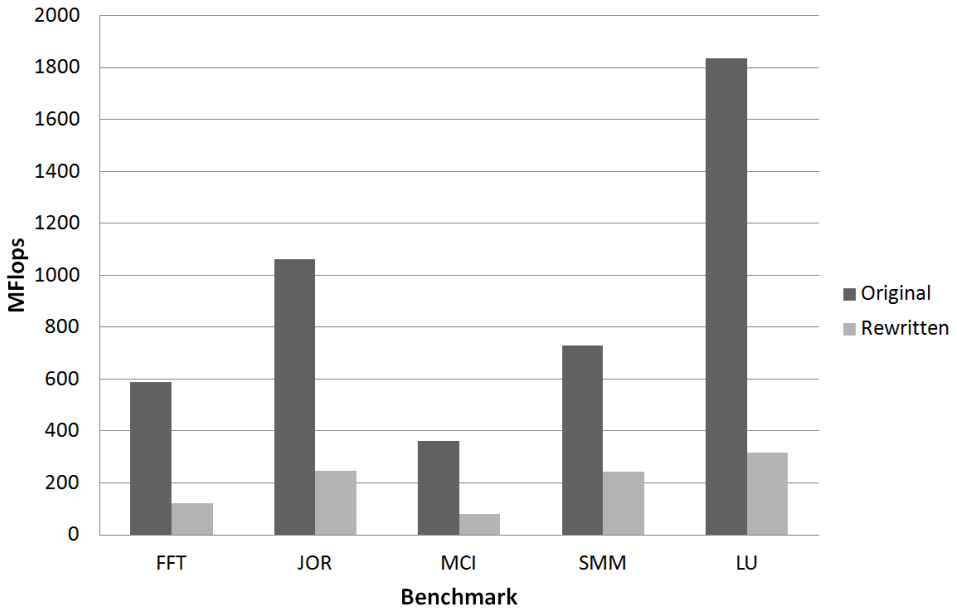

Figure 4.6. Performance overhead from enforcing worst-case security policies on SciMark benchmarks

prevent the application from violating the policy and terminating prematurely. The rewriter enforced this policy by inserting runtime security checks around all 71 `dmul` instructions in the untrusted code.

Figure 4.6 graphs the runtime overhead introduced by our rewriter by comparing the number of MFlops before and after rewriting for each benchmark. Overall we observed an average 78% reduction in performance with individual benchmarks ranging from 66% to 83%. We view these statistics as worst-case scenarios, since this particular policy was designed to force the rewriter to insert many dynamic checks within the innermost loops of intensive numerical calculations. For less pathological policies that we tried (e.g., monitoring method invocations), the runtime overhead introduced was so small as to be unmeasurable. The SciMark JAR file was 34K in size prior to rewriting and dropped to 32K afterward. (See the discussion above for a possible explanation for the size reduction.) Our rewriter loaded and transformed the entire suite in 1.5 seconds.

4.4 Limitations

Although the rewriter accepts programs and policies of arbitrary size and complexity, it has some important current limitations. Some of these are simply implementation details, while a few are limitations of the SPoX language itself. The former can be mitigated by more sophisticated rewriters, while the latter can often be handled through additional, untrusted policy enforcement components. A few limitations are actually desirable, as they make the development of a trustworthy certifying IRM system much more feasible.

Perhaps the most significant limitation is a lack of support for multithreaded programs. This is purely a matter of policy enforcement, not policy definition, and so it is a weakness of our current rewriter, not SPoX. A more complex rewriter that locks security-relevant variables (reified state variables, runtime arguments considered by dynamically decidable pointcuts, etc.) while executing guard code should provide support for parallelization (Hamlen 2006). However, avoiding inefficiency and deadlock is nontrivial, suggesting a need for race-free policies (Dam et al. 2009).

Write-reflective and call-reflective (but not read-reflective) programs are unsupported by our rewriter implementation, and are conservatively rejected by the verifier discussed in Chapter 7. We believe that reflection is used rarely enough that this is not a significant problem. For those programs that do use it, the rewriter itself could be integrated into a wrapper for the reflection API. However, it should be noted that such a solution would be difficult to manage in a certifying IRM system, as it requires certifying the entire rewriting algorithm.

The rewriter's general approach is very simple: every security-relevant instruction gets its own independent block of surrounding guard code. As seen in the SciMark example, this can sometimes lead to inefficiency in the rewritten program. Advanced code analysis could lead to reorganized enforcement code, reducing the impact on performance. Some possible techniques are discussed in Section 3.3.2.

Our system rewrites target programs, *not* the JVM libraries. Any instructions considered security-relevant by the policy are therefore ignored if they occur within JVM library code. In effect, we are trusting these libraries, which is intentional. Enforcing several simultaneous policies on core libraries is usually unnecessary and impractical, especially since they frequently call low-level system code.

In Section 4.2.2, we described how our implementation uses inheritance and interfaces to reify instance state variables for JVM library class objects. This approach does not work for classes that are declared final, disallowing inheritance. Reifying instance state variables for such classes requires an alternative kind of object wrapping, or a mapping scheme to link new instances of state variables to new instances of their corresponding objects.

At this time, SPoX state variables can only be set to integer values. There are situations in which other types would be useful. For example, consider the policy which says that a collection cannot be altered while its iterator is cycling through its contents. Java collection objects spawn independent `Iterator` objects, which must be somehow mapped to their source collection to enforce such a policy. We know of no way to do this in SPoX using integer assignments, but it is possible to emulate a mapping if we assign actual object pointers to the state variables. This pointer could be assigned and accessed within `<nodes>` tags, allowing us to modify the appropriate instance state variable and enforce the policy.

SPoX advice is deliberately limited to automaton state alterations for verification purposes; however, many sophisticated, real-world policies cannot be enforced without the use of specialized actions, code libraries, and method calls. For example, some users may want to enforce a policy that forces a program to add an entry to a log file every time it accesses the network. In Section7.5, we describe a way to combine SPoX with other aspect-oriented IRM systems to enforce such a policy while still allowing certification of the final, rewritten application.

CHAPTER 5
INCONSISTENCY DETECTION[1]

5.1 Overview

Expressing high-level security policies programmatically as aspects is often a difficult and potentially error-prone process. This is especially true when policies are intended to be generic, applying to a broad class of programs rather than just a few known programs. Correctly implementing generic, aspect-oriented security policies often requires highly non-trivial reasoning about how the aspect-weaving process might affect new, previously unseen, untrusted code. Unit testing tends to be an unreliable means of detecting errors in these policy encodings, since an incorrect aspectual policy implementation may correctly enforce the policy for most untrusted programs even if it permits policy-violating behavior or breaks policy-adherent behavior of a few unusual programs.

Two useful approaches to addressing the aspect-oriented security policy specification problem include eliminating side-effects from advice (often strengthening the pointcut language to compensate) (Dantas and Walker 2006; Dantas et al. 2008; Hamlen and Jones 2008) and synthesizing aspect-oriented policy implementations automatically from higher-level specification languages, such as LTL (cf., (Chen and Roşu 2005)) or TemporalZ (Kallel et al. 2009). SPoX (Hamlen and Jones 2008) excludes imperative advice altogether, minimizing the potential for undesired, effectful interactions with untrusted code.

However, policy errors are not limited to advice; they also arise in pointcuts. Pointcut errors tend to arise even in high-level specifications since the policy-writer must still somehow

[1]This chapter includes previously published (Jones and Hamlen 2010) joint work with Kevin Hamlen.

specify the set of security-relevant operations, and the language of operations is often an unfamiliar domain (e.g., Java bytecode instructions).

We have found that one of the most pernicious sources of error when writing complex, aspectual security policies is undesired *pointcut non-determinism*. Pointcut non-determinism arises when multiple pointcuts in an aspect-oriented security policy provide conflicting, inconsistent advice for a shared join point. As a trivial example, a policy that restricts file accesses might mandate different runtime security checks for calls to methods named `*Open*` than for calls to methods named `*Read*`. This policy might have unintended results when applied to a program that calls the `ReadOpenedFile` method, which matches both pointcuts. When pointcuts involve complex boolean expressions, regular expressions, class subtyping constraints, and mixtures of static and dynamic tests, even experts are prone to such mistakes.

We present the design and implementation of a pointcut analysis utility that automatically detects potential non-determinism in pointcut libraries. Our tool targets the SPoX aspect-oriented policy specification language described in Chapter 3. Other aspect-oriented security policy languages such as those supported by JavaMOP (Chen and Roşu 2005) have natural encodings in SPoX, permitting easy analysis of such policies using our utility. We have found this automated analysis invaluable for discovering bugs in policy specifications for real systems.

This chapter is structured as follows. In Section 5.2, we describe our two-phase algorithm for detecting non-determinism in SPoX policies. In Section 5.3, we provide a machine-checked proof of correctness for the pointcut non-determinism detection part of the algorithm. Finally, Section 5.4 discusses several case studies in which our analysis algorithm discovered and reported inconsistencies in real policies.

5.2 Analysis

A SPoX policy is *non-deterministic* if it denotes a non-deterministic security automaton. Formally, we define policy non-determinism as follows.

Definition 5.2.1. *A SPoX policy is non-deterministic if it denotes a security automaton in which there exists a state*[2] $q \in (SV \times O) \rightharpoonup \mathbb{Z}$ *and two edges* (q, q_1) *and* (q, q_2) *labeled with pointcuts* pcd_1 *and* pcd_2 *respectively, such that* $q_1 \neq q_2$ *and* pcd_1 *and* pcd_2 *match non-disjoint sets of join points.*

IRM systems typically exhibit implementation-defined behavior when provided a non-deterministic policy specification as input. The implementation-defined behavior depends upon the order in which the IRM implements the runtime security checks that decide whether to perform each possible state transition at matching join points. The instrumentation (aspect-weaving) process does not detect the non-determinism statically because it cannot decide if multiple checks inserted at the same join point always result in mutually exclusive outcomes at runtime.

Two obvious solutions to this problem are to automatically determinize the security automaton prior to weaving it into untrusted code, or to automatically resolve conflicts by imposing a default ordering on conflicting edges. However, our experience indicates that these approaches often result in an automaton that does not reflect the intentions of the policy-writer. Instead, they often have the counterproductive effect of silently concealing policy design errors from the user. Non-determinism typically arises in practice when separate parts of a specification constrain event sets that the policy-writer expected to be disjoint, but that intersect at a few unusual join points or security states that the policy-writer did not adequately consider. Our goal is to bring these possible design errors to the attention of the policy-writer so that specification bugs can be manually corrected.

[2]Set O denotes the universe of object instances plus a *global meta-object* that models the storage context of global security state variables.

We therefore adopt the approach of detecting and rejecting non-deterministic policies automatically prior to enforcement. To support generic policies, the decision algorithm considers the universe of all possible untrusted target programs rather than a specific program to which the policy is to be applied. When non-determinism is detected, the algorithm yields a witness in the form of a security state and join point for which the non-determinism is exhibited. This allows policy-writers to understand and correct design flaws that may have led to the ambiguity.

At a high level the decision algorithm consists of two phases. In the first phase, state variable pre-conditions and post-conditions of every pair of `<edge>` declarations are compared to determine which edge pairs have common source states but distinct destination states. SPoX supports state transitions both immediately before and immediately after (via the `after` keyword in Figure 3.2) the matched join point. The SPoX language semantics adopt a point-in-time join point model (Endoh et al. 2006) in which after-transitions are distinct from before-transitions of any following join point. Thus, an `after` edge can never conflict with a `before` edge in SPoX, and such edge pairs can be safely ignored when searching for potential non-determinism during the first phase.

In the second phase, pointcut labels of edge pairs identified in the first phase are compared to decide if their intersection is non-empty. If any labeled edge pair is identified as non-deterministic by both phases, a witness is then synthesized from a common source state from the first phase and a common join point from the second phase. Both phases are discussed in detail in the following subsections.

5.2.1 Security State Non-determinism

The problem of deciding whether there exist automaton edges (q, q_1) and (q, q_2) with $q_1 \neq q_2$ can be reduced to a linear programming problem. Each `<edge>` declaration in the policy defines its source (resp. destination) states in terms of pre-conditions (resp. post-conditions) that are expressed as integer equality constraints over state variables and iteration variables. Iteration variables are further constrained by inequality constraints imposed by the

surrounding `<forall>` blocks that declare the iteration variables and the closed intervals over which their values range.

Thus, each set of n nested `<forall>` blocks that surround an `<edge>` containing m `<nodes>` elements defines a convex, rational polytope $T \subseteq \mathbb{Q}^n$ with $2(n+m)$ linear constraints. Each integer lattice point in feasible region T corresponds to a source state for an automaton edge defined by the `<edge>` declaration. The m post-condition constraints in such a structure define an affine transformation $f : T \to \mathbb{Q}^n$ that maps each source state to a destination state.

To decide whether an edge pair e_1, e_2 is potentially non-deterministic, we therefore adopt the following procedure.

1. Alpha-convert iteration variables to unique names. (Instance state variables are renamed since the analysis must conservatively assume that all object references of similar type may alias.)

2. Compute feasible regions T_1 and T_2 for edges e_1 and e_2 by collecting the linear constraints encoded in relevant `<forall>` declarations and the pre-conditions of `<nodes>` declarations.

3. Compute affine transformations f_1 and f_2 by collecting the linear constraints encoded in the post-conditions of the `<nodes>` declarations.

4. Compute the set of common source states $T = T_1 \cap T_2$.

5. Compute extrema of objective function $m(q) = f_1(q) - f_2(q)$ over $q \in T$. If m has a maximum or minimum other than 0 at any $\bar{q} \in T$, then $\bar{q}$ is a common source state for which edges e_1 and e_2 lead to different destination states $f_1(\bar{q})$ and $f_2(\bar{q})$.

As an example, consider the following pair of edge declarations, which refer to the same instance state variable `s`:

```
<forall var="i" from="0" to="10">
 <edge name="edge_p1">
  <argval num="1" obj="x">p1</argval>
  <nodes var="s" obj="x">i,i-3</nodes>
 </edge>
</forall>
```

```
<forall var="i" from="0" to="3">
 <edge name="edge_p2">
  <argval num="2" obj="y">p2</argval>
  <nodes var="s" obj="y">i*3,i+2</nodes>
 </edge>
</forall>
```

Step 1 alpha-converts the iteration variables to unique names i_0 and i_1. Since objects `x` and `y` may alias (e.g., at a join point whose first and second arguments refer to the same object), instance state variable `s` is not alpha-converted. Step 2 then defines feasible region T_1 by linear constraints $(i_0 \geq 0) \wedge (i_0 \leq 10) \wedge (s = i_0)$ and feasible region T_2 by $(i_1 \geq 0) \wedge (i_1 \leq 3) \wedge (s = 3i_1)$. Transformations f_1 and f_2 are defined in Step 3 by $f_1(s) = s - 3$ and $f_2(s) = s/3 + 2$. Step 4 defines intersection $T = T_1 \cap T_2$ by the conjunction of the constraints defining T_1 and T_2.

Step 5 considers objective function $m(s) = s - 3 - (s/3 + 2)$, which is maximized in T at $(s, i_0, i_1) = (9, 9, 3)$ and minimized at $(0, 0, 0)$ (with values 1 and -5, respectively). Thus, nondeterminism could potentially be exhibited when `x.s` = `y.s` $\in \{0, 9\}$ if there exists a program operation that matches both of pointcuts `p1` and `p2`. Disjointness of pointcut expressions is discussed in the next section.

5.2.2 Pointcut Non-determinism

Pointcut disjointness is reducible to pointcut unsatisfiability. That is, pointcuts pcd_1 and pcd_2 match disjoint sets of join points if and only if pointcut `(and` pcd_1 pcd_2`)` is unsatisfiable. Pointcut satisfiability can, in turn, be reduced to boolean satisfiability (SAT). The remainder of the section describes an algorithm $\mathcal{S}$ that transforms a pointcut *pcd* into a boolean sentence $\mathcal{S}(pcd)$ such that sentence $\mathcal{S}(pcd)$ is satisfiable if and only if pointcut *pcd* is satisfiable.

Conjunction, disjunction, and negation in pointcut expressions can be translated directly to boolean conjunction, disjunction, and negation. The only remaining pointcut syntaxes are primitives (forms such as `<call>` that do not include nested pointcuts) and control flow

operators such as `<cflow>`. Control flow operators are removed from pointcuts prior to the analysis by translating them to an equivalent automaton encoding (see Section 4.2.1).

This leaves the various pointcut primitives, such as `<call>`, `<get>`, and `<argval>`. For each unique primitive, a unique boolean variable is introduced. This results in a boolean sentence that is satisfiable if the original pointcut is satisfiable, but which might be satisfiable even if the original pointcut is unsatisfiable. For example, the pointcut

```
<and><call>File.open</call><call>File.close</call></and>
```

yields the sentence $a \wedge b$, which is satisfiable even though the original pointcut is not. To correct this, a constraint term must be added to the sentence for each pair of boolean variables that denote non-independent pointcut primitives. In this case, the appropriate constraint term is $\neg(a \wedge b)$ since a and b denote primitives that cannot both be true for the same join point. A systematic approach to deriving constraint terms is provided later in the section.

Algorithm $\mathcal{S}$ can therefore be summarized as follows:

1. Construct a mapping $V : P \rightarrow B$ from pointcut primitives to unique boolean variables.
2. For each subset of pointcut primitives $S \subseteq P$, potentially generate a constraint term $c(S)$.
3. Construct sentence $\mathcal{S}(pcd)$ as follows:

$$\mathcal{S}(pcd) = \mathcal{T}(pcd) \wedge \big(\bigwedge_{S \subseteq P} c(S) \big)$$

$$\begin{aligned} \text{where } \mathcal{T}(\texttt{and } pcd_1\ pcd_2) &= \mathcal{T}(pcd_1) \wedge \mathcal{T}(pcd_2) \\ \mathcal{T}(\texttt{or } pcd_1\ pcd_2) &= \mathcal{T}(pcd_1) \vee \mathcal{T}(pcd_2) \\ \mathcal{T}(\texttt{not } pcd_1) &= \neg\, \mathcal{T}(pcd_1) \\ \mathcal{T}(pcd) &= V(pcd) \text{ for } pcd \text{ primitive} \end{aligned}$$

Table 5.1. Constraint generation cases

	`call`	`exec`	`get`	`set`	`argv`	`targ`	`argt`	`with`
`call`	CR							
`exec`	E	CR						
`get`	E	E	CR					
`set`	E	E	E	CR				
`argv`	C	C	E	C	RV			
`targ`	CR	CR	CR	CR	I	CR		
`argt`	C	C	E	C	T	I	CR	
`with`	I	CR	I	I	I	I	I	CR

Legend:
I: independent (no constraint required)
E: mutually exclusive (use constraint $\neg(a \wedge b)$)
C: independent except for known classes
R: regular expression non-emptiness check
V: argval check
T: argval–argtyp compatibility check

Generating constraint terms $c(S)$ in Step 2 is the most difficult step in the reduction. Most of the necessary constraints can be generated by considering only pairs of pointcuts $S = \{p_1, p_2\}$ rather than larger sets. (The one exception involves regular expressions, and is described in greater detail below.) The possible cases for such pairs can be divided into the various possible syntactic forms for p_1 and p_2. These cases are summarized in Table 5.1. The columns and rows of the table are labeled with the abbreviated names of the most interesting pointcut primitives listed in Figure 3.3. Note that `argv` and `argt` represent `argval` and `argtyp`, respectively, while `with` refers to `withincode`.

Cells labeled I are always independent; no constraint term is required in these cases. For example, a `call` join point can potentially appear within any lexical scope, so `call` and `withincode` pointcuts are independent. Cells labeled E are mutually exclusive; the necessary constraint term is $\neg(\mathcal{T}(p_1) \wedge \mathcal{T}(p_2))$. For example, no join point is both a call instruction and a field-get instruction, so `call` and `get` are mutually exclusive.

Deriving appropriate constraints for cells labeled C requires a model of the trusted portion of the class hierarchy. Trusted classes typically consist of those implemented by the Java standard libraries or other system-level libraries. When one or both pointcut primitives name a known, trusted class, the constraint generator performs a subclass test over the trusted class model; otherwise the two pointcuts are independent. For example, two `target` pointcuts are typically independent since in an arbitrary untrusted program any type could be a subtype of any other type. However, if one pointcut names a standard library class c that is declared final and the other uses a type pattern that does not match any superclass of c, then the two are mutually exclusive.

A regular expression non-emptiness test is required for cells marked R in Table 5.1, and is the only case that requires consideration of sets $S \subseteq P$ that are larger than size 2. For each set of pointcuts S that place regular expression constraints upon the same join point component (e.g., the same instruction argument), the constraint generator must decide whether there exists a string that satisfies all regular expressions in S and none in $P - S$. If so, no constraint is generated for S; otherwise a mutual exclusion constraint is generated. For example, consider the pointcut fragment

```
<and><call>x*</call><call>xx*</call><call>xy</call></and>
```

which initially reduces to the sentence $a \wedge b \wedge c$ before constraints are added. Regular expressions `xx*` and `xy` are both subsets of `x*`, and `xy` is disjoint from `xx*`. The algorithm above therefore represents these with a conjunction of three constraints: $\neg(b \wedge \neg a) \wedge \neg(c \wedge \neg a) \wedge \neg(a \wedge b \wedge c)$.

The space of subsets $S \subseteq P$ that must be considered is potentially exponential in the size of P, but in practice the space can be significantly pruned through memoizing. That is, if any subset S has an empty intersection, then no supersets of S need be considered further. Thus, working upward from small subsets to larger ones tends to result in a smaller number of regular expression emptiness sub-problems that must be solved.

The cell of Table 5.1 marked V concerns the special case of two `argval` predicates. Two `argval` predicates are always independent unless they regard the same argument index n and both contain value predicates that are relevant to the same type of data (object, integer, or string). String predicates reduce to regular expression non-emptiness problems, described above. Integer predicates result in implication, bi-implication, or mutual exclusion constraints. For example, if p_1 contains `<inteq>3</inteq>` and p_2 contains `<intlt>4</intlt>`, then $c(\{p_1, p_2\}) = (\mathcal{T}(p_1) \Rightarrow \mathcal{T}(p_2))$.

Finally, the table cell marked T concerns the special case of an `argval` and an `argtyp` primitive. These are always independent if they refer to different runtime arguments, they are mutually exclusive if they refer to differing types, and otherwise the `argval` predicate implies the `argtyp` predicate. For example, if p_1 = `<argval num="1"><inteq>3</inteq> </argval>` and p_2 = `<argtyp num="1">int</argtyp>` then we obtain the constraint $\mathcal{T}(p_1) \Rightarrow \mathcal{T}(p_2)$ because any join point satisfying p_1 also satisfies p_2.

Once boolean sentence $\mathcal{S}(pcd)$ has been constructed, it is delivered to a SAT-solving engine. The SAT-solver yields a satisfying assignment of boolean variables if one exists. From such an assignment it is trivial to recover a witness join point that lies in the intersection of the two original pointcuts. This facilitates disambiguation of the flawed policy specification by providing the human expert with an example target program fragment for which the policy is ambiguous.

5.3 Machine-Checked Proof

To prove the correctness of our inconsistency detection algorithm, we used the automated proof assistant ACL2 (Kaufmann and Moore 2006). ACL2 accepts models of system behavior written with LISP syntax, and can be guided by a user to produce proofs of correctness for those models. Many companies and researchers have used ACL2 to provide proofs regarding the behavior of their systems, including the floating point operations of AMD and IBM processors, information flow properties of the AAMP7G system developed by Rock-

well Collins, and key components of the standard JVM bytecode verifier (Kaufmann and Moore 2011). Machine-checkable proofs are widely considered to be the highest standard of mathematical rigor, and therefore appropriate for verification of security-critical components in large software systems. Automated theorem-provers also facilitate rigorous verification of proofs containing large numbers of sub-cases, making them well-suited for proving the correctness of our algorithm.

Our proof establishes the correctness of the constraint-generation cases summarized in Table 5.1. Correctness of the integer-based security state analysis then follows directly from the definition of automaton non-determinism. The first, non-constraint part of the pointcut reduction to SAT is similarly trivial, as it simply produces a boolean sentence that is structurally equivalent to the original pointcut expression. These parts of the algorithm are therefore not included in the ACL2-checked proof.

We have not yet proved any theorems regarding trusted class hierarchies (C in Table 5.1). These are left to future work.

In this section we discuss our ACL2 functions and theorems, and how they accurately model and prove the correctness of various constraint generation cases. The primary data structures are described in Section 5.3.1, the core functions in Section 5.3.2, and the theorems themselves in Section 5.3.3.

5.3.1 Data Structures

We model the abstract syntaxes of both pointcuts and join points with the standard LISP notation for lists. Notation `(a b c)` denotes a true (`nil`-terminated) list with elements `a`, `b`, and `c`. Notation `(a b c . d)` denotes a possibly non-true (not `nil`-terminated) list whose first three elements are `a`, `b`, and `c`, and whose final "`cdr`" is `d` (instead of `nil`).

Join points The abstract LISP syntax for join points is listed in Figure 5.1. A join point `jp` is a list whose first element is an instruction, whose second element is a lexical scope, and whose remaining elements are any relevant arguments on the evaluation stack. The

```
     jp ::= (instr scope . arglist)
  instr ::= ('call . string)
          | 'entry
          | ('get . string)
          | ('set . string)
          | anything
  scope ::= (static-scope . dynamic-scope)
arglist ::= (tp a1 a2 ... an)
     tp ::= nil | anything
      a ::= n | string | nil | (string . anything)
```

Figure 5.1. LISP syntax for join points

first of these dynamic arguments is the "this" pointer if the instruction requires one, or `nil` otherwise. Each remaining argument is an integer, a string, a null pointer (`nil`), or an object. An object is represented as a list whose first element is a string representation of the object's type.

The join point syntax needs only model components that are relevant to our pointcut language. Therefore, we are only concerned with three primary types of instructions: `call`, `get`, and `set`. In order to match against `execution` pointcuts, we also use an artificial `entry` join point to model the entry point of a method.

A join point's `scope` is comprised of both a static element and a dynamic element. The static scope is consulted by `exec` and `withincode` pointcuts, and identifies the lexical context of any join point. The dynamic scope is not used by any existing pointcuts in our language, but is potentially used by anything that considers dynamic context. For example, if class `B` inherits method `m` from class `A`, then any runtime join point occurring within `B.m` would fall into the static scope `A.m` but the dynamic scope `B.m`.

Pointcuts The abstract LISP syntax for pointcuts is listed in Figure 5.2. A pointcut designator `pcd` is a list whose first element is the name of the pointcut and whose remaining

```
pcd ::= ('call . re)
      | ('exec . re)
      | ('get . re)
      | ('set . re)
      | ('argval n . vp)
      | ('argtyp n . re)
      | ('target . re)
      | ('withincode . re)
 vp ::= 'true | 'isnull | ('streq . re) | ('inteq . n) | ('intne . n)
      | ('intle . n) | ('intge . n) | ('intlt . n) | ('intgt . n)
 re ::= string | anything
```

Figure 5.2. LISP syntax for pointcuts

elements are that pointcut's parameters. Note that `n` denotes integers and `re` denotes regular expressions.

A regular expression `re` is either a string constant (to denote an expression that matches exactly and only itself) or a non-string (to denote an expression matching zero or two or more strings). Our theorems do not care exactly how regular expression matching works, so we do not specify exactly how those non-strings are encoded. As seen in the next subsection, we use the stub function `set-member` as a black box to represent the matching process to ACL2.

5.3.2 Core Functions

The most important function used by our ACL2 theorems is `match-pcd` in Figure 5.3, which returns *true* if and only if pointcut `pcd` matches join point `jp`. Observe that there are several helper functions, which we describe below.

Most pointcuts make use of regular expressions, which are matched using `re-member`:

```
(defun re-member (x re)
  (if (stringp re) (equal x re) (set-member x re)))
```

```
(defun match-pcd (jp pcd)
  (or
   (and (equal (car pcd) 'call)
        (equal (caar jp) 'call)
        (re-member (cdar jp) (cdr pcd)))
   (and (equal (car pcd) 'exec)
        (equal (car jp) 'entry)
        (re-member (caadr jp) (cdr pcd))
        (null (cdadr jp)))
   (and (equal (car pcd) 'get)
        (equal (caar jp) 'get)
        (re-member (cdar jp) (cdr pcd)))
   (and (equal (car pcd) 'set)
        (equal (caar jp) 'set)
        (re-member (cdar jp) (cdr pcd)))
   (and (equal (car pcd) 'argval)
        (plusp (cadr pcd))
        (< (cadr pcd) (len (cddr jp)))
        (match-vp (nth (cadr pcd) (cddr jp)) (cddr pcd)))
   (and (equal (car pcd) 'argtyp)
        (plusp (cadr pcd))
        (< (cadr pcd) (len (cddr jp)))
        (re-member (valtype (nth (cadr pcd) (cddr jp))) (cddr pcd)))
   (and (equal (car pcd) 'target)
        (re-member (caaddr jp) (cdr pcd)))
   (and (equal (car pcd) 'withincode)
        (re-member (caadr jp) (cdr pcd))))
```

Figure 5.3. Function `match-pcd`

The function returns *true* if and only if string `x` matches the regular expression `re`. When `re` is a string constant, a match occurs when `x` and `re` are equal. When it is anything else, we use the following function to determine if `x` belongs to the set of strings matched by `re`:

```
(defstub set-member (* *) => *)
```

The `set-member` function is an opaque stub with no implementation, as the process of reasoning about whether a complex regular expression matches a string has no real impact on our pointcut interference theorems. The stub effectively allows us to consider both the case where a match occurs and the case where it doesn't, avoiding any loss of generality in ACL2's proofs.

The `argtyp` pointcut uses the specialized function `valtype` to obtain the type of an argument:

```
(defun valtype (x)
  (cond ((integerp x) "int")
        ((stringp x) "string")
        ((consp x) (car x))
        (nil)))
```

This function is uniquely necessary for `argtyp`, as arguments can be either primitive types (`int` or `string`), objects, or null pointers. The `target` pointcut only ever matches against objects, so it has no need of this function.

The most complex pointcut is `argval`, which utilizes the function `match-vp` from Figure 5.4 to match arguments against value predicates. The only possibly confusing predicate is `streq`, which can be used to match either a string literal (`x` when `(stringp x)` is true) or an object's string representation (`(cdr x)`).

```
(defun match-vp (x vp)
  (or
   (equal vp 'true)
   (and (equal vp 'isnull)
        (null x))
   (and (equal (car vp) 'inteq)
        (integerp x)
        (= x (cdr vp)))
   (and (equal (car vp) 'intle)
        (integerp x)
        (<= x (cdr vp)))
   (and (equal (car vp) 'intne)
        (integerp x)
        (/= x (cdr vp)))
   (and (equal (car vp) 'intge)
        (integerp x)
        (>= x (cdr vp)))
   (and (equal (car vp) 'intlt)
        (integerp x)
        (< x (cdr vp)))
   (and (equal (car vp) 'intgt)
        (integerp x)
        (> x (cdr vp)))
   (and (equal (car vp) 'streq)
        (or (and (stringp x)
                 (re-member x (cdr vp)))
            (and (stringp (cdr x))
                 (re-member (cdr x) (cdr vp)))))))
```

Figure 5.4. Function `match-vp`

5.3.3 Theorems

In order to prove the overall correctness of Table 5.1, we prove smaller theorems for each table entry. In general, for each pair of pointcuts, we prove one of the following: implication, mutual exclusion, or independence. Many of the proofs share some complex theoretical components, necessitating extra ACL2 functions and sub-theorems.

ACL2's support for existential quantification is limited, so we describe many of our theorems in terms of witnesses. For example, wherever we wish to prove something of the form $\exists x.P(x)$, we instead prove $P(w)$ directly for some witness w. Likewise, to prove an implication of the form $(\exists x.P_1(x)) \Rightarrow (\exists y.P_2(y))$, we instead prove $P_1(w_1) \Rightarrow P_2(f(w_1))$, where f is a Skolem function that computes a witness for P_2 from the witness w_1 for P_1. If we can find any f for which this is provable, we have proved the more general existential theorem.

To determine if two pointcuts `pcd1` and `pcd2` both match some witness join point `w`, we use the following function:

```
(defun pcd-intersects (pcd1 pcd2 w)
  (and (match-pcd w pcd1) (match-pcd w pcd2)))
```

If `pcd-intersects` returns *true*, then we have proved that there exists at least one join point matching both `pcd1` and `pcd2`, specifically `w`.

Many pairs of pointcuts share join points if and only if their regular expressions have a non-empty intersection. We use the function `re-intersects` to determine whether two regular expressions both match some witness string `w`:

```
(defun re-intersects (re1 re2 w)
  (and (re-member w re1) (re-member w re2)))
```

That function is used in the two templates below, which describe the if and only if implications, respectively:

```
(defabbrev regexp-if (pcd1 pcd2 s-witness)
  (implies (pcd-intersects pcd1 pcd2 jp-witness)
           (re-intersects re1 re2 s-witness)))

(defabbrev regexp-onlyif (pcd1 pcd2 jp-witness)
  (implies (re-intersects re1 re2 s-witness)
           (pcd-intersects pcd1 pcd2 jp-witness)))
```

Here, `pcd1` and `pcd2` are pointcut expressions that contain free variables `re1` and `re2`. In `regexp-if`, `s-witness` must be a string matched by both `re1` and `re2`; it may be expressed in terms of variable `jp-witness`, which is guaranteed to be a join point matched by both `pcd1` and `pcd2`. In `regexp-onlyif`, `jp-witness` must be a join point matched by both `pcd1` and `pcd2`; it may be expressed in terms of variable `s-witness`, which is guaranteed to be a string matched by both `re1` and `re2`.

Two pointcuts `pcd1` and `pcd2` are mutually exclusive if and only if every join point that satisfies `pcd1` fails to satisfy `pcd2`. Therefore, we can represent the mutual exclusion constraint $\neg(A \wedge B)$ with the theorem $A \Rightarrow \neg B$. We encode this in ACL2 using the `mutex` template:

```
(defabbrev mutex (pcd1 pcd2)
  (implies (match-pcd jp pcd1) (not (match-pcd jp pcd2))))
```

Proving independence is somewhat more complex. Two pointcuts are independent if and only if the existence (or non-existence) of a join point matching one of them does not affect the existence (or non-existence) of a join point matching the other. This can happen when one or both pointcuts `pcd1` and `pcd2` match nothing or everything, or when there exist witness join points `tt`, `tf`, `ft`, and `ff` such that `tt` satisfies both `pcd1` and `pcd2`, `tf` satisfies `pcd1` but not `pcd2`, `ft` satisfies `pcd2` but not `pcd1`, and `ff` satisfies neither `pcd1` nor `pcd2`. This can be formulated as a constructive proof. Given join points `t1`, `t2`, `f1`, and `f2` that

satisfy `pcd1`, satisfy `pcd2`, do not satisfy `pcd1`, and do not satisfy `pcd2` (respectively), we prove that there are join points `tt`, `tf`, `ft`, and `ff` that satisfy `pcd1` and `pcd2`, `pcd1` but not `pcd2`, `pcd2` but not `pcd1`, and neither `pcd1` nor `pcd2` (respectively). The ACL2 template that models this theorem is as follows:

```
(defabbrev independent (pcd1 pcd2 tt tf ft ff)
  (implies (and (match-pcd t1 pcd1) (not (match-pcd f1 pcd1))
                (match-pcd t2 pcd2) (not (match-pcd f2 pcd2)))
           (and (match-pcd tt pcd1) (match-pcd tt pcd2)
                (match-pcd tf pcd1) (not (match-pcd tf pcd2))
                (not (match-pcd ft pcd1)) (match-pcd ft pcd2)
                (not (match-pcd ff pcd1)) (not (match-pcd ff pcd2)))))
```

Note that if either `pcd1` or `pcd2` universally matches everything or nothing, the left-hand side of the implication is false, and the theorem can still be proved.

The following paragraphs describe several representative theorems that describe relationships between pairs of pointcuts in Table 5.1. The remaining cases generally resemble the ones provided in this section. ACL2 has proved the correctness of all of them.

`call-call` Two `call` pointcuts intersect for some set of join points if and only if their regular expressions intersect. This is split into two theorems, each of which describes one implication direction:

```
(defthm call-call-if
  (regexp-if (cons 'call re1) (cons 'call re2)
             (cdar jp-witness)))

(defthm call-call-onlyif
  (regexp-onlyif (cons 'call re1) (cons 'call re2)
                 (cons (cons 'call s-witness) '(nil))))
```

Observe that `(cdar jp-witness)` extracts the regular expression from a given join point witness, using it as the `s-witness` parameter being passed to `regexp-if`. In the reverse direction, `(cons (cons 'call s-witness) '(nil))` constructs a join point witness `jp-witness` from a given regular expression witness and passes it to `regexp-onlyif`.

`call-exec` A `call` matches invoke instructions and an `exec` matches method entry points, so the two pointcuts are mutually exclusive:

```
(defthm call-exec
  (mutex (cons 'call re1) (cons 'exec re2)))
```

`call-withincode` A `call` matches based on instruction identity, while `withincode` matches based on instruction lexical context, so the two pointcuts are independent:

```
(defthm call-withincode
  (independent (cons 'call re1) (cons 'withincode re2)
               (list (cons 'call (cdar t1)) (cadr t2))
               (list (cons 'call (cdar t1)) (cadr f2))
               (list 'non-call (cadr t2))
               (list 'non-call (cadr f2))))
```

From top to bottom, the `list` constructions generate: `tt`, an invoke instruction matching `(call . re1)` within a context matching `(withincode . re2)`; `tf`, an invoke instruction matching `(call . re1)` but within a context not matching `(withincode . re2)`; `ft`, a non-invoke instruction within a context matching `(withincode . re2)`; and `ff`, a non-invoke instruction within a context not matching `(withincode . re2)`.

`exec-withincode` An `exec` and a `withincode` intersect on the same `entry` join point if and only if their regular expressions `re1` and `re2` intersect:[3]

```
(defthm exec-withincode-if
  (regexp-if (cons 'exec re1) (cons 'withincode re2)
             (caadr jp-witness)))

(defthm exec-withincode-onlyif
  (regexp-onlyif (cons 'exec re1) (cons 'withincode re2)
                 (list* 'entry (cons s-witness nil) nil)))
```

`argtyp-argval` The argument pointcuts `argtyp` and `argval` have a specialized implicative relationship. For example, if an `argval` contains the integer comparison predicate `inteq`, any join point matching it must also match an `argtyp` that considers variables of type `int`:

```
(defthm argtyp-inteq
  (implies (match-pcd jp (list* 'argval n 'inteq x))
           (match-pcd jp (list* 'argtyp n "int"))))
```

`argval-argval` There are many subcases for `argval-argval` relationships, due to the number of possible value predicate combinations. For example, suppose `pcd1` contains `inteq x` and `pcd2` contains `intle y`. Then `pcd1` implies `pcd2` when $\texttt{x} \leq \texttt{y}$, otherwise the two pointcuts are mutually exclusive:

```
(defthm inteq-le-intle
  (implies (and (<= x y)
                (match-pcd jp (list* 'argval n 'inteq x)))
```

[3]This relationship is consistent with the pointcut behavior described in the AspectJ documentation (The AspectJ Team 2003). However, `execution` and `withincode` pointcuts usually conflict in version 1.6.12 of the AspectJ implementation.

```
          (match-pcd jp (list* 'argval n 'intle y))))

(defthm inteq-gt-intle
  (implies (> x y)
          (mutex (list* 'argval n 'inteq x) (list* 'argval n 'intle y))))
```

If pcd1 contains `intle x` and pcd2 contains `intge y`, we must prove all of the following: if $x \geq y$, then pcd1 or pcd2 or both must match any instruction with an integer argument; if $x < y$, then pcd1 and pcd2 are mutually exclusive; and if $x = y - 1$, exactly one of pcd1 or pcd2 must match any instruction with an integer argument.

```
(defthm intle-ge-intge
  (implies (and (>= x y)
                (integerp x)
                (integerp y)
                (plusp n)
                (< n (len (cddr jp)))
                (integerp (nth n (cddr jp))))
           (or (match-pcd jp (list* 'argval n 'intle x))
               (match-pcd jp (list* 'argval n 'intge y)))))

(defthm intle-lt-intge1
  (implies (< x y)
          (mutex (list* 'argval n 'intle x) (list* 'argval n 'intge y))))

(defthm intle-lt-intge2
  (implies (and (= x (- y 1))
                (integerp x)
                (integerp y)
```

```
            (plusp n)
            (< n (len (cddr jp)))
            (integerp (nth n (cddr jp))))
      (xor (match-pcd jp (list* 'argval n 'intle x))
            (match-pcd jp (list* 'argval n 'intge y)))))
```

If `pcd1` contains a `streq` predicate and `pcd2` contains any `int` predicate, then the two pointcuts are mutually exclusive:

```
(defthm streq-int
  (implies (or (equal int-vp 'inteq)
               (equal int-vp 'intne)
               (equal int-vp 'intle)
               (equal int-vp 'intge)
               (equal int-vp 'intlt)
               (equal int-vp 'intgt))
           (mutex (list* 'argval n 'streq re) (list* 'argval n int-vp x))))
```

Argument Index If an argument index `n` does not exist for a given join point, then it cannot match any `argval` pointcut:

```
(defthm argval-nonexist
  (implies (and (plusp n) (integerp n)
                (>= n (len args)))
           (not (match-pcd (list* 'any-name 'any-lex args)
                           (list* 'argval n any-vp)))))
```

A similar theorem proves the same for `argval`.

When any two argument pointcuts (either `argval` or `argtyp`) consider different argument indices, the two are independent. Proving this is actually quite challenging, due to the

complexity of constructing the witness argument lists. Therefore, we need some ACL2 machinery to prove any independence theorems.

To start, we use a function `mklist`, which constructs padding lists of a given size `n`. Lists created with `mklist` represent unused argument slots, so we just fill them with `nil` elements.

```
(defun mklist (n)
  (if (> (nfix n) 0) (cons nil (mklist (1- n))) nil))
```

To help ACL2 reason about `mklist`, we use a helper lemma to prove that the function creates lists of a correct size:

```
(defthm mklist-length
  (equal (len (mklist n)) (nfix n)))
```

The key to proving independence is the following lemma:

```
(defthm nth-append-secondlist
  (implies (>= (nfix n) (len x))
           (equal (nth n (append x y))
                  (nth (- n (len x)) y)))
  :hints (("Goal"
           :induct (nth n x))))
```

The lemma proves that the `n`th element of `(append x y)` is the $(\mathtt{n} - |\mathtt{x}|)$th element of `y` whenever $\mathtt{n} \geq |\mathtt{x}|$. This introduces a rewrite rule into ACL2 that encourages length-checking of `x` whenever such expressions appear. To get ACL2 to prove the lemma, we provide a hint that says to induct on `n` and `x` (but not `y`).

We also include a corollary:

```
(defthm nth-append-mklist-sameindex
  (equal (nth n (append (mklist n) x)) (car x)))
```

It's not a necessary theorem, but it helps ACL2 complete subsequent proofs more quickly.

Each witness join point in the argument independence theorems is constructed by creating a new argument list that has the same `i`th argument as witness `w1` and the same `j`th argument as witness `w2`, where `i` $\neq$ `j`. The following function constructs such a list:

```
(defun mklist2 (i w1 j w2)
  (let ((x (list (nth i (cddr w1))))
        (y (list (nth j (cddr w2)))))
    (if (< (nfix i) (nfix j))
      (append (mklist i) x (mklist (1- (- (nfix j) (nfix i)))) y)
      (append (mklist j) y (mklist (1- (- (nfix i) (nfix j)))) x))))
```

Each independence theorem proves that if `i` $\neq$ `j`, then there are 4 witness join points that prove independence of the pointcuts. The 4 witnesses are constructed using `mklist2`. The following template states this in a general form:

```
(defabbrev arg-independent (pcdtyp1 pcdtyp2)
  (implies (and (/= (nfix i) (nfix j))
                (< (nfix i) (len (cddr t1)))
                (< (nfix j) (len (cddr t2)))
                (< (nfix i) (len (cddr f1)))
                (< (nfix j) (len (cddr f2))))
           (independent (list* pcdtyp1 (nfix i) x)
                        (list* pcdtyp2 (nfix j) y)
                        (list* 'any-i 'any-s (mklist2 i t1 j t2))
                        (list* 'any-i 'any-s (mklist2 i t1 j f2))
                        (list* 'any-i 'any-s (mklist2 i f1 j t2))
                        (list* 'any-i 'any-s (mklist2 i f1 j f2)))))
```

Finally, we may prove argument independence for specific pointcuts:

```
(defthm argtyp-argval-independence
  (arg-independent 'argtyp 'argval)
  :hints (("Goal"
           :hands-off (re-member valtype match-vp))))
```

The hint tells ACL2 not to bother expanding the definitions of some functions that lead to rabbit trails, drastically reducing the time required to prove the theorem. The theorems for `argtyp-argtyp` and `argval-argval` have essentially the same form as the one above.

5.4 Case Studies

We implemented our non-determinism detection tool using a combination of Java and SWI-Prolog. The Java component consists of a SPoX parsing library (approximately 5000 lines of Java code) and an analysis engine that includes a Prolog-generating back-end (approximately 4200 lines of Java code). It extracts relevant information from the policy, including automaton transitions and pointcut expressions, and uses this to generate Prolog predicates that model the extracted policy information.

The Prolog half of the implementation is the heart of the analysis engine, and consists of dynamically generated code. It decides whether any pair of pointcut-labeled edges in the policy are non-deterministic using the algorithm described in Section 5.2. State variable non-determinism is decided through the use of a Prolog-based linear constraint solver. Pointcut non-determinism is decided by submitting the boolean sentence derived by algorithm $\mathcal{S}$ of Section 5.2.2 to a C implementation of the MiniSat SAT-solving engine (Eén and Sörensson 2007). If the sentence is satisfiable then the policy is non-deterministic, and the analysis tool identifies the conflicting portions of the specification.

In this section, we discuss six policy scenarios in which our analysis tool discovered policy bugs through non-determinism detection. For each case study, we discuss the origins of the policy, the design flaw that led to unintended non-determinism, and how we removed the error. Runtime statistics for all experiments are summarized at the end of the section.

5.4.1 Filesystem API Protocols

An important class of software security policies are those that prescribe protocols for accessing system API's. For example, the JavaMOP documentation (Formal Systems Laboratory 2011) includes a policy that prevents writing to a file that is not already open. Such a policy can be naturally encoded in SPoX as a 2-state automaton, where the *open* operation transitions the automaton from the *closed* to *opened* state, *write* operations are only permitted in the *opened* state, and the *close* operation transitions the automaton back to the *closed* state.

In practice such protocols can be significantly more complex. For example, a natural extension to the example above makes the set of acceptable operations contingent upon the mode in which the file has been opened. Files opened in read-mode may be read but not written, only those opened in random access mode may be seeked, etc. As the number of possible operations increases, policy complexity and the opportunity for error increase as well.

The `FileMode` policy specification in Figure 5.5 models a simplified filesystem API policy that supports read and write modes, *open* and *close* operations, and *read* and *write* operations. Lines 18 and 23 of the policy use the reserved `#` post-condition to cause the automaton to reject if a write or read operation is attempted in an incompatible mode. In addition, lines 31–34 prohibit all I/O operations when a file is in the closed state.

Line 32 of the policy contains a bug that has the unintended effect of rejecting even *open* operations when a file is in the *closed* state. The bug arises because the regular expression in that line is overly broad.

After submitting this policy to our analysis tool, it reported that the edge at line 31 conflicts with the edges at lines 3 and 9. Specifically, calls to `File.open` with a second argument of `"OpenRead"` or `"OpenWrite"` and a security state of `f` $= 0$ solicit conflicting advice.

```
<state name="f">File</state>

<edge name="openFileRead">
  <and><call>File.open</call>
       <argval num="2"><streq>OpenRead</streq></argval>
       <target obj="x" /></and>
  <nodes var="f" obj="x">0,1</nodes></edge>

<edge name="openFileWrite">
  <and><call "File.open">
       <argval num="2"><streq>OpenWrite"</streq></argval>
       <target obj="x" /></and>
  <nodes var="f" obj="x">0,2</nodes></edge>

<edge name="illegalWrite">
  <and><call>File.write</call>
       <target obj="x" /></and>
  <nodes var="f" obj="x">1,#</nodes></edge>

<edge name="illegalRead">
  <and><call>File.read</call>
       <target obj="x" /></and>
  <nodes var="f" obj="x">2,#</nodes></edge>

<forall var="i" from="1" to="2">
  <edge name="fileClose">
    <and><call>File.close</call>
         <target obj="x" /></and>
    <nodes var="f" obj="x">i,0</nodes></edge></forall>

<edge name="illegalFileOp">
  <and><call>File.*</call>
       <target obj="x" /></and>
  <nodes var="f" obj="x">0,#</nodes></edge>
```

Figure 5.5. `FileMode` policy

Upon discovering the bug, we disambiguated the policy by replacing the pointcut at lines 32–33 with the following refinement of the original pointcut expression.

```
<and><call>File.*</call>
     <not><call>File.open</call></not>
     <target obj="x" /></and>
```

The resulting policy passed the analysis and correctly enforced the desired policy.

5.4.2 Transaction Logging

A classic application of Aspect-Oriented Programming in the literature is transaction logging (e.g., (Laddad 2002)). An AOP-style transaction logger is one specific enforcement of a more general audit policy. The audit policy dictates that impending security-relevant transaction operations must be first logged via a trusted logging mechanism. An IRM or other security implementation can enforce the policy by injecting the necessary logging operations immediately before each transaction operation.

Figure 5.6 provides a fragment of one such audit policy for a hypothetical credit card processing library. The library includes a `CreditCardProcessor` class that contains numerous transaction implementations, all accessed via methods with names ending in `*Transaction`. The desired audit policy mandates exactly one call to the trusted `logTransaction` method before each such transaction.

The specification in Figure 5.6 contains bugs at lines 8 and 16 that mistakenly treat calls to the `logTransaction` method itself as transactions. Our analysis uncovered these bugs in the form of two sources of non-determinism: one associated with the edges at lines 3 and 15, and the other associated with those at lines 7 and 11. In the first case calls to `logTransaction` solicit conflicting advice in security state 0. In the second case the same ambiguity arises in state 1.

To correct the error, we introduced a named pointcut that lists each security-relevant transaction method explicitly in a large disjunctive pointcut. (An alternative would be to

```
1 <state name="logged" />
2
3 <edge name="log">
4   <call>CreditCardProcessor.logTransaction</call>
5   <nodes var="logged">0,1</nodes></edge>
6
7 <edge name="transaction">
8   <call>CreditCardProcessor.*Transaction</call>
9   <nodes var="logged">1,0</nodes></edge>
10
11 <edge name="badLog">
12   <call>CreditCardProcessor.logTransaction</call>
13   <nodes var="logged">1,#</nodes></edge>
14
15 <edge name="badTransaction">
16   <call>CreditCardProcessor.*Transaction</call>
17   <nodes var="logged">0,#</nodes></edge>
```

Figure 5.6. `Logger` policy

explicitly except the logging method using pointcut negation and conjunction operators.) This eliminated the non-determinism and passed the analysis.

5.4.3 Object Aliasing

A particularly elusive form of unintentional non-determinism arises from object aliasing. As an illustration, consider the policy fragment in Figure 5.7, which tracks whether Java `File` objects refer to existent or non-existent files. When an existing file is renamed, the policy marks the source `File` object as referring to a non-existent file and the destination `File` object as referring to an existent one.

However, this policy has a subtle non-determinism bug that arises when the same object is passed as both the source and destination arguments of the `File.renameTo` method. In this case the policy stipulates that the `File` object transitions to both the *existent* and *non-existent* security states. A malicious program could exploit this loophole to transition `File` objects to incorrect security states and potentially circumvent the intended policy.

```
1 <state name="exists">File</state>
2
3 <forall var="i" from="0" to="1">
4   <edge name="rename">
5     <and><call>File.renameTo</call>
6          <target obj="x" />
7          <argtyp num="1" obj="y">File</argtyp></and>
8     <nodes var="exists" obj="x">1,0</nodes>
9     <nodes var="exists" obj="y">i,1</nodes></edge></forall>
```

Figure 5.7. `FileExists` policy

Our non-determinism analysis detected this bug in the form of a join point in which object identifiers `x` and `y` alias to a common `File` object. We corrected the error by changing the `i` in line 9 to the constant 0. This resolves the non-determinism by restricting only `renameTo` operations that change the name of an existing file to an unused filename; thus the source and destination cannot be the same object. The resulting policy was validated as deterministic.

5.4.4 Information Flow

A canonical information flow policy example in the IRM literature prevents untrusted programs from leaking confidential files over the network. One standard encoding of this policy prohibits all network send operations after a confidential file has been read (Schneider 2000); however the resulting policy can sometimes be too draconian to be useful in practice. A useful relaxation introduced in (Aktug and Naliuka 2008) permits subsequent network send operations only after the user has explicitly permitted them via a trusted authentication and authorization mechanism.

We ported this latter policy specification to SPoX in Figure 5.8. The user interface method `GUI.grantConnectPermission` has a trusted implementation that authorizes new connections. (When authorization is denied, the method does not return.) The policy maintains two security state variables: `permission` and `accessed`. The former is set to 1 (line 6) when the user authorizes a new connection, and reset to 0 (line 15) after a new connection

```
<state name="accessed" />
<state name="permission" />

<edge name="authorize">
  <call>GUI.grantConnectPermission</call>
  <nodes var="permission">0,1</nodes></edge>

<forall var="i" from="0" to="1"
  <edge name="read">
    <and><call>File.open</call>
         <argval num="2"><streq>OpenRead</streq></argval></and>
    <nodes var="accessed">i,1</nodes></edge>
  <edge name="send">
    <call>Connection.open</call>
    <nodes var="permission">i,0</nodes></edge></forall>

<edge name="badsend">
  <call>Connection.open</call>
  <nodes var="accessed">1,1</nodes>
  <nodes var="permission">0,#</nodes></edge>
```

Figure 5.8. `GetPermission` policy

is established. Thus, future connections require subsequent authorizations. The `accessed` variable is set to 1 (line 12) when a confidential file is read. Unauthorized, new connections subsequent to such reads are prohibited by line 20, which signals a policy violation.

The policy specification contains a bug in the edge at line 13, which resets `permission` to 0 even when the new connection constitutes a policy violation. This results in a conflict between that edge and line 20 which signals the violation. In this case the unintended non-determinism therefore arises at the security state level rather than at the pointcut level.

To fix the problem, we split the edge at line 13 into two cases, one for new connections after confidential reads and another for new connections prior to any confidential reads:

```
<forall var="i" from="0" to="1">
 <edge name="send1">
  <call>Connection.open</call>
  <nodes var="accessed">0,0</nodes>
  <nodes var="permission">i,0</nodes>
 </edge>
</forall>
```

```
<edge name="send2">
 <call>Connection.open</call>
 <nodes var="accessed">1,1</nodes>
 <nodes var="permission">1,0</nodes>
</edge>
```

This repartitioning of the security state space intentionally omits the case where the send operation occurs after a confidential file-read and without authorization, since that case is adequately covered by the edge at line 17. The resulting policy is deterministic and therefore passed the analysis.

5.4.5 Free-riding Prevention

The original version of the free-riding prevention policy in Section 4.3.2 differed slightly from the one provided in Figure 4.5, and was in fact published in a conference paper (Jones and Hamlen 2009). That original policy is reproduced in simplified form in Figure 5.9.

Surprisingly, this policy specification contains a non-determinism bug, which was uncovered by our analysis utility. Specifically, when counter `i` reaches 2 and another download

```
1 <state name="counter" />
2
3 <forall var="i" from="-10000" to="2">
4   <edge name="download">
5     <call>Connection.download</call>
6     <nodes var="counter">i,i+1</nodes></edge>
7   <edge name="upload">
8     <call>Connection.upload</call>
9     <nodes var="counter">i,i-1</nodes></edge>
10
11 <edge name="illegalDownload">
12   <call>Connection.download</call>
13   <nodes var="counter">2,#</nodes></edge>
```

Figure 5.9. `NoFreeride` policy

operation occurs, the edges at lines 4 and 11 of the specification give conflicting advice. The design flaw can be traced to an off-by-one error in the bounds of the `forall` loop, which permit line 6 to increase the counter beyond 2 and line 9 to decrease the counter beyond -10000.

To correct the error, we reduced the upper bound in line 3 from 2 to 1 and changed the decrement operation in line 9 from `i,i-1` to `i+1,i`. This corrected the bug and yielded a deterministic policy.

5.4.6 Policy Composition

Automated non-determinism detection is also useful for composing related policies because it reveals join points for which the different policies conflict. This affords policy-writers an opportunity to decide how such conflicts should be resolved on a case-by-case basis.

For example, (Douence et al. 2004) discusses challenges related to merging policies that mandate logging and encryption of the same data in untrusted program operations. We adapted that example to SPoX in the form of a data encryption policy for the credit card

```
<state name="encrypted" />

<edge name="encrypt">
  <call>CreditCardProcessor.encryptTransaction</call>
  <nodes var="encrypted">0,1</nodes>

<edge name="transaction">
  <or><call>CreditCardProcessor.creditTransaction</call>
      <call>CreditCardProcessor.debitTransaction</call></or>
  <nodes var="encrypted">1,0</nodes></edge>

<edge name="badEncrypt">
  <call>CreditCardProcessor.encryptTransaction</call>
  <nodes var="encrypted">1,#</nodes></edge>

<edge name="badTransaction">
  <or><call>CreditCardProcessor.creditTransaction</call>
      <call>CreditCardProcessor.debitTransaction</call></or>
  <nodes var="encrypted">0,#</nodes></edge>
```

Figure 5.10. `Encrypt` policy

transaction system discussed in Section 5.4.2. The encryption policy specification is shown in Figure 5.10.

Combining the policy in Figure 5.10 with the one in Figure 5.5 resulted in a composite policy that our analysis tool identified as non-deterministic. The non-determinism was witnessed by join points that both policies considered to be security-relevant, such as calls to `creditTransaction`.

In each case, we were able to resolve the unwanted non-determinism by prescribing an ordering (via an appropriate automaton encoding) on the policy-mandated security checks. For this example we adopted the strategy of requiring the encryption operations to be exhibited before the logging operations. Once all conflicts were resolved, the final composite policy passed the analysis and correctly enforced both policies.

5.4.7 Summary of Results

The results of our experiments are summarized in Table 5.2. As in Section 4.3, 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. In the table headings, *size* refers to the number of characters in each policy specification (including spaces) and *pointcut vars* refers to the number of unique boolean variables introduced during the reduction to SAT described in Section 5.2.2. In one case (`FileExistsFixed`) no pointcut variables were generated because all potential non-determinism was eliminated after the first phase of the algorithm.

The MiniSat Prolog interface converts boolean sentences to conjunctive normal form (CNF) before processing. To indicate the size and complexity of the resulting SAT problems, we list the number of boolean variables and the number of clauses in the resulting CNF sentences in columns 4 and 5 of the table, respectively. Finally, the last column reports the average execution time in milliseconds that our analysis tool required for each policy including parsing, Prolog-generation, and both phases of the algorithm.

In general we found that runtimes and boolean sentence sizes were well within the range of practical use for the policies we tested. The only potential scaling issue we encountered concerns policies that contain large numbers of non-independent regular expressions containing wildcards. In those cases our treatment of regular expressions generates a large number of constraint terms, slowing the analysis. We believe this drawback could be overcome by making better use of the memoizing technique mentioned in Section 5.2.2. An interesting and potentially more elegant alternative is to replace the SAT solver with an SMT solver equipped with a theory of regular languages.

Table 5.2. Inconsistency detection experimental results

Policy	Size (chars)	Pointcut vars	CNF vars	CNF clauses	Runtimes
`FileMode`	1488	9	1764	2061	1294ms
`FileModeFixed`	1570	8	706	850	189ms
`Logger`	722	2	10	12	166ms
`LoggerFixed`	956	3	34	40	112ms
`GetPermission`	938	5	44	57	70ms
`GetPermissionFixed`	1189	5	49	61	75ms
`FileExists`	437	3	10	14	46ms
`FileExistsFixed`	423	0	0	0	22ms
`NoFreeride`	1024	3	22	28	60ms
`NoFreerideFixed`	1075	3	18	22	63ms
`Encrypt`	986	3	34	40	118ms
`Log&Encrypt`	2281	4	972	1180	370ms
`Log&EncryptFixed`	2321	4	578	703	270ms

CHAPTER 6
IN-LINED REFERENCE MONITORING AS A SERVICE[1]

6.1 Overview

In a service-oriented architecture (SOA), non-local resources, systems, and functions are made available to applications through network interfaces called web services. The underlying concept of decomposing application functions across multiple systems and linking them through network APIs has existed for some time, but the "service" terminology did not become widespread until the late 1990s (Townsend 2008). One of the more notable early SOA projects involved making Wells Fargo the first Internet banking company, though at the time the developers called their system Distributed Object Technology (Ronayne and Townsend 1996).

SOA web service interfaces are often called Software as a Service (SaaS), while the back-end datacenter and computation systems have come to be known as the Cloud (Armbrust et al. 2009). Cloud computing offers many benefits to both corporations and end users, most notably the ability to utilize large-scale computing resources at an accessible cost. For example, Amazon's EC2, S3, and SQS services (Garfinkel 2007) and Microsoft's Windows Azure system (Chappell 2010) allow software developers to run applications and store data on remote servers with pricing relative to the quantity of computing power or data involved. Similarly, consumers can now store, access, and share personal documents, photos, videos, and music through a multitude of well-known services, including Google Docs, Picasa, YouTube, and Apple's iCloud. In many of these cases, the cloud is actually a large

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

network of machines, which interoperate using complex protocols such as that of the Google File System (Ghemawat et al. 2003).

Many forms of software security naturally lend themselves to service-oriented solutions. McAfee currently offers a wide suite of SaaS tools, including website and email filters, vulnerability assessment software, and anti-virus scanners (McAfee, Inc. 2011). As the cloud itself is under the control of the software manufacturer, these systems can be kept consistently up-to-date. CloudAV (Oberheide et al. 2008) detects malware by running several anti-virus and behavioral detection engines in parallel, a technique the authors call N-version protection. All associated computing overhead is pushed to the cloud, freeing up resources on the user's local system.

However, current cloud-based security enforcement systems have significant limitations. They are generally not policy-specific, only accepting or rejecting a program based on database matches regarding known malware signatures or behaviors. Although they provide better protection against new malware than traditional, client-based systems (because a centralized cloud is easier to keep up-to-date), truly zero-day attacks can still go undetected. Also, their filtering mechanisms are designed to reject entire programs they deem unsafe, rather than dynamically rejecting only unsafe executions.

To avoid these limitations, clients typically must use their own local protection systems, which are often realized as a fixed part of the execution environment. For example, Java bytecode applets undergo both static validation and dynamic monitoring by the JVM to protect the client from potentially malicious code provided by untrusted service-providers. This arrangement results in a relatively inflexible collection of security policies. For example, the JVM enforces basic memory-safety and object-encapsulation properties, but it does not enforce user- or application-specific policies, such as a policy that prohibits untrusted applets from opening more than 3 pop-up windows per run. While enforcing such custom policies is possible, it frequently requires developing and installing new client-side VM or OS extensions for each new policy to be enforced—an impractical undertaking for many organizations and users.

Mobile devices, such as smartphones and tablets, are in particular need of good service-oriented security solutions. Although it is true that portable hardware is becoming more powerful, the code that runs on it is also getting much more complex, necessitating sophisticated security enforcement software. Client-based solutions therefore require undesirable processor and memory overhead.

As a more flexible alternative, we present an IRM-based, device-independent, service-oriented approach to securing binary code. In this framework, code-consumers submit untrusted Java bytecode and a desired security policy to a trusted in-lining service that instruments the bytecode with an IRM that enforces the policy. The resulting self-monitoring code is digitally signed and returned to the code-consumer, who can then verify the signature and safely execute it.

This approach allows code security to become a separate service in a service-oriented architecture, rather than an obligation that each client must satisfy for itself individually. The significant complexity of policy enforcement implementation is shifted to a trusted third party, affording code-consumers the flexibility of enforcing a wide array of potentially organization- and application-specific policies without implementing that functionality themselves.

The service-oriented approach has numerous potential security advantages, including improved deployment speed and wider client coverage than traditional patching approaches. For example, a web client that is configured by default to always pass untrusted bytecode through an in-lining service before execution receives the benefits of security updates implemented by that service instantly, without needing to download and install security updates or patches for its OS or VM. Moreover, as new vulnerabilities are discovered and new enforcement strategies are invented, third-party in-lining services can often be updated and adapted more rapidly than typical OS/VM patches can be developed. This is because software patches can usually only be developed by a relatively small collection of experts who have access to the OS/VM source code, whereas IRM implementations depend only on bytecode language standards available to the public. The service-oriented approach therefore

provides defenders a means to quickly and comprehensively react to zero-day attacks for which no patch yet exists, and to protect users of legacy software who may be slow to apply patches.

In Section 6.2, we describe our implementation and the three real-world case studies we used to test it.

6.2 Web Service Implementation

We have implemented a version of our rewriter that acts as a Java web service application. Using an HTTP request with the POST method, a client uploads two files: a SPoX policy and an untrusted JAR (Java ARchive) file. The server then provides these two files as input to the rewriter, which creates a new, rewritten JAR that enforces the security policy. Finally, the rewritten JAR file is returned as an HTTP response to the client.

The web service acts as a front end for the rewriter discussed in Chapter 4. The servlet by itself is comprised of 200 lines of Java code. In order to manage the file uploads, we use the Apache Commons `fileUpload` library, which obtains Java `FileInputStream`s from the request data. The files are saved locally on the server in temp files, which are submitted as input to the rewriter. The newly rewritten JAR file is then copied to the HTTP response as a binary stream.

We ran the rewriter service on three different applications, with different policies for each. As with all tests in this thesis, we used 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. The server-side code ran on an instance of GlassFish Server 3.1. Provided runtimes account only for time spent rewriting and performing related file I/O, and do not include any network operations such as uploading and downloading files.

jWeatherWatch For the weather widget application jWeatherWatch (jWeatherWatch 2009), we enforced a policy that prohibits network send operations after the application

```
1 <state name="s" />
2 <edge name="fileAccess">
3   <nodes var="s">0,1</nodes>
4   <and><call>java.io.File*</call>
5        <argval num="1"><streq>.*WINDOWS\\.*</streq></argval></and>
6 </edge>
7 <edge name="illegalSocketOutputStream">
8   <nodes var="s">1,#</nodes>
9   <call>java.net.Socket.getOutputStream</call>
10 </edge>
```

Figure 6.1. Policy that prohibits network sends after sensitive file reads

has accessed a file in the `WINDOWS` directory. Figure 6.1 shows a simplified version of the policy; the complete one utilizes the file access libraries of the policy enforced in Section 4.3.1 and handles both upper-case and lower-case forms of the string `WINDOWS`. The application obeys this policy, so the rewritten program showed no observable change in behavior. The uploaded JAR was 140 KB in size and rewriting increased its size by 6K (4%). Total processing time was approximately 4.3 seconds.

Google.mE Google.mE (Google.mE 2010) is an open source Java application that acts as a client for various Google web applications, including Google Docs, Picasa, YouTube, and GMail. One of its features supports uploading of files to many of those services. We chose to enforce the policy in Figure 6.2, which prohibits uploads of non-picture files to Picasa and non-document files to Google Docs. The file type is identified by whitelisting permissible file extensions; filenames with extensions not explicitly listed in the respective `<streq>` elements are prohibited. When we attempted to upload a file with an unsupported extension to Google Docs, the rewritten application halted execution as expected.

The original JAR was 921 KB in size while the rewritten one was 513 KB—a size reduction of over 44%. The reduction is primarily to the lack of compression in the original JAR,

```
1 <state name="s" />
2 <edge name="badUpload">
3   <nodes var="s">0,#</nodes>
4   <or><and><set>Picasa.UploadPhoto.name</set>
5            <not><argval num="1">
6                   <streq>.*\.(jpg|jpeg|tif|tiff|png|bmp|gif)</streq>
7                 </argval></not></and>
8       <and><set>GoogleDocs.UploadDoc.file</set>
9            <not><argval num="1">
10                  <streq>.*\.(doc|docx|txt|rtf)</streq>
11                </argval></not></and></or>
12 </edge>
```

Figure 6.2. Policy that prohibits uploads of files with non-whitelisted extensions

whereas our rewriter compresses the output JAR by default. The runtime was reported as 21.7 seconds.

Jeti Jeti (Jeti 2007) is a simple Jabber IM client. We enforced the policy in Figure 6.3, which limits the number of simultaneous socket connections to 5. Such a policy might be used to protect a user from DDoS malware disguised as legitimate web service clients. Socket connection events increment security state `s` until $\texttt{s} = 5$, at which point the next connection event signals a policy violation. Socket close events decrement `s`.

Interestingly, enforcement of this policy uncovered an apparent bug in the application. When a login is successful and the user later logs out, the connection is properly closed; however, connections for failed logins are never properly closed. Thus, six successive failed login attempts trigger a policy violation, and the rewritten program halts.

The original JAR file was 533 KB in size and rewriting decreased it by 59 KB (11%). In this case, the size reduction is a result of the rewriter stripping out unnecessary metadata in the JAR's internal class files. The total processing time was 20.4 seconds.

```
<pointcut name="connect">
    <or><and><call>java.net.Socket.new</call>
              <argtyp num="2">int</argtyp></and>
        <call>java.net.Socket.connect</call></or></pointcut>
<state name="s" />
<forall var="i" from="0" to="4">
    <edge name="inc_connections">
        <nodes var="s">i,i+1</nodes>
        <pointcutid name="connect" /></edge>
</forall>
<forall var="i" from="1" to="5">
    <edge name="dec_connections">
        <nodes var="s">i,i-1</nodes>
        <call>java.net.Socket.close</call></edge>
</forall>
<edge name="six_connections">
    <nodes var="s">5,#</nodes>
    <pointcutid name="connect" /></edge>
```

Figure 6.3. Policy that prohibits more than 5 simultaneous connections

CHAPTER 7
VERIFICATION[1]

7.1 Overview

Software security systems that employ purely static analyses to detect and reject malicious code are limited to enforcing decidable security properties. Unfortunately, most useful program properties, such as safety and liveness properties, are not generally decidable and can therefore only be approximated by a static analysis. For example, signature-based antivirus products accept or reject programs based on their syntax rather than their runtime behavior, and therefore suffer from dangerous false negatives, inconvenient false positives, or both (cf., (Hamlen et al. 2009)). This has shifted software security research increasingly toward more powerful dynamic analyses, but these dynamic systems are often far more difficult to formally verify than provably sound static analyses.

To provide exceptionally high assurance guarantees, recent work has sought to reduce the (potentially large) trusted computing bases of IRM frameworks by separately machine-verifying the self-monitoring code they produce (Hamlen et al. 2006; Aktug and Naliuka 2008; Sridhar and Hamlen 2010b; Sridhar and Hamlen 2011). For example, the S3MS project uses a contract-based verifier (Aktug and Naliuka 2008) to avoid trusting the much larger in-liner (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 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-

[1]This chapter includes previously published (Hamlen et al. 2011) joint work with Kevin Hamlen and Meera Sridhar.

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 are discovered (cf., (Li and Wang 2010; Sridhar and Hamlen 2010a; Sridhar and Hamlen 2010b)). 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.

Therefore, rather than proving that one particular IRM framework correctly modifies all untrusted bytecode instances, we instead consider the challenge of machine-certifying a broad class of instrumented code instances with respect to purely declarative (i.e., advice-free) policy specifications. Unlike contracts, which denote code transformations, policies in our system 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.

The result of our efforts is Chekov✓, a light-weight Java bytecode abstract interpreter and model-checker capable of certifying a large class of realistic IRMs fully automatically, but without introducing the significant overheads typically required for model-checking arbitrary code. Chekov✓ is the first IRM certification system that can verify AOP-style IRMs and IRM-policies without appealing to trusted advice. It extends prior work on model-checking IRMs (Sridhar and Hamlen 2010b; Sridhar and Hamlen 2010a; DeVries et al. 2009) with the following substantial additions:

- Support for a full-scale Java IRM framework (the SPoX IRM system from Chapter 3) 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).
- 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. We begin with an overview of our framework in Section 7.2, providing a high-level description of the verification algorithm. In Section 7.3 we present Chekov✓'s formal mathematical model. A detailed treatment of Chekov✓'s soundness and associated proofs is provided in Section 7.4. Chekov✓'s implementation is discussed in Section 7.5. Finally, Section 7.6 presents in-depth case studies of eight classes of security policies that we enforced on numerous real-world applications, along with a discussion of challenges faced in implementing and verifying these policies.

7.2 System Introduction

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 takes the approach of (Sridhar and Hamlen 2010b), using abstract interpretation to non-deterministically explore all control-flow paths of untrusted code, and 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 7.2.1 (Synchronization Point)**.** *A* synchronization point *(SYNC) is an abstract program state with constraints ζ such that proposition $\zeta \wedge \left(\bigvee_{r \sim a}(r \neq a)\right)$ 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 particular, it verifies that all non-marked instructions are *SYNC*-preserving, and each outgoing

```
<state name="s" />

<forall var="i" from="0" to="9">
  <edge name="count">
    <call>Mail.send</call>
    <nodes var="s">i,i+1</nodes>
  </edge>
</forall>

<edge name="10emails">
  <call>Mail.send</call>
  <nodes var="s">10,#</nodes>
</edge>
```

Figure 7.1. A policy permitting at most 10 email-send events

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.

7.2.1 A Verification Example

Figure 7.2 demonstrates a verification example step-by-step. The pseudocode constitutes a marked region in the target program, which includes code injected by the rewriter to enforce the policy in Figure 7.1 (reproduced for convenience from Section 4.2.2). The verifier requires that the abstract interpreter is in the *SYNC* state immediately before and after blocks of marked code.

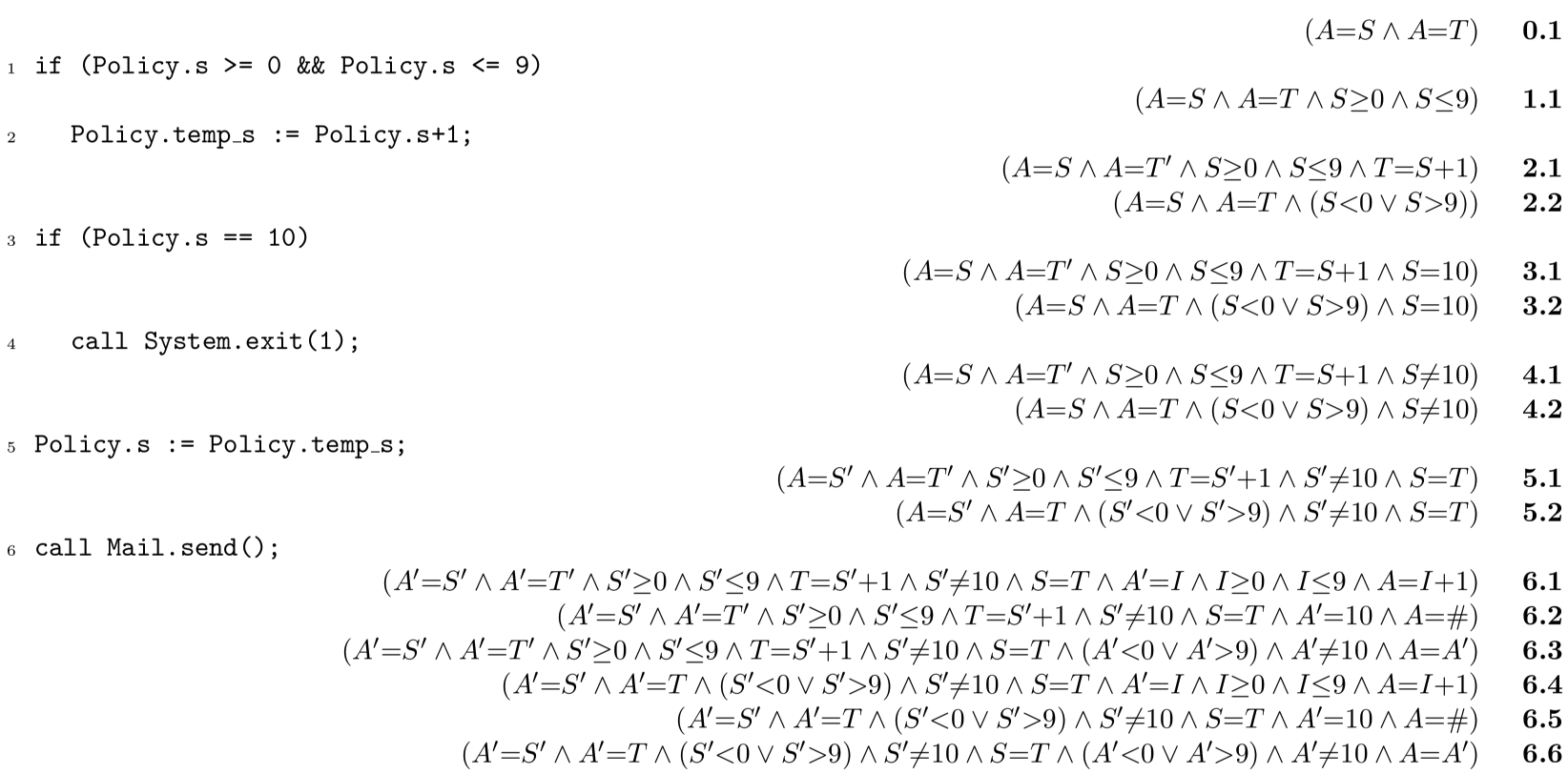

Figure 7.2. An abstract interpretation of instrumented pseudocode

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 7.1. 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.[2]

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 7.1 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.

[2]To account for arithmetic overflow, the + operator actually denotes two's-complement addition over an appropriate bitwidth.

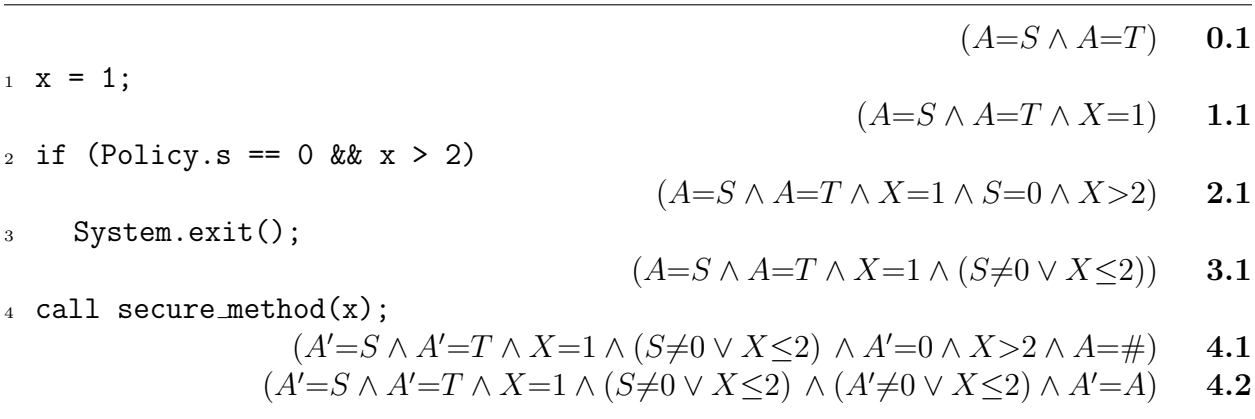

Figure 7.3. An example verification with dynamically decidable pointcuts

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 most $n(m+1)$ states, where m is the number of potential pointcut matches.

If any constraint list is satisfiable and contains the expression $A = \#$, 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.

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.

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 7.3, 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.

Non-numeric dynamic pointcuts are modeled by reducing them to equivalent integer encodings. For example, to support dynamic string regexp-matching (`streq` pointcut expressions), 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.

7.2.2 Limitations

We support some forms of Java reflection, but require that security policies be extended to restrict cases that could circumvent our algorithm. Reflection can be used to invoke methods, instantiate new objects, and alter the values of class fields. Any of those actions can be used to perform security-relevant events in a manner that is difficult to detect, so we prohibit them altogether. Thus, in order to pass verification, rewriters must modify or otherwise restrict reflection in target programs. A less draconian approach would be to have the rewriter modify Java's reflection API to include some of the rewriter's own instrumentation code. Verification would then require a much more complex analysis of the rewritten program, greatly increasing our TCB.

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 Chekov✓ arrives at a fixpoint for every loop by enforcing $SYNC$ at the loop back-edge), 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)). For example, concurrency support requires a race detection analysis such as the one implemented by Racer (Bodden and Havelund 2008).

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 7.4. Core subset of Java bytecode

7.3 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-producing 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.[3] 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 7.4. The proof establishes that executing any program accepted by the certifier never results in a policy violation at runtime.

7.3.1 Java Bytecode Core Subset

Figure 7.4 lists the subset of Java bytecode that we consider. 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**$_y$ n models

[3]The implementation supports the full Java bytecode language.

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

Figure 7.5. Core subset of SPoX

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 $\textbf{event}_y$, in practice it is implemented as a fixed instruction sequence that performs a security-relevant operation (e.g., a system call).

Figure 7.5 defines a core subset of SPoX for the Java bytecode language in Figure 7.4. We here use a simplified LISP-style notation instead of XML for purposes of readability. Without loss of generality, we assume all pointcuts are expressed in disjunctive normal form.

7.3.2 Concrete Machine

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 7.6 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

$$
\begin{array}{ll}
\chi ::= \langle L : i, \rho, \sigma \rangle & \text{(configurations)} \\
L & \text{(code labels)} \\
i & \text{(Java bytecode instructions)} \\
\Sigma : (r \uplus a \uplus \ell) \rightharpoonup \mathbb{Z} & \text{(concrete store mappings)} \\
\sigma \in \Sigma & \text{(concrete stores)} \\
\rho ::= \cdot \mid x :: \rho & \text{(concrete stack)} \\
x \in \mathbb{Z} & \text{(concrete program values)} \\
\chi_0 & \text{(initial configurations)} \\
P ::= (L, p, s) & \text{(programs)} \\
p : L \longrightarrow i & \text{(instruction labels)} \\
s : L \longrightarrow L & \text{(label successors)}
\end{array}
$$

Figure 7.6. Concrete machine configurations and programs

security state variables r to their integer values. A security automaton state is σ restricted to the abstract state variables, denoted $\sigma|_a$.

Figure 7.7 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 7.4 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.

7.3.3 SPoX Concrete Denotational Semantics

The contents of this section are adapted from Section 3.3.1.

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

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

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

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

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

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

Figure 7.7. Concrete small-step operational semantics

$v \in \mathbb{Z}$	integer values
$jp ::= \langle\rangle \mid \langle k, v^*, jp\rangle$	join points
$k ::= \texttt{call}\ c.md \mid \texttt{get}\ c.fd \mid \texttt{set}\ c.fd$	join kinds

Figure 7.8. Join points

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 point, defined in Figure 7.8, 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 $\langle\rangle$.

Figure 7.9 provides denotational semantics for SPoX, and is adapted from Figure 3.5. Without loss of generality, we simplify the semantics such that $q^{\leftarrow} = \{a\}$. The semantics 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 $f[g] = \{(x, f(x)) \mid x \in f^{\leftarrow} \backslash g^{\leftarrow}\} \sqcup g$ 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 $\delta : (Q \times E) \rightarrow 2^Q$. 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).

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 7.9 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

$q \in Q = a \to \mathbb{Z}$	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
$\mathcal{ES} : edg \to \Psi \to 2^{(JP \to \{Succ, Fail\}) \times SM \times SM}$	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 $\delta(q, jp) = \{q[S'] \mid (f, S, S') \in \cup_{1 \le i \le n} \mathcal{ES}[\![edg_i]\!]\bot, S \sqsubseteq q, f(jp) = Succ\}$

$$\mathcal{ES}[\![\texttt{(forall } \hat{v}\texttt{=}e_1\texttt{..}e_2 \ edg\texttt{)}]\!]\psi = \cup_{\mathcal{E}[\![e_1]\!]\psi \le j \le \mathcal{E}[\![e_2]\!]\psi} \mathcal{ES}[\![edg]\!](\psi[j/\hat{v}])$$

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

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

$$\mathcal{EP}[\![\texttt{(nodes "}sv\texttt{" } e_1 \ e_2\texttt{)}]\!]\psi = \big(\{(sv, \mathcal{E}[\![e_1]\!](\psi, \bot))\}, \{(sv, \mathcal{E}[\![e_2]\!](\psi, \bot))\}\big)$$

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

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

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

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

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

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

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

Figure 7.9. Denotational semantics for SPoX

$$
\begin{array}{l}
\mathit{match\text{-}pcd}(\texttt{(call } c.md\texttt{)})\langle \texttt{call } c.md, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(get } c.fd\texttt{)})\langle \texttt{get } c.fd, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(set } c.fd\texttt{)})\langle \texttt{set } c.fd, v^*, jp\rangle = \mathit{Succ} \\
\mathit{match\text{-}pcd}(\texttt{(argval } n\ vp\texttt{)})\langle k, v_0 \cdots v_n \cdots, jp\rangle \\
= \mathit{Succ} \text{ if } vp = \texttt{(true)} \text{ or } (vp = \texttt{(isnull)} \text{ and } v_n = \textsf{null}) \\
\mathit{match\text{-}pcd}(\texttt{(and } \mathit{pcd}_1 \mathit{pcd}_2\texttt{)})jp = \\
\quad \mathit{match\text{-}pcd}(\mathit{pcd}_1)jp \wedge \mathit{match\text{-}pcd}(\mathit{pcd}_2)jp \\
\mathit{match\text{-}pcd}(\texttt{(or } \mathit{pcd}_1 \mathit{pcd}_2\texttt{)})jp = \\
\quad \mathit{match\text{-}pcd}(\mathit{pcd}_1)jp \vee \mathit{match\text{-}pcd}(\mathit{pcd}_2)jp \\
\mathit{match\text{-}pcd}(\texttt{(not } \mathit{pcd}\texttt{)})jp = \neg \mathit{match\text{-}pcd}(\mathit{pcd}) \\
\mathit{match\text{-}pcd}(\texttt{(cflow } \mathit{pcd}\texttt{)})\langle k, v^*, jp\rangle = \\
\quad \mathit{match\text{-}pcd}(\mathit{pcd})\langle k, v^*, jp\rangle \vee \mathit{match\text{-}pcd}(\texttt{(cflow } \mathit{pcd}\texttt{)})jp \\
\mathit{match\text{-}pcd}(\mathit{pcd})jp = \mathit{Fail} \quad \text{otherwise}
\end{array}
$$

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

Figure 7.10. Matching pointcuts to join points

$$
\begin{array}{ll}
\hat{\chi} ::= \bot \mid \langle L{:}i, \zeta, \hat{\rho}, \hat{\sigma} \rangle & \text{(abstract configs)} \\
\zeta ::= \bigwedge_{i=1\ldots n} t_i \quad (n \geq 1) & \text{(constraints)} \\
t ::= T \mid F \mid e_1 \leq e_2 & \text{(predicates)} \\
\hat{\rho} ::= \cdot \mid e :: \hat{\rho} & \text{(abstract stack)} \\
\hat{\Sigma} : (r \uplus \ell) \longrightarrow e & \text{(abstract store mappings)} \\
\hat{\sigma} \in \hat{\Sigma} & \text{(abstract stores)} \\
\hat{\chi}_0 & \text{(initial abstract config)}
\end{array}
$$

Figure 7.11. Abstract machine configurations

syntax in Figure 7.10. Note that that this version of *match-pcd* is largely identical to the one in Figure 3.6, except that it ignores instance state variables and only ever returns *Succ* or $Fail$.

7.3.4 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 transition relation in the abstract domain. Figure 7.11 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.

The small-step operational semantics of the abstract machine are given in Figure 7.12. Rules (AIflePos), (AIfleNeg), and (AEvent) are non-deterministic—the abstract machine

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

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

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

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

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

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

Figure 7.12. Abstract small-step operational semantics

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

Rule (AEvent) is the model-checking step. Its premise appeals to an *abstract denotational semantics* $\hat{\mathcal{P}}$ for SPoX, defined in Figure 7.13, 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 7.13, $\widehat{jp} \in \widehat{JP}$ denotes an *abstract join point*—a join point (see Figure 7.8) 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.

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

$$\hat{\mathcal{P}}[\![edg_1 \ldots edg_n]\!]\hat{jp} = \bigcup_{i=1}^{n} \widehat{\mathcal{ES}}[\![edg_i]\!]\hat{jp}$$

$$\widehat{\mathcal{ES}}[\![\texttt{forall}\ \hat{v}{=}e_1..e_2\ edg]\!]\hat{jp} = \{(\hat{v} \geq e_1) \wedge (\hat{v} \leq e_2) \wedge \zeta \mid \zeta \in \widehat{\mathcal{ES}}[\![edg]\!]\hat{jp}\}$$

$$\widehat{\mathcal{ES}}[\![\texttt{edge}\ pcd\ ep_1 \ldots ep_n]\!]\hat{jp} = \{\zeta \wedge (\wedge_{i=1}^{n} \widehat{\mathcal{EP}}[\![ep_i]\!]) \mid \zeta \in \widehat{\mathcal{PCD}}[\![pcd]\!]\hat{jp}\}$$

$$\widehat{\mathcal{PCD}}[\![\texttt{or}\ pcc_1 \ldots pcc_n]\!]\hat{jp} = \{\widehat{\mathcal{PCC}}[\![pcc_i]\!]\hat{jp} \mid 1 \leq i \leq n\}$$

$$\widehat{\mathcal{PCC}}[\![\texttt{and}\ pct_1 \ldots pct_n]\!]\hat{jp} = \bigwedge_{i=1}^{n} \widehat{\mathcal{PCC}}[\![pct_i]\!]jp$$

$$\widehat{\mathcal{PCC}}[\![\texttt{not}\ pca]\!]\hat{jp} = \neg(\widehat{\mathcal{PCC}}[\![pca]\!]\hat{jp})$$

$$\widehat{\mathcal{PCC}}[\![\texttt{event}_y\ n]\!]\langle \texttt{event}_z\ m, e^*, \hat{jp}\rangle = (n{=}m)$$

$$\widehat{\mathcal{PCC}}[\![\texttt{arg}\ n\ (\texttt{intleq}\ m)]\!]\langle k, e_1{::}\cdots{::}e_n{::}e^*, \hat{jp}\rangle = (e_n \leq m)$$

$$\widehat{\mathcal{EP}}[\![\texttt{nodes}\ a\ e_1\ e_2]\!] = (a_0 = e_1) \wedge (a = e_2)$$

Figure 7.13. Abstract Denotational Semantics

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

Figure 7.14. State-ordering relation $\preceq_{\hat{\chi}}$

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.

7.3.5 Abstract Interpretation

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 $(\mathcal{A}, \preceq_{\hat{\chi}})$ is defined in Figure 7.14. This reduces the number of control-flows that an implementation of the abstract machine must explore.

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

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

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

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

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

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

Figure 7.15. Soundness relation $\sim$

7.4 Soundness

The abstract machine (defined in Section 7.3.4) is *sound* with respect to the concrete machine (defined in Section 7.3.2) 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. 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 7.15.

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 7.4.1 (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 7.4.2. *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 7.4.3. *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 7.4.4. *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 7.4.5. *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* $\psi''(a_0) = \sigma(a)$.

Proof. Proofs of Lemmas 7.4.2–7.4.5 follow from a straightforward expansion of the definitions in Figs. 7.9 and 7.5. □

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 7.4.6 (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 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 $jp = \langle \textbf{event}_y\ n, x_1{::}x_2{::}\cdots{::}x_y{::}\cdot, \langle\rangle\rangle$. 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 7.4.1. 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 7.4.1 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 7.4.7. *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 7.4.8 (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 $i =$ **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 $\langle L :$ **ifle** L', ζ, $e_1{::}e_2{::}\hat{\rho}_r$, $\hat{\sigma} \rangle$. 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 (Stk-Sound) proves that $\mathcal{E}[\![e_1]\!]\mu = x_1$ and $\mathcal{E}[\![e_2]\!]\mu = x_2$. 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 (CGetLocal) proves that $i = \textbf{getlocal}\ \ell$, and $\chi' = \langle s(L) : p(s(L)), \sigma(\ell)::\rho, \sigma\rangle$. Relation $\chi \sim \hat{\chi}$ proves that $\hat{\chi}$ has the form $\langle s(L) : p(s(L)), \zeta, \hat{\rho}, \hat{\sigma}\rangle$. Choose $\hat{\chi}' = \langle L : \textbf{getlocal}\ \ell, \zeta,\ \hat{\sigma}(\ell)::\hat{\rho}, \hat{\sigma}\rangle$, 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) $\mu, \hat{\sigma}(\ell)::\hat{\rho} \models \sigma(\ell)::\rho$ can be derived with Rule (Stk-Sound) by combining $\mathcal{E}[\![\hat{\sigma}(\ell)]\!]\mu = \sigma(\ell)$ (from Rule (Str-Sound)) and (2) above.

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

Case (CSetLocal)**:** Rule (CSetLocal) proves that $i = \textbf{setlocal}\ \ell$, 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 $\langle L : \textbf{setlocal}\ \ell, \zeta, e::\hat{\rho}_r, \hat{\sigma}\rangle$.

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 7.4.7 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 7.4.7 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)**:** Rule (CEVENT) proves 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 7.4.1 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 7.12.

(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 7.4.9 (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 7.7, whose premise guarantees that the event does not violate the policy. □

7.5 Implementation

Our prototype verifier implementation consists of 5200 lines of Prolog code and 9100 lines of Java code. The Prolog code runs under 32-bit SWI-Prolog 5.10.4, which communicates with our Java libraries using the JPL interface. The Java side handles the parsing of SPoX policies and input Java binaries, and compares Java 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.

In this section, we discuss interesting parts of the implementation that were not previously addressed in Section 7.2. In Section 7.5.1, we describe how we use SWI-Prolog's CLP system to do linear constraint analysis. In Section 7.5.2, we discuss how the abstract interpreter considers all possible control flows through a program.

7.5.1 Linear Constraints

Linear constraints are modeled as lists of lists of mathematical expressions. For example, consider the following Prolog list:

```
C = [[A=S,A=T,X<1],[A=S,A=T1,T=S+1,X>=1]]
```

The list `C` contains two possible lists of constraints. In the first, the abstract state variable `A` remains in synchronization with both reified state variables `S` and `T`, and a runtime variable `X` is less than `1`. In the second, `A` remains equal to `S`, but `T` has been assigned a new value equal to `S+1`, and the old `T` has been replaced by a new placeholder variable `T1`. Also, runtime variable `X` is known to be greater than or equal to `1`.

Constraints are continuously and aggressively checked for satisfiability through SWI-Prolog's CLP system. To check a single list of constraint expressions, we use the recursive `add_constraints` predicate:

```
add_constraints([]).
add_constraints([C|Cs]) :- {}(C),add_constraints(Cs).
```

The predicate fails if the list of expressions proves to be unsatisfiable. If an entire list of lists of constraints is unsatisfiable, then Chekov✓ knows that a control flow is impossible and halts abstract interpretation along that execution path.

An undesired side effect of SWI-Prolog's CLP system is its tendency to unify variables with actual values wherever possible, so we alpha-convert all constraint variables to fresh variables prior to calling `add_constraints`. The predicate below alpha-converts a single variable in a single expression:

```
% alpha_convert(+OldVar,+NewVar,+OldExpr,-NewExpr)
alpha_convert(OldVar,NewVar,OldExpr,NewVar) :-
    OldExpr == OldVar.
alpha_convert(OldVar,_NewVar,Expr,Expr) :-
    \+compound(Expr),
    Expr \== OldVar.
alpha_convert(OldVar,NewVar,OldExpr,NewExpr) :-
    compound(OldExpr),
    OldExpr \== OldVar,
    OldExpr =.. [Functor,Arg],
    alpha_convert(OldVar,NewVar,Arg,NewArg),
    NewExpr =.. [Functor,NewArg].
alpha_convert(OldVar,NewVar,OldExpr,NewExpr) :-
    compound(OldExpr),
    OldExpr \== OldVar,
    OldExpr =.. [Functor,Arg1,Arg2],
    alpha_convert(OldVar,NewVar,Arg1,NewArg1),
    alpha_convert(OldVar,NewVar,Arg2,NewArg2),
    NewExpr =.. [Functor,NewArg1,NewArg2].
```

Alpha-conversion allows us to maintain ambiguity in the values of our constraint variables, ensuring soundness throughout abstract interpretation.

In order to prove an implication of the form $A \Rightarrow B$ (e.g., to prove that a constraint list implies $SYNC$), we submit an expression of the form $A \wedge \neg B$ to the constraint solver. The implication is true if and only if that expression is unsatisfiable.

SWI-Prolog's CLP system natively supports conjunction and disjunction using the "," and ";" characters, but it does not directly support negation. Therefore, we use DeMorgan's Laws to convert expressions into their negated forms. The following predicate submits the negation of a given constraint list to the CLP solver:

```
add_constraints_neg([(',')(C1,C2))|Cs]) :-
    add_constraints_neg([C1,C2]) ;
    add_constraints_neg(Cs).
add_constraints_neg([(';')(C1,C2))|Cs]) :-
    (add_constraints_neg([C1]),add_constraints_neg([C2])) ;
    add_constraints_neg(Cs).
add_constraints_neg([(C1=C2)|Cs]) :-
    {}(C1=\=C2) ; add_constraints_neg(Cs).
add_constraints_neg([(C1=\=C2)|Cs]) :-
    {}(C1=C2) ; add_constraints_neg(Cs).
add_constraints_neg([(C1>C2)|Cs]) :-
    {}(C1=<C2) ; add_constraints_neg(Cs).
add_constraints_neg([(C1>=C2)|Cs]) :-
    {}(C1<C2) ; add_constraints_neg(Cs).
add_constraints_neg([(C1<C2)|Cs]) :-
    {}(C1>=C2) ; add_constraints_neg(Cs).
add_constraints_neg([(C1=<C2)|Cs]) :-
    {}(C1>C2) ; add_constraints_neg(Cs).
```

7.5.2 Non-deterministic Abstract Interpretation

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 contains no potential pointcut matches or possible writes to reified security state. If any such code is detected during that scan, the program is rejected before abstract interpretation even begins.

The abstract interpreter steps through each marked bytecode instruction, inferring any necessary information regarding local variables, the runtime operand stack, and objects on the heap. Local variable stores and the operand stack are both modeled using lists. In the case of the operand stack, the top element is the head of its representative list. All heap variables (including object and field references) are encoded as simple Prolog variables, and are moved in and out of local variable registers and the stack as appropriate. Constants are tracked as equivalent Prolog atoms. Wherever possible, Chekov✓ adds relational information to the constraints.

In our implementation, the target program is required to maintain the *SYNC* invariant from Definition 7.2.1 at the beginning and end of every loop. This speeds up verification, as there is no need to consider the effect of multiple loop iterations on the security state.

Similarly, all methods are required to maintain *SYNC* at their entrypoints and at all return instructions. This obviates the need to perform complex inter-procedural analysis, and allows Chekov✓ to independently verify any methods that generate intractable control flows, such as recursive methods, static initializers, and call-backs (e.g., event handlers). Marked policy enforcement methods are treated as if they were in-lined in place of their invocations. A call graph analysis confirms that there is no way for a policy enforcement method to recursively call itself; if it can, the verifier rejects.

The *execution environment* is represented throughout our code as `EE`, and is a compound variable of the following form:

```
EE = env(C, M, OS, LVs, IH)
```

Lists `OS` and `LVs` represent the operand stack and the local variables, respectively. Variables `C`, `M`, and `IH` are actually pointers to Java BCEL `ClassGen`, `MethodGen`, and `InstructionHandle` objects, respectively. Each such object contains detailed information about the containing class, the containing method, and the current instruction. Whenever the abstract interpreter needs specific information (e.g., the name of the method), it calls Java methods through the JPL interface.

We treat invoked methods as in-lined, but we must still have some way of tracking the call stack so that we can maintain separation of operand stacks, local variables, and lexical context. We therefore model the call stack as a list of execution environments, where `EE`s are appended and removed at method invocations and returns, respectively. Recall that we do not do this for recursive method calls, as they would cause the call stack to become infinitely large.

Note that we always begin abstract interpretation at the beginning of the program's execution, so the bottom `EE` on the call stack always refers to the starting method even if it contains no marked code. Likewise, any method involved in a control flow to marked code is included in the call stack, even if it contains no marked code itself. For example, if starting method A calls method B, which in turn calls method C, and C contains marked, security-relevant code, then the bottom three elements of the call stack are execution environments with A, B, and C as lexical contexts.

Wherever a conditional branch occurs, all paths are explored non-deterministically, and any possible inferences are stored in the constraints. For example, the integer comparison instruction `if_icmple` is handled by the following (simplified) state transition predicate:

```
% branch_instr('if_icmple',+State,-NewState)
branch_instr('if_icmple', State, NewState) :-
    State = state([EE|EEs],Constraints),
    EE = env(C, M, OS, LVs, IH),
    OS = [O1,O2|OS1],
    update_constraints((O2=<O1),Constraints,NewConstraints),
```

```
    check_all_constraints(NewConstraints),
    get_next_instr(C,M,IH,true,PosIH),
    PosEE = env(C, M, OS1, LVs, PosIH),
    NewState = state([PosEE|EEs],NewConstraints).
branch_instr('if_icmple', State, NewState):-
    State = state([EE|EEs],Constraints),
    EE = env(C, M, OS, LVs, IH),
    OS = [O1,O2|OS1],
    update_constraints((O2>O1),Constraints,NewConstraints),
    check_all_constraints(FW),
    get_next_instr(C,M,IH,false,NegIH),
    NegEE = env(C, M, OS1, LVs, NegIH),
    NewState = state([NegEE|EEs],NewConstraints).
```

The predicate `update_constraints` appends the given expression to every constraint list in the list of constraint lists. The predicate `check_all_constraints` checks every constraint list for satisfiability, and removes any that are unsatisfiable. If all constraints are unsatisfiable, it fails, thus pruning that control flow from the abstract interpreter's search tree. Finally, the predicate `get_next_instr` obtains the next instruction in the chosen control flow. The code above uses an alternate rule for the predicate that selects the appropriate "`true`" or "`false`" branch of the `if` instruction.

Whenever the abstract interpreter encounters an invocation whose called method is resolved at runtime (e.g., an interface call), it non-deterministically branches to all possible methods. To make this approach feasible, the verifier's initialization stage looks at all methods in the Java classpath and stores their signatures in a hierarchical database. Chekov✓ uses this database to determine all methods that could be called by a given invoke instruction.

For any instruction that can throw one or more exceptions, Chekov✓ non-deterministically branches to all appropriate handlers. For example, an array reference instruction that can throw an `ArrayIndexOutOfBoundsException` may spawn a control flow that branches to a

generic `Exception` handler. If the current method lacks a handler that catches the exception, we must determine whether one could exist elsewhere on the runtime call stack. In such cases, our interpreter pops execution environments from the stack until either the stack is empty or a handler is found. If a handler is not found, then we may safely say that the exception would be uncaught at runtime, causing the program to halt. Therefore, no new control flows are considered by the abstract interpreter.

If the interpreter does not directly enter a method at its point of invocation (e.g., a library or recursive method), then it branches from that invoke instruction based upon all possible exceptions that the method could throw. These include all thrown exceptions listed in the method's signature, as well as any unchecked exceptions (e.g., `ArithmeticException`). If an exception is later thrown within a method undergoing independent analysis, it is not caught by anything on the (necessarily incomplete) call stack, and occurs while the current constraints all imply *SYNC*, then it is ignored. Any such exception-driven control flow must have been considered during a prior analysis, when the method was not entered at its point of invocation. In order to maintain soundness with this approach, we require all exception handlers to begin with constraints equal to *SYNC*.

Chekov✓ memoizes to drastically reduce time spent in verification. Exceptions in particular can produce an intractable number of control flows, but most of those flows are likely to be abstractly equivalent. We use Prolog assertions to globally declare that the interpreter has reached a given instruction with particular local variables, stack contents, and constraint lists. If a control flow creates a branch to that instruction with constraints and an execution environment of the same form as one that was previously asserted, that branch is pruned from the search tree. The pruning is safe because no new information could be learned on a repeat traversal of that control flow; if a policy violation was ruled out before, we know that it would be ruled out again.

Table 7.1. Verification 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

7.6 Case Studies

We have used our prototype implementation to rewrite and subsequently verify numerous Java applications. These case studies are discussed throughout the remainder of the section, and statistics are summarized in Table 7.1. Once again, 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.

In Table 7.1, each cell in the File Sizes column has three parts: the original size of the main program before rewriting,[4] the size after rewriting, and the size of included system libraries that needed to be verified (but not rewritten) as part of verifying the rewritten code. Verification of system library code is required to verify the safety of various control-flows that pass through them. Likewise, each cell in the No. 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 7.1 (`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 the issue 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.

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.

[4] In order to make the comparison more meaningful, we stripped out unnecessary meta-data and compressed the original class files to obtain initial size.

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

Figure 7.16. `NoExecSaves` policy

7.6.1 Filename Guards

Figure 7.16 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 5 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 hieroglyphics 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. For `vrenamer`, a mass file-renaming application, we prohibited files being renamed to include executable extensions.

```
<edge name="FileRead">
  <nodes var="s">0,1</nodes>
  <and>
    <call>java.io.File.*</call>
    <argval num="1"><streq>[A-Za-z]*:\\windows\\.*"</streq></argval>
  </and>
</edge>
<edge name="NetworkSend">
  <nodes var="s">1,#</nodes>
  <call>java.net.Socket.getOutputStream</call>
</edge>
```

Figure 7.17. `NoSendsAfterReads` policy

7.6.2 Event Ordering

Figure 7.17 is essentially the same policy as the one in Figure 6.1, and encodes the canonical information flow policy in the IRM literature that prohibits all network-send operations after a sensitive file has been read (cf. Figure 3.1). Specifically, this policy prevents calls to `Socket.getOutputStream` 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`. These programs are listed in Table 7.1 as `jWeather` and `YTDownload`, respectively. 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 the program's built-in encryption method. The two-state security policy is structurally similar to the one in the Figure 7.17.

```
<edge name="no_gui">
  <nodes var="s">0,#</nodes>
  <and>
    <call>jfilecrypt.GuiMainController.new</call>
    <withincode>jfilecrypt.Application.main</withincode>
  </and>
</edge>
```

Figure 7.18. `NoGui` policy

7.6.3 Pop-up Protection

The `NoGui` policy in Figure 7.18 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.

7.6.4 Port Restriction

Policies such as the one in Figure 7.19 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.

We also enforced a similar policy on `jrdesktop`, a remote desktop client, prohibiting the use of ports less than 1000. For `jknightcommander` (`jknight` in Table 7.1), 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.

```
<edge name="badPort">
  <nodes var="s">0,#</nodes>
  <and>
    <set>Config.port</set>
    <or>
      <argval num="1"><intgt>29</intgt></argval>
      <argval num="1"><intlt>20</intlt></argval>
    </or>
  </and>
</edge>
```

Figure 7.19. `SafePort` policy

7.6.5 Resource Bounds

In Section 4.2.2, we described a policy which prohibits an email client from sending more than 10 emails in a given execution. The policy is reproduced in Figure 7.1. We enforced it 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.

7.6.6 Anti-freeriding

Figure 7.20 reproduces the policy from Section 4.3.2 that prohibits freeriding in file-sharing clients. We enforced the policy on `xnap` and `Phex`.

```
1 <forall var="i" from="-10000" to="1">
2   <edge name="download">
3     <nodes var="s">i,i+1</nodes>
4     <call>Download.download</call>
5   </edge>
6 </forall>
7 <forall var="i" from="-9999" to="2">
8   <edge name="upload">
9     <nodes var="s">i,i-1</nodes>
10    <call>Upload.upload</call>
11  </edge>
12 </forall>
13 <edge name="too_many_downloads">
14   <nodes var="s">2,#</nodes>
15   <call>Download.download</call>
16 </edge>
```

Figure 7.20. `NoFreeRide` policy

7.6.7 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 7.21. 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 8 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 por-

```
<edge name="SQL_Injection_occurred">
  <nodes var="s">0,#</nodes>
  <and>
    <call>Login.login"</call>
    <not><argval num="1"><streq>[a-zA-Z0-9]*</streq></argval></not>
  </and>
</edge>
<edge name="XSS_injection_occurred">
  <nodes var="s">0,#</nodes>
  <and>
    <call>Employee.new</call>
    <not>
      <and>
        <argval num="2"><streq>[A-Za-z_0-9,.\-\s]*</streq></argval>
        <argval num="3"><streq>[A-Za-z_0-9,.\-\s]*</streq></argval>
                 ...
        <argval num="16"><streq>[A-Za-z_0-9,.\-\s]*</streq></argval>
      </and>
    </not>
  </and>
</edge>
```

Figure 7.21. `NoSqlXss` policy

tion 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.

7.6.8 Ensuring Advice Execution

Most aspectual policy languages (e.g., (Chen and Roşu 2005; Aktug and Naliuka 2008; Erlingsson 2004; Ligatti et al. 2009)) allow policies to include explicit advice code that implements IRM guards and interventions. Such systems can be applied to create custom implementations of SPoX policies, such as those that perform custom actions when impending violations are detected. Chekov✓ can then take the SPoX policy as input and verify that the implementation correctly enforces 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 added encryption and logging calls immediately prior to email-send events in `JVMail`. Each email is therefore encrypted, then logged, then sent. The `LogEncrypt` policy in Figure 7.22 requires these events to occur in that order. After inserting the advice, we 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`), only allowing file deletion after execution of advice code that consults the user. Finally, we enforced a policy on `sshwebproxy` that requires the proxy to encrypt messages before sending.

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

Figure 7.22. `LogEncrypt` policy

CHAPTER 8
CONCLUSIONS

This dissertation presents a purely declarative, aspect-oriented security policy language, along with an accompanying IRM enforcement system. We have aggressively sought to minimize the system's TCB, excluding advice from the policy language and providing a verifier for rewritten code. We have also developed helpful tools for debugging policies and rewriting untrusted code via a web service.

Chapter 3 presented SPoX. We discussed how SPoX code uses pointcuts to denote security-relevant events, and how it declaratively describes automaton state transitions without imperative advice. In addition, we defined a formal denotational semantics for SPoX, which allows us to precisely identify the meaning of a policy, and what it means for a program to adhere to it. We provided a general algorithm for rewriting untrusted programs, including a way to synthesize enforcement code from declarative state transition information.

In Chapter 4, we presented a rewriter that automatically modifies untrusted binaries to enforce SPoX policies. We described the basic approach of the system, and discussed our solutions to some of the interesting problems we encountered in its development. Case studies showed that the rewriter could be used on a variety of real-world applications, with a realistic runtime overhead.

In Chapter 5, we presented a tool that automatically detects inconsistencies in SPoX policies. Specifically, it determines if the security automaton denoted by a policy is non-deterministic. We presented an algorithm that detects security state non-determinism and pointcut non-determinism; when both occur simultaneously, the policy is considered ambiguous. We proved the correctness of the constraint generation part of the pointcut non-determinism detection algorithm using ACL2. Our implementation of the tool found am-

biguities in several SPoX policies, exhibiting its usefulness in debugging errors that are not always obvious.

Chapter 6 described how IRMs lend themselves to service-oriented architectures, and presented a web service implementation of our rewriter. Our approach targets devices which cannot reliably obtain and run the rewriter itself, as all code analysis and instrumentation occurs at a central server.

Finally, Chapter 7 presented Chekov✓, a trusted certifier for rewritten binaries. Chekov✓ abstractly interprets a binary, non-deterministically tracking the security state using a linear constraint analysis, and either accepts the program as safe with respect to a SPoX policy or conservatively rejects. We formally modeled the operational semantics of a concrete machine over a core Java bytecode subset, and proved that the corresponding behavior of our abstract machine is sound. We discussed interesting parts of the verifier implementation, the core of which was written in Prolog. We provided the experimental results of submitting twenty-five real-world programs to both our rewriter and Chekov✓.

We consider these systems to be significant contributions to the field; however, there remain numerous avenues of future work that would improve the practicality and widen the applicability of our approach. Several of these future directions are summarized below.

Multithreading and reflection remain outstanding problems for certifying IRM systems. In order to manage multithreading properly, rewriters and verifiers must consider race conditions while avoiding issues of deadlock and inefficiency. Policy writers can also help to mitigate such issues by creating race-free policies. Fully supporting reflection requires that reflection APIs be instrumented with the rewriter itself, greatly complicating the verification process.

SPoX is expressive enough to denote many kinds of policies, but some specialized policies cannot be described without additional tools and extensions. For example, policies that involve mappings between objects require declarative extensions to the language that can be used to track such relationships at runtime. Sophisticated management of policy violations

(e.g., roll-back) typically requires specific enforcement advice, which could be provided to rewriters as a secondary input apart from the declarative SPoX policy.

Rewriting sometimes impacts code efficiency, necessitating more sophisticated instrumentation algorithms. For example, guard code placed inside a processor-intensive loop can sometimes be lifted outside of that loop, effectively combining multiple updates to the security state. However, complex rewriting approaches make certification more difficult, requiring even more complex verification algorithms.

Correctly specifying security policies can be difficult in any language, suggesting a need for good visualization tools. Bytecode analysis tools make it easier to reason about the effects of program events on the security state, as well as how rewriting will affect an application's execution. As SPoX policies denote security automata, a graphical policy viewer and development environment has the potential to be a very useful tool.

Finally, although our system targets Java bytecode, there are many other types of binaries to consider in any large-scale IRM system. Adapting SPoX rewriters and verifiers to other bytecode languages, such as .NET and Python, should be straightforward. IRM frameworks targeting x86 machine code are much more difficult to develop, but are a necessary part of future work.

REFERENCES

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

Apache Software Foundation (2006). Byte code engineering library. `http://jakarta.apache.org/bcel/`.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia (2009, February). Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, USA.

Barvinok, A. and J. E. Pommersheim (1999). An algorithmic theory of lattice points in polyhedra. In *New Perspectives in Algebraic Combinatorics*, Volume 38, pp. 91–147.

Bauer, L., J. Ligatti, and D. Walker (2005, June). Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, USA, pp. 305–314.

Bodden, E. and K. Havelund (2008, July). Racer: effective race detection using AspectJ. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, WA, USA, pp. 155–166. ACM.

Chappell, D. (2010, October). Introducing the Azure services platform. `http://microsoft.com/windowsazure/Whitepapers/introducingwindowsazureplatform/`.

Chen, F. and G. Roşu (2005, April). 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)*, Edinburgh, UK, pp. 546–550. Springer.

Chudnov, A. and D. A. Naumann (2010, July). Information flow monitor inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, Edinburgh, UK, pp. 200–214.

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 Symposium on Principles of Programming Languages (POPL)*, Los Angeles, CA, USA, pp. 238–252. ACM.

Cousot, P. and R. Cousot (1992, August). Abstract interpretation frameworks. *Journal of Logic and Computation 2*(4), 511–547.

Dam, M., B. Jacobs, A. Lundblad, and F. Piessens (2009, July). Security monitor inlining for multithreaded Java. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, Genova, Italy, pp. 546–569. Springer-Verlag.

Dantas, D. S. and D. Walker (2006, January). Harmless advice. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL)*, Charleston, SC, USA, pp. 383–396.

Dantas, D. S., D. Walker, G. Washburn, and S. Weirich (2008, May). AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems 30*(3), 14:1–14:60.

DeLine, R. and M. Fähndrich (2004, June). Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*, Oslo, Norway, pp. 465–490.

Desmet, L., W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe (2007, November). A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of the 2007 ACM Computer Security Architecture Workshop (CSAW)*, Fairfax, VA, USA, pp. 19–28. ACM.

Desmet, L., W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe (2009, December). The S3MS.NET run time monitor. *Electronic Notes in Theoretical Computer Science 253*(5), 153–159.

DeVries, B. W., G. Gupta, K. W. Hamlen, S. Moore, and M. Sridhar (2009, June). ActionScript bytecode verification with co-logic programming. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Dublin, Ireland. ACM.

Douence, R., P. Fradet, and M. Südholt (2002, October). A framework for the detection and resolution of aspect interactions. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE)*, Pittsburgh, PA, USA, pp. 173–188. Springer-Verlag.

Douence, R., P. Fradet, and M. Südholt (2004, March). Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, pp. 141–150.

Dragoni, N., F. Massacci, K. Naliuka, and I. Siahaan (2007, June). Security-by-contract: Toward a semantics for digital signatures on mobile code. In *European PKI Workshop: Theory and Practice (EuroPKI)*, Mallorca, Balearic Islands, Spain, pp. 297–312.

ECMA (2002, December). *ECMA-335: Common Language Infrastructure (CLI)* (Second ed.). Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems).

Eén, N. and N. Sörensson (2007). MiniSat. `http://minisat.se/`.

Endoh, Y., H. Masuhara, and A. Yonezawa (2006, March). Continuation join points. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop (FOAL)*, Bonn, Germany, pp. 1–10.

Erlingsson, U. (2004). *The inlined reference monitor approach to security policy enforcement*. Ph. D. thesis, Cornell University, Ithaca, NY, USA. Adviser-Schneider, Fred B.

Erlingsson, Ú., M. Abadi, M. Vrable, M. Budiu, and G. C. Necula (2006, November). XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, USA, pp. 75–88. ACM.

Erlingsson, Ú. and F. B. Schneider (1999, September). SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, Caledon Hills, Ontario, Canada, pp. 87–95. ACM.

Evans, D. and A. Twynman (1999, May). Flexible policy-directed code safety. In *Proceedings of the 20th IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 32–45.

FileInfo.com (2011). Executable file types. `www.fileinfo.com/filetypes/executable`.

Flatt, M., S. Krishnamurthi, and M. Felleisen (1998, January). Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, USA, pp. 171–183. ACM.

Formal Systems Laboratory (2011). JavaMOP 2.0 Finite State Machine (JavaFSM).

Garfinkel, S. L. (2007). An evaluation of Amazon's grid computing services: EC2, S3 and SQS. Technical Report TR-08-07, School for Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA.

Ghemawat, S., H. Gobioff, and S.-T. Leung (2003, October). The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, pp. 29–43. ACM.

Google.mE (2010). `http://sourceforge.net/projects/googleme`.

Hamlen, K. W. (2006, August). *Security Policy Enforcement by Automated Program-rewriting*. Ph. D. thesis, Cornell University, Ithaca, NY, USA.

Hamlen, K. W. and M. Jones (2008). Aspect-oriented in-lined reference monitors. In *Proceedings of the ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, Tucson, AZ, USA, pp. 11–20. ACM.

Hamlen, K. W., M. Jones, and M. Sridhar (2011, May). Chekov: Aspect-oriented runtime monitor certification via model-checking (extended version). Technical Report UTDCS-16-11, Computer Science Department, The University of Texas at Dallas, Richardson, Texas.

Hamlen, K. W., V. Mohan, M. M. Masud, L. Khan, and B. Thuraisingham (2009, November). Exploiting an antivirus interface. *Computer Standards & Interfaces Journal 31*(6), 1182–1189.

Hamlen, K. W., V. Mohan, and R. Wartell (2010, June). Reining in Windows API abuses with in-lined reference monitors. Technical Report UTDCS-18-10, Computer Science Department, The University of Texas at Dallas, Richardson, Texas.

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

Jaffar, J. and M. J. Maher (1994). Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jeti (2007). `http://jeti.sourceforge.net`.

Jones, M. and K. W. Hamlen (2009, June). Enforcing IRM security policies: Two case studies. In *Proceedings of the IEEE Intelligence and Security Informatics Conference (ISI)*, Dallas, TX, USA, pp. 214–216.

Jones, M. and K. W. Hamlen (2010, March). Disambiguating aspect-oriented policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*, Rennes and Saint Malo, France, pp. 193–204. ACM.

Jones, M. and K. W. Hamlen (2011, September). A service-oriented approach to mobile code security. In E. Shakshuki and M. Younas (Eds.), *Proceedings of the 8th International Conference on Mobile Web Information Systems (MobiWIS)*, Niagara Falls, Ontario, pp. 531–538.

jWeatherWatch (2009). `http://code.google.com/p/jweatherwatch`.

Kallel, S., A. Charfi, M. Mezini, M. Jmaiel, and K. Klose (2009, February). From formal access control policies to runtime enforcement aspects. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)*, Leuven, Belgium, pp. 16–31.

Katz, E. and S. Katz (2010, March). Semantic aspect interactions and possibly shared join points. In *Proceedings of the 9th International Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, Rennes, France, pp. 7–12.

Kaufmann, M. and J. S. Moore (2006). ACL2. `http://www.cs.utexas.edu/~moore/acl2/`.

Kaufmann, M. and J. S. Moore (2011). ACL2: Interesting Applications. `http://www.cs.utexas.edu/users/moore/acl2/v4-3/INTERESTING-APPLICATIONS.html`.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold (2001, June). An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, Volume 2072, Budapest, Hungary, pp. 327–355. Springer.

Kiczales, G., J. Lamping, A. Medhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin (1997, June). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Volume 1241, Jyväskylä, Finland, pp. 220–242. Springer.

Kim, M., M. Viswanathan, S. Kannan, I. Lee, and O. V. Sokolsky (2004, March). Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design 24*(2), 129–155.

Laddad, R. (2002, January). I want my AOP!, part 1. `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html`.

Li, Z. and X. Wang (2010, December). FIRM: capability-based inline mediation of Flash behaviors. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, USA, pp. 181–190. ACM.

Ligatti, J., L. Bauer, and D. Walker (2005, September). Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, Milan, Italy, pp. 355–373. Springer.

Ligatti, J., L. Bauer, and D. Walker (2009, January). Run-time enforcement of nonsafety policies. *ACM Transactions on Information and Systems Security 12*(3), 19:1–19:41.

Lindholm, T. and F. Yellin (1999). *The Java Virtual Machine Specification* (Second ed.). Addison-Wesley.

Massacci, F., F. Massacci, K. Naliuka, and K. Naliuka (2006, November). Multi-session security monitoring for mobile code. Technical Report DIT-06-067, University of Trento, Trento, Italy.

McAfee, Inc. (2011). McAfee SaaS Total Protection. `http://www.mcafee.com/us/products/saas-total-protection.aspx`.

McCamant, S. and G. Morrisett (2006, July). Evaluating SFI for a CISC architecture. In *Proceedings of the 15th Conference on USENIX Security Symposium*, Vancouver, BC, Canada, pp. 209–224. USENIX Association.

Necula, G. C. (1997, January). Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, Paris, France, pp. 106–119. ACM.

Oberheide, J., E. Cooke, and F. Jahanian (2008, July). CloudAV: N-version antivirus in the network cloud. In *Proceedings of the 17th conference on Security Symposium (SS)*, San Jose, CA, USA, pp. 91–106. USENIX Association.

Patwardhan, A., K. W. Hamlen, and K. Cooper (2010, October). Towards security-aware program visualization for analyzing in-lined reference monitors. In *Proceedings of the International Workshop on Visual Languages and Computing (VLC)*, Oak Brook, IL, USA, pp. 257–260.

Ronayne, M. L. and E. S. Townsend (1996, October). A case study: Distributed object technology at Wells Fargo bank. The Cushing Group, Inc.

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

Sekar, R., V. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney (2003, October). Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, USA, pp. 15–28. ACM.

Shah, V. and F. Hill (2003, April). An aspect-oriented security framework. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, Volume 2, Washington, DC, USA, pp. 143–145.

Sridhar, M. and K. W. Hamlen (2010a, January). ActionScript in-lined reference monitoring in Prolog. In *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, Madrid, Spain, pp. 149–151. Springer.

Sridhar, M. and K. W. Hamlen (2010b, January). Model-checking in-lined reference monitors. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Madrid, Spain, pp. 312–327. Springer.

Sridhar, M. and K. W. Hamlen (2011, January). Flexible in-lined reference monitor certification: Challenges and future directions. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV)*, Austin, TX, USA, pp. 55–60. ACM.

The AspectJ Team (2003). The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.

Townsend, E. (2008, July). The 25-year history of service-oriented architecture. `http://www.eriktownsend.com/white-papers-technology/doc_view/4-1-the-25-year-history-of-service-oriented-architecture.raw`.

Vanoverberghe, D. and F. Piessens (2008, June). A caller-side inline reference monitor for an object-oriented intermediate language. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, Oslo, Norway, pp. 240–258. Springer-Verlag.

Vanoverberghe, D. and F. Piessens (2009, July). Security enforcement aware software development. *Information and Software Technology 51*(7), 1172–1185.

Viega, J., J. Bloch, and P. Chandra (2001, February). Applying aspect-oriented programming to security. *Cutter IT Journal 14*(2), 31–39.

Wand, M., G. Kiczales, and C. Dutchyn (2004, September). A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS) 26*(5), 890–910.

Yee, B., D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar (2009, May). Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, pp. 79–93.

Yu, D., A. Chander, N. Islam, and I. Serikov (2007, January). JavaScript instrumentation for browser security. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Nice, France, pp. 237–249. ACM.

VITA

Micah Jones was born in Fort Worth, Texas, on November 12, 1984, as the son of Bill and Beverly Jones. Early on, they elected to take the route of homeschooling, a remarkably efficient and flexible form of education that left him time to pursue his childhood love of books and video games. He soon grew tired of simply reading and playing other people's work, and sought to create his own. Before he developed actual writing skills, he narrated stories to his mother, but she quickly grew tired of that and introduced him to that brilliant piece of antique equipment known as a typewriter. He later upgraded to DOS-based word processors (which finally allowed him to erase mistakes), and in the process discovered that computers could be used to write not only stories, but also games!

Micah's early attempts at programming assumed that his 80386 IBM-compatible computer handled complex natural language processing, so his first BASIC program contained code such as, "`REM This is a game with a space ship. It will have 5 levels.`" He was thoroughly disappointed when the BASIC interpreter did absolutely nothing, and looked for books containing printed code. After meticulously copying down and running code for programs like Wumpus and Eliza, he decided to make his own games. BASIC's limitations became apparent, and he upgraded to the much more powerful and challenging C language. After making a small game that involved robots shooting at each other, he decided he lacked the motivation to learn much more on his own.

During high school, Micah further pursued interests in writing and acting, but he decided to return to computers for his studies at Oklahoma Baptist University. There he encountered professors like Dale Hanchey, Cindy Hanchey, and John Nichols, whose teaching furthered his appreciation and knowledge of mathematics, logic, and software development. He also thoroughly enjoyed his courses in history, literature, and theater, and was disappointed that he could only realistically major in one field. He spent one month abroad during this time,

teaching English as a Second Language in Urumqi, China, and worked for one summer as an intern at Lockheed Martin in Fort Worth. After four years of study, he graduated in May of 2007, Summa Cum Laude with College Honors, with a B.S. degree in Computer Science. In addition, he wrote a thesis entitled, *Mathematics and Physics Applications in Two-Dimensional Video Games: Two Case Studies.*

Micah immediately entered the Ph.D. program at the University of Texas at Dallas in the Fall of 2007. Kevin Hamlen took him on as a research assistant, and introduced him to the fascinating world of programming language-based security, automated bytecode rewriters, and abstract interpretation. Finally, after four years of work in the area, including one semester as a teaching assistant, Micah wrote this dissertation.