THE INLINED REFERENCE MONITOR APPROACH TO SECURITY POLICY ENFORCEMENT

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Úlfar Erlingsson

January 2004

THE INLINED REFERENCE MONITOR APPROACH TO SECURITY POLICY ENFORCEMENT

Úlfar Erlingsson, Ph.D.

Cornell University 2004

Embedding security enforcement code into applications is an alternative to traditional security mechanisms. This dissertation supports the thesis that such *Inlined Reference Monitors*, or IRMs, offer many advantages and are a practical option in modern systems. IRMs enable flexible general-purpose enforcement of security policies, and they are especially well suited for extensible systems and other non-traditional platforms. IRMs can exhibit similar, or even better, performance than previous approaches and can help increase assurance by contributing little to the size of a trusted computing base. Moreover, IRMs' agility in distributed settings allows for their cost-effective and trustworthy deployment in many scenarios.

In this dissertation, IRM implementations are derived from formal automata-based specifications of security policies. Then, an IRM toolkit for Java is described in detail. This Java IRM toolkit uses an imperative policy language that allows a security policy, in combination with the details of its enforcement, to be given in a single complete specification. Various example policies, including the stack-inspection policy of Java, illustrate the approach. These examples shed light on practical issues in policy specification, the support needed from an IRM toolkit, and the advantages of the IRM approach.

BIOGRAPHICAL SKETCH

Úlfar Erlingsson was born in Reykjavík, Iceland, in January 1972, son of Erlingur Steingrímson and Vilborg Sigurjónsdóttir.

Úlfar spent a quiet childhood playing with LEGO's, religiously avoiding all athletic activity, and reading whatever he could get his hands on. At age 10 he chanced to take a summer course in computers—and since then he has never stopped playing with them

When not programming his 8-bit Sinclair Spectrum ZX and Atari 800XL computers, teenage Úlfar could often be found playing text adventure games developed by Infocom, a Cambridge, MA, company founded by MIT graduates. This made him interested in natural-language processing and U.S. graduate schools, a critical development, although his interest in the former later subsided.

In high school, Úlfar was enthusiastic about mathematics, due to the Derive symbolic algebra package and his math teachers, Jón Hafsteinn Jónsson and Agnar Magnússon. So, in fall 1991, he began studies in Mathematics at the University of Iceland. In 1994 he graduated with a B.Sc. in Computer Science. Celebrating the occasion, he shore his hair which had grown unfettered during his three years of study.

Perhaps due to his early Infocom influence, that following fall, Úlfar joined the Ph.D. program in Computer Science at Rensselaer Polytechnic Institute in Troy, NY—this time swearing to forgo eating meat, rather than having his hair unkempt, during his studies. Unfortunately, Troy, "Home of Uncle Sam," did not appeal, and after only two years of study (and an M.Sc.), he moved to Cornell University, in another of upstate New York's pseudo-Greek towns.

The companionship and culture of Ithaca—home of the world-famous Moosewood vegetarian restaurant—was more to Úlfar's liking. Cornell's academic flexibility allowed him to concurrently develop his interests in computer systems research and English literature. He stayed in this happy environment and did work on this thesis, apart from a summer spent at Digital's Systems Research Center in Palo Alto, CA.

In 1999, Úlfar found a reason to graduate—and return to being a carnivore. Failing to finish writing his dissertation, he moved back to Iceland, and Lotta. Úlfar spent some years working for deCODE Genetics, a biotechnology startup in Iceland, and later Green Border Technologies, a Silicon Valley startup that he co-founded. Only after joining Microsoft Research's Silicon Valley Center, and growing a beard, did he finally finish this dissertation.

To Lotta

O, rocks! Tell us in plain words.

—J.J.

ACKNOWLEDGEMENTS

My advisor, Fred B. Schneider deserves the highest acknowledgement for coming up with the seeds for Inlined Reference Monitors, for his collaboration in developing the ideas, but especially for his help, guidance, and patience, in the writing of this dissertation. Greg Morrisett, my second advisor, was also involved throughout this work and gave much assistance when needed.

Erich Kaltofen at RPI introduced me to academic computer science, and Dan Schwarz at Cornell entertained my interest in English literature. Thanks, both. Special acknowledgement must also go to Mukkai Krishnamoorthy and Andrew Shapira from RPI, Mike Burrows from DEC SRC and MSR SVC, and Snorri Gylfason from Cornell and California.

Takk Mamma, for sending me to that summer introductory computer course, Pabbi, for buying me my first micro, Ulla, for letting me stay up all night with your Sinclair, Agnar, for letting me monopolize your computers for all those years, and Jón Hafsteinn, for being an encouraging and motivating teacher.

Thanks Lotta, for being such a wonderful partner in life.

My work on this dissertation has benefited from the generous support of Intel, deCODE genetics, and Microsoft Research, and this document has improved from the review of Michael Isard, Colleen Cullerton, and Mike Schroeder.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EXAMPLE SECURITY POLICIES

Chapter 1

Introduction

The thesis this dissertation supports is that a non-traditional implementation of software security enforcement—one that merges the code of the enforcement mechanism into the software whose activity is to be restricted—has numerous advantages and is practical in many modern systems.

Such *Inlined Reference Monitors* (IRMs) use a trusted *rewriter* that inserts security code into a *target application* in a manner that prevents the target from subverting the security code. A security policy specification guides this rewriting and determines where *security update* code is inserted and what *security state* is added to the application. Security updates are inlined program fragments that (i) maintain in the added security state a summary of the application's execution history, as relevant to the policy being enforced, and (ii) take remedial action if the policy is violated. The resulting *secured application* is guaranteed not to violate at runtime the security policy embedded within it.

By residing within a target, an IRM can observe more potentially security-relevant activity than could a traditional enforcement mechanism positioned at an underlying system interface. Observing these actions is, arguably, a prerequisite for security enforcement in an increasingly common class of software that includes extensible and scriptable systems, and where security policies must be expressed in terms of application abstractions. Triggering security updates by such a rich set of actions—and allowing updates to modify the added security state arbitrarily and, thus, to maintain any property of the application's execution—gives IRMs unprecedented flexibility in enforcing security policies.

The IRM approach is facilitated by the trend towards using higher-level languages, especially type safe languages, for software development. Not only do those languages define application abstractions on which policies can be enforced, but they also provide strong guarantees that can be used to ensure a secured application cannot compromise its IRM. By leveraging these guarantees, an IRM security policy can provide a single cohesive description of both the intent and the means by which a policy is enforced. This potentially allows the IRM approach to give greater assurance, since enforcement now relies on a trustworthy component of moderate size whose full specification can be studied in isolation.

1.1 Motivation

Software is increasingly developed using the abstractions of high-level programming languages, object-oriented or component-based programming, and modular software engineering methodologies [CO65, GMS77]. By abstracting from implementation details, high-level languages and modular design hide complexity and permit the programmer's task to be partitioned into simpler pieces. This enhances programmer productivity and allows the construction of more complex and flexible software [Boe99, Ous98] but often degrades the performance of the final system, effectively exploiting increases in hardware performance to increase software-developer productivity. Recently, this has continued with the advent of the Java programming language [GJS96] which employs program analysis, interpretation, and run-time garbage collection—techniques that provide strong guarantees but often suffer from poor performance [MTC+96, RLV+96, Wil92, DAK00].

The modular development of modern software systems often results in a layered design, where software components extend the functionality of other soft-

ware components, with the former now relying on the latter's correct behavior [Lam03, HLP98]. Building software systems in layers of such *extensions* facilitates component reuse as well as the partitioning of complex software (components often can provide functionality useful in several software systems). Extensibility can even allow reuse of entire software systems by permitting a system's users to specialize its behavior for particular needs [Lam83]. Today, commonly used software consists of such *extensible systems* that allow for end-user extension or programming: operating systems can be extended and perform scripted activity, databases allow stored procedures and reusable queries, word processors and spreadsheets support macro programming, graphics software allows addition of new filters and shaders, and Internet web browsers can be extended with plugins and webpage applets and can be programmed with scripting languages.[1]

The concept of a software "application," and its relation to its operating system, has thus changed. Today, all applications extend libraries built on top of operating system services and much software is application extensions, executing within application platforms such as spreadsheets or databases, rather than as traditional operating system processes [Sal74, Tan92].

One of the major failures in software construction—and of extensible systems—is the lack of assurance about application behavior. The roots of this failure are both social and technical: production of high-assurance software holds few commercial rewards and is technically difficult [Lam00, Sch99a]. One may wish to defend against this failure by relying on *security enforcement mechanisms* which

[1] Examples of this trend include: Microsoft Word™ and Microsoft Excel™ macros, Adobe PhotoShop™ plugin filters, Microsoft DirectX™ shaders, Informix DataBlades™, Netscape Navigator™ plugins, Sun's Java™ webpage applets, webpages programmed in JavaScript™, and Microsoft Windows™ Internet Explorer™ scripting.

aim to ensure automatically that software behavior complies with a *security policy*, a formal specification of the restrictions to be enforced [Ste91].

Unfortunately, available security enforcement mechanisms are unlikely to provide the assurance desired—a problem compounded by software's modular and extensible structure. Many security mechanisms are too inflexible or tied to a single interface or module, others are too complex or too closely tied to ill-defined semantics, and few, if any, allow for isolated specification and study of policies. As a result, desired security policies may be hard or impossible to specify and, in practice, are incompletely specified in ways that allow their subversion [LBMC94, Asl95, LJ97, How97].

At the same time, there is increased need for assured application behavior. With communication overtaking computation as the major use of computers, users are more inclined to enter data of uncertain origin and of unknown effect into applications—a known security risk [Sib96, Wag99]. The problem is acute when the untrustworthy "data" contains user-level extensions that run on the users' extensible software systems. Examples include macros for Microsoft Word™ (MSWord) as well as plugins and Java applets for web browsers [DFW96, Vig98]. In numerous recent incidents, such as the Melissa and ExploreZip "email viruses" [CER99a, CER99b, Sch99b], the security flaws of extensible systems have been exploited and, as predicted, these vulnerabilities are subject to successful attacks at an increasing rate [Sch99a, JK99].

1.2 Reference Monitors

Reference monitors observe software execution and take remedial action on operations that violate a policy. Thus, reference monitors encompass most current

a. The reference monitor must fully mediate all operations relevant to the enforced security policy.

b. The integrity of the reference monitor must be protected, either by the reference monitor itself or by some external means.

c. The correctness of the reference monitor must be assured, in part by making the reference monitor be small enough to analyze and test.

Figure 1.1: Requirements for a reference monitor implementation.

runtime security enforcement. Reference monitors were introduced in a 1972 U.S. Air Force report [And72] based on ideas derived from early work at Cambridge University and by Lampson [Lam69, GD72, WN79]. Their original motivation was the security problems raised by shared access to general purpose computers by multiple categories of users.[2] A key problem in these systems was limiting the set of resources that could be directly and indirectly referenced as a result of each user's activity through their invocation of system services or support routines. Such errant references posed risk to system resources, potentially allowing malicious users to learn secrets by dishonest reading, to violate integrity by writing of incorrect values, or to deny legitimate use by others (e.g., by exhausting resources).

Reference monitors addressed the problem by capturing all references made by user activity, either directly or indirectly, and subjecting each reference to a validity check. Reference monitor implementations were required to provide high-assurance and complete mediation of relevant activity by meeting the requirements shown in Figure 1.1 [And72]. Because the set of resources at risk included

[2]At the time, the terminology for this type of security was *protection*, defined as those issues in computer security not solvable by the traditional means of physical security and organizational methods. Unfortunately, the term has fallen out of favor and the phrase "computer security" is more commonly used today although in technical literature (as in this dissertation) "computer security" usually refers only to this restricted form of security.

all memory accessible to user and operating system activity, efficiency concerns dictated that the above requirements were best discharged with the support of specialized hardware. Thus a standard hardware-assisted implementation became prevalent, the *traditional reference monitor*, which combines memory virtualization with a restricted set of protected operating system entry points or system calls [Ame81, RT78, GS91, SR99].

Reference monitors are part of the *trusted computing base* (TCB)—that part of the computer system whose correct behavior is necessary to guarantee enforcement of the system's security policy [SS84]. Ensuring TCB correctness for realistic systems has proven surprisingly hard. That goal was at the heart of the Secure Computing Initiative, a grand programme embraced by the computer security research community [Tas81, DoD85]. Although heavily funded by the U.S. Department of Defense for more than a decade, this initiative is seen as a failure by most [Sch99a].

Despite the difficulty of constructing a high-assurance TCB, which has both technical and social causes [Tan76, DLP79], TCBs remain fundamental to security. No alternative exists for assuring software trustworthiness than to create a separate component that guarantees the semantics of fundamental interfaces and data structures [Sho97]; eventually, such a trusted component must exist, even if only at the level of the underlying hardware. Assurance comes from having this component simple enough and constructed in such a way that it can be examined in isolation and shown to provide the right semantics.

1.3 Reference Monitor Implementations

When introduced, the reference monitor implementation was seen as separate from the validity check for the mediated references:

> It is clear that the reference validation mechanism described above is not a model of secure computing. It is a device to provide containment of programs in execution, and as such, is at the heart of any implementation of these ideas. Surrounding this particular element [i.e., the reference monitor] are others that collectively make up the security part of a system. These include the authorization mechanism, the access control mechanism, ... [And72, p. 18]

Modern reference monitors implement both the mechanism for mediating security-relevant references as well as the mechanism that judges the validity of those references. These *validity checks* commonly involve examining page tables for virtual memory access or filesystem access-control metadata for file access.[3] This expanded role for reference monitor implementations has not, however, changed their three requirements given in Figure 1.1.

Example 1.1 shows one such expanded reference monitor that combines both mediation and validity checks; this reference monitor prevents the opening of more than one graphical window. This security policy specification enumerates what references to mediate and specifies for each the details of how it should be validated; a form of pseudo-code is used that can be converted into a full-fledged modern reference monitor implementation using techniques described in Chapter 3. In this example, the opening and closing of windows are the policy-relevant operations that the reference monitor mediates. At these *security events*, the policy performs security updates which modify its security state (here the boolean flag openWindow records whether a window is open) and encodes the validity checks, which here

[3]This somewhat blurs the reference monitor concept with that of the security kernel [And72] but, arguably, at no great loss.

```
STATE {
    Boolean flag openWindow, initially false.
}
EVENT Any graphical user interface operation
CONDITION The operation opens a graphical window
UPDATE {
    If openWindow {
        REJECT the operation.
    } Else {
        Set openWindow to true.
        ALLOW the operation.
    }
}
EVENT Any graphical user interface operation
CONDITION The operation closes a graphical window
UPDATE {
    Set openWindow to false.
}
```

Example 1.1: A security policy that disallows more than one open window.

reject the opening of multiple windows.

Extending reference monitors to include validity checks has not increased their capabilities as there are hard limits to what security policies can be enforced by monitoring software execution at runtime. Schneider [Sch00] defines a class of such enforcement mechanisms as *EM* (for Execution Monitoring) and proves that security policies enforceable by EM-class mechanisms are safety properties [AS87]. Unfortunately, safety properties[4] do not encompass all security policies of interest [ZL97, McL90]. Liveness properties [AS87][5] cannot be enforced with EM-class reference monitors, and "availability"—the target of denial-of-service attacks—is an example of a liveness property. Information-flow policies, which preclude inappropriate disclosure of sensitive information [DD77, Mye99, McL94], are another

[4]Informally, safety properties prevent "bad events" from occurring.

[5]Informally, liveness properties ensure "good events" do occur (e.g., guaranteed service after payment).

example of a security policy that is not a safety property and, therefore, not in the class EM.[6]

Recent work has sought to more accurately characterize the exact properties of runtime enforcement mechanisms [BLW02b, HMS03, Vis00]. These characterizations encompass all practical reference monitor implementations—in particular those whose remedial actions are not simply termination, as necessary in EM. Also included in these recent formulations are reference monitors that perform static analysis of the target software to establish properties of all possible execution paths. Although theoretically these extended formulations describe reference monitors capable of enforcing virtually all security policies, in practice they introduce only a few that are not in EM and that actually can be implemented. Most importantly, these extended formulations allow for the formal specification of IRM toolkits that leverage static analysis, like that described in Chapter 3.

Extending reference monitor formulations with static analysis and alternate remedial actions no more eliminates the practical limitations of traditional reference monitors than did the inclusion of validity checks. The primary restriction on what security policies are enforceable by a reference monitor is the difficulty of mediating arbitrary application operations. Traditional reference monitors cannot, for instance, easily distinguish whether a system call originates in a web browser or in a Java applet running inside that browser. This prevents a traditional reference monitor from enforcing the security policy of Example 1.1 to defend against

[6] General enforcement of liveness and information-flow policies requires analysis of all possible application executions, so the monitoring of a single execution is not sufficient. However, a reference monitor may well enforce a more restrictive security policy which implies either of these security policies. For example, prohibiting all filesystem access—a security policy that can be enforced with a reference monitor—can preclude disclosure of any information stored in the filesystem.

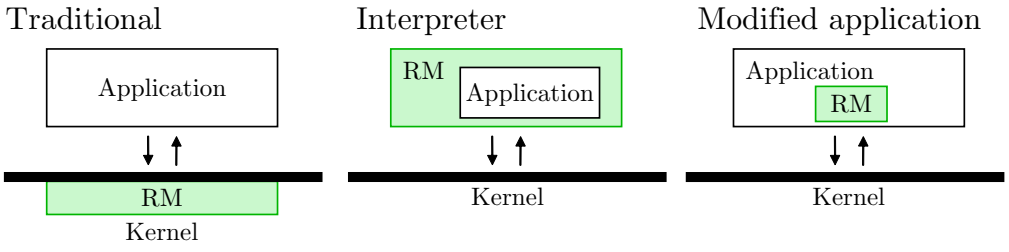


Figure 1.2: Three approaches to reference monitor implementation.

a denial-of-service attack on the graphical user interface (viz., opening a myriad of windows).[7]

In particular, traditional reference monitors do not have access to the abstractions of extensible systems, which precludes enforcement of most security policies on those extensions, because the policies are likely to be specified in terms of those abstractions. For example, a security policy restricting the operations of MSWord macros [Mic00] will probably refer to MSWord abstractions, such as "documents" and "paragraphs." Traditional reference monitors, tied to a single interface, are unable to enforce these application-specific policies. Because such reference monitors only mediate system calls, they can restrict use of the operating system's abstractions (e.g., files) but not the use of application abstractions.

As shown in Figure 1.2, there are several ways that reference monitors can mediate all application operations. A traditional reference monitor is implemented by halting execution before certain machine instructions and invoking the reference monitor with the instruction as input. An alternate implementation, not limited by hardware support, runs applications inside an interpreter that executes the application code and invokes a reference monitor before each instruction. This

[7]Reportedly, this security policy is desirable for HTML web pages [RHJ98], and any JavaScript [Fla98], Dynamic HTML [Rul98], and Java applets [LaD96] contained within them.

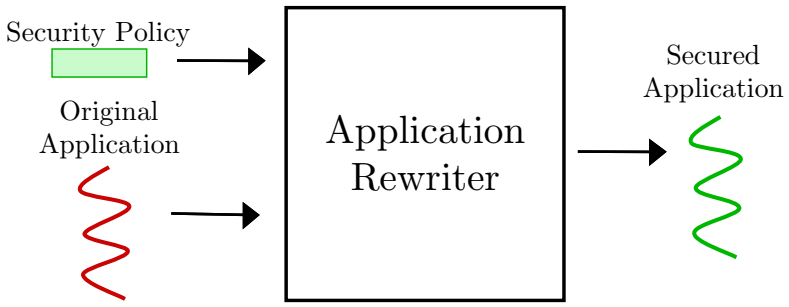


Figure 1.3: The IRM approach to security policy enforcement.

alternative was dismissed in [And72] as having unacceptable performance overhead, since a cost is incurred on every executed machine instruction regardless of its relevance to the security policy.[8] This dissertation examines the third option shown in Figure 1.2—embedding reference monitors in the target software—and shows how this approach overcomes the limitations of traditional reference monitors, yet exhibits reasonable performance.

1.4 Inlined Reference Monitors

An inlined reference monitor is obtained by modifying an application to include the functionality of a reference monitor. IRMs are *inserted* into applications by a rewriter that reads a target application and a security policy and produces a secured application, whose execution is guaranteed not to violate that security policy. This is depicted in Figure 1.3. The security policy is enforced on the secured application by the *IRM enforcement code* inserted by the rewriter. This IRM enforcement code encompasses both the reference monitor security state and security updates needed for this policy.

Like all reference monitors, IRMs are subject to the requirements given in Fig-

[8]Even so, this approach is used, for instance, in the Berkeley Packet Filter [SV93].

a. The IRM must fully mediate all security-policy-relevant operations in the secured application.

b. The integrity of the IRM must be protected, either by the IRM itself or by some external means. In particular, the secured application must not be able to circumvent or subvert IRM enforcement code.

c. The correctness of IRMs must be assured, in part by having the IRM rewriter and its input security policy be small enough to analyze and test them.

d. In the absence of security policy violations, the secured application output by the IRM rewriter must be functionally equivalent to the original; rewriting the target application must not modify its observable behavior.

Figure 1.4: Requirements for IRMs, based on those in Figure 1.1.

ure 1.1. IRMs are additionally constrained, however, because despite the insertion of IRM enforcement code, the semantics of the original target application must be preserved. Figure 1.4 gives this additional requirement as well as restating the general requirements of Figure 1.1 instantiated for the IRM approach.

Compared to traditional reference monitors, IRMs can enforce a richer set of security policies. In particular, because the code and state of the IRM exist as a part of the application, the IRM can mediate the execution of any application instruction and, therefore, can restrict uses of application-level abstractions. To enforce such richer security policies, the rewriter must identify where the application uses application-level abstractions and insert at those points the appropriate IRM enforcement code—i.e., the rewriter must identify where security-relevant actions occur and insert an appropriate security check or update. An IRM thus can enforce application-specific security policies such as "A Java applet running inside the Netscape browser may only open one graphical window," or "An MSWord macro may only modify the MSWord document that contains the macro." Neither

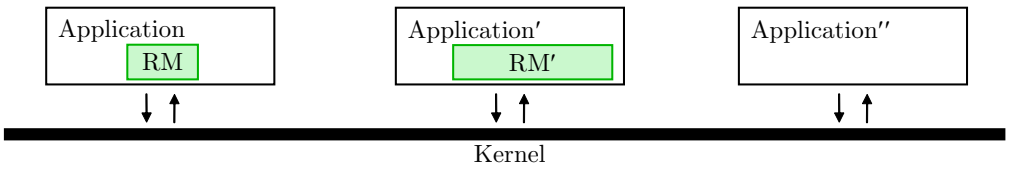


Figure 1.5: Three different applications, two of which have a distinct IRM.

of these security policies can be enforced by a traditional reference monitor because the policies involve application operations—not operating system operations.

IRMs are also able to enforce policies of traditional reference monitors, such as filesystem access control and memory protection. IRM enforcement code can be inserted into the application before and after each system call [PH98] and each memory-relevant operation [WLAG93]. In fact, because IRM security state can summarize an application's execution history, IRMs are strictly more flexible than traditional reference monitors for enforcing policies involving operating system abstractions. This permits IRMs to enforce a policy like "The application may not send network messages after it has read the filesystem" which is not enforceable with existing traditional reference monitor implementations, e.g., using the standard file permission bit flags of UNIX [RT78, GS91].

Typically, operating systems implement only such simple low-overhead security mechanisms that track state relevant to almost every circumstance. A more general-purpose mechanism that tracks additional state comes at a higher enforcement overhead, which penalizes all those applications that do not make use of this state. With IRMs, the enforcement mechanism can be made to fit the intent, and state kept in proportion to the security policy.

IRMs also have the potential to be more efficient than traditional reference monitors. Not only do IRMs not incur overhead by context switching to an oper-

ating system for enforcement: as shown in Figure 1.5, applications can be subject to different specialized IRMs, reducing the total overhead of security enforcement. Also, moving security enforcement from the operating system into an IRM is likely to make the operating system itself smaller and more efficient, in particular for activity not subject to security enforcement, such as Application″ in Figure 1.5.

This said, the use of IRMs introduces new challenges that must be resolved in practice. For instance, if each application has a different IRM then many copies may be made of otherwise common code such as libraries, with each copy slightly different. This can adversely affect code sharing and locality, resulting in lower performance and increased resource consumption. In addition, because IRMs are embedded in secured applications, it can be difficult understand how IRMs compose, e.g., as a result of multiple rewritings, and to determine what is the security policy being currently enforced. This dissertation presents several such challenges to the implementation of IRMs and how they may be resolved.

1.5 Terminology

IRMs are best described using a consistent vocabulary, since notational clarity both helps explain what is novel as well as resolve what is ambiguous. This section introduces the terminology that is specific to IRMs and is used in this document. Terms are italicized twice, both in the below paragraphs and also at the time of their first use.

An IRM is a security enforcement mechanism that is *inserted* into a *target application* by a *rewriter* that is part of an IRM system implementation, or *toolkit*. The rewriter is guided by a security policy in its insertion of *IRM enforcement code*, which consists of *security state* and *security updates* to that state. The output of

the rewriter is a *secured application* which is guaranteed to comply with the input security policy.

IRMs mediate runtime application activity, subject that activity to *validity checks*, and take remedial action if those checks fail. Validity checks and remedial actions are specific to the security policy being enforced. However, the *low-level actions* comprising the target application's execution on its runtime platform are independent of any specific security policy, and it is those low-level actions that an IRM mediates.

The structure of a target application's static image determines what low-level actions can occur during its execution. When rewriting a target application, the IRM rewriter uses this structure to *identify* places in the application where low-level actions may *occur* at runtime; each such *place* forms an *insertion point* for IRM enforcement code.[9] Also, low-level actions that do not occur in the application may be *exposed* by the IRM toolkit rewriter through the addition of code that forms an insertion point for that action.

Given a particular security policy, only a subset of a target application's low-level actions may actually be *security-relevant actions* and thus at runtime be relevant to compliance with the security policy. An IRM toolkit makes mediation specific to a security policy by identifying and exposing only security-relevant actions. At a particular insertion point, the rewriter inserts—as specified by the security policy—the IRM enforcement code that should be *triggered* at runtime by that particular security-relevant action.

Although security enforcement only requires the mediation of security-relevant actions, those actions are too low-level to be the only basis for the validity checks

[9]Thus, insertion points correspond to *join points* and IRM enforcement code to *advice* in the work of [KLM+97, WZL03].

of actual security policies. Typically, therefore, the security policy input to an IRM toolkit *synthesizes*, or abstracts, from the low-level activity higher-level *security events* by using security updates and security state. As a result, in order to separate policy and mechanism, some security policy specifications may restrict their attention to synthesis of security events, leaving the validity checking of those events to future *security policy extensions*. With careful event synthesis, an IRM toolkit can provide the facilities of a higher-level reference monitor that operates by mediating such higher-level security events.

1.6 Dissertation Structure

This dissertation explores the IRM approach, discusses the issues raised, and demonstrates the generality and practicality of the approach by describing some prototype implementations, thereby expanding on the work published in [ES99, ES00].

Chapter 2 describes the *SASI* (Security Automata SFI Implementation) prototypes, which process security policies specified using a form of finite-state automata. SASI was implemented both for the Intel x86 [INT94] and JVML [YL96] instruction set architectures. The x86 SASI used the *SFI* (Software-based Fault Isolation) technique of [WLAG93] to protect the integrity of the IRM; the integrity of the JVML SASI prototype relies on guarantees provided by the JVML verifier. Chapter 2 offers a close look at how SASI would enforce an example security policy, the Chinese Wall policy of [BN89], and offers a retrospective on the prototypes.

Chapter 3 refines the ideas of inlined reference monitors, describing how to overcome practical challenges in their implementation as well as discussing issues of deployment and management. The chapter introduces a second-generation proto-

type IRM implementation based on JVML: the *PoET* Policy Enforcement Toolkit and its *PSLang* security Policy Specification Language. Numerous examples, written as PSLang security policy specifications (including those listed in Appendix B), are given to illustrate the applicability of the IRM approach.

Chapter 4 offers a detailed case study of applying the IRM approach to Java 2 (or JDK 1.2) stack inspection [Gon99, WF98], describing the tradeoffs and performance of different implementation strategies. The chapter compares IRM implementations of stack inspection with other known implementations.

Chapters 5 and 6 describe related and future work, and offer concluding remarks, respectively.

Appendix A gives the syntax and informal semantics of the PoET and PSLang prototypes, relative to JVML, as well as describing relevant details of their implementation. Finally, Appendix B provides complete listings for PSLang policies relevant to the discussion in Chapters 3 and 4.

Chapter 2

Security Automata SFI Implementation

Most reference monitors observe the execution of a target system and halt that system whenever it is about to violate some security policy of concern. Security mechanisms found in system software and hardware typically either directly implement reference monitors or are intended to facilitate the implementation of reference monitors. For example, an operating system might mediate access to files and other abstractions it supports, thereby implementing a reference monitor for policies concerning those objects. As another example, the context switch (trap) caused when a system call instruction is executed forces a transfer of control, thereby facilitating invocation of a reference monitor whenever a system call is executed.

The IRM approach, described in Chapter 1, constitutes an alternative to placing the reference monitor and target systems in separate address spaces: modify the target system code, effectively merging the reference monitor in-line. This is the basis for *SFI* (Software-based Fault Isolation) [WLAG93], which provides memory protection by enforcing the security policy that prevents reads, writes, or branches to memory locations outside of certain predefined memory regions associated with a target system. In theory, a reference monitor for any security policy—not just memory protection—could be inlined into a target application, provided the target can be prevented from circumventing the inlined code. This chapter describes two prototype systems that have successfully reduced this theory to practice.

The prototypes implement the IRM approach by inlining security policy enforcement code into the object code for a target system. The prototypes work at

the level of object code, and not source code, both to be source-language independent and, also, to minimize the size of the trusted computing base. In particular, by rewriting object code, high-level language processors are not made part of the TCB. However, the IRM rewriter (which performs analysis and modification of object code) *is* added to the TCB, but this rewriter can be relatively modest is size (as described in §2.3). Also, working at the object code level makes available a rich vocabulary of low-level activity—the machine-language instructions of the target system—over which any security policy can presumably be crafted.[1]

One of the prototypes transforms Intel x86 [INT94] assembly language output from the `gcc` compiler; another prototype transforms JVML (Java Virtual Machine Language) [YL96]. With each, security policies are specified using security automata, a notation that has been shown to be in the class EM (described in the previous chapter), and expressive enough to define any security policy that halts the target system and is enforceable through execution monitoring [Sch00].

The chapter proceeds as follows. The use of security automata for specifying security policies is discussed in §2.1. A general approach to merging enforcement code into a target system is the subject of §2.2. The two prototype realizations of this approach are then discussed in §2.3. For each prototype, ensuring enforcement code integrity is discussed and results of performance experiments are given. In §2.4 a detailed example of how the prototypes could be used to enforce the Chinese Wall security policy [BN89] is given. And, §2.5 critiques the prototypes, offering the conceptual basis for further implementations of the IRM approach.

[1] As will become clear, this belief—although appearing reasonable—is a fallacy. By observing only machine instructions, a reference monitor is unable to mediate certain security-relevant actions and, also, security events often require complicated analysis that reduces both assurance and runtime performance. Resolving these concerns is the subject of Chapter 3.

2.1 Security Automata

As defined in [Sch00], a *security automaton* involves a (not necessarily finite) set of states, a (not necessarily finite) input alphabet, and a transition relation. The transition relation defines a next state for the automaton given its current state and an input symbol. It is often convenient to define this transition relation using first-order predicates: such a transition predicate is *true* for a current state q, an input symbol s, and a next state q' iff whenever the security automaton is in state q and input symbol s is read, the automaton state changes to state q'.

Security automata are similar to ordinary non-deterministic finite-state automata [HU69]. Both classes of automata involve a set of states and a transition relation such that reading an input symbol causes the automaton to make a transition from one state to another. However, an ordinary finite-state automaton rejects a sequence of symbols if and only if that automaton does not make a transition into an accepting state upon reading the final symbol of the input. In a security automaton, all states are accepting states; the automaton rejects an input upon reading a symbol for which the automaton's current state has no transition defined.

Formally, a security automaton is defined by

- a set Q of *automaton states*,
- a set $Q_0 \subseteq Q$ of *initial automaton states*,
- a (countable) set I of *input symbols*, and
- a *transition function*, $\delta \subseteq (Q \times I) \rightarrow 2^Q$.

The set I of input symbols depends on the security policy being enforced; it may correspond to the set of unique system states or to the set of unique atomic actions

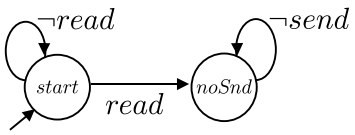


Figure 2.1: Security automaton: "No messages sent after reading a file".

in the target system (such as machine instructions). To process a sequence $s_1 s_2 \ldots$ of input symbols in I, the automaton starts with its *current state set* Q' equal to Q_0 and reading the sequence one symbol at a time changes its current state set Q' to the set Q'', where

$$Q'' = \bigcup_{q \in Q'} \delta(q, s_i).$$

If the current state set Q' is ever empty, the input is rejected (and does not comply with the policy being defined by the automaton); otherwise the input is accepted.

As mentioned above, transition function δ may be given in terms of *transition predicates*, which are Boolean-valued effectively computable total functions with domain I. If p_{ij} denotes the predicate for the transition from automaton state q_i to automaton state q_j, then the security automaton, upon reading an input symbol s with current state set Q', changes it's current state set Q' to Q'', where

$$Q'' = \{q_j \mid q_i \in Q' \wedge s \models p_{ij}\}.$$

Security automata can be regarded as defining reference monitors in the class EM described in Chapter 1. The input alphabet corresponds to low-level actions that are mediated by the reference monitor. The transition relation encodes a security policy—the automaton rejects sequences of inputs corresponding to target system executions in which the security policy would be violated. For example, Figure 2.1 depicts a security automaton for a security policy that prohibits message sends after file reads. The automaton's states are represented by the two nodes labeled *start* and *noSnd*. (Automaton state *start* is the initial state of this security

automaton.) The transition predicates *read* and *send* characterize target-system instructions that cause files to be read and messages to be sent, respectively. Thus, the security automaton of Figure 2.1 rejects any input corresponding to a target system's attempt to execute a send instruction while in state *noSnd* (i.e. after a file read).

The remainder of this chapter only considers deterministic security automata with only a finite number of states. Standard automata techniques allow this simplification with no practical limitation to the discussion or to the applicability of the conclusions drawn [ASU85, Sch00].

2.2 Inlining a Security Automaton

Security Automata SFI Implementation (SASI) is an instantiation of the IRM approach that generalizes SFI to any security policy specified as a security automaton. With SFI, new target system code is inserted immediately before any instruction that accesses memory (i.e. any read, write, branch, subroutine call, or subroutine return). This new code ensures that (i) all reads and writes to memory will access addresses within the target's data region, (ii) all branches, calls, and returns will transfer control to an instruction within the original target program, and (iii) the functionality of this added security checking cannot be circumvented by the target system.[2]

With SASI, new code is inserted into the target system immediately preceding every instruction. The added code simulates a security automaton. Specifically,

[2] In addition, with SFI, the instruction itself may be modified in a way that preserves semantics in absence of security policy violation. Thus, an instruction that writes to a memory location contained in register *r1* might be modified to use register *r2*, with *r2* containing the original address of *r1* modified to lie inside a certain memory region.

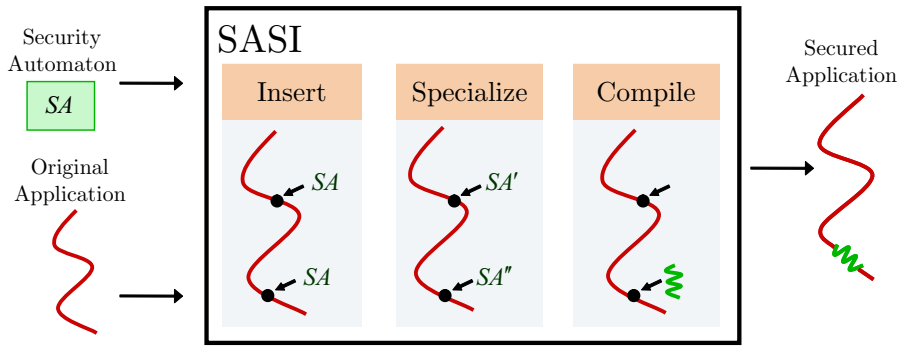


Figure 2.2: An overview of the SASI implementation of the IRM approach.

new variables—accessible only to the code added for SASI—represent the current state of the security automaton, and new code—that cannot be circumvented—simulates an automaton state transition. The new code also causes the target system to halt (because the current automaton state does not allow a transition for the next target instruction) whenever the automaton rejects its input. Thus, the automaton simulation is equivalent to inlining a reference monitor into the target system (as shown in Figure 2.2).

The set of security automata input symbols—in SASI the set of target system instructions—determines which security policies can be enforced by the security automata. For instance, the set of automata input symbols could be limited to the set of function calls and returns. Such a restriction would reduce the number of insertions of IRM enforcement code, thereby reducing security enforcement overhead. However, such a restriction would also preclude SASI from enforcing some security policies—for example, policies concerning individual machine instructions like SFI (which restricts memory access operations) or a policy that disallows division by zero. This is because both memory access and division by zero do not necessarily involve any function calls.

SASI reduces security enforcement overhead (without restricting the set of en-

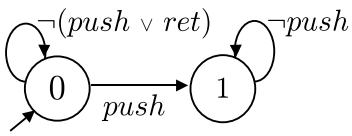


Figure 2.3: Security automaton: "Push once, and only once, before returning."

forceable policies) by simplifying the inserted code. Simple analysis of the target system often allows the code for simulating a security automaton to be greatly reduced by using the context in which that code appears. To achieve this, SASI performs the processing diagrammed in Figure 2.2:

Insert security automata. A copy of the security automaton SA, which enforces the desired security policy, is associated with every application instruction.

Specialize security automata. Each copy of SA is specialized according to its associated instruction. In Figure 2.2, SA' and SA'' are the result of such a specialization.

Compile security automata. Emulation code is generated for the each specialized security automaton and inserted before its associated instruction. In Figure 2.2, SA' happens to be effect-free (i.e., does not restrict the target system execution) and, thus, only emulation code for SA'' is inserted. For example, this is the case if SA specifies a security policy that prohibits division by zero, SA' is associated with a load instruction, and SA'' is associated with a division instruction.

By specializing the transition predicates of security automata as well as by using the automaton structure, irrelevant tests and updates to the security automaton state can be removed during the security automaton specialization step. SASI specializes security automata to each insertion point using methods from program

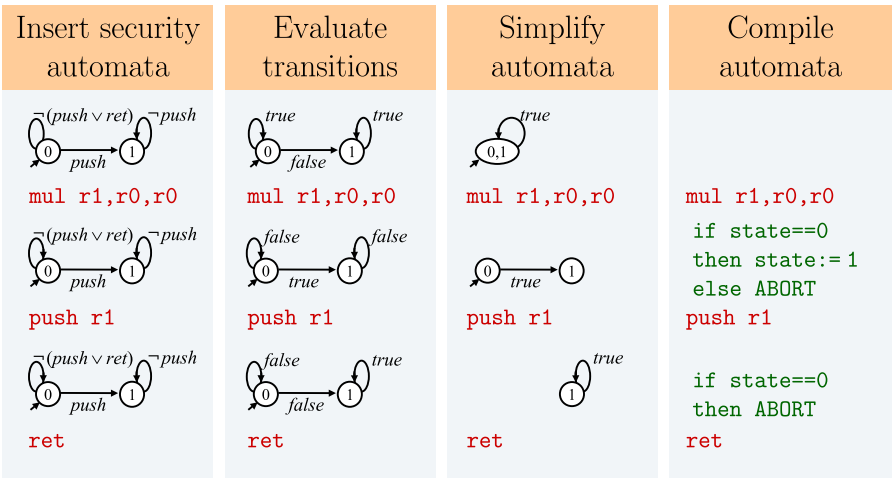


Figure 2.4: Simplification of inserted code.

optimization [ASU85] and partial evaluation of programs [JGS93]. Figure 2.4 depicts the inlining of a security automaton specification (given in Figure 2.3) into a three-instruction routine that squares a value in register `r0`. The security policy in Figure 2.3 restricts execution to pushing exactly one value onto a stack before returning.[3]

In Figure 2.4 the "Specialize" column of Figure 2.2 has been expanded into two separate columns to better illustrate the processing:

Evaluate transitions. Partially-evaluate the transition predicates of each security automaton, given information at each insertion point.

Simplify automata. Delete any transitions whose predicates evaluated to *false* in the previous step of evaluating the transition predicates.

The final step (shown in the last column of Figure 2.4) compiles the simplified security automata into code that, if inserted at these points, simulates the operation

[3] For exposition purposes, the security policy shown doesn't handle some cases, e.g., those involving the `pop` instruction.

of the security automaton. `ABORT` is invoked by the inserted code if the automaton being simulated must reject its input.

SASI eliminates security enforcement code—and thereby reduces security enforcement overhead—by utilizing the structure of security automata in two ways:

1. If each state of an automaton has a transition labeled with predicate p back to that same state, the automaton can be simplified to a one-state automaton that only accepts inputs that satisfy p. In addition, when $p \equiv \mathit{true}$ the resulting automaton will accept all inputs, and when $p \equiv \mathit{false}$ the resulting automaton will reject all inputs.

2. If each state of an automaton has a transition labeled with predicate p to a distinguished state q, the automaton can be simplified to a two-state automaton that only accepts inputs that satisfy p and update the state to q.

The above automaton-structure simplification methods are an instance of an optimization for "switch statements" in programming languages like C [EKR96, HS94]. A security automaton can be emulated by branching (or "switching") on the current automaton state to code that updates that state, contingent on the satisfaction of a transition predicate. In such emulation, simplifications 1 and 2 above apply when all branch targets update the state in the same way, collapsing multiple equivalent branch targets.

Because SASI evaluates and simplifies security automata using only local information, a more global analysis can sometimes show inserted code to be redundant. For example, in Figure 2.4, if the `ret` instruction can be reached only through straight-line execution, then `state==0` will necessarily be *false* before `ret` is executed. In this case, the code appearing in Figure 2.4 before the `ret` instruction

might not be needed. The partial evaluator in SASI doesn't attempt this global analysis because this would increase both the size and complexity of the SASI TCB.[4] Recent work indicates, however, that only a small addition to the TCB may be required for this type of analysis, if integrated into existing partial-evaluation frameworks [Thi03].

2.3 Two SASI Prototypes

Security policies for the SASI prototypes are represented in *SAL* (for Security Automaton Language), a language devised to provide a textual way to specify security automata.[5] Each SAL specification consists of a (finite) list of states, with each state having a list of transitions to other states. Macros are defined at the start of the SAL specification and are expanded fully bottom-up before use (and therefore may not be recursive).

SAL transition predicates are expressions constructed from constants, variables, C-style arithmetic and logical operators, and function calls. SAL functions are either general-purpose and platform-independent, or platform-specific:

- For the x86 SASI prototype, the platform-specific functions evaluate to the opcode and operands of the next target instruction, as well as defining the sets of addresses corresponding to data values that can be read/written and

[4]For this particular example, the analysis would need to statically propagate between instructions the set of potential "current" automata states and give strong guarantees about execution of branches. For arbitrary object code, this analysis is complex and somewhat unreliable. However, if restricted to individual JVML basic blocks, the analysis becomes simple enough to even be directly encoded in security policies, as shown by the IRM toolkit of Chapter 3.

[5] SAL is just one language, of many, that could be used to specify security automata for security policies. Another language—of strictly greater expressiveness than SAL—is introduced in Chapter 3 and described in detail in Appendix A. Therefore, this dissertation does not give details of SAL syntax and semantics.

```
/*
** Macro definitions
*/
MethodCall(name) ::= op=="invokevirtual"
                  && param[1]==name;
FileRead() ::= MethodCall("java/io/FileInputStream/read()I");
Send()     ::= MethodCall("java/net/SocketOutputStream/write(I)V");
/*
** The Security Automaton
*/
start ::=
  !FileRead() -> start
  FileRead()  -> noSnd
;
noSnd ::=
  !Send() -> noSnd
;
```

Example 2.1: SAL specification for "No messages sent after reading a file."

defining addresses of instructions in the target system that can be branch, call, or return destinations.

- For the JVML SASI prototype, the platform-specific functions allow access to class name, method name, method type signature, JVML opcode, instruction operands, as well the JVM state in which the target instruction will be executed.

As an illustration, Example 2.1 contains SAL for a JVML SASI specification of the security policy (given in Figure 2.1) prohibiting message sends after file reads.[6]

Associated with each SASI prototype is a rewriter that inlines a security automaton simulation into the object code for a target system. The rewriter operates as outlined in §2.2, i.e., it inserts IRM enforcement code for the security automaton

[6] For expository simplicity, this SAL specification does not consider all ways in which JVML programs can access the filesystem and the network. The relevant details are given in §3.4.

simulation immediately before each target instruction. To construct this code, the platform-specific SAL functions appearing in transition predicates are instantiated with actual values, if they are known, and with code that will compute the values at runtime, otherwise. A generic partial evaluator is next run to simplify the resulting automaton. Finally, object code for the simplified security automaton is generated and inserted in the target system code.

The integrity of a reference monitor inlined by the SASI rewriter into the object code of a target system depends on preventing corruption of that security automaton simulation. This entails

- preventing the target system from modifying variables being used in security automaton transition predicates and variables being used to encode the state of the security automaton,
- preventing the target system from circumventing the code that implements transitions by the security automaton, and
- preventing the target system from improperly modifying its own code or causing other code to be executed (e.g. by using dynamic linking, which most operating systems support), since this could nullify the measures just described for preserving security automaton integrity.

Thus, the secured application must not be able to violate the integrity of the IRM—by circumventing the security updates, by compromising the security state, or by generating some security-related action not mediated by the IRM. Discharging these obligations is platform dependent, but can, for instance, verify the object code to establish that the unwelcome behavior is impossible or modify the object code to rule out the unwelcome behavior. The prototypes make use of both approaches.

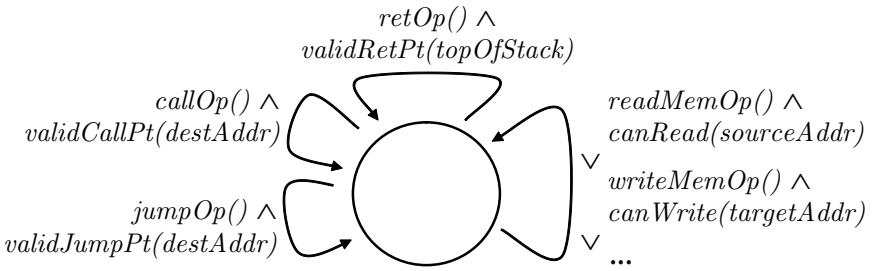


Figure 2.5: A security automaton for SFI-like memory protection.

2.3.1 The x86 Prototype

The x86 SASI prototype works with the assembly language output of the GNU `gcc` C compiler [Woe94]. Like most SFI implementations, x86 SASI makes several assumptions about its target programs. For instance, x86 SASI excludes self-modifying programs, relies on the register-usage conventions of `gcc`, and the following assumptions:

- Program behavior is insensitive to silent modifications such as adding stutter-steps, i.e., functional null operations, or no-ops.
- Variables and branch-targets are restricted to the set of labels identified by `gcc` during compilation.

These restrictions considerably simplify the task of preventing a security automaton simulation from being corrupted by the target system.

Figure 2.5 shows the x86 SASI security automaton for a security policy that enforces the guarantees of SFI,[7] with Example 2.2 showing some details of its SAL specification. The transitions of Figure 2.5 are labeled with predicates that

[7] The policy of Figure 2.5 even deals with difficulties that arise because valid x86 instructions might exist at non-instruction boundaries in a target system—a bit pattern for one instruction might encompass the bit pattern for another. By limiting branch destination to valid labels, jumping into the middle of an instruction is prevented.

correspond to macros in the SAL specification. These macros serve to restrict allowed inputs to each type of machine instruction.

In the figure, each transition uses a predicate for a certain type of machine instruction—branch instructions in the case of *jumpOp()*—and it holds when that type of instruction is executed at runtime. Each of these instruction types operates on arguments that must be restricted at runtime. These instruction operands are represented in the figure by distinct macro names, e.g., *targetAddr* for the recipient operand in instructions that write a new value.

It suffices to apply x86 SASI with the simple memory-protection policy of Figure 2.5 in order to obtain target-system object code that cannot subvert an inlined security automaton simulation. The x86 SASI prototype, therefore, prepends a SAL description of the security policy in Figure 2.5 to the SAL input describing any security policy P to be enforced. This effectively inlines into the target system a security automaton that enforces security policy P and cannot be circumvented or corrupted.

This recursive use of x86 SASI works because the transition predicates in Figure 2.5 are defined entirely in terms of SAL platform-specific functions which, by virtue of being constructed from information provided by `gcc`, accurately characterize the target program. An informal proof that the transformation suffices proceeds by contradiction, along the following lines. Only branch, call, return, and write instructions can subvert the security automaton simulation. Let i be the first instruction that accomplishes the subversion. Before each branch, call, return, and write instruction, code to check that instruction's operand is added by x86 SASI for the policy in Figure 2.5. Thus, such checking code must immediately precede instruction i. Since, by assumption, i is the first instruction that

```
/*
** Macro definitions
*/
...
/* define trusted operands */
trusted(N) ::= indirectBaseRegisterName[N]=="%ebp"
            && ...;
/* ensure address is trusted or in right segment */
sandbox(N,address) ::= trusted(N)
                    || (address & writeMask)==writeSegment;
/* can operand N be written to? */
canWrite(N) ::= ( type[N]=="register" )
             || ( type[N]=="direct"
                  && sandbox(N,memoryLabel[N]) )
             || ( type[N]=="indirect"
                  && sandbox(N,indirectAddress[N]) );
...
/*
** The Security Automaton
*/
start ::=
  /* exchange instruction writes to both its operands */
  op=="xchg" && canWrite(1) && canWrite(2)  ->  start

  /* return instruction must have valid target address */
  op=="ret"  && validReturnPoint(%esp[0])   ->  start
...
;
```

Example 2.2: Excerpts from the SAL specification for x86 SASI SFI.

accomplishes the subversion, the checking code that precedes it must have been reached and executed. And since the transition predicates are, by construction, accurate, the checking code that precedes i will prevent instruction i from executing. This contradicts the assumption that i is able to execute and subvert the security automaton simulation.

x86 SASI SFI Implementation

Example 2.2 shows parts of the x86 SASI SAL specification for the SFI security policy of Figure 2.5. Shown are the parts relevant to the `xchg` and `ret` instructions. In the specification, the instruction-type predicates of Figure 2.5 have been expanded to designate particular opcode names; similarly, the argument macros have been expanded, so that, for example, *topOfStack* has become `%esp[0]`.

The `xchg` instruction swaps the contents of two memory locations and, thus, must have write access to both locations (here, write access implies read access). The x86 SASI SAL specification of Example 2.2 ensures that only certain memory addresses can be written through *sandboxing* target addresses, a method developed in previous SFI implementations [WLAG93]. Sandboxing a write instruction, such as `xchg`, entails comparing the first few bits of the target address to a specific value, for which all writes are legal—i.e., this sandboxing requires addresses in the writable memory segment to have a specific prefix.[8]

The `ret` instruction returns from a function call by performing an indirect jump to the address on top of the call stack. The SFI security policy requires this address to be a valid return point, i.e., the address of an instruction following a `call` instruction. In Example 2.2 this is enforced by the *validReturnPoint* platform-specific function, which is constructed by the x86 SASI rewriter from the set of labels following `call` instructions in the target program. Indirect jumps and calls are subject to a similar constraint, also enforced by platform-specific functions constructed by the rewriter from target-program labels.[9]

[8] SFI implementations typically sandbox addresses in this manner rather than comparing against a lower and upper bound, because this allows for more efficient security enforcement code [WLAG93].

[9] The x86 SASI prototype implements these functions using balanced binary trees, generated using the relative order of labels in the input assembly. A more

For return instructions, indirect jumps, and some calls, these platform-specific functions must be executed at run time in order to determine whether the target address is a valid label. However, when the target address is a directly-named label (e.g., as in a function call to a known function), the rewriter can statically determine the platform-specific function's result. This is the case for the function `memoryLabel` in Example 2.2; that function is constructed by the rewriter and only invoked during the rewriting of the target program.

Finally, the SFI specification in Example 2.2 exploits the fact that code generated by `gcc` makes indirect stack references by using x86 register `%ebp`, and `%ebp` is guaranteed by `gcc` to point to a valid location in the stack. Thus, the SFI security policy does not require run-time checks for indirect references through `%ebp` (see the `trusted` macro in Example 2.2). Since indirect references through `%ebp` constitute a significant fraction of the indirect memory references in executables produced by `gcc`, avoiding such checks significantly reduces enforcement overhead. But exploiting assumptions about code generated by `gcc` in order to reduce the cost of an enforcement mechanism does expand the TCB to include `gcc`—a questionable trade-off.

For SFI processing of machine object code it seems necessary to make additional simplifying assumptions of the type described in this section. In particular, the SFI implementations in the literature, have had difficulty with function pointers, and other computed jumps, self-modifying and dynamically-loaded code, and variable-length machine opcodes (as mentioned on page 30, in a footnote). As a result, SFI implementations often unrealistically constrain the set of their input programs,

efficient x86 SASI implementation could make use of the labels' absolute memory addresses, determined when the target program is linked into memory, and implement these functions by hashing on target addresses.

```
pushl  %ebx                        pushl  %ebx
leal   dirty(,%eax,4), %ebx        leal   dirty(,%eax,4), %ebx
andl   offsetMask, %ebx            andl   segmentMask, %ebx
orl    writeSegment, %ebx          cmpl   writeSegment, %ebx
movl   %edx, (%ebx)                jne    SASIx86_ABORT
popl   %ebx                        popl   %ebx
                                   movl   %edx, dirty(,%eax,4)
```

MiSFIT | **x86 SASI SFI**

Figure 2.6: SFI'd x86 assembly instruction `movl %edx, dirty(,%eax,4)`.

even after the guaranteed use of a certain compiler, e.g., excluding C programs with trampolines or other inline assembly. Only by modifying the compiler and language semantics has this SFI limitation been overcome [ATLLW96]—thereby moving towards the verifiable certification of code discussed in §3.1.2.

x86 SASI in Action

The SFI memory protection policy given by the SAL specification of Example 2.2 is the same security policy as implemented by MiSFIT, a special-purpose SFI transformation tool for x86 operating system extensions [Sma97]. MiSFIT thus constitutes a benchmark for measuring the SFI performance of x86 SASI.[10] Consequently, a set of target systems was input both to MiSFIT and to x86 SASI with no additional SAL input (so that it enforces the same policy as MiSFIT).

The modifications MiSFIT and x86 SASI SFI make to a target system are not very different. Figure 2.6 shows the `gcc` assembly output (target operand on the right) generated by MiSFIT and x86 SASI SFI for a `movl` instruction that transfers the contents of register `%edx` into integer array `dirty` at a position specified by the

[10] Like the SFI security policy of Example 2.2, MiSFIT relies on the `%ebp` register being inaccessible to actual application code. The primary reason for including this performance optimization in x86 SASI was to allow meaningful comparisons with MiSFIT; without this assumption, overhead would approximately double.

Table 2.1: Relative performance of MiSFIT and x86 SASI SFI.

Benchmark	**MiSFIT**		**x86 SASI SFI**	
Page-eviction hotlist	2.378	(0.3%)	3.643	(2.6%)
Logical log-structured disk	1.576	(0.3%)	1.654	(0.5%)
MD5 message digest	1.331	(1.4%)	1.363	(0.1%)

contents of register `%eax`. (In Figure 2.6, code inserted by x86 SASI is typeset in a slanted font; original target system code is typeset in an upright font.) Notice that MiSFIT actually replaces the original `movl` whereas x86 SASI only prepends additional instructions. In both, the `%ebx` register is made usable by saving its contents on the stack.[11] In some rare cases, the stack is also used to save the processor flags, which on the x86 architecture are implicitly changed by execution of most instructions. For efficiency, the x86 SASI rewriter uses a load-time generated function that performs a conservative data-flow analysis to determine when these flags must be saved.

Table 2.1 gives running times for three target systems from [SS96, Sma97] that have been processed by MiSFIT and by the x86 SASI prototype. The running times are relative to execution of the unmodified target system, and the numbers shown are averages (with standard deviation in parentheses) over 30 runs on a 266mhz Pentium II running Linux 2.0.30. Thus, executing MD5 processed by MiSFIT took on average 33.1% longer than running an unmodified MD5, with 31.2% and 35% overhead at one standard deviation from the norm. The "Page-eviction hotlist" benchmark is a memory intensive application; not surprisingly, it has a high overhead in both implementations, since a check must be executed on

[11] MiSFIT and x86 SASI thus both require that enough stack space be available for saving the `%ebx` register. This can be ensured at load time. Note, however, that MiSFIT's use of `%ebx` after the sandboxed memory access may be incorrect if the array `dirty` overlaps the stack, a subtle bug that may result in insidious modifications of program behavior.

each indirect memory access.

As Table 2.1 shows, x86 SASI and MiSFIT produce target systems having comparable performance. But there are target systems where MiSFIT performs considerably better. This might be expected, since MiSFIT is a specialized tool, customized to enforce one specific policy, and MiSFIT optimizes the code it adds. Also, adding additional analysis and optimization capabilities to the x86 SASI rewriter would improve its relative performance, albeit at the cost of increased complexity, creating more risk for security bugs, like that described for MiSFIT in the footnote on page 36. In general, however, the flexibility of being able to enforce any policy should make IRM-based security enforcement attractive, even at the cost of some increase in enforcement overhead.

2.3.2 The JVML Prototype

Type safe languages, such as JVML, provide guarantees about execution, including guarantees that imply the simple memory-protection policy given in Figure 2.5 cannot be violated. A program that does not satisfy this simple memory-protection policy is simply not type safe. The JVML SASI prototype exploits the type safety of JVML programs to prevent a security automaton simulation from being corrupted by the target system in which it resides, thus ensuring that IRM integrity is preserved.

Variables that JVML SASI adds to JVML object programs as part of a security automaton simulation cannot be compromised because they are inaccessible to the target-system object code by virtue of their names and types, given the type safety of JVML. Code that JVML SASI adds for the security automaton simulation cannot be circumvented because JVML type safety prevents jumps to unlabeled

```
...
ldc 1                                   ; noSnd state number
putstatic SASIJVML/state                ; change state to noSnd
invokevirtual java/io/FileInputStream/read()I  ; read file
...
getstatic SASIJVML/state                ; get current state number
ifeq SUCCEED                            ; if state = start goto SUCCEED
   invokestatic SASIJVML/ABORT()V       ; else violation
SUCCEED:
invokevirtual java/net/SocketOutputStream/write(I)V  ; send msg
...
```

Figure 2.7: JVML SASI enforcement of "no messages sent after reading a file."

instructions, and the security automaton simulation code segments are constructed so they do not contain labels.[12]

The type safety of JVML also empowers the JVML SASI user who is formulating a security policy that concerns application abstractions. JVML instructions contain information about classes, objects, methods, threads, and types. This information is made available (though platform-specific functions) in SAL to the author of a security policy. Security policies for JVML SASI thus can define permissible computations in terms of these application abstractions. In contrast, x86 code contains virtually no information about a C program it represents, so the author of a security policy for x86 SASI may be forced to *synthesize*, or abstract, from sequences of object code instructions those security events that use abstractions from the programming language or the application. Such synthesis is necessarily brittle, since those instruction sequences depend on target application implementation details (such as what high-level language it was written in).

[12] JVML SASI security policies must also rule out indirect ways of compromising the variables or circumventing the code added for policy enforcement. For example, JVML dynamic class loading and program reflection must be disallowed, or its behavior suitably modified to ensure IRM integrity (as discussed in Chapter 3).

JVML SASI in Action

Figure 2.7 shows code that was produced by the JVML SASI prototype for enforcing the security policy specification of Example 2.1 (corresponding to the security automaton of Figure 2.1 which prohibits sends after file reads). Code inserted by the JVML SASI prototype is typeset in a slanted font; original target system code is typeset in an upright font. Because the target system's instruction to invoke the `FileInputStream/read` method satisfies the security automaton's *read* predicate, this instruction has been prefixed by security automaton simulation code; that code causes an automaton transition to state *noSnd*. And because the target system's instruction to invoke the `write` method satisfies the *send* predicate, it too has been prefixed with security automaton simulation code; that code halts the application if *start* is not the current state of the security automaton.

To gain some understanding about the performance overhead of policy enforcement with JVML SASI, the prototype was used to implement the functionality of Sun's Java 1.1 SecurityManager (SM). There, enforcement is based on a security automaton that specifies checks to be performed at exactly those points in the Java system libraries where Sun's SM performs runtime security checks in the `java.lang`, `java.io`, and `java.util` libraries.

Parts of this specification are shown in Example 2.3. The parts shown stipulate that whenever a file is opened for reading, given a string filename, the SecurityManager (if one is present) must check for read access to the file denoted by that filename. This is implemented by—at the very beginning of the constructor method for `java.io.FileInputStream` object instances—invoking the SecurityManager's `checkRead` method with the string filename argument of that constructor.

To ensure the JVML SASI implementation is behaviorally equivalent to Sun's

```
/*
** Define SecurityManager
*/
SM() ::= invokestatic("java/lang/System",
                      "getSecurityManager",
                      "()Ljava/lang/SecurityManager;");
...
/*
** Perform SecurityManager check, if it's present
*/
check(sm,name,sig,arg) ::=
       sm==null
    || invokevirtual(sm,"java/lang/SecurityManager",name,sig,arg);
...
/*
** Checks related to ``file input streams''
*/
FileInputStream_Emulation_Points() ::=
       class=="java/io/FileInputStream"
    && atStartOfMethod()
    && (
          /* must have read access at time of construction */
          (   method=="<init>"
           && signature=="(Ljava/lang/String;)V"
           && check(SM(),"checkRead",signature,objectArg(1)) )
        ||
          /* other FileInputStream emulation points */
          ...
       );
...
/*
** The Security Automaton
*/
start ::=
  FileInputStream_Emulation_Points() -> start
  ...
  non_emulation_points() -> start
;
```

Example 2.3: Excerpts of the JVML SASI SecurityManager SAL specification.

original implementation, it calls the same Java 1.1 `check` functions as Sun's SM does. However, the JVML SASI implementation is more flexible than Sun's SM—`check` functions as well as new `check` points can be added to ours simply by modifying the security automaton. Modifying the SM in Sun's implementation requires a new release of Java: just subclassing the SM would not suffice, because the SM's invocation points are fixed by Sun's implementation.

The JVML SASI implementation of the Java 1.1 SecurityManager turns out to be quite efficient. Microbenchmarks to compare the security overhead of the implementation with Sun's SM show no real differences in the overhead. Moreover, the JVML SASI implementation has the possibility of being cheaper than the Java 1.1 SM implementation. Sun's SM is invoked at predefined points, whether the check will succeed or not. In settings where access checks are known to succeed—say, because a component is trusted or because of pre-existing access-control rights—the JVML SASI prototype does not add checking code (because the rewriter simplifies the security automaton as it is being inserted). Data for the Blast and the Tar benchmarks in [ET99] suggest that as much as a fourfold performance improvement can be expected when checking code is eliminated in what arguably are realistic applications.

When the JVML SASI SM is violated, the SASI-inserted security enforcement code does not halt the execution of the target system but, rather, throws a Java exception of type `java.lang.SecurityException`. This corresponds to the behavior of Sun's SM, but does not fit in the framework of security automata: the security automaton rejects its input without halting the target system's execution, and the thrown exception will modify the target system's behavior. This suggests that SASI might be extended to handle security policies such as this one—and, in

```
STATE {
    Assume Categories maps each company to a category.
    Let usedCategories be an empty set.
    Let seenCompanies be an empty set.
}
EVENT Any application-level operation
CONDITION The operation involves an access to a company
UPDATE {
    Let company be the company being accessed.
    Let category be company's category in Categories.
    If category ∈ usedCategories and company ∉ seenCompanies {
        REJECT the operation.
    } Else {
        Add category to usedCategories.
        Add company to seenCompanies.
        ALLOW the operation.
    }
}
```

Example 2.4: The Chinese Wall security policy.

fact, Chapter 3 describes such an extension.

2.4 The Chinese Wall Security Policy in SASI

To gauge how flexible SASI is in enforcing various security policies, it is worthwhile to consider how SASI might be used to enforce the Chinese Wall security policy [BN89], one concrete policy from the literature.

Chinese Wall, outlined in Example 2.4, was introduced into the computer security literature as an example of a security policy that could not be enforced using the access control mechanisms typically found in operating systems. The policy aims to prevent the same individual (e.g., a consultancy analyst) from handling sensitive data about competing companies by separating companies into disjoint categories and only allowing the analyst access to one company from each category. As shown in this section, SASI IRMs can easily enforce Chinese Wall policies. How-

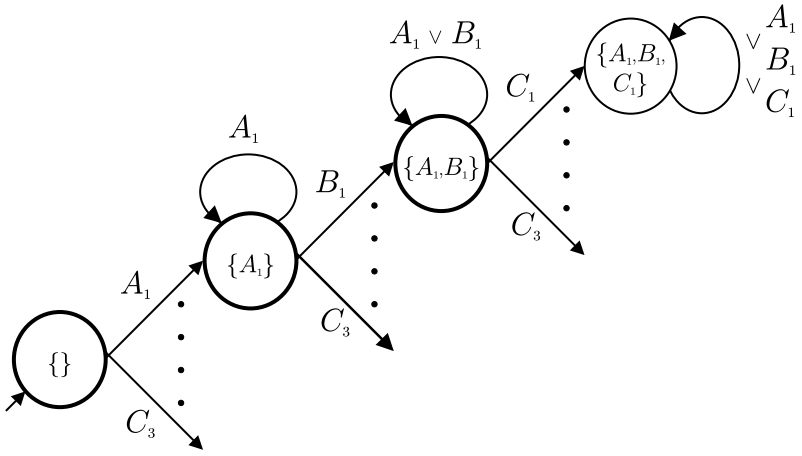


Figure 2.8: Security automaton encoding all possible states of Chinese Wall.

ever, the necessary SAL specifications quickly become both awkward and opaque, which leads to the introduction, in the next chapter, of a new policy language similar to that used in Example 2.4, supported by a new IRM implementation.

For expository purposes, the remainder of this section makes some simplifying assumptions about the security policy particulars. First, the assumption is of a Chinese Wall security policy with a fixed number of companies and categories. In particular, the security policy is assumed to have three company categories, named A, B, and C, with three companies in each category. Within a category X, companies are named X_1 through X_3. (E.g., companies in category A are A_1, A_2, and A_3.) The only way a user can access data concerning a company X_i is assumed to be through invoking the command read(X_i), which may, e.g., display information about company X_i on the user's screen. Further, users are assumed to be perpetually subject to the security policy—i.e., each user has a copy of the security state, and that security state is never reset.

A Multi-state Security Automaton

Figure 2.8 shows part of a multi-state security automaton for the Chinese Wall security policy, specified with regard to the simplifying assumptions. This security automaton encodes each valid combination of concurrently accessed companies—that is any subset of companies with at most one company from each category—in a distinct automaton state. For example, the security automaton will be in a state that encodes the set $\{A_1, B_1\}$ of accessed companies after companies A_1 and B_1 (and only those two) have been accessed. Each such automaton state, which encodes a set S of accessed companies, has a transition back to itself iff company X is accessed and $X \in S$. (Here, only read(X) commands are assumed to form input symbols to the security automaton; all other program actions are being ignored.) In Figure 2.8, this transition predicate is denoted by $A_1 \vee B_1$ for the automaton state that encodes the set $\{A_1, B_1\}$. Finally, an automaton state for the subset S has a transition to the automaton state for each strict superset $S \cup \{X\}$, whose transition predicate is satisfied on read(X). Thus, the automaton state for $\{A_1\}$ has a transition with predicate B_1 to the automaton state for $\{A_1, B_1\}$.

The size of a SAL security automaton specification is relative to number of automaton states, transitions, and transition predicate size. This Chinese Wall security automaton has 64 states, for three categories and three companies in each category. In general, for N company categories and k companies in each category, this Chinese Wall security automaton will be of size:[13]

$$\text{states}(N,k) = \sum_{i=0}^{N} \binom{N}{i} k^i, \qquad \text{transitions}(N,k) = \sum_{i=0}^{N} \binom{N}{i} k^i \left((N-i)k+1\right).$$

[13] The first formula sums over states required to encode that i categories (of k companies each) have been accessed. The second formula sums over the transitions from states of i categories to states of $i+1$ categories and adds the self-loop.

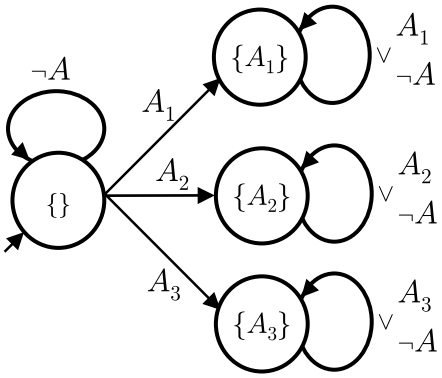


Figure 2.9: Category-specific security automaton for Chinese Wall.

Thus, for 33 categories of three companies each, it will have approximately 2^{66} states and 2^{71} transitions. Thus, writing a SAL specification for this formulation of the Chinese Wall security policy quickly becomes infeasible.

One Security Automaton per Category

The Chinese Wall security policy is less difficult to specify in SAL if it is formulated as a collection of security automata, rather than one large automaton. This formulation associates each company category with a security automaton that encodes whether any company in that category has been accessed, and, if so, ensures that only that company can be accessed from that category.

Figure 2.9 depicts the security automaton for category A. This security automaton has an initial state to encode that no company from category A has been accessed, and three states that, respectively, encode the three companies in category A: A_1, A_2, and A_3. From the initial state there is a transition whose predicate is satisfied for accesses to company A_i and whose target is the state for A_i. Also, there is a transition from the initial state to itself whose predicate is denoted $\neg A$ and is satisfied for operations that access companies not in category A. (If com-

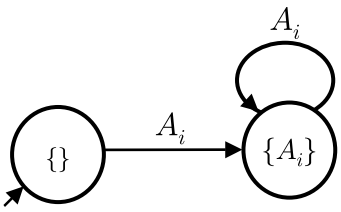


Figure 2.10: Incorrect attempt at constructing a security automaton.

mands besides read(X) form input symbols to the security automaton, then the predicate $\neg A$ must also be *true* for operations that do not access companies.) Finally, from each state encoding company A_i, there is a transition back to that state for accesses to company A_i or to companies in other categories, denoted $A_i \vee \neg A$.

This Chinese Wall security policy formulation is more convenient to specify in SAL because its security automata are of size linear in the number of companies. Even so, this policy formulation may incur a large security enforcement overhead, because there are as many security automata as there are company categories. If each security automaton receives an input symbol whenever a company is accessed, i.e., on each read(X) command, the enforcement overhead is linear in the number of company categories. However, this runtime overhead will be greatly reduced if the company name X forms a static argument to read(X) since the simplification performed during rewriting by SASI will eliminate all but one of the automata. Thus, depending on the circumstances, this formulation of Chinese Wall as security automata can be both simple to manage and offer good performance.

This last formulation of Chinese Wall used static simplification to reduce overhead by forwarding an input symbol to a category's security automaton only when a company in that category is accessed. If the set of input symbols is guaranteed to always be restricted in this manner then the security automaton of Figure 2.9 can be simplified by removing the $\neg A$ from its transition predicate. It is tempting

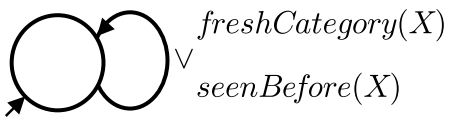


Figure 2.11: Single-state security automaton for Chinese Wall.

to further simplify the security automaton of Figure 2.9 into the "security automaton" of Figure 2.10. However, security automata can only summarize the execution history of the target system through distinct automata states. The right state of Figure 2.10 implicitly encodes three different states that summarize the execution history—that of companies A_1, A_2, and A_3 having been accessed—and, therefore, the figure does not show a security automaton.

A Single-state Security Automaton

Security automata must summarize in a distinct state each security-policy-relevant execution history of the target system—this causes the prohibitively large size of Figure 2.8's automaton and disallows attempts like that of Figure 2.10. Some policy-relevant state might already be maintained, either internally in the target system itself, or externally (e.g., by the operating system). For example, applications that access files are likely to contain handles to currently open files and, for each application, the operating system will know which files those are. Referring to this pre-existing state might well allow the number of states in a security automaton to be reduced. For instance, a security policy that prohibits more than 10 open files can be enforced by a single-state security automaton that queries the operating system about the number of open files.

The security automaton of Figure 2.11 is a single-state formulation of the Chinese Wall security policy. The automaton uses functions, *freshCategory* and *seen-*

Before, that summarize what companies have been accessed and that are assumed to be supported by the target system. Thus, when a company X is accessed, *freshCategory*(X) function returns *true* if company X is in a category in which no company has been previously accessed, and *seenBefore*(X) function returns *true* if company X has been accessed before.

If general-purpose state and computation can be explicitly added to the target system—such as the map usedCategories and the security update of Example 2.4—security enforcement need not rely on policy-relevant state already existing and being correctly maintained in the target system. For instance, in Example 2.4 the specification of the Chinese Wall security policy updates the variables usedCategories and seenCompanies when a DeMorgan's form of the predicate in Figure 2.11 holds true, thereby implementing *freshCategory* and *seenBefore*.

Instead of specifying security policies as state-by-state automata, as done in SAL, security automata can equivalently be specified using structured state and computation. Explicitly specifying the addition of state and computation into the target system also removes the need for many assumptions. The x86 SASI rewriter had special support for SFI, through platform-specific functions, and the security automaton of Figure 2.11 can only be enforced if *freshCategory* and *seenBefore* already exist in the target system or are added by the rewriter. Also, neither the security automata of Figure 2.8 nor that of Figure 2.9 can support the addition of new companies and categories. This flexibility is easily supported by a security policy specification for the Chinese Wall security policy that uses structured state and general-purpose computation.

2.5 SASI in Retrospect

The SASI prototypes proved instructive, but neither x86 SASI or JVML SASI is a practical tool. First, SAL is an awkward language for writing real security policies because SAL forces relevant execution history of target system to be summarized in a single, unstructured "current" state of one or more security automata. Values from the target system that are instances of application-level abstractions must be encoded in order for them to play any role in subsequent enforcement decisions. It is difficult, for example, to write in SAL a policy specifying that the same character string is used in a sequence of operations—SAL cannot directly store strings in automaton states. Changing SAL so that security state can contain typed variables—with types that may be application-level abstractions (e.g., strings)—eliminates this difficulty.

A second difficulty with the SASI prototypes concerns what low-level actions can be used in defining security policies. A reference monitor that checks every machine language instruction initially seemed like a powerful basis for defining application-specific security policies. In practice, this power has proved difficult to harness. Most x86 object code, for example, does not make explicit the application-level abstractions that are being manipulated by that code. There is no explicit notion of a "function" in x86 assembly language, and "function calls" must be found by searching for code sequences resembling the target system's calling convention. The author of a security policy thus finds it necessary to embed a disassembler, or other analysis mechanism, within each SAL security policy description. This is cumbersome and error-prone. One solution would be to build a SASI that modified high-level language programs rather than object code. A security automata could be inlined into the C++ program (say) for the target system rather than being inlined

into the object code produced by the C++ compiler.[14] But this is unattractive, because a SASI that modifies C++ programs adds the C++ compiler to the TCB.

The approach taken in JVML SASI is the more promising way to handle security policies involving application-level abstractions. That is, rely on annotations of the object code that are easily checked and that expose application-level abstractions. This approach is not limited to JVML code, or even to type safe high-level languages; even x86 object code can include the necessary annotations [MWCG98]. The next chapter, therefore, builds on the SASI experience and develops the IRM approach to its full potential.

[14]This is the approach taken by the Aspect-oriented programming methodology [KLM+97, WZL03].

Chapter 3

Inlined Reference Monitors Refined

The previous chapter described one strategy for implementing an IRM toolkit that used the SAL policy language, and two prototypes based on that strategy. These SASI prototypes failed by not fully providing the properties required of reference monitors, namely (to paraphrase from Figure 1.1), to be a trustworthy mechanism that fully mediates all security-relevant actions in the execution of a target application and subjects each to a validity check. There are three primary reasons why the SASI prototypes failed to meet this criterion.

1. The low-level actions fundamental to SASI were the set of machine language operations in the target application (i.e., the machine's *Instruction Set Architecture*, or ISA). This is problematic because the ISA is both too rudimentary for some critical security-relevant actions, like function calls, and too high-level for others, like interrupt and signal handling.

2. Certain characteristics of the SAL policy language proved an obstacle to writing useful IRM security policies. In particular, SAL forces users to encode all security-relevant information in an automaton state, and when security-relevant actions occur SAL allows only the evaluation of side-effect-free state-transition expressions. This awkward automata-state encoding of intended security policies facilitates automatic analysis and optimization [Wal99a, BR02], but makes writing SAL specifications difficult.

3. Finally, SAL and SASI cannot be seen as trustworthy, because they might require convoluted constructions for those security events that do not di-

rectly correspond to machine opcodes. The occurrence of higher-level security events must be synthesized from a sequence of fundamental ISA actions and encoded into the single "current" automaton state, along with all other security-relevant information. This is unfortunate, because security policies are more likely to proscribe higher-level events (like launching missiles or sending email) than they are to forbid integer multiplication.[1] Inference from low-level execution does not form a sound basis for mediating these events, especially in SAL policies, where a single automaton state must encode all relevant information.

This chapter describes how to overcome the above limitations and provides a more satisfactory implementation of the IRM approach, the PoET Policy Enforcement Toolkit. The writer of IRM security policies is given a new language, the PSLang Policy Specification Language, where security state is maintained explicitly in named variables, security updates are imperative program fragments, and IRM enforcement code can use the types and abstractions of the target application. PSLang/PoET IRMs operate on a set of fundamental low-level actions strictly richer than the machine ISA and can easily specify how higher-level security events are synthesized and what enforcement activity they trigger. Security policies that operate on synthesized higher-level security events are just one example of a new capability of *security policy extensions*; in particular, a PSLang/PoET IRM can fully implement the mediation of all security-relevant actions, including any required security event synthesis, and leave the specification of validity checks to other PSLang policies, or even to the run-time system. As a result, PSLang/PoET

[1]Note that even a security policy restricting how integers can be multiplied might need to synthesize when the multiplication security event has happened, for example, as a result of a sequence of addition actions.

IRMs are more trustworthy, because they can observe relevant activity without unnecessarily complicated analysis or inference, and because specifications are written in a familiar structured style that promotes reuse.

The next section, §3.1, expands on what constitutes a complete set of security-relevant actions and how to mediate those actions in a trustworthy manner. The PSLang/PoET IRM toolkit for JVML and Java target applications is described in §3.2, with §3.2.1 and §3.2.2, respectively, giving details on the toolkit's PoET rewriter and PSLang security policy language. Section 3.3 describes how, for JVML, a full set of fundamental security-related events can be synthesized using PoET and PSLang. The remaining issues facing PSLang users, many of which are common to implementors of any security mechanism, are discussed in §3.4. Deployment of IRM-based security enforcement, and how it can be adopted in existing infrastructures, is the topic of §3.5. Finally, §3.6 shows how the scope of the IRM approach, and of PSLang/PoET, can be expanded to include activity not generally considered to be security enforcement.

3.1 Mediating Security Events

Before embarking on a new implementation of the IRM approach, it is prudent, given the experience of SASI, to carefully consider the different trustworthy means for the synthesis and mediation of security events, as well as whether any new low-level actions must added to the proposed IRM toolkit.

An IRM toolkit must, at least, provide high-assurance mediation of the system's low-level actions—as any security policy could be formed by such actions. These low-level actions include activity like the callbacks and interrupts that are generated by the target application's execution environment. For such actions an

IRM toolkit must build upon their actual system implementation to capture the correct semantics. Fortunately, IRMs can usually interpose on these particular actions by registering a handler with the runtime system and by preventing the secured application from overriding that handler.

It seems natural, however, to dismiss approaches that address with ease low-level actions such as ISA operations but can only with great difficulty mediate other significant and common activity such as exceptions or function calls.

3.1.1 Dealing with High-level Languages

Function calls, and other language-based structures in the target application, will typically be important for security policies. For these (except in a handful of interpreted systems), an IRM cannot receive explicit notification from the runtime system—such language-based primitives are most often implicitly encoded in the ISA operations of the target application as a result of compilation and translation from a higher-level language. Attempts like those of x86 SASI that synthesize the original language-based events from their ISA encoding will necessarily be both complex and incomplete, and therefore untrustworthy.[2]

Fortunately, the compilation mechanism that creates the binaries of target applications also creates evidence in the form of debug information, types, and tables of imported and exported symbols, for example. This evidence is persistently maintained throughout the translation process, compilation, linking, and loading, and can show where high-level language abstractions are encoded in an applica-

[2]In general, such synthesis might not even be feasible, because it must solve combinatorial pattern matching and semantic equivalence problems [KS94], as illustrated by the policy forbidding multiplication mentioned in a footnote on page 52. Although this may be an acceptable limitation for abstruse policies, like that of the footnote, an IRM toolkit must do better on key activities, like function calls.

tion [KBA02, SBN+97]. This information should be sufficient for IRM needs, at least for target applications written in type-safe languages.

Unfortunately, an IRM toolkit cannot directly trust this type of compiled-in evidence. Unlike signals in Unix [GS91] and other low-level actions triggered by the system runtime, language-based information is usually not produced by part of the TCB, but rather, by an external, possibly malicious, entity. The common means of establishing that information originates from a trusted party, such as cryptographic signatures or trusted servers, all introduce additional complexity and the potential for new vulnerabilities.

Even if a trusted external party has truly vouched for some compiler-generated evidence, the integrity of that evidence may later be invalidated by runtime activity. Languages like C and C++, although popular, fail to guarantee that an application's control-flow and memory will not be corrupted in a way that violates the intent of the high-level source code and compiler [Koz98]. For such common languages, reliance on compiler annotations, even when certified by a trusted entity, is likely only to increase the size of the TCB and allow novel types of attack. In particular, cryptographic evidence showing the use of a trusted compiler (a strategy used, for example, in Vino [SESS96]) or the inclusion of a compiler in the IRM toolkit will not provide complete assurance [CER00].

As an alternative, and to be able to rely on strong guarantees about behavior, an IRM toolkit may choose to only trust external certificates for target applications that are compiled from type safe languages. This was done, for example, in the SPIN operating system [BSP+95]. In this case, however, the high-level language compiler must in all cases correctly enforce its type safety guarantees—a statement unlikely to hold true for a modern optimizing compiler [Mor95, MTC+96, Aba98].

Here, adding a compiler to the TCB fails to increase assurance, as it fails with C and C++, because the compiler itself is untrustworthy.

As a final attempt to increase assurance, static type-safety guarantees can be augmented by runtime checking of the compiled-in evidence that is part of the target application. Such dynamic safety checks can provide significant guarantees, even for target applications compiled from legacy languages, but those guarantees are likely to come at a high performance cost [KBA02, SD02]. Anyway, these runtime checks, like SFI, are just an instantiation of a limited form of the IRM approach, and do not alleviate the need for a trustworthy foundation for language-based low-level actions in a general IRM toolkit.

3.1.2 Using Verifiable Code Certification

Verifiable Code Certification (VCC)[3] is a technique that guarantees low-level activity of a target application satisfies properties specified by a high-level language (or, more generally, a formal theorem). VCC does not rely on certificates from external trusted parties. Instead, VCC provides assurance by making use of a low-complexity *verifier*, which avoids placing trust in compilers [NL98], analysis engines or other proof generators [Nec97], or further translation [MWCG98]. Typically, what is verified is a high-level-language type safety property, such as the assertion that all operations are on data of correct type.

To enable verification of each target application, VCC requires the application to be accompanied by a formal statement that proves that its execution satisfies the desired properties. This requirement enables VCC to make use of a natural

[3]This dissertation uses the term Verifiable Code Certification, or VCC, for what is sometimes called self-certified code, but more often (and misleadingly) called language-based security [Koz99, BL01].

asymmetry—that the construction of a proof may be complex (and sometimes impossible [Göd31]), but once a formal proof has been created, checking its correctness can be relatively straightforward. By using this asymmetry, the TCB can remain trustworthy, despite its enforcement of language-based guarantees, as only the VCC verifier—which is simple, and can be used at the time of IRM rewriting—needs to be added as a trusted component.

There is a natural connection between IRMs and the creation of VCC-based applications. Some compilers, such as for the Lisp language [Ste90], enforce a type safety policy on their output applications only through embedded runtime checks—in effect, rewriting their output with an IRM for this high-level language-based policy. Such a compiler could be enhanced with an analysis that removes superfluous runtime checks and thus obtains higher performance. For each removed check, this analysis should be able to produce a proof that the type safety policy was still in effect for all executions of the target application. The VCC approach works by verifying exactly this type of proof, with runtime checks remaining where the compiler cannot statically establish type safety. Therefore, an IRM, such as that for x86 SASI SFI, can, in a sense, provide the same benefits as VCC, but VCC achieves superior performance by eliding checks and also contributes less to the TCB—providing the desired guarantees without trusting either a compiler or a complicated runtime analysis.

VCC was popularized by Java applets—web page extensions expressed in Java Virtual Machine Language (JVML) "bytecodes" that provide a form of VCC [YL96, SA99]—but the VCC approach is not limited to Java. Several technologies have extended the principles of VCC to properties other than type safety or to support the x86 ISA [Nec97, MWCG98].

The demonstrated practical applicability of the VCC approach, in combination with VCC-based industrial initiatives such as Java and Microsoft's .NET [BS02], makes it an attractive option for security enforcement. In particular, VCC can be leveraged as a trustworthy substrate for the mediation of language-based low-level actions in implementations of the IRM approach.

3.1.3 On Application and Platform Interfaces

For a given security policy, even the set of all low-level actions (i.e., ISA operations, actions triggered by the system runtime, or use of language-based primitives) is unlikely to capture—without additional context—exactly the security-policy-specific security events that an IRM must mediate. Most security policies do not apply so directly to language or system abstractions; they instead originate as informal requirements outside any computer system. So, a security policy is likely to describe invariants specific to a particular organization or application, and to restrict processes so that these invariants are maintained [Ste91, CW87]. Such high-level security policies typically map to restrictions on the use of certain components in a particular target application, or to conditions that must hold before and after transactions or function calls. Enforcing such security policies, therefore, requires introspection into a target application.

Application-specific security policies like these can, fortunately, be enforced in modern software: layered design and modular development enables the required insight into the structure of applications. Target applications written as extensions to an application platform will make use of fully-specified interfaces with known semantics, both for convenience and for interoperability. These platform interfaces comprise a rich set of events on which any practical security policy can

be constructed, analogous to (at least for type safe target programs), but at a significantly higher level than, the system-call interface of traditional reference monitors [GS91, SR99].

IRM enforcement of security policies is also more trustworthy when specified in terms of the *APIs*, or Application Programmer Interfaces, used by the target application. These APIs are at a higher level of abstraction, so they are likely to have semantics close to that of a given security policy's security events, thereby reducing the need for intricate and error-prone synthesis of security events. In fact, although the IRM approach allows mediation of any low-level action, individual policies will often primarily work by mediation of higher-level API security events. This, because popular target platforms like Java and Microsoft's .NET [GJS96, BS02] typically fully encapsulate all low-level resources (including security-critical data, such as the operating system) in one or more APIs, allowing such resources to be protected at that level.[4]

Specifying the mediation of higher-level security events, such as the use of platform APIs, is only a first step in the creation of a security policy. A policy may, in subsequent steps, delineate how such security events are constrained or combined to more accurately capture activity the policy aims to regulate. For example, filesystem security events might be restricted to open only user files, and applets might only receive file handle objects created for reading. A security policy

[4]Note that this platform-level enforcement also helps ensure that IRM rewriting maintains the functional equivalence of the target application and the secured application—the one goal (introduced in Figure 1.4) that IRMs do not share with normal reference monitors—as platform users must ignore the type of perturbations caused by IRMs: they may occur normally in different platform versions. Empirical evidence from virtual machine products shows that such a *de-facto* platform exists even for machine ISAs, as most software must provide the same function despite a wide variety of hardware- and system-induced perturbations [SVL01].

should be able to fully specify all security events of interest through a sequence of such steps, with each step defining a set of new security events more relevant to the task at hand from the existing security events, either through simple conditions or more complicated synthesis.[5]

3.1.4 To Enforce Least Privilege

Few rules-of-thumb exist for implementing security policies and enforcement mechanisms, because such implementations are naturally tied to intricate details of particular platforms and even specific target applications. However, the design principles in Saltzer and Schroeder's tutorial paper on protection of information in computer systems remain as relevant today as when they were written, three decades ago [SS75]. These principles mostly overlap with the properties of reference monitors, described in §1.2, except for their *Principle of Least Privilege*, which states that a target application should be granted only the bare-minimum capabilities its functionality requires.

Full implementation of the principle of least privilege is a worthy goal, but, in practice, is difficult to achieve [Sch03]. It is greatly facilitated, however, by choosing to enforce security at the right abstraction level. For instance, if a security policy that aims to preclude filesystem access in a target application only restricts the use of high-level platform APIs, the libraries that implement those APIs will still be able to perform such filesystem accesses even though these libraries form part

[5]The strategy described here, of bottom-up synthesis of security events, first from security-relevant actions, and then further from other security events, purposefully blurs the distinction between policy and mechanism—even though maintaining this distinction is a central tenet of system design [EKO95]. For security enforcement, unlike many software systems, implementation and mechanism details cannot be abstracted from policies [Sch99a, Gar03, AG03].

of the application. Such situations occur frequently in systems (e.g., a graphical application that is to be restricted from file access might still need to use a platform service that reads font outline files and writes back to a file cache of rendered bitmaps). Thus, choosing to enforce policies by mediating a high-level API can hide platform implementation details, allowing both greater functionality and simpler enforcement.

Striving for least privilege exposes a balancing act between the simplicity and exactness of security policies and mechanisms. The file access example above has a simple policy if font drawing operations are treated as opaque and the platform semantics are known and trusted by the policy writer. This trust is misplaced if the font drawing operations are flawed or misunderstood—for example, if they allow arbitrary files to be read and exposed as bitmaps. However, not trusting platform semantics requires a security policy that explicitly permits all file operations required for font-related functionality and, to do this, the policy must correctly encode the necessary platform configuration and implementation details. Avoiding this increased complexity might lead to policies that preclude more activity than necessary, simply to allow for trustworthy enforcement.

In this manner, one proposal advocates applying the principle of least privilege by creating a multitude of policies, each of which puts simple restrictions on some platform interfaces whose semantics are trusted [Sch03]. On its own, each such policy would grant strictly more than the least privilege needed but would be simple enough to allow for trustworthy enforcement. A more restrictive policy could then be created for a given category of target applications by composing sets of these simple policies, although such composed policies would likely still err on the side of being too permissive for any particular target application. However, if generally

accepted, such categories could allow assured enforcement of restrictions known to both end users and creators of applications [SVB+03]. Despite the moderate success of current systems that implement this proposal [CER00], few equally promising alternatives exist.

Another useful source of general security policy guidelines is Clark and Wilson's work on commercial security policies, which addresses the issue of application-specific validity checks [CW87, TW89]. In [CW87] the authors observe that real-world security objectives limit, in an application-specific manner, application-specific accesses and updates, and therefore any automatically-enforced security policy must both be specific to the application and place trust in some application procedures. Their proposed framework takes a transactional view of applications and makes access contingent on application-specific runtime validity checks of necessary preconditions. Finally, the paper lays down a challenge: discovering automatic ways to increase assurance in preconditions, giving the two-phase-commit solution to database serializability [LS81] as motivation. Here, VCC can play a role by automatically assuring the type-safety of target applications, a requirement previously only discharged by trusting a complex language infrastructure.

Eventually, security policies are inexorably tied to details of the system they protect, the restrictions they aim to impose, and the activity to be permitted in target applications. A general security enforcement framework can, at best, hope to give policy writers the expressive power and flexibility they need to accurately and correctly specify their intent. That is, the ability to specify exactly what defines a security event, upon what invariants that definition relies (and how it is maintained), which security events are mediated, and what validity checks are performed upon those events. Moreover, to be trustworthy, the resulting specification

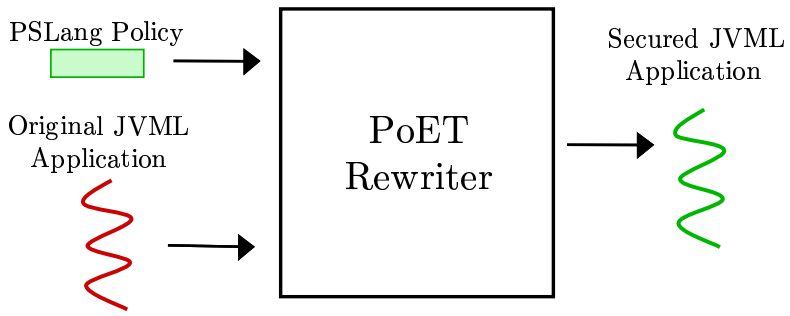


Figure 3.1: The PSLang/PoET implementation of the IRM approach.

must allow for its inspection and analysis in isolation. The next section describes an IRM toolkit that implements such a framework.

3.2 PSLang/PoET for Java

This section gives an overview of the PoET and PSLang implementation of the IRM approach; its operation is depicted in Figure 3.1 and described in detail in Appendix A. PoET and PSLang operate on JVML target programs [YL96], typically compiled from source code in the Java language [GJS96]. PoET builds on the VCC guarantees of JVML to identify and expose a rich set of fundamental JVML low-level actions. Security policies are expressed in PSLang[6], an imperative language with transparent and familiar semantics that are closely tied to JVML target programs. Higher-level security events are captured easily in PSLang policies through decomposition and factoring, or by synthesis using arbitrary data structures and computation. This IRM toolkit overcomes the three critical obstacles to IRM trustworthiness that existed in SASI and that were detailed at the beginning of this chapter.

The PoET policy enforcement toolkit includes an IRM rewriter, which oper-

[6]The "P" in PSLang is silent, as that in Psmith [Wod10].

ates on target programs' JVML class files. IRM security policies, specified in the PSLang policy specification language, are also input to the PoET rewriter to create a secured version of the target program. The secured application output by PoET includes both added security state variables and added JVML code for security updates, simplified by partial evaluation. The PoET rewriter inserts IRM enforcement code either at places where security-relevant actions can be directly identified or at places where such actions can be exposed by the insertion of additional code, like exception handlers. PSLang policies define the implementation of security updates that occur at each policy-relevant action, specify any validity checks that are included in those updates, and specify what remedial action is taken (e.g., to terminate the secured application) if those checks fail. The full implementation of PSLang/PoET consists of approximately 15 thousand lines of Java code.[7]

3.2.1 The Policy Enforcement Toolkit

The PoET IRM rewriter, whose operation is outlined in Figure 3.1, turns JVML class files of a target program into a secured application—including rewriting any JVML code that is dynamically generated or implicitly or explicitly loaded at runtime. This is depicted in Figure 3.2. Command-line arguments and other invocation details for PoET are given in Appendix A.2.

All low-level actions of the JVML ISA are exposed by PoET to PSLang policies. These low-level actions include class and object initialization and garbage collection, method calls, and member field access, as well as execution of basic blocks,

[7]Included is a JVML class file reflection library comprising 5K lines, which would be unnecessary if PoET was made specific to a particular JVM implementation. PoET itself comprises only 2.5K lines, with the ADT libraries of Appendix A included, and the remaining 7.5K lines implement PSLang, although, there, 4K lines are automatically-generated lexer and parser code.

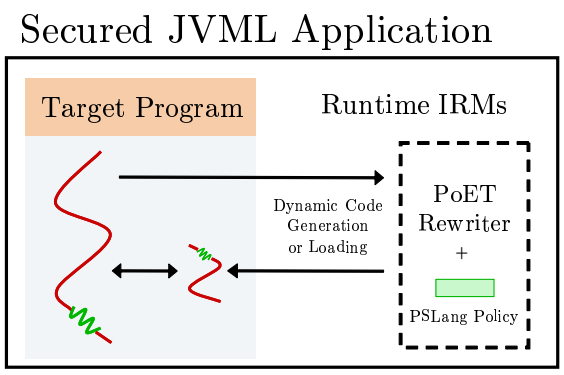


Figure 3.2: IRM rewriting of dynamically-loaded or -generated code.

exception handlers, and individual JVML instructions; Appendix A.1 describes all PoET events. The details of each JVML low-level action are exposed by PoET through a set of runtime libraries, `Event`, `Reflect`, and `State`, explained in the Appendix. PoET runtime libraries also expose certain JVML interfaces (e.g., for handling primitive types) and form a basis for extending PSLang with new data structures and functionality.

PoET uses the VCC guarantees of verified JVML bytecodes as the foundation for ensuring the integrity of IRM enforcement code, the PoET toolkit, and PSLang policies, all of which are co-located inside the secured application at runtime. Figure 3.2 shows how PoET ensures that a secured application complies with a PSLang security policy. Both the initial JVML bytecodes of a target program and code from explicit use of the JVML ClassLoader [Gon99] are run through the JVML type safety verifier and rewritten by PoET. The details of how PSLang/PoET ensures IRM integrity are discussed in §3.2.3.

IRM security state added by PoET is globally visible to all IRM enforcement code and may use any JVML type. In addition, the `State` runtime library allows security state to be embedded into JVML classes, object instances, and methods. This allows individual data objects of the target application to carry annotations

that dictate their handling by IRM enforcement code, reducing enforcement overhead for many policies. For instance, a policy that limits the amount of information read from files could associate a counter with every file object, incrementing that counter on read operations—saving an indirection to such a counter through some global data structure, whose maintenance is likely to be expensive. The VCC guarantees of JVML help ensure this added state, like the IRM enforcement code added by PoET, remains inaccessible to the target program.[8]

3.2.2 The Policy Specification Language

PSLang is designed to allow security policies to be specified in a straightforward fashion that clearly describes both the high-level semantics of security enforcement and its detailed implementation, while remaining a transparent language that is simple enough to be trustworthy. To this purpose, PSLang is a small imperative language (described in Appendix A) whose syntax and semantics are based on Java and JVML and therefore easily understood by writers of policies for JVML programs.

PSLang syntax contains little sugar but focuses on providing primitives natural to the factoring of security policies. Therefore, PSLang policies are structured around security updates, where a JVML low-level action is coupled with an explicit condition stating when it is relevant to the security policy at hand, thereby specifying a higher-level security event. PSLang security updates also specify an

[8]JVML also obscures from target programs implementation details such as intra-instruction distance and memory layout of data. This helps ensure that a secured application will be functionally equivalent to the original. However, nothing is perfect, and a devious JVML target programs might still be able to detect the presence of PoET additions, for example, by inference from the number of object allocations required to exhaust memory.

```
USES LIBRARY Lock;
GLOBAL SECURITY STATE {
    int openWindows = 0;
    Object lock = Lock.create();
}
ON EVENT at start of method
WITH Event.methodPrototypeIs("void java.awt.Window.show()")
PERFORM SECURITY UPDATE {
    Lock.acquire(lock);
    if( openWindows = 10 ) {
      FAIL[ "Too many open GUI windows" ];
    }
    openWindows = openWindows + 1;
    Lock.release(lock);
}
ON EVENT at start of method
WITH Event.methodPrototypeIs("void java.awt.Window.dispose()")
PERFORM SECURITY UPDATE {
    Lock.acquire(lock);
    openWindows = openWindows - 1;
    Lock.release(lock);
}
```

Example 3.1: PSLang security policy that allows at most 10 open windows.

imperative program fragment that forms the IRM enforcement code to be triggered by this security event (i.e., executed when the secured application performs the low-level action and the condition is satisfied).[9] PSLang policies can add security state to the secured application, either globally through special syntax, or (as discussed in the previous section) to individual JVML elements using the `State` library.

PSLang libraries offer a rich set of data structures and functionality through the syntax of *Abstract Data Types* (ADTs), and they form a back door through which PSLang can be augmented with new ADTs written in Java. Default PSLang ADTs include data structures for sets, associations, vectors, tuples, stacks, and queues;

[9] Appendix A.1, gives more detailed information on the syntax and semantics of PSLang security updates, starting on page 128.

ADT extensions expose fundamental JVML operations, the runtime system, locks, delay timeouts, and the Java `Permissions` framework [Gon99]. The resulting PSLang increases overall assurance in security enforcement by enabling security policies to be specified in a natural and clear manner. Example 3.1 illustrates this with a PSLang specification for multi-threaded JVML target programs that enforces a security policy that generalizes Example 1.1 from §1.3.

PSLang policies can also define functions, which allow common expressions and statements to be factored out into a single imperative program fragment that may then be invoked anywhere in a policy. Last, but not least, PSLang allows one policy to extend another—thus, security policies may be layered into reusable IRM enforcement code and security event specifications, increasingly specific to the intent of particular application-specific security policies.

PSLang was designed to impose low performance overhead. Like SASI, PoET achieves this by simplifying security policies using techniques that build on partial evaluation of imperative programs [ASU85, JGS93, Thi03]. PSLang policies are partially evaluated at each security-relevant action in the target program, using information derived from the static context of each insertion point. The PoET rewriter only inserts IRM enforcement code at security events, which are likely to be a small fraction of the JVML low-level actions. To help ensure this, the event predicate expression of each PSLang update, which defines security events, is restricted to use only *side-effect-free* functions and ADT operations that are guaranteed to return a value when statically evaluated. The writer of a PSLang security policy can define his own side-effect-free functions, and they also help in the simplification of security updates and other IRM enforcement code.

There are only cosmetic differences between PSLang and the SAL of Chap-

ter 2—the imperative style of PSLang is simply an alternative presentation of deterministic security automata, albeit more familiar. Admittedly, PSLang is enhanced by PoET's identification and exposure of a large set of JVML low-level actions (such as finalizer garbage-collection callbacks), as well as by the functionality in ADT libraries. However, SAL could accommodate such enhancements. PSLang has one clear advantage for writing policies—namely, the availability of typed state variables (e.g., strings, floating-point numbers, etc.). Apart from this, PSLang is simply a more convenient way to specify security automata. But because PSLang is less opaque and has fewer hidden assumptions than SAL, it offers increased transparency which results in greater reliability and assurance.

3.2.3 Protecting IRM Integrity

Not only do the VCC guarantees of verified JVML programs allow PoET to identify language-based low-level actions, but they also create a trustworthy foundation for protecting IRM integrity. PoET can be certain that target programs do not refer to security state inserted by IRM rewriting, because any such reference to non-existent state would have been rejected by the JVML verifier.[10] Also, JVML semantics guarantee at runtime that the instructions of the secured program cannot be tampered with, that the stack and registers of a method are local to the thread currently executing in that method, and that in a method some registers can be set aside for exclusive use by IRM enforcement code.

Unfortunately the above guarantees are not enough to protect IRM integrity.

[10] PoET was implemented on top of closed commercial JVMs and, therefore, currently only the rewritten bytecodes of secured programs are passed through a JVML verifier to ensure type safety. This vulnerability can easily be remedied by running PoET on an open JVM implementation (such as Jikes [AAB+00]) where the target program's original bytecodes can also be verified.

Because JVML allows both dynamic class loading and reflection, a malicious target application might be able to access and subvert security state and IRM enforcement code. PoET partially addresses these problems by rewriting all classes loaded at runtime with a PoET rewriter in the secured application that applies the PSLang policy to be enforced (as shown in Figure 3.2).

The key to IRM integrity protection lies in a unique prefix that is given to all names used by IRM enforcement code.[11] This prefix is used for the global security state of the IRM, all local security state added to classes, ADT functions, the PSLang policy, and the PoET runtime itself. At runtime, no access is allowed from the target program code to any state named with this prefix—whether the access comes directly from loaded bytecode instructions or through invocation of reflection APIs. This restriction is sufficient to protect IRM integrity because, as described above, state that cannot be given such a prefix (like the stack and registers of methods) is already protected by JVML semantics.

PoET implements the above name-based access restrictions through a set of PSLang security policies instead of hard-coding this functionality into the rewriting process. Two of these policies are given in Appendices B.1 and B.2. During rewriting, these PSLang policies are prepended to any other PSLang security policy to be enforced; that is, they are used recursively by the PoET toolkit in a manner analogous to the enforcement of SFI guarantees in x86 SASI (discussed on page 31). By using explicit PSLang policies, IRM integrity enforcement itself benefits from the increased trustworthiness and flexibility of isolated and complete security policy

[11] Currently, in PoET, this prefix is pre-determined and fixed, which does not scale to realistic deployments. A full solution could use random prefixes, or hashes of the IRM enforcement code, in combination with the identity of those that perform rewriting (discussed in §3.5).

```
USES LIBRARY JVML;
USES LIBRARY Set;
GLOBAL SECURITY STATE {
    Object usedCategories = Set.create();
    Object seenCompanies = Set.create();
}
SIDE-EFFECT-FREE FUNCTION Object getCategory(Object companyName) {
    if( JVML.strEq(companyName, "IBM") )   { return "COMPUTERS"; }
    if( JVML.strEq(companyName, "Apple") ) { return "COMPUTERS"; }
    if( JVML.strEq(companyName, "GM") )    { return "CARS"; }
    if( JVML.strEq(companyName, "BMW") )   { return "CARS"; }
    return null;  // should never happen
}
ON EVENT at start of method
WITH Event.methodPrototypeIs("void accessCompany(java.lang.String)")
PERFORM SECURITY UPDATE {
    Object company = State.methodGetObject( "$methodArg1" );
    Object category = getCategory( company );
    if( Set.has(usedCategories, category)
        && !Set.has(seenCompanies, company) )
    {
        FAIL[ JVML.strCat6("Can't access new company ",company,
                  ": category ",category," is already used." ];
    }
    Set.put( seenCompanies, company );
    Set.put( usedCategories, category );
}
```

Example 3.2: PSLang security policy for Chinese Wall.

specification.

3.2.4 Implementing Chinese Wall

Example 3.2 shows PSLang for the Chinese Wall security policy [BN89], previously discussed in §2.4. As the figure shows, PSLang allows a straightforward specification of Chinese Wall in the spirit of the pseudo-code of Example 2.4.

Example 3.2 identifies, using a PSLang security event condition, the start of the method `accessCompany` as the single place in the target program that forms a

security-relevant action for the Chinese Wall security policy. The policy's security update maintains security state on used categories and seen companies and also performs a validity check: accesses to the unseen companies in previously used categories are deemed invalid. The overhead of enforcing this policy is likely to be small, because partial evaluation of the PSLang event results in only one insertion point for IRM enforcement code, namely the beginning of the company access method.

The security policy of Example 3.2 is in many ways simplistic: it has a simple fixed notion of categories, companies and their access, it assumes that only a single "user" is responsible for all activity in the application, and it makes no attempt to update the security state indivisibly in presence of multiple threads. These shortcomings are easily remedied without significantly complicating the policy's PSLang representation. By using policy extensions, thread-safe PSLang primitives for company access and required security state are easily constructed. PSLang extensions and libraries can also associate policy enforcement with whatever notion the target application or its platform has of users (e.g., login session, thread group, etc.).

Finally, instead of a fixed mapping of strings representing companies and categories, the PSLang specification can simply use the target application's own abstractions for those concepts. And, unlike the strategies proposed for SASI in §2.4, PSLang directly supports this refinement without requiring the PoET rewriter to be modified or policy-specific changes to be made to PSLang.

3.3 Security Event Synthesis

This chapter has argued that relying on security event synthesis (i.e., inferring from observations of low-level actions the higher-level security events that are relevant to the policy being enforced) reduces IRM trustworthiness and therefore, when possible, IRMs should build on high-assurance static analysis like VCC. Security event synthesis can never be completely avoided, however: to do so would require generic methods, like static verification of type safety, to exactly capture the security-relevant actions and higher-level security events for all possible security policies. Therefore, it is not surprising that even when relying on the guarantees of the JVML verifier PoET is unable to identify many potential security events in JVML target programs, making it necessary to synthesize those events in PSLang policies. As this section shows, even seemingly fundamental language-based actions and events, not specific to any particular application or policy, may need to be synthesized.

3.3.1 Exposing Security-relevant Actions

First, to enforce certain security policies, an IRM might need to mediate low-level actions that would not otherwise occur in the execution of the original application. For instance, a JVML target program may not specify a class initialization method for a given class, providing no place, or IRM insertion point, for the low-level action that occurs when that class is loaded and initialized. Such hidden actions must be exposed by PoET so they can be referred to in PSLang security policy specifications.

Each security-relevant action without any IRM insertion point in the JVML

target program is exposed by PoET in a simple manner: the rewriter adds to the target program JVML code that corresponds to the handlers and methods where those actions occur. Typically, when such handlers are optional in JVML, the addition of an "empty" handler routine, or program fragment, is guaranteed not to alter JVML program semantics. Exposing these security-relevant actions does not unnecessarily complicate PoET, since the rewriter must already discover their insertion points in the target program and, when absent, "empty" handlers are easily created.

A more difficult case, for PoET, is the misidentification of security-relevant actions in the target application—that is, when PoET identifies the same low-level action to PSLang policies in two or more different manners. This puts the onus on PSLang policies to consider alternative means of effecting actions. Such alternative means do exist, not just hypothetically like the reading of files as font bitmaps, mentioned on page 61, but also in practice, as shown by the Java `java.lang.reflect` reflection library. In fact, all JVML code outside the scope of the PoET rewriter might encode such alternatives, and such external code might include anything from the entire Java standard class libraries, the operating system, or rare callbacks from the native windowing system.

The use of JVML reflection forms an example of how alternative forms of security-relevant actions can be exposed to and dealt with by PSLang policies. The PSLang policy of Appendix B.3, which is excerpted in Example 3.3, overcomes the fact that PoET does not identify activity resulting from use of the reflection APIs in the same way as other actions. (See page 130 of Appendix A for a description of the PoET actions that can be used in PSLang policies.)

Of course, the security policy excerpted in Example 3.3 might be too strict for

```
...
SIDE-EFFECT-FREE FUNCTION boolean isBlocked(Object string) {
    if(JVML.strStartsWith(string,"java/lang/reflect/")) {
        return true;
    }
    ...
    return false;
}
EVENT at start of loading class initialization
WITH isBlocked( Reflect.className(Event.class()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot declare class in package" ];
}
EVENT at start of loading instruction
WITH Event.instructionIs("new")
     && isBlocked( Reflect.instrClassName(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot allocate object in package" ];
}
...
```

Example 3.3: Part of a PSLang policy precluding use of a Java package.

some purposes, because it immediately terminates any target program that starts to reflect. A more permissive policy could be formed along the lines of the PSLang of Appendix B.2, which checks the arguments to the reflection APIs at runtime and restricts the target program to only accessing its own abstractions. At the cost of some added specification complexity, functions could factor out the PSLang code for event-condition and security updates that are common to both PoET low-level actions and JVML reflection events. Such a convention for writing PSLang would be a simple way to ensure policy enforcement was agnostic to whether low-level actions originate through reflection or not.[12]

[12]At first glance, this style of writing PSLang would appear to result in very high enforcement overheads, since it involves runtime checks. This is not the case, because PoET simplification through partial evaluation of PSLang policies would eliminate runtime checks except on reflection events, where they add negligible extra overhead.

Reflection is just one example where details of a given JVM implementation or a PoET installation might significantly impact the semantics of IRM enforcement. A PSLang policy written for early versions of Java (which did not include reflection capabilities) might offer no protection against a knowledgeable attacker, if run on modern JVMs. In many cases, like reflection, this problem can be eliminated by applying the principle of *fail-safe defaults* articulated in [SS75].

However, adopting fail-safe defaults does not excuse policy writers from the burden of carefully considering how both current and future implementations of permitted interfaces might inadvertently invalidate their assumptions. For instance, a callback mechanism in an apparently irrelevant component, like a native-code implementation of a windowing system, can easily subvert the enforcement of a PSLang policy that restricts each call-site of a certain method. Even more important, the semantics of a particular PSLang policy could depend on details of the PoET rewriter, such as how PoET sees the Java standard libraries and how it interacts with JVML ClassLoaders [Gon99]. PoET, currently, can only rewrite the Java libraries if run in *static mode* (see Appendix A.2)—otherwise, it will only rewrite code in the target program.

3.3.2 Building a Better Event

In addition to synthesizing security events that correspond to alternate or hidden security-relevant actions, IRM security policies must sometimes also bear the burden of defining events that perhaps should be identified by the IRM toolkit as low-level actions. For certain security-relevant activity, such as the end of an application's execution or the invocation of a function, there may not be any particular low-level action whose occurrence should trigger IRM enforcement code. This am-

biguity results from a tension between what constitutes activity relevant to security policies and what semantics are provided by the ISA and the runtime platform. In particular, there may be such a semantic gap between the ISA and the security policy's view of certain language-based concepts, such as program modules.

The start of the execution of a JVML target program illustrates this category of security event synthesis of what might appear to be fundamental low-level actions. Most JVML programs start their execution through a static `main` function in some class. However, this distinguished class must first be properly initialized, which might involve executing target program code. As a result of initializing the class, other program classes might be initialized, arbitrary methods might be called, and any object instantiated. In fact, such initialization activity can comprise the entire execution of a JVML program and—if this activity terminates by throwing an uncaught exception—the `main` method might never be invoked. Of course, a writer of PSLang security policies might believe the start of the program to mean an even-earlier point, before the initialization of the JVM runtime and the Java standard libraries. Thus, policy writers may be disappointed with the PoET action `at start of program`, which occurs only at the beginning of the aforementioned `main` method.

It might seem best if PoET marked the start of executing a JVML target program on a JVM using some well-defined PoET action. Although convenient, this would only provide one fixed resolution of the above ambiguities—and the exact semantics might remain either unclear or unsatisfying to policy writers.[13] These dis-

[13]Even so, for some low-level actions, including the start of execution of a JVML program, the definition of such a PoET action is perfectly feasible. Most JVMs come with standard profiling and debugging interfaces that allow mediating such activity. For the reasons described above, these low-level actions are not exposed by PoET.

```
IMPORT LIBRARY Lock;
GLOBAL SECURITY STATE {
    boolean programNotBegunFlag = true;
    Object beginProgramFlagLock = Lock.create();
}
EVENT at start of class initialization
PERFORM SECURITY UPDATE {
    Lock.acquire(beginProgramFlagLock);
    if( programNotBegunFlag ) {
        programNotBegunFlag = false;
        globalProgramStartFunction( Event.class() );
    }
    Lock.release(beginProgramFlagLock);
}
```

Example 3.4: PSLang synthesis for security event at start of target program.

advantages are avoided by defining the target program's beginning through PSLang security event synthesis, as illustrated by the PSLang policy in Example 3.4. That policy defines the program's beginning at the occurrence of the first initialization of a class that has been rewritten by PoET. Such a synthesis might incur extra performance overhead and lengthen PSLang policies, but, at the same time, it provides to policy writers a concise, reliable, and exact specification, with transparent semantics, of this important security event.

In PSLang, a number of other fundamental language-based security events must be synthesized in this manner. A naive definition of the target program termination, like that of the PoET action `at end of program`, might be taken to mean the end of the `main` method. In a more realistic specification the PSLang policy might mediate the use of threads and decide the target program has terminated when its last thread has done so. The synthesis of this security event could be further refined to include system activity, such as callbacks, that may happen even after all program threads have terminated, by mediating all registration for such future actions.

Garbage collection and finalization of JVML objects is one example where—after the target program might be expected to have terminated—program code might still be executed. The PoET action `object instance garbage collection` corresponds to the JVM execution of an object's `finalize` method, which is called exactly once before that object is garbage collected. The `finalize` method poses a challenge to PSLang event synthesis because, just like class initializers, it can encapsulate a target program's entire execution. More troublesome, however, is that the invocation of an object's finalizer does not guarantee that the object won't be used further, as a new "live" reference to the object instance might result from execution of the `finalize` method. JVML provides no mechanism to detect such re-registration of object instances and, even worse, when it occurs, that object's `finalize` method will not be invoked again before garbage collection. To resolve these ambiguities and allow the trustworthy synthesis of garbage collection and program termination events, a PSLang policy can limit finalization activity to a reasonable subset, such as one that does not create new threads or re-register the object instance variable. Such restriction can be imposed without runtime penalty by load-time PoET security updates that effect a static analysis of `finalize` methods.

However, the single most important language-based low-level action that must be synthesized in PSLang policies is probably JVML virtual method invocation. By the definition of a virtual method, JVML only refers to such methods by name and determines the actual method code to execute from the type of an object instance provided at runtime. This makes synthesizing a security event that removes the ambiguity caused by this indirection both challenging and important, because security policies are concerned not with names, but with execution behavior.

Table 3.1: Virtual method map for a `Socket` with a `write` method.

Class	Superclass	Class of Virtual Method Implementation
`Object`	—	`(toString,Object), ...`
`Writer`	`Object`	`(write,`*abstract*`), ...`
`Socket`	`Writer`	`(write,Socket), (toString,Socket) ...`
`Bar`	`Socket`	`(write,Socket), (toString,Bar), ...`

One way of resolving this ambiguity is to insert IRM enforcement code in the bodies of JVML methods, since this ties the IRM to the method's execution regardless of what name it used to invoke it. The example PSLang policies of this dissertation are typically written in this manner. In some cases, however, PoET may be unable to rewrite the actual code of JVML object methods, for example, because that code is inaccessible or outside the scope of the rewriter, or the code is shared by activity that is subject to different security policies.

A PSLang policy can attempt to resolve what method will be executed at runtime by examining the constant class name given as an operand to the JVML virtual method invocation instruction. However, relying on this call-site class name is challenging for two distinct reasons. First, the method executed may be that of a subclass of the call-site class: during execution the runtime object present at the call site can either be an instance of the call-site class or an instance of any of its subclasses. Second, the method executed may be that of a superclass of the call-site class: even if the runtime object is an instance of the call-site class the call-site class may not implement the method named—and in this case a superclass's implementation is executed. Thus, the call-site class name only constrains to a limited extent what method implementation will be invoked.

Instead of the static call-site class name, designers of PSLang policies can always

use a runtime check to determine, for each instruction that invokes a method the class of the provided object instance.[14] Table 3.1 shows an example where, if writing to sockets is to be precluded, such runtime checking might be needed. In the table, an abstract superclass `Writer` defines a prototype for the `write` method, which `Socket` then implements. JVML instructions that invoke the `Writer/write` method might, therefore, write to a socket when executed.

The use of virtual methods is a common pattern in object-oriented languages, and runtime checks like the above might result in significant enforcement overhead. This runtime overhead can sometimes be avoided by careful construction of PSLang policies. In the example of Table 3.1, for instance, no runtime check of `write` methods is needed if `Socket` is the only `Writer` subclass used by the target program. Such simple static analysis can be implemented using PSLang load-time security updates, and is particularly useful for the second type of virtual method ambiguity described earlier: when superclass method's code is executed at runtime even though the invocation instruction specified the name of a subclass.

The last row of Table 3.1 illustrates this second type of ambiguity. In this case, the `Socket` class, whose `write` method must be restricted, has been extended by a class `Bar`. A JVML target program with this class hierarchy will actually invoke `Socket`'s `write` method when executing a JVML `invokevirtual` instruction that refers to `Bar/write`.

The PSLang security policy of Appendix B.4 resolves this ambiguity without runtime performance penalty by maintaining for all loaded classes (in the order

[14] JVML, unlike some object-oriented languages, ignores runtime types of method arguments in the selection of what method implementation to execute on virtual method invocations. Therefore, without loss of generality, this dissertation sometimes uses JVML method names without specifying the argument type definitions.

```
EXTEND POLICY synthProgStart;     // From Example 3.4
EXTEND POLICY virtualMethodMap;   // Defined in Appendix B.4

FUNCTION void globalProgramStartFunction(Object class) {
    restrictVirtualMethod("java/net/Socket", "getOutputStream");
}
EVENT at start of loading instruction
WITH virtualCallTo(Event.instruction(),
                   "java/net/Socket", "getOutputStream")
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot retrieve output part of TCP socket" ];
}
```

Example 3.5: PSLang policy restricting disambiguated virtual method calls.

of their resolution, as specified by JVML semantics), a data structure like that of Table 3.1. Example 3.5 shows how a PSLang specification can use the policy of Appendix B.4 as a building block in restricting use of sensitive methods.[15]

This section has demonstrated, through examples in the context of JVML and PSLang, that IRM policies might need to synthesize even the most common security events, often using non-trivial data structures and computation. This may appear disappointing, as building on VCC guarantees held the promise of alleviating the need for such synthesis. Such disappointment is misplaced, since the security events discussed in this section cannot be exposed as low-level actions on the JVML platform and, therefore, some synthesis would always be necessary. Encoding security event synthesis in IRM policy specifications makes transparent the intended and actual semantics, and is preferable to alternatives, such as fixing one specific notion of these security events into the IRM rewriter.[16]

[15]In security policies that restrict network communication, the `Socket` class and `write` method of this section might correspond to the `java.net.Socket` class in the standard Java libraries and to its `getOutputStream` method.

[16]This said, if a strong advantage such as reliability can be gained, an IRM toolkit for a particular platform could always register for external notification of hard-to-synthesize security events and expose them as low-level actions. In the

3.4 Writing Good IRM Security Policies

IRM security policies encode both policy and implementation—that is, both the intent of security enforcement as well as the specifics of how that intent is accomplished. An IRM toolkit, like PoET, can therefore be thought of as a policy-free rewriting mechanism that puts the entire burden of security enforcement onto the writers of security policy specifications. As the PSLang examples of the previous section illustrate, this burden can be lightened through the synthesis of higher-level security events, which can be opaque and simple to use in further policy extensions.

Even so, the writing of good IRM security policies is likely to remain a difficult task, just like the creation of any infrastructure software. Security enforcement mechanisms are often those system components that are hardest to implement correctly, because they are sensitive to (even hidden) implementation details and may be subjected to non-independent faults during an attack. To help alleviate this, PSLang allows security policy specifications to make use of many of the methodologies that are known to facilitate the creation of correct software. PSLang can structure security policies as pre- and post-conditions on security events [Sch97], as abstract state machines [BS01], directly as security automata, like in SAL, or through most other imperative styles of programming. As this thesis argues, the difficulty and complexity of creating security mechanisms is best addressed with a well-written out-of-band IRM specification that fully specifies all enforcement details.

The difficulty of hidden implementation details has already been mentioned in §3.3.1, which discussed how to synthesize security events that expose security-

---

case of PoET, for example, a JVM debug handler could be registered to enable mediation of program termination events without restricting the set of allowed input programs.

relevant actions. Unfortunately, the set of IRM low-level actions is likely to be incomplete in a way that makes it impossible for IRMs to mediate certain security-relevant actions. For instance, an IRM toolkit intended for securing applications is unlikely to expose the low-level actions that correspond to the operating system kernel forwarding network packets. In this example, it is even unlikely that an IRM security policy would be able to synthesize a security event corresponding to this forwarding activity.

The above failure demonstrates how IRMs may not be able to implement all security policies that are currently enforced by traditional reference monitors. Kernel-based reference monitors may have the advantage of being able to mediate internal operating system abstractions, such as packet forwarding by the network stack. As a result of such limitations, IRMs may sometimes be unable to enforce important security policies. In those scenarios, the IRM toolkit may have to be extended to mediate internals of the execution platform, e.g., by rewriting or interpreting the operating system kernel.[17] Yet, despite any such extensions, the underlying IRM toolkit will always be subject to some limitations that must be taken into consideration during the specification of IRM security policies.

Often the most convenient strategy is to write a more restrictive security policy that completely eliminates the troublesome activity, when faced with the task of writing IRM specifications that must resolve some difficult issue (like low-level actions hidden from the IRM toolkit). This is the approach chosen in the policy for dealing with JVML reflection excerpted in Example 3.3; that policy allows no reference to the JVML reflection libraries, even though many of their uses (such

[17] This is feasible in practice, even in commercial operating systems. The profiler described in [BEL+00] interpreted all of the executed code in a running system, apart from a handful of critical interrupt handlers.

```
USES LIBRARY JVML;
SIDE-EFFECT-FREE FUNCTION boolean isStdIn(Object s) {
   return JVML.strEq(s,"java/lang/Runtime/inLjava/io/InputStream;");
}
FUNCTION stdInInvariant( Object newInputStream );

ON EVENT at start of instruction
WITH Event.instructionIs("putstatic")
  && isStdIn( Reflect.instrRefStr(Event.instruction()) )
PERFORM SECURITY UPDATE {
   Object proposedInputStream = State.methodGetObject("$instrArg1");
   if( ! stdInInvariant(proposedInputStream) ) {
       FAIL[ "Illegal modification of standard input stream" ];
   }
}
```

Example 3.6: PSLang security event synthesis for the modification of stdin.

as creating assertion error messages) are of little security concern.

This same draconian tactic could be used for the problem of forwarded network packets: the IRM could, for example, disable packet forwarding, or (if unable to be so selective) simply turn off all processing by the network stack. A well-written IRM specification should not resort to such restrictive measures, because they unnecessarily preclude execution of applications that would never violate the intended security policy. Rather, IRMs should allow the greatest number of legitimate target applications to execute, perhaps at a cost of increased enforcement overhead when they exercise boundary cases. In the case of JVML reflection, Appendix B.2 gives a PSLang specification for such a forgiving security policy.

In fact, the greater flexibility of IRMs should allow a larger set of target applications to be executed in a secure fashion: an IRM can easily mediate at runtime many security-relevant actions that previous enforcement mechanisms cannot, and have therefore statically prohibited. Example 3.6 shows one PSLang policy that

leverages the flexibility of IRMs to mediate direct access to a global variable.[18] Allowing such direct modifications of global state allows target applications of both higher performance and greater expressiveness since system state has traditionally only been accessible through restricting APIs. IRM toolkits in general, and PoET in particular, provide full support for enforcing security policies on any state, even in face of arbitrary direct access to that state. IRM specifications should, therefore, not allow such activity in applications while ensuring it complies with the intended security policy.

If an invariant must hold for the contents of multiple states simultaneously, instead of just for a single variable, like in Example 3.6, security policies must consider concurrency and synchronization. If all interleaving of IRM enforcement code is allowed then a multi-threaded security application can compromise the security state by using a "Time of Check to Time of Use" attack [BH78, BD96]. For instance, this attack would be possible in the security policy of Example 3.1 if it had been written without using locks (in the way that Example 3.2 uses no locks). Then, the secured application could conspire to have multiple threads concurrently evaluate the security event condition in one state (e.g., `openWindows` = 2, when about to open a window) and, consequently, these threads would allow the mediated activity and incorrectly update the security state (e.g., so `openWindows` = 3).

PSLang security policy specifications must make use of some locking discipline to ensure that consistency is maintained for both the IRM security state as well as the target program state. PSLang supports synchronization through the `Lock`

[18] This example is drawn from an early vulnerability reported for Java, where the globally-accessible field `System.in` was changed through direct access. Later Java versions were modified to prohibit all direct modifications of this field, causing incompatibility with many applications.

```
IMPORT LIBRARY Lock;
GLOBAL SECURITY STATE {
    boolean programNotBegunFlag = true;
    Object beginProgramFlagLock = Lock.create();
}
EVENT at start of class initialization
PERFORM SECURITY UPDATE {
    if( programNotBegunFlag ) {
        Lock.acquire(beginProgramFlagLock);
        if( programNotBegunFlag ) {
            programNotBegunFlag = false;
            globalProgramStartFunction( Event.class() );
        }
        Lock.release(beginProgramFlagLock);
    }
}
```

Example 3.7: Incorrect PSLang synthesis of target program beginning.

ADT library, which builds upon the fundamental JVML `monitorenter` and `monitorexit` operations [YL96]. Maintaining consistency is an arduous task in PSLang, as in any concurrent programming language [Bir89]. Without exercising extreme caution, it is easy to introduce race conditions, cause deadlock, or simply write an incorrect specification. In Example 3.7, for instance, the policy writer's attempt at efficiency causes an insidious race condition. (described in [Bir03]). Even in the correct version of this policy, in Example 3.4, the `globalProgramStartFunction` must maintain the partial order of locks in the target program (e.g., by using only locks exclusive to the IRM and this function) or risk runtime deadlock.

The "no network sending after file reading" security policy, discussed in Chapter 2, further illustrates the dangers of concurrency. Without use of synchronization, a secured application subject to this security policy might interleave the reading of a file between the execution of a network send operation and the IRM enforcement code that precedes that operation. A PSLang specification can encode the security policy of Figure 2.1 in several ways that remove this race condition. For

```
...
EVENT at start of instruction
WITH virtualCallTo( Event.instruction(),
                    "java/net/URL", "openConnection" )
PERFORM SECURITY UPDATE {
    Object str = JVML.toStr(State.methodGetObject("$instance"));
    if( notAcceptableNetworkURL( str ) ) {
        FAIL[ "Attempt to access proscribed network URL" ];
    }
    Object url = JVML.newObjectInstance( "java/net/URL" );
    JVML.objectInvokeMethod( "java/net/URL", "<init>",
                             "(Ljava/lang/String;)V", url, str );
    State.methodSetObject( url, "$instance" );
}
...
```

Example 3.8: Outline of PSLang resolution of `java.net.URL` race conditions.

instance, PSLang security updates could acquire a specific lock before all reading and sending operations, and only release that lock after those operations complete.

Sometimes the use of IRM locks is neither necessary nor sufficient to achieve integrity of a concurrent secured application. In JVML and Java, for example, the file system and network APIs typically operate on immutable object instances with the properties of capabilities [WN79, Gon99]. Thus, a PSLang policy that wants to restrict file access through `java.io.File` needs to consider only the immutable string passed as the file name argument when creating such object instances. Such immutable arguments cannot be changed (e.g., to refer to another string) because they reside on the JVML stack and are therefore inaccessible to all other threads. This type of immutable thread-local arguments often gives stronger guarantees than use of the `Lock` ADT library, making it easy to specify many PSLang policies, in spite of potential concurrency.

Unfortunately, not all potentially dangerous APIs in the Java libraries accept only immutable arguments. For instance, in the `java.lang.Runtime` library, one

of the `exec` methods that spawns execution of arbitrary programs takes its arguments as an array of strings—and, in this case, the array might be modified after its observation by IRM enforcement code and before its use in the `exec` method implementation [Gar03]. However, often such potential race conditions can be converted to use immutable arguments by making a copy of the arguments onto the stack, and only considering those copies in IRM enforcement code. Because the only reference to the copied arguments resides on the thread-local stack, JVML semantics guarantee their inaccessibility to other threads. Example 3.8 shows part of a PSLang security policy that could remove such a race condition in use of `java.net.URL` objects, which are not immutable. In the example, the `openConnection` method is performed only after the URL object instance it uses has been replaced by a thread-local copy.

Locking, or some other synchronization, is especially important when multiple IRMs cooperate to enforce a single security policy. As discussed in §2.4, security policies can, in general, be specified either as a conjunction of many IRMs, each with their own security state, or as a single IRM that updates centralized security state. Every PSLang specification, in particular, can be thought of as a conjunction of many separate security updates, each of which has its own local security state. In PSLang, the IRM security state may be even more decentralized, because it may be embedded within the JVML classes and object instances of the secured application. Eventually, however, these distributed IRMs must somehow coordinate with each other to enforce their intended security policy—just as the policy at the end of §3.2.1, which sets a global threshold on file reading, must at some point aggregate the local counters embedded in file objects.

Unlike traditional reference monitors—which can centrally maintain the secu-

rity state for all enforcement activity in the operating system kernel—an IRM toolkit is likely to require each secured application to use a separate IRM with its own distinct security state. In PSLang/PoET, IRMs must cooperate in this manner if a single security policy is to be enforced on all secured applications, even if they are all running on the same JVM. To coordinate with other IRMs, PSLang security policies can communicate through the primitives of the underlying system (such as shared memory or network channels), as long as the integrity of those primitives is also protected in the same PSLang specification. More generally, a single global security policy may be concurrently enforced through any composition of IRMs in different secured applications or on different systems, as long as consistency is somehow ensured, e.g., using the standard methods of distributed systems [Sch90, Lam01, Lyn96].

Whenever the same target application is simultaneously subject to multiple security policies, the writers of PSLang specifications must carefully consider their composition. Ascertaining the behavior of such composed systems, or policies, is known to be a difficult task with no simple solution [Hin98, BLW02a]. The IRM approach introduces an additional composition problem: How to ensure correct security enforcement when the same target application is rewritten multiple times sequentially in succession, perhaps with different IRM toolkits. In this case IRM enforcement code will be rewritten to include other IRM enforcement code, which intuitively seems likely to cause inconsistency.

Successive rewriting of different IRM security policies is not equivalent to the concurrent case and may not always give the expected result. To see this, consider two PSLang/PoET IRMs A and B that both restrict some system APIs, and also protect their own integrity, e.g., by limiting use of JVML reflection and reducing

the target program's privileges. Now, assume a JVML target program is rewritten with A, to form a secured program, and that secured program is then rewritten with B. Then, the IRM enforcement code of B may truncate the execution of the final secured program as soon as the IRM enforcement code of A makes use of JVML reflection or uses a privilege forbidden by B (such as instantiating a JVML ClassLoader for IRM rewriting). This early termination is not a fault of the target program, but results because the two security policies were applied sequentially—since, if an IRM toolkit had simultaneously rewritten with both A and B, the secured application might always execute to completion.

One solution to the above IRM composition problem might be to identify the IRM enforcement code embedded within a secured application. Then, this IRM enforcement code could be excluded from further rewriting by other IRM toolkits. However, for such a solution to work, trust would have to be established between the two different IRM toolkits and security policies. This brings up the issue of deployment and management of implementation of the IRM approach, a frequent stumbling block for security technologies.

3.5 Deploying the IRM Approach

The level of assurance provided by a computer security enforcement mechanism can sometimes be increased by simple changes to its deployment and management, i.e., how, and at what time and place, security is actually enforced. Network firewalls are a good example, as their primary advantage as a security mechanism is their ability to centralize enforcement to a single point. Such centralization offers no technological advantage over distributing security enforcement over all network nodes, yet has immense practical value, since it makes efficient use of

management resources and easily allows a single policy to be consistently enforced across a whole network [CBR03]. The cryptographic certification of software code is another example where a shift in security enforcement brings benefits—namely the potential of moving the onus of examining software for errors (or malicious intent) from the software consumers to its producers or vendors by leveraging existing trust relationships [Gon99]. Because of their deployment and management advantages the above technologies are now ubiquitous, despite having shortcomings that limit their value [CER00].

The IRM approach offers great flexibility in the timing and placement of security enforcement and how that enforcement leverages existing trust relationships—an advantage that IRMs share with some variants of VCC [SGB+98, Nec98]. Applications can be rewritten to include an embedded IRM by the application developers, vendors, distributors, administrators, or end users; IRM rewriting can take place dynamically on demand, or statically ahead-of-time, and the IRM can either be an explicitly identified or integral part of the resulting secured application. This flexibility allows IRMs to reap many benefits, such as, in corporate settings, the centralized application of security policies with an IRM toolkit operated by network administrators. In that scenario, all applications that originate from the network (e.g., through email or web browsing) could be transparently rewritten to include IRMs that enforced a corporate policy during their execution.

Centralized IRM deployment has several benefits, but other distinct advantages can come from leaving IRM rewriting to the discretion of the end user.[19] One such advantage is the potential for increased efficiency with lower security enforcement

[19] Such end-user rewriting is easily accomplished, since each secured application typically contains both an IRM rewriter and a specification of the intended security policy in order to deal with potential runtime code generation.

overheads: Because each secured application may include a different IRM, the end user can use simple IRMs, or no enforcement, for those applications that do not handle sensitive data or where other external circumstances dictate that little security enforcement is needed. The problem of composing IRM security policies gives another reason to postpone IRM rewriting of applications. Trusting the end user (or the end user's operating system) to perform IRM rewriting allows the conjunction of all intended security policies to be independently enforced on the original target application and, thus, avoids the problem of nested rewriting mentioned in the last section.

One scheme that combines many of the above benefits is to perform early and centralized IRM rewriting but make explicit and certify all IRM enforcement code embedded in the target application. Then—if he has the privilege to do so—an end user could recover the original target application (by removing the inserted IRM enforcement code), retain those IRMs that are certified by parties he trusts, verify other IRMs (by rewriting them again from their PSLang policy), or the end user could use his own IRM toolkit to enforce further security policies. A PSLang/PoET implementation of this scheme could identify the inserted IRM bytecode using the extended method attributes of JVML class files, and certify this inserted code, as well as the PSLang specification, using Java's code signing infrastructure [YL96, Gon99].

Another alternative would be to separate the verification of IRM processing from actual IRM rewriting, i.e., in the manner of VCC, separating the generation of a rewriting proof from its validation [Nec97]. When compared to cryptographic signatures, such a validation scheme is preferable since it does not rely on any external trust relationships. As mentioned in §3.1.2, the process of IRM rewrit-

ing can generate evidence that discharges a verification condition generated from the security policy—whether or not any IRM enforcement code is inserted. As in [Nec00], the simplification and elimination of inserted IRM code, if sound, can typically create evidence that proves its correctness. In the case of declarative policies, like the SAL security automata of Chapter 2, target program modification might even be proved unnecessary or one policy proved contained in another [BR02].

The adoption of some standard IRM deployment scheme, like the above, would have a major benefit in the increased end-to-end transparency of software semantics [SRC84]. Currently, it is often difficult to determine what security policy is being enforced during execution, and, as a result, not only does lax security go unnoticed but sometimes its enforcement is purposefully discontinued in an attempt to resolve unrelated problems. If software vendors and administrators embedded identifiable IRM security policies within applications there would be no such ambiguity about what security policy was currently being enforced: a simple disassembly would reveal exactly the security policy applied to their execution.

By using the IRM approach, software developers—or other interested parties that know the details of platform and application API semantics—can produce and publish multiple IRM security policies. When combined with the above deployment scenarios, this could enable security administrators to choose different security policies for each occasion, and yet make the currently enforced security policy be transparent to the end users. This style of decoupled IRMs has the potential to greatly simplify the management of security updates, since the late binding of security enforcement to the time of end-user execution provides a point of indirection where security policies can be patched or updated to a newer version. Thus, IRMs could help ease the problem of security configuration management, one

of the current challenges of computer security.

Adopting IRM enforcement does not come without its costs in terms of complexity and other overhead. For example, as discussed in §3.3.2, synthesis of security events as simple as virtual method calls can sometimes be both difficult and inefficient. The simple solution, for this particular security event, may spend significant resources by using multiple copies of the Java libraries, each rewritten with a different IRM. The possible nesting of IRMs mentioned earlier can also cause complexity and waste of resources, as multiple ClassLoaders, each originating in a separate IRM toolkit, form a chain of IRM rewriting at runtime. Such problems would have to be resolved, e.g., by using the identity of certified IRMs to negotiate a single rewriter to use at runtime [MHS00].

Despite such difficulties, deployment and management of the IRM security enforcement is certainly preferable to one current alternative: the development of a distinct built-in security mechanism for each new software application. If each application comes with its own notion of security and its own enforcement mechanism then it becomes intractable to establish a coherent system security policy or even to determine what policy is currently being enforced. Numerous applications, e.g., the Netscape web browser [Ros96], have adopted this approach and implement complicated mechanisms that enforce security policies. However, even if those applications allow some tuning of their security policies, and correctly enforce those policies, it is very difficult to manage and reconcile the abstractions of such multiple disparate mechanisms. Finally, as briefly discussed in §1.1, the nature of computer security and software application development gives little hope that such application security enforcement would be either sound or complete.

3.6 Extending the Scope of IRMs

So far, IRMs have been presented according to their definition in Chapter 1, which—in addition to satisfying the general requirements for reference monitors—explicitly prohibited IRMs from changing the function or behavior of their target application. This restriction, labeled (d) in Figure 1.4, is attractive because it not only forms a reasonable constraint on IRMs, circumscribing their task, but it also justifies their specification and analysis as security automata by placing IRMs in the class EM of enforceable security properties.

Experience, however, shows that this restriction is both quite difficult to achieve and also that it prevents the enforcement of a number of interesting security policies well suited to IRMs. Example 3.8 illustrates the difficulty of fulfilling this obligation: security enforcement requires the removal of certain race conditions, but—in doing so—non-deterministic behavior is eliminated and the secured application may not behave the same as the original target application. Thus, the IRM embedded in a secured application unavoidably changes some properties of that application. At the same time by forbidding changes to application semantics, IRMs are prevented from performing many tasks, such as the following:

- Creation of audit logs and traces of target application behavior, either using side channels or existing public APIs. With IRMs, it can be simple to implement coherent system-wide audit policies and record use of otherwise opaque application-level interfaces.

- Preventing resource completion and denial-of-service attacks by limiting the consumption rate of some resources, such as memory, computation cycles, and disk and network bandwidth [CvE98]. For those resources where this is

not sufficient, such as locked data structures, IRMs could potentially perform simple transactional rollback on timeouts [SESS96, HF03].

- Implementing security policies with alternate remedial actions that do not truncate application execution. Such security policies might inform the application when IRM validity checks fail, for instance by returning an innocuous error status, by throwing a security exception for the application to catch, or by reverting to some previous state [Pov99].

- Virtualizing the application's execution environment by modifying each access in the secured application to refer to alternate state. IRMs can easily perform such virtualization, which is known to provide strong isolation and security guarantees [KF91].

- Sandboxing the behavior of the application by silently eliding or otherwise changing the semantics of certain activity. Such sandboxing is used by some security mechanisms, for instance, by the MiSFIT SFI implementation discussed in §2.3.1.

There are a number of possible alternatives to the notion of application semantic equivalence that is specified in Figure 1.4(d), and some of those alternatives can enable IRMs to perform a number of the above tasks. Intuitively, such alternative IRM definitions must allow for more non-determinism and reasonable perturbations in the application's execution environment—permitting activity that should not be visible to the target application as well as changes that should not be under its control. One attempt at formalizing such IRM requirements might require, for each secured application trace, that there exist a target application trace in some execution environment (including different interleaving due to concurrency),

in which the sequence of named state values in that trace maps onto the same sequence in the secured application trace. Recently, several such alternative formal definitions have been introduced [BLW02b, WZL03, BLW02a].

PSLang/PoET is already capable of performing all of the extended tasks listed above, even without the support of a formal definition. For instance, the PSLang security policy of Appendix B.2 is written to modify the semantics of the JVML reflection API to hide the inserted IRM and the PoET runtime. The current PoET implementation is only limited in that it cannot completely elide selected security-relevant actions from a JVML target program. This limitation can, however, typically be overcome by changing the arguments or return values of those actions in the secured version of the JVML program.

Another potentially useful extension to the IRM approach might add to IRMs the capability of performing static analysis of the target application. It is clear that this enhancement would greatly increase the power of IRMs, since VCC and the JVML verifier are one example of such static analysis, and the IRM approach greatly benefited by leveraging their guarantees. Again, PSLang/PoET already implements a form of this extension in its load-time security updates. This capability, and those security updates, can be crucial to the PSLang synthesis of more intricate security events, such as §3.3.2's unambiguous virtual method calls.

However, great care must be taken whenever the role of IRMs is extended beyond that discussed in the previous sections of this dissertation. For instance, if IRMs can modify application behavior, enforcing a collection of IRMs is no longer equivalent to the concurrent, but independent, enforcement of each of those IRMs—as is the case with the IRM definition of Figure 1.4 and was discussed in

Chapter 2 in the context of security automata.[20] In particular, such modification to IRM abilities and semantics can make some of their difficulties into intractable problems. For example, when sequentially composing so extended IRMs, an early IRM may be completely removed: The later IRMs may simply silently elide the installation of the IRM rewriter (in PoET, a JVML ClassLoader).

The trustworthiness of IRMs as a security enforcement mechanism is the most significant factor in any consideration of IRMs of extended scope. When weighed against this standard, some of the extensions proposed in this section appear unattractive—either because they make it more difficult to reason about the effects of IRM enforcement code or because they add too much to the trusted computing base. However, the simpler of the proposed extensions, such as logging and auditing, seem perfectly reasonable. The next chapter gives a detailed example of how the IRM approach can be used to implement a complex security policy whose remedial actions inform the application of security violations.

[20] To see this, consider two IRM security policies that both wish to prevent the occurrence of $F(1,1)$ and $F(-1,-1)$. Now, one of these policies might sandbox F operations by negating the first argument, and the other might sandbox F by negating the second argument. Then, however, the composition of these two security policies might turn $F(1,1)$ into $F(-1,-1)$, and vice versa, subverting the intent of both policies.

Chapter 4

Java Stack Inspection IRMs

Stack inspection is a recently introduced security enforcement mechanism that originated with Java. The stack inspection mechanism is designed to enforce security policies that are concerned with "confused deputy" attacks, where an unauthorized party are able to (indirectly) cause proscribed activity through interaction with a privileged entity. To avoid this, a stack inspection security policy requires that all parties responsible for a given activity have been granted suitable permissions [Gon99, FG02].

4.1 Security Enforcement in Java

Java was designed to support construction of applications that import and execute untrusted code from across a network. The language and runtime system enforce access control policies that support downloading JVML programs, *applets*, from a host computer and executing them safely on a client computer. In Sun's Java implementation [GJS96, YL96, Gon99], this access control is enforced by runtime checks in the standard Java libraries, supported by VCC guarantees of JVML bytecodes (described in §3.1.2), which are derived from the syntax and semantics of the Java programming language.

The JVM access-control policies associate access rights with the bytecode of JVML classes and, thus, regulate access based on the requesting class. The *sandbox policy* of early (pre Java 2) JVM implementations distinguished between code residing locally and code obtained from across the network. The more recent Java 2 *stack inspection policy* refines this. Hence, in Java 2, whether an access is

permitted can depend on the current nesting of method invocations. Enforcement of the stack inspection access-control policy therefore relies on information found on the JVM runtime call stack [Gon99, FG02].

Changing which access-control policy is supported by the JVM requires changing the JVM. Thus, programs expecting Java 2's stack inspection policy to be enforced will not execute correctly on earlier-generation JVM implementations. Applications requiring other access-control policies might be ruled out altogether, might require awkward constructions, or might be forced to employ their own application-level custom enforcement mechanisms.[1] Finally, a JVM that enforces the stack inspection policy includes mechanisms that may or may not be needed for executing any given Java application. For applications where memory is at a premium, such as those intended for use in embedded systems, the size of the JVM footprint is crucial; there is considerable incentive to omit unused enforcement mechanisms.

This chapter describes how IRMs can provide an alternative to enforcing access-control on runtime platforms, like the JVM, without requiring changes to the platform. To illustrate this, IRMs for Java 2 stack inspection are specified in PSLang, added to JVML programs by PoET, and then executed with the policy (now, possibly, on any JVM, even those predating Java 2). Two IRM implementations of stack inspection are discussed—one is a reformulation of security passing style proposed in [Wal99b, WF98]; the other is new and exhibits performance competitive with existing commercial JVM-resident implementations.

[1]With elaborate construction, the stack inspection mechanism can be made to support more common policies. These constructions typically involve creating multiple copies of the same class (in different code bases) or creating multiple instances of identical class loaders.

Java 2's stack inspection policy is a particularly challenging one to enforce with an IRM because state relevant to policy enforcement (the JVM runtime call stack) is not directly accessible to Java applications. That PSLang/PoET can obtain a new implementation exhibiting competitive performance reflects well on the practicality of the IRM approach. Also, because an IRM implementation makes stack inspection optional, it allows the use of simpler (or older) JVMs that can be more efficient and more trustworthy, and—in the future—when alternate means of enforcement supersede stack inspection, only a PSLang policy needs be changed.

The chapter is organized as follows. Section 4.2 reviews Java 2's stack inspection policy and the primitives that implement this policy. An IRM version of the security-passing style [Wal99b, WF98] implementation of stack inspection is described in §4.3; an IRM implementation for more lazy enforcement of Java 2's stack inspection policy is given in §4.4. Finally, §4.5 concludes with some remarks about the clarification of Java's stack inspection policy that resulted from the experience of implementing it with the IRM approach.

4.2 Review of Java Stack Inspection

Java 2's stack inspection access-control policy is based on *policy files* which associate *permissions* with *protection domains*. The policy file, which is read when the JVM starts, defines application access controls as follows.

Protection domains. Each application initially is a sequence of bytes stored outside the JVM. The bytes are fetched by a class loader and then executed by the JVM. Prior to execution, the bytes are assigned to a protection domain in accordance with the source of the bytes (a network address or a file name)

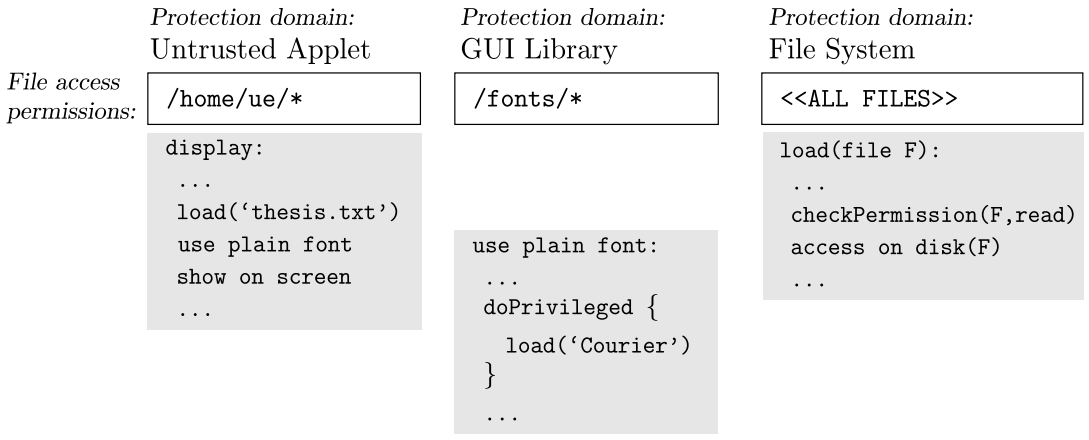


Figure 4.1: Three protection domains set up for stack inspection.

and any attached cryptographic signature.[2]

Permissions. Each protection domain implies a set of permissions. This set includes all those permissions associated with the protection domain by the policy file, as well as other implied permissions. The definition of a permission—a class—states what permissions it implies by defining an `implies` method.

As an example, Figure 4.1 depicts three protection domains: Untrusted Applet, GUI Library, and File System. Permissions associated with each domain appear in the box below the name of the protection domain; code associated with that domain appears below the permissions. Notice that file access permissions are given in the figure using patterns rather than complete file names—the `implies` method would decode those patterns to generate permissions for actual files in the expected way.

[2] The Java class loader used to fetch those bytes can also be involved in determining the protection domain of the bytecode [LB98, Gon99]. Since new class loaders can be created at runtime, protection domains can be created dynamically, thereby helping to overcome the static nature of policy files.

For a permission P, invoking the `checkPermission`(P) method of Java 2 throws a security exception if access should not be allowed to proceed; it otherwise has no visible effect. Whether a security exception is thrown depends on the protection domains assigned to the methods from which control has not yet returned—methods having frames on the JVM call stack when `checkPermission`(P) is invoked. Specifically, when `checkPermission`(P) is invoked, the JVM call stack is traversed from top to bottom (i.e., starting with the frame for the method containing the `checkPermission`(P) invocation) until either the entire stack is traversed or an invocation is found within the scope of a `doPrivileged` block. In that traversal, the stack frames encountered are checked to make sure their associated protection domains imply permission P; if some frame doesn't, a security exception is thrown.

Observe that `doPrivileged` supports a form of rights amplification. Without `doPrivileged` or some equivalent, it would be impossible to invoke methods that require permissions not already held by the invoker. Such rights amplification is crucial, for example, when untrusted code invokes a system routine. A system routine is trusted to perform adequate checks before exercising the power that comes with the more powerful permissions in its associated protection domain; it should also be trusted to invoke only methods that are similarly prudent. So, a construct like `doPrivileged` that allows an invoked method to exercise permissions beyond those of its invoker is both sensible and useful.

The code in Figure 4.1 illustrates how `doPrivileged` is used. `display` directly invokes the `load` method of File System and invokes the `use plain font` method of GUI Library. Also note that `use plain font` invokes `load`—loading a font may require loading a file that contains bitmaps for the font. Thus:

- In invoking `load('thesis.txt')`, the `checkPermission` will throw a security exception if protection domains File System (the frame at the top of the stack) and Untrusted Applet (the next and bottom frame on the stack) do not each imply the needed permissions for reading that file. They do if `thesis.txt` resides in `/home/ue`.

- In invoking `load('Courier')` from within `use plain font`, the resulting call to `checkPermission` will throw a security exception if protection domains File System (the frame at the top of the stack) and GUI Library (the next frame on the stack) do not each imply the needed permissions for reading that file. They do if `Courier` resides in `/fonts`. Untrusted Applet is not checked for permissions, because the invocation of `load` in GUI Library is within the scope of a `doPrivileged`.

Java's stack inspection policy also handles dynamic creation of threads. When a new thread T is created, T is given a copy of the existing runtime call stack to extend. The success of subsequently evaluating `checkPermission` in thread T thus involves permissions associated with the call stack when T is created.

4.3 A Security-Passing Style IRM

The first work on modifying JVML programs to enforce stack inspection is described in [Wal99b, WF98]. There, an additional variable is introduced to replicate information from the JVM runtime call stack. This variable is changed upon invoking or returning from a method call as well as upon entering or exiting the scope of a `doPrivileged` block; the variable is scanned when `checkPermission` is evaluated. The resulting scheme is called *security-passing style* (SPS) because the

new variable is passed to method invocations as an additional argument.

SPS is an example of the IRM approach, so it is no surprise that PSLang/PoET can be used to create IRM_{SPS}, an implementation of SPS. The updates associated by IRM_{SPS} with each security event—method call and return, `checkPermission`, `doPrivileged`, and thread creation—are sketched in Table 4.1; the actual PSLang formulation appears in Appendix B.5.

In the PSLang that specifies IRM_{SPS}, variable `domainStack` replicates policy-relevant information from the JVM runtime call stack; this variable is local to each thread (and is equivalent to the additional explicit argument to method invocations employed in [Wal99b, WF98]). It is worth noting exactly how IRM_{SPS} handles security updates associated with a method call from A to B. Permissions for B could be added to security state `domainStack` either inside method A or inside method B. But performing the update inside method A turns out to be less desirable in part because when B is a virtual method (the Java equivalent of a function pointer), a dynamic lookup would be required to determine its permissions. Therefore, IRM_{SPS} does the security update inside method B.

4.3.1 Performance Overhead

In order to understand the performance of stack inspection implementations, the frequency and cost of relevant security events in actual applications must be assessed. Four applications were measured for these attributes: the Jigsaw 2.01 web server [BS96], Sun's `javac` Java compiler [GJS96], the `tar` utility [End98], and an MPEG video player [And99]. All were run using modern JVMs[3] with garbage

[3]For JDK 1.1.x, Symantec Java! Version 3.10.107(i) was used; for Java 2, Sun's JDK 1.2 distribution was used, which employs Symantec Java! Version 3.00.078(x).

Table 4.1: IRM_{SPS} implements security-passing style.

Method call/return: $A \rightarrow B$
Before calling B, look up permissions P_A for A's code and push P_A on the thread-local `domainStack`. After returning from B (either normally or by a thrown exception), pop `domainStack`, removing P_A.

doPrivileged $\{S\}$
Push a distinguished token **doPriv** on `domainStack`, at the beginning of the `doPrivileged`, and pop the token off at the end (whether an exception was thrown or not).

checkPermission(P)
Scan `domainStack` from top to bottom (without modifying it), and look at each set of permissions p. Throw a security exception if p does not imply P, but accept if $p =$ **doPriv** or the bottom of `domainStack` is reached.

Create thread: T
Set the `domainStack` of T to contain a copy of the contents of the `domainStack` of its parent thread.

collection disabled on a 300 Mhz Pentium II running Windows 98 and connected to a dedicated 100Mb Ethernet network. Since quantifying access-control overhead was of interest, the first three benchmark applications used the same set of 500 small synthetic Java source files as their input, with each source file defining a class containing only a single integer variable.

Table 4.2(a) shows how many times the various stack inspection primitives were invoked in the benchmarked applications. The cost of performing `doPrivileged`, `checkPermission`, and thread creation can be relative to the size of the JVM call stack, and—because `checkPermission` is dominant—the average number of accessed stack frames ("avg checked") for that operation is also reported. So that the numbers are less dependent on irrelevant implementation details, stack inspection primitives used in the construction of permission objects have not been counted. For instance, not counted are the `doPrivileged` invocations for creating each `java.io.FilePermission` object in Sun's implementation.

Table 4.2: Assessing stack inspection performance.

	Method calls	**doPrivileged**	**checkPermission**		**New threads**
			count	avg checked	
Jigsaw	2,476,731	1,002	5,333	18.7	71
`javac`	1,456,970	0	1,067	12.4	0
`tar`	19,580	0	6,509	8.6	0
MPEG	35.997.662	101	205	5.7	201

(a) Frequency of stack inspection primitives.

Method call	**doPrivileged**	**checkPermission**	**New thread**
1.00μs	1.66μs	7.7μs	6.5μs

(b) Benchmarked cost of IRM$_{SPS}$ primitives (at stack depth 10).

	JVM	IRM$_{SPS}$
Jigsaw	6.2%	20.1%
`javac`	2.9%	46.2%
`tar`	10.1%	3.0%
MPEG	0.9%	72.5%

(c) Overhead of JVM-resident implementation and IRM$_{SPS}$.

Table 4.2(b) shows the overhead, in microseconds, for the IRM$_{SPS}$ stack inspection primitives. The values shown are averages from a synthetic benchmark of the primitives. The primitives in the last three columns were benchmarked using a stack depth of 10—each operation accessed 10 stack frames.

Table 4.2(c) compares the runtime overhead of Sun's JVM-resident implementation of stack inspection and IRM$_{SPS}$. The column labeled JVM gives the percentage overhead between running the application on Java 2's JVM with stack inspection enabled versus without stack inspection enabled; the column labeled IRM$_{SPS}$ gives the percentage overhead between running the application with IRM$_{SPS}$ on Java 1.1[4] versus without any IRM. Measurements did not include the cost of con-

[4]Java 1.1's JVM was employed to measure IRM$_{SPS}$ because otherwise the stack inspection already present in Java 2's JVM would perturb the results.

structing permission objects or evaluating their `implies` methods, in order to better quantify the relative difference in overhead between implementations. For each average, the computed standard deviation was small enough to be ignored in interpreting the numbers.

The JVM-resident implementation is considerably cheaper for Jigsaw, `javac`, and MPEG. This is not surprising because of the per method call cost of IRM_{SPS} and the large number of method calls each of these applications makes. However, when an application has many permission checks relative to the number of method calls, IRM_{SPS} can exhibit less overhead than the JVM-resident implementation. This is because IRM_{SPS} is able to amortize the cost of creating `domainStack` over a large number of `checkPermission`'s. The results for `tar` illustrate this benefit.

4.3.2 An Improved SPS Implementation Scheme

The overhead of an SPS stack inspection implementation would be improved if the security state (i.e., `domainStack`) were not updated on each method call. In fact, updates need to be made only when a method call crosses protection domains—method calls within the same protection domain repeatedly push the same permission onto `domainStack`, and `checkPermission` is unaffected by replacing sequences of identical stack frames with a single frame.

The implementation of [Wal99b, WF98] exploits this insight. The implementation comprises 12,800 lines of Java code, of which 1700 lines implement an analysis to determine whether invoked methods are in the same or different protection domains as the invoker and 6900 lines are produced by JOIE, the generic JVML rewriter [CCK98]. With these optimizations, [Wal99b, WF98] reports overall security enforcement overheads of between 13% and 17% of total execution time—still

relatively high when compared to the overheads on the same applications run under the JVM-resident implementation stack inspection. Adding this optimization to IRM_{SPS} does not seem worthwhile, given the performance gains achieved in other ways with the IRM implementation of the next section.

4.4 A Lazy Stack Inspection IRM

Sun's implementation of stack inspection profits from having direct access to the JVM call stack, because then no overhead is incurred at method calls in order to keep track of nested invocations for subsequent `checkPermission` evaluation. Since method calls are the common case, the performance advantages of this design should be obvious.

In order to specify such a scheme in PSLang, some facility is needed for accessing the JVM runtime call stack. Fortunately, Java provides one. First, Java provides an interface so that exceptions can print a textual description of the JVM call stack when they are thrown; second, the Java `SecurityManager` contains a protected method `getClassContext` that returns the JVM call stack as an array of `Class` objects, each a unique identifier for the code at that JVM call stack frame. The PoET runtime makes this latter interface accessible to PSLang specifications (as part of PoET's `System` library) by extending the `SecurityManager`.

Table 4.3 sketches security events and updates for IRM_{Lazy}, an IRM stack inspection implementation that uses the JVM call stack. The actual PSLang formulation appears in Appendix B.6. Notice how work has been moved from method call/return to the implementation of `doPrivileged`, `checkPermission`, and new thread creation (which all must make a copy of the call stack when they are invoked). `doPrivileged` pushes the frame number for the stack frame at the top of

Table 4.3: IRM_{Lazy} uses the JVM call stack.

Method call/return: $A \rightarrow B$ Nothing.	**doPrivileged** $\{S\}$ At the start of `doPrivileged` push the frame number of the current JVM call stack onto `privStack`; at the end pop it off (whether an exception was thrown or not).
checkPermission(P) Let `bottom` be the position of the top-most privileged stack frame on `privStack`, or 0 if there is no such stack frame. Scan the current JVM call stack from top to `bottom` and find the permissions p for each stack frame—reject if ever p does not imply P. If there was no privileged stack frame, likewise scan the `ancestralStack`.	**Create thread:** T Let the `ancestralStack` of the new thread T be either a copy of the `ancestralStack` that is present on its parent thread, with the current JVM call stack pushed on top, or—if there is a privileged stack frame somewhere to be found on `privStack`—the top portion of the current JVM call stack up to that privileged frame.

the current JVM call stack onto a separate thread-local variable, `privStack`. This frame number then serves to bound the segment of the JVM call stack that must be traversed in evaluating `checkPermission`—stack frames appearing lower on that call stack are not checked. For each thread, the relevant stack frames of parent threads are stored in thread-local variable `ancestralStack`, since this information cannot be derived from the current JVM call stack and it is needed in evaluating any `checkPermission` that does not terminate early by reaching a `doPrivileged` frame.

Table 4.4(a) shows the cost of the stack inspection primitives with IRM_{Lazy}. As with Table 4.2(b), reported measurements are averages from a synthetic benchmark that repeatedly performed the subject operation.

Notice that, except for method calls, the measured costs for each stack in-

Table 4.4: Assessing the IRM_{Lazy} stack inspection implementation.

Method call	**doPrivileged**	**checkPermission**	**New thread**
0μs	23.4μs	22.4μs	29.8μs

(a) Cost of JVM-call-stack implementation primitives.

	JVM	IRM_{SPS}	IRM_{Lazy}
Jigsaw	6.2%	20.1%	6.4%
`javac`	2.9%	46.2%	2.0%
`tar`	10.1%	3.0%	5.4%
MPEG	0.9%	72.5%	0.4%

(b) Overhead of JVM-resident and IRM_{Lazy} implementations.

spection primitive in Table 4.4(a) are higher than the IRM_{SPS} costs given in Table 4.2(b). These higher costs arise because the entire stack is now being copied by the implementations of all but the method call/return stack inspection primitives. Even so, for the benchmark applications, IRM_{Lazy} exhibits overall performance that is superior to IRM_{SPS} and that is competitive with Sun's JVM-resident implementation. This is seen in Table 4.4(b), and it is a consequence of method call/return invocations dominating performance of the benchmarks. Where IRM_{Lazy} performs better than the JVM-resident implementation, it is because of optimizations in the PSLang specification, which do a better job of eliminating redundant work in permission checking.[5]

4.5 Concluding Inspections

This section has shown how enforcement of stack inspection could be eliminated from the JVM. The result is a smaller and simpler—hence, more trustworthy—JVM, especially since the enforcement mechanism for stack inspection policies is

[5]Similar optimizations are done in IRM_{SPS}.

now isolated (as a PSLang specification) rather than being distributed throughout the JVM.

The idea of separating mechanism from the policy that directs this mechanism is advocated often. Java 2's support for the stack inspection access-control policy involves a mechanism (in the JVM) and the flexibility to direct that mechanism through policy files, protection domains, and permission classes (with their `implies` methods). The IRM realizations of stack inspection actually draw a somewhat different line between policy and mechanism. With no JVM-resident mechanism, there is considerable flexibility about what policies can be enforced using the IRM approach and about when that choice of policy must be made.

This flexibility allows enforcement of policies that alter or extend what the JVM implements today. One might now contemplate remedying the various deficiencies in the Java 2 stack inspection access-control policy, allowing

- protection domains, permissions, and the `implies` method to be changed after execution of an application is commenced, enabling straightforward creation of new protection domains as execution proceeds;

- the coupling between protection domains and bytecode origin to be refined so that, for example, an application's state is used in determining the protection domain for code; and

- the operation of `doPrivileged` to be extended so that only some of the privileges in a protection domain are amplified in a block of code.

It now even becomes possible to enforce different security policies on different Java applications, raising questions about detecting and resolving incompatibilities between those policies. However, these questions about policy composition are

independent of whether or not the IRM approach is being used to enforce policies.

Flexibility is a double-edged sword. The IRM approach is not only flexible enough to implement Java 2's stack inspection (in multiple ways!) and to implement a host of variants that address apparent limitations in the policy, but it is also flexible enough to allow policies to be defined that have unanticipated consequences or vulnerabilities. There is no way to guarantee that the PSLang formulations of stack inspection in Appendices B.5 and B.6 are indeed the policy supported by Sun's distribution. To get such assurance, a formal specification of Sun's stack inspection implementation would be needed, as well as a logic for PSLang specifications. Neither exists. But PSLang could be given a formal semantics in terms of security automata, and then it would not be difficult to reason about and/or simulate PSLang policies in order to gain confidence that they describe what is intended.

Even without a logic for reasoning about PSLang specifications, the exercise of formulating stack inspection in PSLang, a formal language, did prove enlightening. Writing the PSLang security updates forced the asking of questions about what really happens when security events occur. In seeking the answers to those questions, several surprising things about the exact semantics of stack inspection came to light:

- If a new thread is created from within a `doPrivileged` block then that thread will continue to enjoy amplified privileges—even though its code might not be within the scope of a `doPrivileged` block and even after its creator has exited from within the `doPrivileged`. This is because the new thread starts execution with a copy of its creator's call-stack (whose top frame is marked as being within the scope of a `doPrivileged`).

- When a class B extends some class A but does not override A's implementation of a method *foo*(), then the permissions associated with A (and not B) are used by `checkPermission` for *foo*'s stack frame. Because B can extend A in ways that may affect the semantics of *foo*, (such as by overriding other methods), one might argue that the wrong permissions are being consulted.

Both of these "features" of stack inspection will become apparent to attentive readers of the PSLang formulations in Appendices B.5 and B.6.[6] This is not to say that there aren't also surprises in the PSLang formulations or there aren't aspects of the Java 2 behavior that were missed in constructing these formulations. But having—in just a few pages—a complete and rigorous description of the security policy being enforced seems like a necessary condition for understanding that policy.

[6] For example, in Appendix B.5, the security event `ON EVENT begin method WITH appMethod()` results in a security update that adds the protection domain for the current method on the top of `domainStack`. This protection domain will not reflect inheritance, since it is read from the class variable `siDomain`. Also in Appendix B.5, the `domainStack` for a newly created thread is a direct copy of its creating thread's `domainStack`—including any **doPriv** tokens—and, therefore, the new thread may be privileged throughout its lifetime.

Chapter 5

Related and Future Work

To the author's knowledge, the first known use of program modification to enforce security policies was in 1969 in the Informer execution profiler [DG71] for the Berkeley SDS 940 time-sharing system. In that system, modules—object code program fragments typically hand-crafted in machine language—could be inserted into the operating system kernel, where they gathered and filtered profiling events. Informer would reject modules that used privileged instructions or whose execution time could not be statically bounded. In accepted modules, each memory reference machine instruction would be modified so it was preceded by a runtime check that restricted the access to profiler memory.

Later, SFI [WLAG93] re-discovered the use of software modification to achieve runtime protection more typically enforced by hardware. SFI, again, used a combination of static analysis and program modification to enforce runtime memory-protection guarantees [ATLLW96, Sma97]. Meanwhile, program rewriting had been used for execution monitoring and debugging purposes in multiple specialized tools that, like Informer and SFI, implemented one particular fixed functionality, or policy [HJ92]. These evolved into ATOM [SE94], the first general-purpose program rewriting mechanism, which implemented a policy-free instrumentation framework that was used to construct several tools for execution monitoring.

Until recently, general-purpose security enforcement mechanisms were implemented in the manner of traditional reference monitors—by interpositioning between hardware-supported security boundaries [Sal74, Tan92]. Two examples of this approach are Generic Software Wrappers [FBF99] and Janus [Wag99], both of

which intercepted calls to the operating system kernel [Gar03]. Recently, a system has enhanced system-call interception with policies based on a security automata variant [SVB+03].

Apart from in SASI and PSLang/PoET, rewriting programs for security enforcement has recently been used in work that continues this tradition of wrapping system interfaces. Naccio [ET99] modifies method-call instructions, redirecting them through a wrapper method; Ariel [PH98] and Grimm and Bershad [GB98] mediate method calls by inserting enforcement code between JVML bytecode instructions.

These refinements of traditional reference monitors were spurred on by the advent of Java [GJS96], which both facilitated program modification and also spurred interest in novel security mechanisms. Java 2's stack inspection [Gon99, WF98] and its successor by Abadi and Fournet [AF03] are two such novel mechanisms where the security policy depends on the previous execution history. Deeds [EAC98] is another, and more general, history-based access control mechanism for Java.

Relying on high-level languages for security policy enforcement was pioneered with the Burroughs B5000 computer [Org73], which required that applications be written in Algol [OT97]. This approach has recently been adopted not only in Java, but also in the SPIN and Vino extensible operating systems [BSP+95, SESS96], which respectively relied on Modula-3 and `gcc` compilers [Nel91, Woe94], and the Agent TCL mobile code platform [Gra97], which relied on a Safe TCL interpreter [OLW96]. In these systems, the use of a high-level language was imposed either by only accepting source code, and performing compilation internally in the systems, or through verification of a cryptographic signature [RSA78] that a particular trusted compiler has been used.

Java innovated by not requiring that the Java compiler be trusted, but only requiring that its JVML [YL96] output be verifiable. A JVML program is verified using a relatively simple analysis that can establish the type-safety of that program. Such trustworthy analysis techniques have become an active area of research. Efficient Code Certification [Koz98] can check the safety of memory, jump, and stack operations. Typed Assembly Language [MWCG98] provides a flexible type-safe assembly language that prevents abstractions from being violated. Most generally, Proof-Carrying Code [Nec97] allows for the verification of arbitrary proofs of program behavior and also considers the implications of distributing the establishment and verification of properties between multiple parties [Nec98]. Such relocation of security enforcement work allows for more agile deployment of security policies, and has been explored by Sirer, Grimm, and Bershad [GB98, SGGB99, SGB+98].

Another current path of security research relies not on language-based guarantees but, rather, on SFI-like runtime checks inserted during dynamic translation of programs [SCK+92]. Program shepherding [KBA02] uses this approach to implement non-executable memory in programs for the Intel x86 [INT94]; Strata [SD02] uses the same approach but implements a more general framework.

Even so, approaches that build upon language-based guarantees—like those provided by Java and Microsoft's .NET [BS02]—seem destined to dominate, because of their many advantages in addressing concerns about reliability, flexibility, and security. However, type safety is no panacea, as shown by PSLang/PoET as well as by attacks that manage to subvert type-safety guarantees, e.g., by inducing random bit errors in memory hardware [AG03]. Even with a flexible enforcement mechanism that builds upon type-safe foundations, many challenges remain, as demonstrated in this dissertation.

Recently, progress has been made in addressing those challenges. Bauer, Ligatti, and Walker, in particular, in a series of papers [BLW02b, BLW02a, BLW02c, BLW03] have improved the theoretical foundations for IRMs to encompass static analysis, non-terminating remedial actions, and the composition of such IRMs. With Zdancewic, this theory has also been applied in the context of the aspect-oriented programming methodology [WZL03, KLM+97], which shares many properties with IRMs. On another front, Hamlen, Morrisett, and Schneider have clarified the connection between set-theoretic specification of safety properties and the runtime enforcement of those properties, constraining enforcement mechanisms to allow maximum progress for each program execution [HMS03]. Meanwhile, the SLAM effort at Microsoft Research has successfully used model checking and declarative reasoning to statically verify that Windows device-drivers (given as C source code) can never violate a security-automata-based policy at runtime [BR02].

Future implementations of IRM toolkits can build on these additional formal results and also make use of new results that formalize the connection between partial-evaluation and IRMs [Thi03]. Such IRM toolkits should, therefore, make it easier to understand IRMs that have been extended along the lines of §3.6, as well as the composition of such IRMs. Those IRM toolkits should not only be more trustworthy, by lending themselves better to analysis, but also be more efficient, as they could make use of complete verifiable partial-evaluation frameworks in the insertion and simplification of IRM enforcement code.

Preferably, the environment of future IRM toolkits would be designed to expose common execution activity as security-relevant actions—eliminating the need for their synthesis as security events, like described for JVML in §3.3.2. Also, those future IRM toolkits could provide transactional support, perhaps along the lines

of [HF03], for both security state and updates, as well as for the execution of target program fragments. These two improvements might be combined with a standard IRM runtime environment and a standard for IRM embedding and identification, as discussed in §3.5. In this case, few obstacles would remain to wide-spread deployment of the IRM approach.

Chapter 6

Conclusions

Security enforcement mechanisms have been one of the most influential factors in the design of operating systems [Lam74] and remain a topic of research for all infrastructure systems [Sch99a]. Multi-processing, virtual machines, processes, isolated address spaces, threads, inter-process communication, shared memory, kernels, and microkernels, all aim to to share computing resources in a controlled, yet efficient, manner that prevents both accidental and deliberate protection violations [Tan92]. These abstractions are the framework in which computing is currently performed; however, their design primarily results from that of hardware-supported traditional reference monitors. IRMs—combined with strong language-based guarantees—are an alternative to this traditional design for protection and, as such, hold the potential to fundamentally change the abstractions of computing.

IRMs are strictly more powerful than previous security enforcement mechanisms, because IRMs can mediate more activity and because more information is accessible to them. This makes IRMs especially well suited for the increasingly common class of applications that form extensible systems. In addition to this expressive power, IRMs have the potential for greater efficiency, because they can be customized to each activity—for instance, allowing trusted applications to execute with zero security enforcement overhead.

IRM security policies are specified in a structured manner that allows the complete details of their implementation and enforcement strategy to be given, alongside their higher-level policy objectives. IRMs can enforce any practical security policy, whether it be traditional access control or sandboxing, virtualization and

isolation, or simply the auditing of execution activity—the extended notion of IRMs discussed in §3.6 can even enforce liveness and other policies outside of the class EM. Such IRM enforcement can build upon any notion of security primitives, such as principals, objects, and authentication, e.g., by using results from static analysis or the primitives of an underlying platform. IRMs can also themselves implement any required security primitives, thus eliminating the need for security support from the execution environment.

By specifying both intended security policies as well as their implementation details, IRMs can provide higher assurance in security enforcement. Apart from the IRM toolkit (which can be of moderate size), and the actual semantics of low-level actions (such as fundamental ISA operations or operating system primitives), a single isolated IRM specification captures all details relevant to security enforcement. This makes protection more trustworthy, because the properties of this IRM specification can be derived in isolation, through its examination, analysis, and testing. By careful construction of the IRM primitives, such analysis and declarative reasoning is placed on a sound foundation by the many formal results on the general semantics of low-level languages [MWCG98] as well as recent results specific to modification of program behavior [BLW03].

Because IRMs become an integral part of a secured application (and because they comprise both a security policy and the means for its enforcement) the IRM approach applies especially well to distributed settings, where trust may be decentralized. With IRMs, security enforcement can be effected anywhere from the development of a software application, through its deployment and management, to its execution by an end user—whether or not the final execution environment explicitly supports IRMs. In addition, IRM security enforcement can make use

of any existing trust relationships, such as trusted network connections or trusted cryptographic data-signing keys.

This agility of IRMs in distributed settings allow their deployment to be crafted so that security is enforced in the most cost-effective and trustworthy fashion. Any IRM adoption will, however, most likely be based on IRM specifications provided by the developers and vendors of software applications and platforms: Those parties are most likely to understand the semantics and implementation details of any particular APIs, and this knowledge is necessary for the correct synthesis of fundamental security events. Even so, with the IRM approach, basic security specifications can be enhanced by interested parties, such as system administrators or third-party security professionals, for instance, to allow greater usability or security. Eventually, such enhancements might form a diverse market for IRM security policy specifications tailored to every scenario.

IRMs form a general-purpose security enforcement mechanism that can be practically implemented, as demonstrated by PSLang/PoET and this dissertation. Their flexibility, end-to-end transparency, and many other advantages, make the IRM approach to security policy enforcement an attractive framework for tackling current challenges in computer security.

Appendix A

PoET and PSLang:

Java Inlined Reference Monitors

A.1 PSLang/PoET: Syntax and Semantics

PoET, the Policy Enforcement Toolkit, is a mechanism that rewrites the JVML class files of target programs and inserts IRM enforcement code implementing a reference monitor into those programs. The security policy to be enforced by the IRM is defined in PSLang, the Policy Specification Language, and specifies to PoET what security state the policy needs to keep for the target program, what JVML actions identified or exposed by PoET are relevant to this policy, and what security updates those security-relevant actions should trigger. PoET can be seen as confining its duty to monitoring references—mediating to allow security decisions to consider any JVML low-level action—and leaving it to PSLang policies to specify how each monitored reference should be handled. However, as PoET performs all the rewriting for IRM enforcement, PoET is also responsible for inlining both the security state and updates specified by PSLang policies.

This appendix describes the syntax of PSLang security policies in an extended form of Backus-Naur grammar notation [ASU85] and describes the semantics of both PoET and PSLang in annotations to that grammar. The intent is neither to formally specify the syntax nor the semantics but, rather, to enable a person familiar with the Java language to understand PSLang policies and how they interact with PoET.

Policy: *Preamble* **(** *GlobalState* **)?** **(** *Function* **|** *Update* **)***

A PSLang security policy is given in a file with a `.psl` extension; such files are input to the PoET rewriter along with the target program. Each PSLang policy contains a description of a set of security events and the security updates they should trigger. Updates can make use of global security state, which may be optionally defined at the start of a PSLang file. To factor out activity that is common to more than one event, PSLang allows function definitions to be interspersed between events.

PoET interprets the content of a PSLang policy in an environment containing the global state and any functions defined (or forward-declared) up to that point. Not only is the information in a PSLang policy interpreted by PoET in the sequence it appears, but the runtime behavior of a PoET-inserted IRM also reflects the relative order and construction of the policy. So, if two PSLang updates that result in runtime IRM activity are triggered by the same JVML low-level action in the target program, PoET ensures the IRM enforcement code executes at runtime in the order given by the original PSLang policy.

Preamble: *Extensions* *Libraries*

Extensions: **(** `EXTEND POLICY` *policyFileName* `;` **)***

Libraries: **(** `USES LIBRARY` *libraryFileName* `;` **)***

The start of each PSLang policy can have a preamble that optionally specifies a set of PSLang policies that the current policy extends. Each policy that is to be extended should be a PSLang `.psl` policy file found in the current working

directory of the file system. The policies in the list of *Extensions* may themselves specify other policies to be extended and PoET will interpret the policies bottom-up based on their order of appearance.

The starting preamble of a PSLang policy also specifies what libraries are available to that policy. Libraries expand the set of abstractions and interfaces available in the PSLang language using syntax similar to Abstract Data Types (ADTs) [AHU74]. Like `#include` in C [HS94], library inclusion propagates from extended policies, so that if policy A uses a library and is extended by policy B then B also uses the library. Three PoET ADTs, `Event`, `Reflect`, and `State`, described in §A.3.1, §A.3.2, and §A.3.3, are always available, even if not explicitly included in *Libraries*, since their use is integral to any PSLang policy.

The set of available libraries is extendible by implementing a new Java (or JVML) class in the `PoET.runtime` package. In the Java program for a library class, all ADT operations to be exposed are given as public static methods whose name is prefixed with either `proc$` or, if they have no side effects, `func$`. The ADT library operations are then invoked by their library name, a dot, and their method name, as in: `Event.classNameIs("Foo")`. If a library exposes mutable state to PSLang, that state must be encapsulated as an opaque object. For example, `Set.create()` returns an object that can be used in other `Set` operations like `Set.size(SomeSetObject)`.

GlobalState: `GLOBAL SECURITY STATE {` **(** *VarDef* `;` **)*** `}`

VarDef: *Type variableName* `=` *Expr*

Type: **(** `Object` **|** `boolean` **|** `int` **|** `double` **|** `void` **)**

Each PSLang policy can optionally define globally accessible state just after

the *Preamble*. Global state consists of a set of variables (which must be initialized to `null` or to some less trivial expression) that can be used directly by name anywhere after being defined. Like all PSLang state, global state can only be of a type supported by PSLang, where PSLang types are a subset of the Java types that were chosen so information isn't lost when casting to a PSLang type. The initialization of global variables takes place before PoET rewrites or executes any part of the target program. Thus, initialization expressions are executed, in the order they appear, before anything else.

Function: **(** `SIDE-EFFECT-FREE` **)?** `FUNCTION` *FuncDef*

FuncDef: *Type functionName* (*FormalArgs*) *FuncBody*

FormalArgs: **(** *FormalArg* **(** , *FormalArg* **)*** **)?**

FormalArg: *Type argumentName*

FuncBody: **(** ; | *StmtBody* **)**

As mentioned before, PSLang allows definitions of functions to factor out common policy code. A function definition specifies its return type (possibly `void`), name, zero or more typed formal arguments, and, optionally, an implementation. To support mutual recursion, function bodies can be omitted, but each such forward-declared function must be specified with an implementation somewhere in the PSLang policy, exactly once.

PSLang is designed to support simplification by partial-evaluation-based techniques, so functions can be declared to be side-effect free.[1] Judicious use of side-effect-free functions is the key to efficient PSLang policies, since such functions primarily add overhead to the rewriting and not to the execution of the target

[1]It is the responsibility of the policy writer to ensure that functions so declared actually have no side effects.

program. PoET simplifies functions and updates by evaluating expressions and side-effect-free functions in an environment of known information, including known constants and information about the security-relevant action exposed through the `Event` ADT of §A.3.1. All PSLang code is simplified in this manner, but side-effect-free functions are simplified at least twice—first in the environment existing at the time of their definition and again in environment existing at each point of their use, effectively inlining their invocation.

One distinguished user-defined PSLang function, the `stealthCheck` function is treated specially by PoET (and is side-effect free). This function exists to help implement IRMs on commercial JVMs, where implementation irregularities can preclude inlining arbitrary code. PSLang policies specify the implementation of a `stealthCheck` as a function returning a boolean, and accepting a single `Tuple` ADT object argument. Then, on PoET events that should invoke the `stealthCheck` at runtime, PSLang policies use a special `JVML` ADT operation to pass arguments to the function, including the runtime PoET instance object and a command string.[2] PoET implements `stealthCheck` by leveraging JVML virtual method calls to masquerade a call to the PSLang-defined function as a `java.lang.Object.equals` method call in the inserted IRM code. This indirection allows the `stealthCheck` function to be called at runtime when the target program executes without any perturbation of the order in which the JVM loads its system classes.

Update: `ON EVENT` *Event* `PERFORM SECURITY UPDATE` *StmtBody*

Event: *ActionName* **(** `WITH` *Expr* **)?**

[2]See the `getSpecObject State` ADT operation in §A.3.3.

Updates are the heart of any PSLang policy, with each *Update* defining a set of PSLang statements and an *Event* that specifies, for those statements, what insertion points should be used by the rewriter and when to trigger them at runtime. For any *ActionName*, or named JVML low-level action (detailed in the next paragraphs), the optional `WITH` predicate expression allows for the synthesis of a more relevant security event. The `WITH` expression is limited to calling only side-effect-free functions, which ensures it can always be fully evaluated during rewriting. This does not limit the expressible set of security events because the ADTs likely to be used, in particular `Event` and `Reflect` of §A.3.1 and §A.3.2, are side-effect-free, and other operations can be moved into the statement body.

Each JVML low-level action that is identified or exposed by PoET, is named in PSLang by that action's *place*, the structural part of the target program the action is tied to. The PoET rewriter uses those places as insertion points for the IRM enforcement code for the PSLang policy. Then, security updates are triggered at runtime whenever target program activity occurs in those places. For example, a PSLang policy update named with a `method` action and no `WITH` predicate will be triggered whenever a thread in the target program executes a method.

```
ActionName:   ( ActionTime )?  ActionPlace

ActionTime:   ( at start of )?  ( loading )?
            | at exception thrown in
            | at normal completion of
            | at finally completed  ( loading )?

ActionPlace:  program
            | class initialization
            | object instance initialization
            | object instance garbage collection
            | method
            | exception handler
            | basic block
            | instruction
```

PSLang policies can optionally refine the named JVML low-level action with an *ActionTime*, which specifies whether the update is triggered before or after the action occurs at runtime. The execution of an update's statements can be triggered `at start of` an action, i.e., immediately before the execution of that action at runtime. Alternatively, the update may be triggered at the end of the action (by using `at normal completion`), only if the activity throws an exception (`at exception thrown in`), or at either of those two times (`at finally completed`). If not specified for an update in a PSLang policy, `at start of` the action is the default. The triggering of updates nests in the natural manner, with (several) instructions in each basic block, basic blocks in exception handlers and methods, and exception handlers in methods. Finally, as shown in Table A.1, initialization and garbage collection are also named as a method action, as is the program action.

Table A.1 shows how PoET actions are named[3] by each major JVML struc-

[3]As discussed in §3.3, the names in Table A.1 are simplistic, for example, only providing "correct" semantics for the `program` actions for certain classes of single-threaded target programs.

Table A.1: The place of PoET low-level actions and their insertion points.

PoET *ActionPlace*	**Insertion point**
`program`	The `main` method in the target program's "main" class, as specified to PoET according to the rules of §A.2.
`class initialization`	The special JVML class initialization method `<clinit>` of a loaded class.
`object instance initialization`	Every JVML object instance initialization method `<init>` of a loaded class.
`object instance garbage collection`	The JVML `finalizer` method of a loaded class.
`method`	Every JVML method of a loaded class (including those above in this table).
`exception handler`	Every exception handler in every JVML method of a loaded class.
`basic block`	Every basic block in every JVML method of a loaded class.
`instruction`	Every JVML instruction in every JVML method of a loaded class.

tural element, which also determines their insertion point. As discussed earlier, specification of an *ActionTime* can refine the semantics of each insertion point in Table A.1, so that IRM enforcement code will be triggered either before or after the program's runtime execution at that place. Not shown in Table A.1 is the nesting of PoET actions. Just as a JVML method always consists of one or more JVML instruction, one or more PoET `instruction` actions are embedded within a PoET `method` action. At runtime one action may be nested within another action, such that all updates for certain instructions are triggered between the triggering

of updates at the beginning and end of those instruction's method.

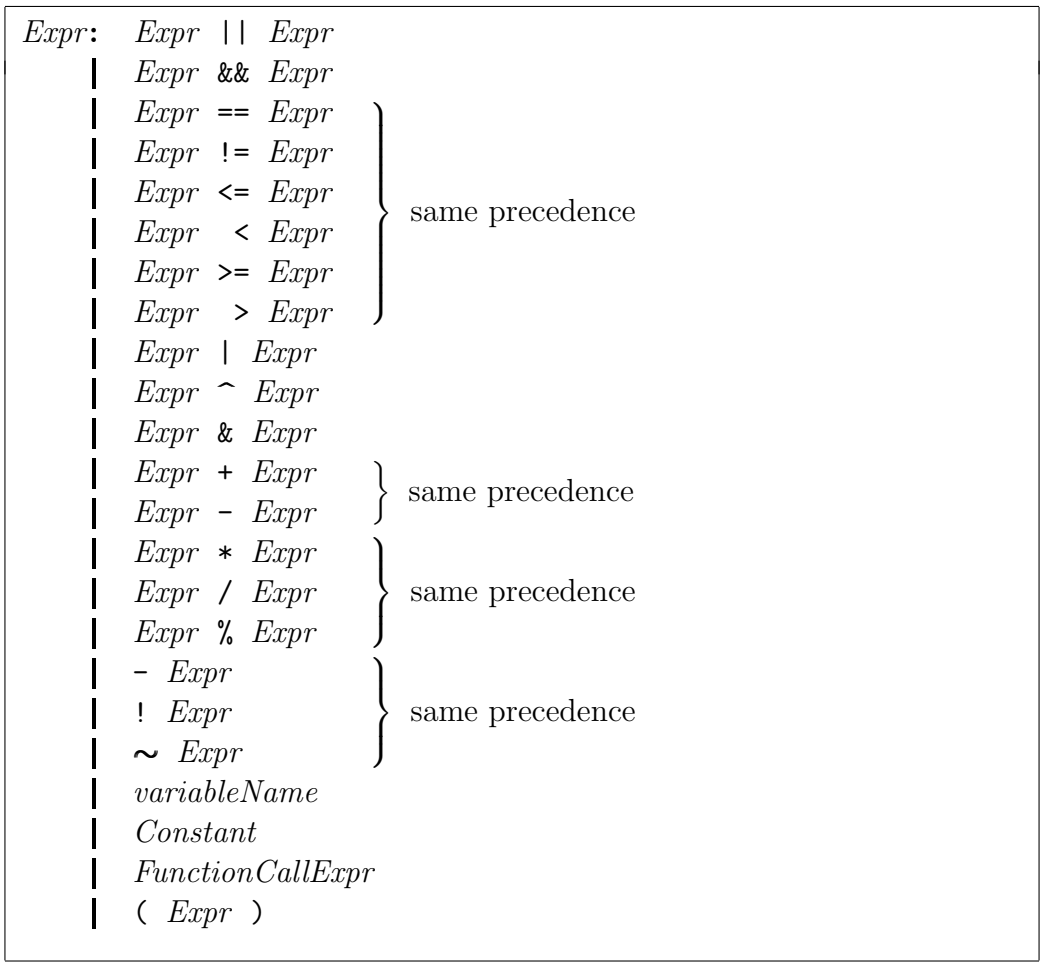
PoET can also trigger updates before and after loading each structural component, or place, in the target program from its class file, deriving the order for loading from that of the Java ClassLoader [GJS96, Gon99]. Static rewriting (also discussed in §A.2) creates special difficulties for load-time updates as it introduces a program load-time that is disjoint from execution time. Thus, if static rewriting is used, PSLang policies must be careful not to specify load-time updates that refers to runtime information.[4] Because of this limitation, PSLang load-time updates should primarily initialize and maintain information on target program structure and insert target program security state.

```
StmtBody:  { ( Statement )* }

Statement:  VarDef ;
          | variableName = Expr ;
          | if ( Expr ) StmtBody ( else StmtBody )?
          | while ( Expr ) StmtBody
          | for ( VarDef ; Expr ; AssignExpr ) StmtBody
          | FunctionCallExpr ;
          | return Expr ;
          | FAIL [ Expr ] ;
          | StmtBody
```

The PoET structure of named low-level actions, shown in Table A.1, lacks all details such as the name of classes or methods. The predicate expressions and statement body of PSLang updates allows for the examination of such details and for the synthesis of higher-level security events. The use of the `Event`, `Reflect`, and `State` PoET ADTs, described in §A.3.1, §A.3.2, and §A.3.3, is integral to the interpretation of each action in Table A.1.

[4]This limitation could be partially alleviated by persisting all security state (e.g., using Java serialization) at the end of static rewriting and restoring that state when execution begins. Currently, this is not implemented in PoET.



PSLang statements inherit their semantics mostly from Java [GJS96] and indirectly from C [HS94]. The most noteworthy exception is guaranteed simplification based on partial evaluation that produces, for example, the inlining of side-effect-free function calls. Expressions using conditional operators are always short-circuit evaluated—even during simplification—with control flow potentially simplifying as expected. The `for` statement encodes a while statement, as in C [HS94], but `for` is further constrained in its pre- and post-statements. The `break` or `continue` of Java are not supported, because their behavior can be programmed using multiple functions and `return`. The `FAIL` statement can be thought of as a function that prints its argument as a string and then halts the JVM.

PSLang expressions, like PSLang statements, inherit their semantics mostly from Java [GJS96], with the exception of guaranteed simplification. PSLang expressions operate on the limited set of PSLang types, propagating them as in Java. Operator precedence is non-surprising and, as mentioned before, logical operators short-circuit. Left-to-right order of evaluation is guaranteed for all operators, including those of equal precedence.

Constant: **(** *StrConst* **|** *IntConst* **|** *DblConst* **|** `true` **|** `false` **|** `null` **)**

Constant expressions in PSLang are like those in Java [GJS96], for the subset of types supported in PSLang; strings and `null` are the only `Object` constants. PSLang does not support any escape sequences in string constants.

FunctionCallExpr: *functionName* `(` **(** *Expr* **(** `,` *Expr* **)*)?** `)`

PSLang function calls are as expected, with argument expressions evaluated in left-to-right order.

A.2 PoET Invocation and Runtime

PoET is invoked with a PSLang policy by calling the public method `main` in the class `PoET.Main`, which in most Java environments can be specified on the command line. The arguments to PoET are an array of strings of the following form:

javaCommand `PoET.Main` **[** *options* **]** *policy*
< `-classes` *listFile* **|** *MainClass args...* **>**

The primary non-optional argument to PoET is the name of a PSLang policy (which can be a file path, but should exclude the `.psl` suffix). PoET applies this policy in one of two modes: statically on a list of classes specified in a file or dynamically on a running Java program as it loads classes during execution. To run PoET in the first mode, the `-classes` argument is given followed by a text file listing classes by their JVML name (using "`/`" as package separator), one class on each line. The first class listed in this file is taken to be the main program class—that is, the class containing the `main` method on which PoET triggers the `program` event. When PoET runs in the second mode, the name of the main program class is given on the command line and PoET calls its `main` method with the remaining command-line arguments.

The *javaCommand* used to execute `PoET.Main` should have a classpath to the PoET classes, either through a `CLASSPATH` shell environment variable or through an explicit argument, like `-cp` *poetPath*, that sets the path. At runtime, PoET must know this file path, the root of its installation directory, and refers to the `PoET.path` Java property value for that information. If PoET is not installed in the default directory, `c:\poetdist` on Microsoft Windows, the `PoET.path` property can be set with a *javaCommand* argument, e.g., `-DPoET.path=e:\poet`.

[`-silent`**]**	Suppress all PoET output
[`-debug`**]**	Prints extra debug information
[`-in paths`**]**	Input classpaths ("`-in .;..`")
[`-out path`**]**	Path to output PoET-processed classes
[`-java2support`**]**	Java2-support classes in runtime

The *options* for PoET include the ability to produce no output, with `-silent`, allowing non-corrupted target program output, and the ability to produce verbose output, with `-debug`. PoET loads class files from the current working directory or

from paths given with the `-in` option. If PoET is run with the `-out` option then it will output all class files it uses at runtime (including the PoET runtime library) to the path specified there. Finally, if PoET is given the `-java2support` option, then it will include in the PoET runtime versions of the Java 2 classes supporting stack inspection, allowing the policies of §B.5 and §B.6 to enable that type of security with older Java class libraries.

A.3 PSLang Libraries for PoET

The following three ADT libraries form a key component of PoET semantics that allows PSLang to specify a finer granularity of actions than the *ActionName* syntax of §A.1 and also allows the synthesis of higher-level security events. For each PSLang *Update*, the `Event` library exposes a reflective data type that allows the inspection and modification of target program structure at the place (or insertion point) of the update. For this purpose, the opaque ADTs exposed by the `Event` library are used as arguments to ADT operations in the `Reflect` and `State` libraries. Thus, the return value of `Event.class()` can be used as the argument to `Reflect.className(arg)` to retrieve the name of the class in which an event is occurring.

A.3.1 The `Event` Library

The `Event` is intended for primary use in the `WITH` expression of an *Event*. The library exposes a set of opaque objects that allow inspection and modification of elements of the target program. All ADT operations in the `Event` library are side-effect free and, therefore, can be used in event predicate expressions or in the statements of their security updates.

```
Object Event.class()
Object Event.method()
Object Event.instruction()
```

The above three operations return opaque reflection objects for the current insertion point—that is, the place of the current action. Depending on the action's place (i.e., the *ActionPlace*), the `Event.instruction()` may return `null`; however, all actions are guaranteed to be placed in classes and all instructions are guaranteed to be placed in methods.

```
Object Event.classSource()
int    Event.basicBlockLen()
Object Event.basicBlockInstr(int ithInstruction)
```

The `Event.classSource` operation returns a URL to the source of the current class, such as, for instance, `file://c:/Main.class`. The remaining two operations can be used with basic block and instruction actions, allowing retrieval of reflective objects for individual instructions in the basic block.[5]

```
boolean Event.classNameIs(Object nameString)
boolean Event.methodNameIs(Object nameString)
boolean Event.methodTypeIs(Object typeString)
boolean Event.instructionIs(Object nameString)
boolean Event.methodPrototypeIs(Object javaPrototypeString)
```

The first four operations above are shorthand for common reflective inspection activity. For example, the first saves the PSLang policy writer from passing the `Event.class()` object to `Reflect.className` and comparing the result with `JVML.strEq`. The fifth operation goes a step further and allows PSLang policy

[5]Note that these operations allow several simple analysis to be directly encoded in PSLang policies; for example, they allow the analysis for the policy of Figure 2.4 that is discussed in a footnote on page 27.

writers to specify method names as in Java prototypes, instead of by using the awkward JVML notation.

A.3.2 The `Reflect` Library

The `Reflect` library provides reflective operations on the attributes of `Event` objects. Like the `Event` library, all ADT operations in the `Reflect` library are side-effect-free. For brevity, not all possible `Reflect` library operations corresponding to JVML structural elements are shown below.

```
Object Reflect.instrName(Object instruction)
Object Reflect.instrClassName(Object instruction)
Object Reflect.instrRefStr(Object instruction)
Object Reflect.instrRefClassName(Object instruction)
Object Reflect.instrRefName(Object instruction)
Object Reflect.instrRefType(Object instruction)
```

The above operations return for an `Event.instruction()` a string of the instruction opcode name or its JVML class name argument if the instruction only operates on classes. Otherwise, if the instruction references a field or method in a class or object, a string with the full class name, `/` separator, and the name and type of the class member can be retrieved. The last three operations individually return the components of class member references; of these three, the last does not return a string, but a `Reflect.type` object, that is to say, an object that may be passed to `Reflect.type` operations.

```
Object  Reflect.methodName(Object method)
Object  Reflect.methodType(Object method)
boolean Reflect.methodIsEmpty(Object method)
```

For an `Event.method()` object, the above operations return a string of the method name (excluding its class or package name), the method's `Reflect.type` object,

and a boolean indicating whether the method is effect-free (i.e., contains only null operations), respectively.

```
Object  Reflect.methodAccess(Object method)
boolean Reflect.methodIsSynchronized(Object method)
boolean Reflect.methodIsNative(Object method)
```

The above operations return the method's `Reflect.access` object, in the first operation, and Booleans, in the next two, that indicate whether the method is synchronized or native (i.e., locked or referring to a non-JVML implementation).

```
Tuple   Reflect.methodExceptions(Object method)
boolean Reflect.methodThrows(Object method,
                             Object exceptionName)
Tuple   Reflect.methodExceptionHandlers(Object method)
```

For methods, a `Tuple` library object of the names of exceptions declared to be thrown by the method can be retrieved. Alternatively, the second operation above allows querying whether the method throws an exception by giving its string name. The third operation allows retrieval of a `Tuple` of the actual exception handlers within the method's code.

```
Object  Reflect.className(Object class)
Object  Reflect.classSuperName(Object class)
Tuple   Reflect.classInterfaces(Object class)
boolean Reflect.classImplements(Object class,
                                Object interfaceName)
```

For an `Event.class()` object, the first operation above returns a string of the class name. The superclass of the class (i.e., the class's immediate ancestor in the class hierarchy) can be retrieved as a string. (The empty string is returned for the root of the class hierarchy.) A `Tuple` library object of the names of interfaces

implemented by the class can be retrieved, and this set can alternatively be queried directly by name.

```
Object  Reflect.classAccess(Object class)
boolean Reflect.classIsInterface(Object class)
Tuple   Reflect.classFields(Object class)
```

Finally, for a class, its `Reflect.access` object can be retrieved, as can its status as an interface (as opposed to a "real" class). The final operation returns a `Tuple` of objects to be accessed by the `Reflect.field` operations below.

```
Object  Reflect.fieldName(Object field)
Object  Reflect.fieldType(Object field)
Object  Reflect.fieldAccess(Object field)
boolean Reflect.fieldIsVolatile(Object field)
boolean Reflect.fieldIsTransient(Object field)
```

For each object representing a class or object field, the name and `Reflect.type` object can be retrieved. The last three operations return the member variable's `Reflect.access` object and Booleans showing whether it's volatile or transient.

```
Object  Reflect.typeStr(Object type)
boolean Reflect.typeIsMethod(Object type)
boolean Reflect.typeIsObject(Object type)
boolean Reflect.typeIsArray(Object type)
```

For each object representing a type, a properly formatted string naming the JVML type can be retrieved. Also, the type can be examined to show whether it's referring to a method call (i.e., a list of formal arguments and a return type), whether it's any type of object, or whether it is an array type.

```
Tuple   Reflect.typeMethodArgTypes(Object methodType)
Object  Reflect.typeMethodRetType(Object methodType)
Object  Reflect.typeObjectClassName(Object objectType)
Object  Reflect.typeArrayType(Object arrayType)
```

For the more complex types, type objects for the actual formal arguments of the method type can be retrieved, as can its return type. For object types, the JVML class name of the object can be examined. Finally, for array types the type of the array elements can be retrieved (note that the element type can also be an array type).

```
boolean Reflect.accessStatic(Object access)
boolean Reflect.accessPublic(Object access)
boolean Reflect.accessPrivate(Object access)
boolean Reflect.accessProtected(Object access)
boolean Reflect.accessPackage(Object access)
boolean Reflect.accessFinal(Object access)
boolean Reflect.accessAbstract(Object access)
```

For access objects, whether the access is static, public, private, protected, package, final, or abstract can be retrieved. Note that these may or may not apply or be combined depending on what JVML structural element they describe access.

A.3.3 The `State` Library

The `State` library allows the examination and modification of the state of JVML structural elements. In this library, only the reading of state is considered side-effect-free—although this might be moot because only the rewriter makes use of this distinction (to direct simplification) and this library is typically used later, at runtime.

Very importantly, although this section only shows the integer version of `State` library operations, appropriate operations exist for all PSLang types—in particular, for objects and floating-point doubles. The syntax for the `State` operation of these other types is as expected. The operations for object instances are useful,

because they allow the addition of `Tuple` and other PSLang data-structure state to elements of the target program.

```
void State.classAddInt(Object class, Object fieldName)
void State.classSetInt(int value, Object classAndFieldName)
int  State.classGetInt(Object classAndFieldName)
```

The above operations allow PSLang policies to add named integer state values to JVML classes in the target program and then to read and write them. The addition operation—which takes an `Event.class()` object and a name for the integer field to be added to that class—must be used before operations that access the integer. This means the addition operation should be invoked in a PSLang update guaranteed to be triggered at runtime before any other updates where the added field may be used; practically, this means the addition should be triggered at a load-time class initialization action. The read and write operations on the added variable must specify the full field name in JVML format (e.g., `/java/io/Foo/barName`) and can be used in all runtime PSLang updates.

```
void State.instanceAddInt(Object class,
                          Object fieldName)
void State.instanceSetInt(Object instance, int value,
                          Object classAndFieldName)
int  State.instanceGetInt(Object instance,
                          Object classAndFieldName)
```

The above three operations add and access integer instance variables in objects. These operations exactly correspond to the `State.class` operations above, and they are used in the same manner. The only difference is an extra argument—of the actual object instance—to the access operations that read and write the added field.

The PSLang policy writer must use these operations to access only object instances whose type is a class where they've added the instance variable (and this can be tricky because fields may be inherited into subclasses). PSLang writers can simplify this, and their life, by sacrificing some performance and checking the object type at runtime using a `JVML` library operation.[6]

```
void State.methodAddInt(Object method, Object variableName)
void State.methodSetInt(int value, Object variableName)
int  State.methodGetInt(Object variableName)
```

Local state can also be added to individual methods, using the above three operations, which are analogous to the `State.class` operations. In these operations, the name of the added variable need not be in JVML member format, but can be an arbitrary string as long as it does not start with $.

`$instance`	Current object instance, if not a static method
`$methodArgK`	K-th argument to the current method (starting at 1)
`$methodRet`	Return value of the current method
`$instrArgK`	K-th runtime argument to this instruction (start at 1)
`$instrRet`	Value returned (on the stack) by the current instruction
`$exception`	Exception object for the current exception (in handlers)

`State.method` operations are also used expose to PSLang certain critical state about PoET method actions. Thus, the operations are overloaded to access state of the methods that existed in the target program, as well as the state added by PoET. This pre-existing state includes, e.g., the method arguments, and is distinguished by being available only at runtime PoET method actions (or actions nested within them at runtime, as discussed in §A.1). This state is described in

[6]One possible enhancement to PoET might be to make all objects in the target program implement an interface allowing access to added state. This would not, however, remove the need for dynamically checking the object instance's type to determine whether the added field has any meaning for the object.

the box above, which also gives the special variable names used by PSLang policies to access this state.

```
Object State.getSpecObject()
```

Finally, in the `State` library, the above operation allows retrieval of the runtime object instance that represents this PSLang policy. This operation is primarily used to implement the `stealthCheck` described in §A.1, as these checks masquerade as object equality tests against this runtime object instance.

Appendix B

PSLang Security Policies

The examples in this appendix are specific to Java 2 and Sun's JDK 1.2.

B.1 IRM Integrity Policy for JVML Bytecodes

```
IMPORT LIBRARY JVML;

//
// PoET defines the unique prefix of names for this IRM instance
//
SIDE-EFFECT-FREE FUNCTION boolean isIRM(Object string) {
    return JVML.strStartsWith(string,#irmInstanceNamePrefix#);
}

ON EVENT at start of loading class initialization
WITH isIRM( Reflect.className(Event.class()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot declare class with IRM prefix" ];
}

ON EVENT at start of loading class initialization
WITH isIRM( Reflect.classSuperName(Event.class()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot inherit from class with IRM prefix" ];
}

ON EVENT at start of loading instruction
WITH Event.instructionIs("new")
     && isIRM( Reflect.instrClassName(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot allocate object with IRM prefix" ];
}

ON EVENT at start of loading instruction
WITH ( Event.instructionIs("invokevirtual")
       || Event.instructionIs("invokeinterface")
       || Event.instructionIs("invokespecial")
       || Event.instructionIs("invokestatic") )
  && isIRM( Reflect.instrRefStr(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot access method with IRM prefix" ];
}

ON EVENT at start of loading instruction
WITH ( Event.instructionIs("putfield")
       || Event.instructionIs("getfield")
       || Event.instructionIs("putstatic")
       || Event.instructionIs("getstatic") )
  && isIRM( Reflect.instrRefStr(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot access field with IRM prefix" ];
}
```

B.2 IRM Integrity Policy for Reflection

```
IMPORT LIBRARY JVML;
IMPORT LIBRARY Tuple;
IMPORT LIBRARY Stack;

//
// PoET defines the unique prefix of names for this IRM instance
//
SIDE-EFFECT-FREE FUNCTION boolean isIRM(Object string) {
    return JVML.strStartsWith(string,#irmInstanceNamePrefix#);
}

//
// Helper: We are restricting methods in java.lang.Class
//
SIDE-EFFECT-FREE FUNCTION boolean isReflection(Object strMethodName) {
    return Event.classNameIs("java/lang/Class")
        && Event.methodNameIs(strMethodName);
}

//
// Prevent access by name to IRM classes
//
ON EVENT at start of method WITH isReflection("forName")
PERFORM SECURITY UPDATE {
    if( isIRM(State.methodGetObject("$methodArg1")) ) {
        JVML.throwException("java/lang/ClassNotFoundException");
    }
}

//
// Prevent access by name to IRM fields and elide them from listings
//
ON EVENT at start of method
WITH isReflection("getField") || isReflection("getDeclaredField")
PERFORM SECURITY UPDATE {
    if( isIRM(State.methodGetObject("$methodArg1")) )
        JVML.throwException("java/lang/NoSuchFieldException");

}
ON EVENT at normal completion of method
WITH isReflection("getFields") || isReflection("getDeclaredFields")
PERFORM SECURITY UPDATE {
    Object oldRet = Tuple.createArray( State.methodGetObject("$methodRet") );
    Object retStack = Stack.create();
    for( int i = 0; i < Tuple.size(oldRet); i = i+1 ) {
        Object f = JVML.typeCast(Tuple.get(oldRet,i),"java/lang/reflect/Field");
        Object name  = JVML.objectInvokeMethod("java/lang/Class","getName",
                                               "()Ljava/lang/String;",f);
        if( ! isIRM(name) ) {
            retStack.push( f );
        }
    }
    Object rs = Stack.toTuple( retStack );
    Object ret = JVML.typeCast(Tuple.toArray(rs),"[Ljava/lang/reflect/Field;");
    State.methodPutObject( "$methodRet", ret );
}
```

B.3 Blocking use of a Java Package

```
IMPORT LIBRARY JVML;

//
// Define the packages and classes we are restricting use of
// This example blocks all use of reflection.
//
SIDE-EFFECT-FREE FUNCTION boolean isBlocked(Object string) {
    if(JVML.strStartsWith(string,"java/lang/Class")) { return true; }
    if(JVML.strStartsWith(string,"java/lang/reflect/")) { return true; }
    return false;
}

ON EVENT at start of loading class initialization
WITH isBlocked( Reflect.className(Event.class()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot declare class in package" ];
}

ON EVENT at start of loading class initialization
WITH isBlocked( Reflect.classSuperName(Event.class()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot inherit from class in package" ];
}

ON EVENT at start of loading instruction
WITH Event.instructionIs("new")
     && isBlocked( Reflect.instrClassName(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot allocate object in package" ];
}

ON EVENT at start of loading instruction
WITH ( Event.instructionIs("invokevirtual")
       || Event.instructionIs("invokeinterface")
       || Event.instructionIs("invokespecial")
       || Event.instructionIs("invokestatic") )
  && isBlocked( Reflect.instrRefStr(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot access method in package" ];
}

ON EVENT at start of loading instruction
WITH ( Event.instructionIs("putfield")
       || Event.instructionIs("getfield")
       || Event.instructionIs("putstatic")
       || Event.instructionIs("getstatic") )
  && isBlocked( Reflect.instrRefStr(Event.instruction()) )
PERFORM SECURITY UPDATE {
    FAIL[ "Cannot access field in package" ];
}
```

B.4 Static Resolution of Virtual Method Calls

```
IMPORT LIBRARY JVML;
IMPORT LIBRARY Association;
IMPORT LIBRARY Lock;

/* Initialize policy state: the virtual method map
*/
GLOBAL SECURITY STATE {
    Object virtualMethodMap = Association.create();
    Object virtualMethodMapLock = Lock.create();
}

/*  Map helpers
 */
SIDE-EFFECT-FREE FUNCTION Object cloneSuperClassMethods(Object class) {
    Object o = Association.get(virtualMethodMap,Reflect.classSuperName(class));
    if( o != null ) {
        return Association.clone(o);
    }
    return Association.create();
}
SIDE-EFFECT-FREE FUNCTION Object getClassMethods(Object className) {
    return Association.get(virtualMethodMap, className);
}

/*
** Each new class inherits all virtual methods from parent class
*/
ON EVENT at start of loading class initialization
PERFORM SECURITY UPDATE {
    Lock.acquire( virtualMethodMapLock );
    Object dad = cloneSuperClassMethods( Event.class() );
    Association.put(virtualMethodMap, Reflect.className(Event.class()), dad);
    Lock.release( virtualMethodMapLock );
}

// For each newly defined method, is it a virtual definition?
// NOTE: Should handle class and object initializers in another policy
SIDE-EFFECT-FREE FUNCTION boolean nonVirtualMethod(Object method) {
    return Reflect.accessStatic(Reflect.methodAccess(method))
        || Reflect.accessFinal(Reflect.methodAccess(method))
        || JVML.strEq(Reflect.methodName(method),"<clinit>")
        || JVML.strEq(Reflect.methodName(method),"<init>");
}
/*
** Each virtual method defined in a class overrides any from a parent
*/
ON EVENT at start of loading method
WITH ! nonVirtualMethod(Event.method())
PERFORM SECURITY UPDATE {
    Lock.acquire( virtualMethodMapLock );
    Object map = getClassMethods( Reflect.className(Event.class()) );
    Association.put( map,
                     Reflect.methodName(Event.method()),
                     Reflect.className(Event.class()) );
    Lock.release( virtualMethodMapLock );
}
```

```
/* For an invokevirtual instruction, its real target code
 */
SIDE-EFFECT-FREE FUNCTION
Object realVirtualCallClassName(Object instruction, Object methodName) {
    Lock.acquire( virtualMethodMapLock );
    Object callClassName = Reflect.instrRefClassName(instruction);
    Object virtualMethods = Association.get(virtualMethodMap, callClassName);
    if( virtualMethods != null ) {
        return Association.get(virtualMethods, methodName);
    }
    Object realName = Association.get(virtualMethods, methodName);
    Lock.release( virtualMethodMapLock );
    return realName;
}
SIDE-EFFECT-FREE FUNCTION
boolean isVirtualCallTo(Object instr, Object cName, Object mName) {
    Object realClassName = realVirtualCallClassName(instr, mName);
    return JVML.strEq(cName, realClassName);
}

/*
** Helper function for users of this policy
*/
FUNCTION void restrictVirtualMethod(Object cName, Object mName) {
    Lock.acquire( virtualMethodMapLock );
    Object methodMap = getClassMethods( cName );
    if( methodMap == null ) {
        methodMap = Association.create();
        Association.put(virtualMethodMap, cName, methodMap);
    }
    Association.put( methodMap, mName, cName );
    Lock.release( virtualMethodMapLock );
}
SIDE-EFFECT-FREE FUNCTION
boolean virtualCallTo(Object instr, Object cName, Object mName) {
    return JVML.strEq( Reflect.instrName(instr), "invokevirtual" )
        && isVirtualCallTo(instr, cName, mName);
}
```

B.5 PSLang Formulation of IRM_{SPS}

```
IMPORT LIBRARY JVML;
IMPORT LIBRARY System;
IMPORT LIBRARY Stack;
IMPORT LIBRARY Java2Permissions;

//
// define the default permissions and doPriv token
//
GLOBAL SECURITY STATE {
    Object doPrivToken = Lock.create();
    Object defaultPerms = Java2Permissions.createAllDomain("");
}

//
// limit most rewriting to application classes and methods
//
SIDE-EFFECT-FREE FUNCTION boolean appClass() {
    return ! JVML.strStartsWith(Reflect.className(Event.class()),"java/");
}
SIDE-EFFECT-FREE FUNCTION boolean appMethod() {
    return appClass() && ! Event.methodNameIs("<clinit>");
}

//
// initialize extra member in loaded classes with default permissions
//
ON EVENT at start of loading class initialization
WITH appClass()
PERFORM SECURITY UPDATE {
    State.classAddObject(Event.class(), "siDomain");
}
ON EVENT at start of class initialization
WITH appClass()
PERFORM SECURITY UPDATE {
    State.classSetObject(defaultPerms, "siDomain");
}

//
// create the domainStack at the start of the application
//
ON EVENT at start of loading class initialization
WITH Event.classNameIs("java/lang/Thread")
PERFORM SECURITY UPDATE {
    State.classAddObject(Event.class(), "domainStack");
}
ON EVENT at start of program
PERFORM SECURITY UPDATE {
    Object stack = Stack.create();
    Object thread = JVML.typeCast(System.currentThread(),"java/lang/Thread");
    State.instanceSetObject(thread, stack, "java/lang/Thread/domainStack");
    System.useSecurityManager();
}

//
// Maintain domainStack at method calls
//
ON EVENT at start of method
WITH appMethod()
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread,"java/lang/Thread/domainStack");
    Stack.push(stack, State.classGetObject("siDomain"));
}
ON EVENT at finally completed method
```

```
WITH appMethod()
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread,"java/lang/Thread/domainStack");
    Object discard = Stack.pop( stack );
}

//
// on doPrivileged, push the doPriv token onto the domainStack
//
SIDE-EFFECT-FREE FUNCTION boolean doPrivilegedCall(Object instr) {
    return Event.instructionIs("invokestatic")
        && JVML.strEq(Reflect.instrRefStr(instr),
                      JVML.strCat("java/security/AccessController/doPrivileged",
                            "(Ljava/security/PrivilegedAction;)Ljava/lang/Object;"));

}
ON EVENT at start of instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread, "java/lang/Thread/domainStack");
    Stack.push( stack, doPrivToken );
}
ON EVENT at finally completed instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread, "java/lang/Thread/domainStack");
    Object discard = Stack.pop( stack );
}
```

```
//
// Check the domainStack on a checkPermission
//
SIDE-EFFECT-FREE FUNCTION boolean checkPermissionCall(Object instr) {
    return Event.instructionIs("invokestatic")
        && JVML.strEq(Reflect.instrRefStr(instr),
                      JVML.strCat("java/security/AccessController/",
                                  "checkPermission(Ljava/security/Permission;)V"));
}
ON EVENT at start of instruction
WITH checkPermissionCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object permissionToCheck = State.methodGetObject("$instrArg1");
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object stack = State.instanceGetObject(thread,"java/lang/Thread/domainStack");
    Object stackCopy = Stack.clone(stack);

    boolean finished = false;
    Object prevDomain = null;
    while( !finished && !Stack.empty(stackCopy) ) {
        Object domain = Stack.pop(stackCopy);
        if( domain == doPrivToken ) {
            finished = true;
            if( !Stack.empty(stackCopy) ) {
                domain = Stack.pop(stackCopy);
            }
        }
        if( domain != null
            && domain != prevDomain
            && ! Java2Permissions.implies(domain, permissionToCheck) )
        {
            FAIL[ JVML.strCat4("DOMAIN ",domain,
                               "DOESN'T IMPLY ",permissionToCheck) ];
        }
        prevDomain = domain;
    }
}

//
// clone the domainStack whenever a thread is created
//
SIDE-EFFECT-FREE FUNCTION boolean newThreadCreation(Object instr) {
    return Event.instructionIs("invokespecial")
        && JVML.strEq(Reflect.instrRefClassName(instr), "java/lang/Thread")
        && JVML.strEq(Reflect.instrRefName(instr), "<init>");
}
ON EVENT at normal completion of instruction
WITH newThreadCreation(Event.instruction())
PERFORM SECURITY UPDATE {
    Object newThread = State.methodGetObject("$instrArg1");
    Object oldT = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object oldS = State.instanceGetObject(oldT, "java/lang/Thread/domainStack");
    Object newStack = Stack.clone( oldS );
    State.instanceSetObject(newThread, newStack, "java/lang/Thread/domainStack");
}
```

B.6 PSLang Formulation for IRM_{Lazy}

```
IMPORT LIBRARY JVML;
IMPORT LIBRARY System;
IMPORT LIBRARY Association;
IMPORT LIBRARY Stack;
IMPORT LIBRARY Tuple;
IMPORT LIBRARY Java2Permissions;

GLOBAL SECURITY STATE {
    Object classToDomain = Association.create();
    Object defaultPerms = Java2Permissions.createDomain("",
                          "java.lang.RuntimePermission:createClassLoader");
    Object domainPerms = Java2Permissions.createAllDomain("");
}

// constant that tells us to ignore top two stack frames
SIDE-EFFECT-FREE FUNCTION int stackSkipPart() {  return 2;  }

//
// Add thread-local security state to java.lang.Thread
//
ON EVENT at start of loading class initialization
WITH Event.classNameIs("java/lang/Thread")
PERFORM SECURITY UPDATE {
    // is this thread a security-relevant thread, i.e., not a deamon thread?
    State.instanceAddInt(Event.class(), "notDeamon");
    // stack of call stacks of ancestral threads, at thread-creation time
    State.instanceAddObject(Event.class(), "parentStacks");
    // stack of doPriv call-frames, always relative to curr thread call stack
    State.instanceAddObject(Event.class(), "privStack");
    // "collapsed" parentStacks, lazily computed at first checkPermission
    State.instanceAddObject(Event.class(), "parentTuple");
}

//
// add domain state for each class at load time, init to correct domain
//
SIDE-EFFECT-FREE FUNCTION boolean appClass() {
    return ! JVML.strStartsWith(Reflect.className(Event.class()),"java/");
}
ON EVENT at start of class initialization
WITH appClass()
PERFORM SECURITY UPDATE {
    Object class = JVML.getClassByName(Reflect.className(Event.class()));
    Association.put(classToDomain, class, domainPerms);
}

//
// set the initial empty stack when the program begins
//
ON EVENT at start of program
PERFORM SECURITY UPDATE {
    // this is a security-relevant thread
    Object thread = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    State.instanceSetInt(thread, 1, "java/lang/Thread/notDeamon");
    // empty ancestral call stack
    Object parents = Stack.create();
    State.instanceSetObject(thread, parents, "java/lang/Thread/parentStacks");
    // empty doPriv stack
    Object privs = Stack.create();
    State.instanceSetObject(thread, privs, "java/lang/Thread/privStack");
    // use security manager
    System.useSecurityManager();
}
```

```
//
// Create parentStacks for child thread: copy parentStacks & add curr stack
//
FUNCTION Object childParentStacks(Object parentStacks, Object thisStack) {
    Object ret = Stack.clone( parentStacks );
    Stack.push( ret, thisStack );
    return ret;
}
//
// Create parentStacks for a child thread constructed inside a doPriv block
//
FUNCTION Object computePrivChildParentStacks(int first, Object thisStack) {
    // get privileged part of stack---keep PoET part of stack: it's expected
    int privSize = Tuple.size(thisStack) - first + 1;
    Object privStack = Tuple.create( privSize ); {
        for(int i = 0; i < privSize; i = i+1) {
            Tuple.put(privStack, i, Tuple.get(thisStack, i));
        }
    }
    // return a stack of one elementStack
    Object newDads = Stack.create();
    Stack.push( newDads, privStack );
    return newDads;
}
//
// combine ancestral "stack of stacks" into one tuple of parent stack frames
//
FUNCTION Object computeParentTuple(Object parentStacks) {
    // compute size of collapsed stack, always ignoring PoET at each bottom
    Object reverseStacks = Stack.create();
    int size = 0; {
        while( ! Stack.empty(parentStacks) ) {
            Object elementStack = Stack.pop( parentStacks );
            size = size + Tuple.size(elementStack) - stackSkipPart();
            Stack.push( reverseStacks, elementStack );
        }
    }
    // copy into return tuple, and re-create parentStacks from reverseStacks
    Object ret = Tuple.create(size); {
        int pos = size - 1;
        while( ! Stack.empty(reverseStacks) ) {
            Object elementStack = Stack.pop( reverseStacks );
            int elementSize = Tuple.size(elementStack);
            for(int i = elementSize-1; i >= stackSkipPart(); i=i-1) {
                Tuple.put(ret, pos, Tuple.get(elementStack, i));
                pos = pos - 1;
            }
            Stack.push( parentStacks, elementStack );
        }
    }
    return ret;
}

//
// clone threadStack whenever a thread is created
// NOTE: This follows Sun's Java2 implementation so a thread created inside
//       doPriv block is privileged for all of its lifetime
// NOTE: A thread created inside a doPriv never has to check it's parents'
//       stack frames before the doPriv, hence we trim that prefix away
//
SIDE-EFFECT-FREE FUNCTION boolean newThreadCreation(Object instr) {
    return Event.instructionIs("invokespecial")
        && JVML.strStartsWith(Reflect.instrRefStr(instr),"java/lang/Thread/<init>");
}
```

```
ON EVENT at normal completion of instruction
WITH newThreadCreation(Event.instruction())
PERFORM SECURITY UPDATE {
    // get security state in current thread
    Object this = JVML.typeCast(System.currentThread(),"java/lang/Thread");
    Object dads = State.instanceGetObject(this,"java/lang/Thread/parentStacks");
    Object privs = State.instanceGetObject(this,"java/lang/Thread/privStack");

    // create and initialize security state in new thread
    Object child = State.methodGetObject("$instrArg1");
    Object newPrivs = Stack.create();
    State.instanceSetObject(child, newPrivs, "java/lang/Thread/privStack");
    State.instanceSetInt(child, 1, "java/lang/Thread/notDeamon");

    // compute child's parentStacks---use only curr stack if we're in doPriv
    Object thisStack = System.stackTrace();
    if( Stack.empty(privs) ) {
        Object ndads = childParentStacks(dads, thisStack);
        State.instanceSetObject(child, ndads, "java/lang/Thread/parentStacks");
    } else {
        int privEnd = JVML.toInt( Stack.peek(privs) );
        Object ndads = computePrivChildParentStacks(privEnd, thisStack);
        State.instanceSetObject(child, ndads, "java/lang/Thread/parentStacks");
    }
}

//
// Check permissions
//
SIDE-EFFECT-FREE FUNCTION boolean checkPermissionCall(Object instr) {
    return Event.instructionIs("invokestatic")
        && JVML.strEq(Reflect.instrRefStr(instr),
                      JVML.strCat("java/security/AccessController/",
                                  "checkPermission(Ljava/security/Permission;)V"));
}
ON EVENT at start of instruction
WITH checkPermissionCall(Event.instruction())
PERFORM SECURITY UPDATE {
    // get thread-local data
    Object this = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    int notDeamon = State.instanceGetInt(this,"java/lang/Thread/notDeamon");
    if( notDeamon > 0 ) {
        Object privs = State.instanceGetObject(this,"java/lang/Thread/privStack");
        Object curr = System.stackTrace();
        int currSize = Tuple.size(curr);
        Object toCheck = State.methodGetObject("$instrArg1");
        // if not privileged, do normal check on current & ancestral stacks
        if( Stack.empty( privs ) ) {
            checkStack(curr, stackSkipPart, currSize, toCheck);
            Object d = State.instanceGetObject(this,"java/lang/Thread/parentTuple");
            if( d == null ) { // lazily compute collapsed version of dads
                d = State.instanceGetObject(this, "java/lang/Thread/parentStacks");
                d = computeParentTuple( d );
                State.instanceSetObject(this, d, "java/lang/Thread/parentTuple");
            }
            if( Tuple.size(d) > 0 ) {  // if dadSize > 0
                checkStack(d, 0, dadSize, toCheck);
            }
        }
        // if privileged, do simple check on the current stack only
        else {
            int privEnd = currSize - JVML.toInt(Stack.peek(privs)) + 1;
            checkStack(curr, stackSkipPart, privEnd, toCheck);
        }
    }
}
```

```
// FAIL inside a doPriv block, so we are sure to have exitVM privilege
//
FUNCTION void stackFail(Object domain, Object permissionToCheck) {
    Object this = JVML.typeCast(System.currentThread(), "java/lang/Thread");
    Object privs = State.instanceGetObject(this,"java/lang/Thread/privStack");
    int privStackFrameNumber = Tuple.size(System.stackTrace()) - stackSkipPart;
    Stack.push(privs, JVML.intToObject(privStackFrameNumber));
    FAIL[ JVML.strCat4("DOMAIN ",domain," DOESN'T IMPLY ",permissionToCheck) ];
}
// Check the permissions of all domains
//
FUNCTION void checkStack(Object stack, int first, int last, Object toCheck) {
    Object prevDomain = null; // skip domain and class repetitions
    Object prevClass = null;
    for(int i = first; i < last; i=i+1) {
        Object class = Tuple.get(stack,i);
        if( class != prevClass ) {
            if( verbose ) {
                Object className = JVML.getNameOfClass(class);
                System.printStr(className);
            }
            Object domain = Association.get(classToDomain, class);
            if( domain != null  {
                && domain != prevDomain
                && ! Java2Permissions.implies(domain, toCheck) )
            {
                stackFail(domain,toCheck);
            }
            prevClass = class;
            prevDomain = domain;
        }
    }
}

//
// push a privileged bit on doPriv calls
//
SIDE-EFFECT-FREE FUNCTION boolean doPrivilegedCall(Object instr) {
    return Event.instructionIs("invokestatic")
        && JVML.strEq(Reflect.instrRefStr(instr),
                      JVML.strCat("java/security/AccessController/doPrivileged",
                            "(Ljava/security/PrivilegedAction;)Ljava/lang/Object;"));

}
ON EVENT at start of instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object this = JVML.typeCast(System.currentThread(),"java/lang/Thread");
    int notDeamon = State.instanceGetInt(this,"java/lang/Thread/notDeamon");
    if( notDeamon > 0 ) {
        Object privs = State.instanceGetObject(this,"java/lang/Thread/privStack");
        int privStackFrameNumber = Tuple.size(System.stackTrace()) - stackSkipPart;
        Stack.push(privs, JVML.intToObject(privStackFrameNumber));
    }
}
ON EVENT at finally completed instruction
WITH doPrivilegedCall(Event.instruction())
PERFORM SECURITY UPDATE {
    Object this = JVML.typeCast(System.currentThread(),"java/lang/Thread");
    int notDeamon = State.instanceGetInt(this,"java/lang/Thread/notDeamon");
    if( notDeamon > 0 ) {
        Object privs = State.instanceGetObject(this,"java/lang/Thread/privStack");
        Object discard = Stack.pop( privs );
    }
}
```

BIBLIOGRAPHY

[AAB+00] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litvinov, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S.E. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[Aba98] M. Abadi. Protection in programming-language translations. In *Proc. of 1998 Colloquium on Automata, Languages and Programming*, pages 868–883. Springer-Verlag, July 1998.

[AF03] M. Abadi and C. Fournet. Access control based on execution history. In *Proc. of 2003 Symposium on Network and Distributed System Security*, February 2003.

[AG03] A. Appel and S. Govindavajhala. Using memory errors to attack a virtual machine. In *Proc. of 2003 IEEE Symposium on Security and Privacy*, May 2003.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Ame81] S.R. Ames, Jr. Security kernels: A solution or a problem. In *Proc. of 1981 IEEE Symposium on Security and Privacy*, pages 141–150, May 1981.

[And72] J.P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), October 1972.

[And99] J. Anders. Java MPEG player. Technical report, Fakultät für Informatik, Technische Universität Chemnitz, 1999.

[AS87] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed computing*, 2:117–126, 1987.

[Asl95] T. Aslam. A taxonomy of security faults in the UNIX operating system. Master's thesis, Dept. of Computer Science, Purdue University, August 1995.

[ASU85] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[ATLLW96] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe. Efficient and language-independent mobile programs. In *Proc. of 1996 Symposium on Programming Language Design and Implementation (PLDI'96)*, pages 127–136, May 1996.

[BD96] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, Spring 1996.

[BEL+00] M. Burrows, Ú. Erlingsson, S-T.A. Leung, M.T. Vandevoorde, C.A. Waldspurger, K. Walker, and W.E. Weihl. Efficient and flexible value sampling. In *Proc. of 2000 ACM Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[BH78] R. Bibsey and D. Hollingworth. Protection analysis project final report. Technical Report ISI-RR-78-13, Institute of Information Sciences, University of Southern California, 1978.

[Bir89] A. Birrell. An introduction to programming with threads. Technical Report SRC Research Report 35, Digital Equipment Corporation, January 1989.

[Bir03] A. Birrell. An introduction to programming with C# threads. Technical report, Microsoft Corporation, May 2003. Manuscript available at http://research.microsoft.com/~birrell/papers/ThreadsCSharp.pdf.

[BL01] A. Bernard and P. Lee. Enforcing formal security properties. Technical Report CMU-CS-01-121, School of Computer Science, Carnegie Mellon University, April 2001.

[BLW02a] L. Bauer, J. Ligatti, and D. Walker. A calculus for composing security policies. Technical Report TR-655-02, Department of Computer Science, Princeton University, August 2002.

[BLW02b] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Proc. Workshop on Computer Security Foundations*, July 2002.

[BLW02c] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In *Software Security–Theories and Systems. Proc. Mext-NSF-JSPS International Symposium*, November 2002.

[BLW03] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2003. To appear.

[BN89] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proc. of 1989 IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[Boe99] B. Boehm. Managing software productivity and reuse. *IEEE Computer*, 32(9):111–113, September 1999.

[BR02] T. Ball and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. Principles of Programming Languages (POPL'2002)*, pages 1–3, January 2002.

[BS96] A. Baird-Smith. Jigsaw: An Object Oriented Server. Technical Report W3C Note, World Wide Web Consortium, MIT Laboratory for Computer Science, June 1996. http://www.w3.org/Jigsaw/.

[BS01] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *Proc. of OOPSLA 2001 Workshop on Specification and Verification of Component-based Systems*, October 2001.

[BS02] D. Box and C. Sells. *Essential .NET, Volume I: The Common Language Runtime*. Addison-Wesley, 2002.

[BSP+95] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. Symposium on Operating System Principles (SOSP'95)*, pages 267–284. ACM Press, December 1995.

[CBR03] W.R. Cheswick, S.M. Bellovin, and A.D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 2 edition, 2003.

[CCK98] G. Cohen, J. Chase, and D. Kaminsky. Automatic program transformation with JOIE. In *Proc. of 1998 USENIX Annual Technical Symposium*, pages 167–178, June 1998.

[CER99a] Computer Emergency Response Team Coordination Center CERT. CERT advisory CA-99-04 Melissa macro virus. Technical Report CA-99-04, Software Engineering Institute, Carnegie Mellon University, March 1999.

[CER99b] Computer Emergency Response Team Coordination Center CERT. CERT advisory CA-99-06 ExploreZip Trojan horse program. Technical Report CA-99-06, Software Engineering Institute, Carnegie Mellon University, June 1999.

[CER00] Computer Emergency Response Team Coordination Center CERT. Results of the security in activex workshop. Technical report, Software Engineering Institute, Carnegie Mellon University, August 2000.

[CO65] E.L. Glaser J.F. Couleur and G.A. Oliver. System design of a computer for time sharing applications. In *AFIPS Conf. Proc.*, volume 28, pages 197–202. Spartan Books, 1965.

[CvE98] G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. In *Proc. of 1998 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 21–35, October 1998.

[CW87] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proc. of 1987 IEEE Symposium on Security and Privacy*, pages 184–194, May 1987.

[DAK00] R. Dimpsey, R. Arora, and K. Kuiper. Java server performance. *IBM System Journal*, 39(1), February 2000.

[Dav65] M. Davis. *The undecidable. Basic papers on undecidable propositions, unsolvable problems and computable functions.* Raven Press, 1965.

[DD77] D.E. Denning and P.J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[DFW96] D. Dean, E.W. Felten, and D.S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. of 1996 IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[DG71] P. Deutsch and C.A. Grant. A flexible measurement tool for software systems. In *Information Processing (Proc. of the IFIP Congress)*, pages 320–326, 1971.

[DLP79] R. Demillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems. *Communications of the ACM*, 22(5):271, May 1979.

[DoD85] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria, Department of Defense 5200.28-STD, the "Orange Book.".* National Computer Security Center, December 1985.

[EAC98] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proc. 5th Conf. on Computer & Communications Security*, May 1998.

[EKO95] D.R. Engler, M.F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of Symposium on Operating System Principles (SOSP'95)*, pages 251–266. ACM Press, December 1995.

[EKR96] Ú. Erlingsson, M. Krishnamoorthy, and T.V. Raman. Efficient multiway radix search trees. *Information Processing Letters*, 60(3):115–120, 1996.

[End98] T. Endres. Java Tar Package. Technical report, ICE Engineering, Inc., 1998. http://www.ice.com/java/tar/.

[ES99] Ú. Erlingsson and F.B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. 1999 New Security Paradigms Workshop*. ACM Press, September 1999.

[ES00] Ú. Erlingsson and F.B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of 2000 IEEE Symposium on Security and Privacy*, May 2000.

[ET99] D. Evans and A. Twyman. Policy-directed code safety. In *Proc. of 1999 IEEE Symposium on Security and Privacy*, May 1999.

[FBF99] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with Generic Software Wrappers. In *Proc. of 1999 IEEE Symposium on Security and Privacy*, pages 2–16, May 1999.

[FG02] C. Fournet and A.D. Gordon. Stack inspection: Theory and variants. In *Proc. Principles of Programming Languages (POPL'2002)*, pages 307–318, January 2002.

[Fla98] D. Flanagan. *JavaScript Pocket Reference*. O'Reilly & Associates, Inc., Newton, MA, 1998.

[Gar03] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proc. of 2003 Symposium on Network and Distributed System Security*, February 2003.

[GB98] R. Grimm and B.N. Bershad. Providing policy-neutral and transparent access control in extensible systems. Technical Report UW-CSE-98-02-02, University of Washington, Dept. of Computer Science, February 1998.

[GD72] G.S. Graham and P.J. Denning. Protection — Principles and Practice. In *Proc. SJCC*, volume 40, pages 417–429, 1972.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GMS77] C.M. Geschke, J.H. Morris, and E.H. Satterthwaite. Early experience with MESA. *Communication of the ACM*, 8(20):540–552, August 1977.

[Göd31] K. Gödel. Uber Formal Unentscheidbare Satze der *Principia Matematica* und verwandter Systeme I. *Monatshefte fur Matematik und Physik*, 38:173–98, 1931. Translation by E. Mendelsohn printed in [Dav65].

[Gon99] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, 1999.

[Gra97] R. Gray. *Agent Tcl: A flexible and secure mobile-agent system*. PhD thesis, Dept. of Computer Science, Dartmouth College, June 1997. Dartmouth Computer Science Technical Report TR98-327.

[GS91] S. Garfinkel and E. Spafford. *Practical Unix Security*. O'Reilly & Associates, Inc., 1991.

[HF03] T. Harris and K. Fraser. Language support for lightweight transaction. In *Proc. of 2003 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2003.

[Hin98] H.M. Hinton. Composing partially-specified systems. In *Proc. of 1998 IEEE Symposium on Security and Privacy*, pages 27–37, May 1998.

[HJ92] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 USENIX Technical Conference*, pages 125–136, September 1992.

[HLP98] R. Harper, P. Lee, and F. Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, Carnegie Mellon University, School of Computer Science, January 1998.

[HMS03] K.W. Hamlen, G. Morrisett, and F.B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR 2003-1908, Cornell University, Dept. of Computer Science, August 2003.

[How97] J.D. Howard. *An Analysis of Security Incidents On The Internet 1989–1995*. PhD thesis, Carnegie Mellon University, April 1997. http://www.cert.org/research/JHThesis/.

[HS94] S.P. Harbison and G.L. Steele. *C: A Reference Manual*. Prentice Hall, 4 edition, 1994.

[HU69] J.E. Hopcroft and J.D. Ullman. *Formal Languages and their relation to Automata*. Addison-Wesley, 1969.

[INT94] INTEL. *Pentium Family User's Manual Volume 3: Architecture and Programming Manual*. Intel Corporation, Mt. Prospect, Illinois, 1994.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[JK99] B. Joy and K. Kennedy, editors. *Information Technology Research: Investing in Our Future*. President's Information Technology Advisory Committee, February 1999.

[KBA02] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. 11th USENIX Security Symposium*, 2002.

[KF91] N.L. Kelem and R.J. Feiertag. A separation model for virtual machine monitors. In *Proc. of 1990 IEEE Symposium on Security and Privacy*, pages 2–19, May 1991.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, and C. Lopes. Aspect-oriented programming. In *Proc. of 1997 European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[Koz98] D. Kozen. Efficient code certification. Technical Report TR98-1661, Cornell University, Dept. of Computer Science, January 1998.

[Koz99] D. Kozen. Language-based security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.

[KS94] S. Kumar and E. Spafford. An application of pattern matching in intrusion detection. In *Proc. 17th National Computer Security Conference*, pages 11–21, October 1994.

[LaD96] M. LaDue. Hostile applets home page, 1996. Reliable Software Technologies, Sterling, VA, `http://www.rstcorp.com/`.

[Lam69] B.W. Lampson. Dynamic protection structures. In *AFIPS Conf. Proc.*, volume 35, pages 27–38, 1969.

[Lam74] B.W. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[Lam83] B.W. Lampson. Hints for computer system design. *ACM Operating Systems Review*, 15(5):18–24, October 1983.

[Lam00] B.W. Lampson. Computer security in the real world. In *18th Annual Computer Security Applications Conference*, December 2000.

[Lam01] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, December 2001.

[Lam03] B.W. Lampson. Software components: Only the giants survive. In *Computer Systems: Papers for Roger Needham*, pages 113–121, February 2003.

[LB98] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. In *Proc. of 1998 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 36–44, October 1998.

[LBMC94] C.E. Landwehr, A.R. Bull, J.P. McDermott, and W.S. Choi. A taxonomy of computer program security flaws, with examples. *ACM Computing Surveys*, 26(3):221–254, September 1994.

[LJ97] U. Lindqvist and E. Jonsson. How to systematically classify computer security intrusions. In *Proc. of 1997 IEEE Symposium on Security and Privacy*, pages 154–163, May 1997.

[LS81] B.W. Lampson and H. Sturgis. Atomic transactions. *Distributed Systems-Architecture and Implementation*, 1981.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[McL90] J. McLean. Security models and information flow. In *Proc. of 1990 IEEE Symposium on Security and Privacy*, pages 180–189, May 1990.

[McL94] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. of 1994 IEEE Symposium on Security and Privacy*, pages 79–93, May 1994.

[MHS00] G. Morrisett, R. Harper, and F.B. Schneider. A language-based approach to security. *Lecture Notes in Computer Science*, 2000:86–101, 2000.

[Mic00] Microsoft. *Microsoft Office 2000 Visual Basic Language Reference*. Microsoft Corporation, Redmond, WA, 2000.

[Mor95] G. Morrisett. *Compiling with Types*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, December 1995. CMU Technical Report CMU-CS-95-226.

[MTC+96] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Proc. Workshop on Compiler Support for Systems Software*, February 1996.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, pages 85–97, January 1998.

[Mye99] A.C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Lab. for Computer Science, Massachusetts Institute of Technology, January 1999. Technical Report MIT/LCS/TR-783.

[Nec97] G. Necula. Proof-carrying code. In *Proc. Principles of Programming Languages (POPL'97)*, pages 106–119, January 1997.

[Nec98] G. Necula. *Compiling with Proofs*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, December 1998. CMU Technical Report CMU-CS-98-154.

[Nec00] G.C. Necula. Translation validation for an optimizing compiler. In *Proc. of 2000 Symposium on Programming Language Design and Implementation (PLDI'98)*, pages 83–94, June 2000.

[Nel91] G. Nelson, editor. *System Programming in Modula-3*. Prentice Hall, 1991.

[NL98] G.C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proc. of 1998 Symposium on Programming Language Design and Implementation (PLDI'98)*, June 1998.

[OLW96] J.K. Ousterhout, J.Y. Levy, and B.B. Welch. The Safe-Tcl Security Model, 1996. Sun Microsystems Laboratories, Mountain View, CA.

[Org73] E.I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[OT97] P.W. O'Hearn and R.D. Tennent, editors. *Algol-like Languages*. Progress in Theoretical Computer Science. Birkhauser, 1997.

[Ous98] J.K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[PH98] R. Pandey and B. Hashii. Providing fine-grained access control for mobile programs through binary editing. Technical Report TR98-08, University of California, Davis, Dept. of Computer Science, August 1998.

[Pov99] D. Povey. Optimistic security: A new access control paradigm. In *Proc. 1999 New Security Paradigms Workshop*. ACM Press, September 1999.

[RHJ98] D. Raggett, A. Le Hors, and I. Jacobs. HTML 4.0 Specification. Technical Report W3C Recommendation, World Wide Web Consortium, MIT Laboratory for Computer Science, April 1998.

[RLV+96] T.H. Romer, D. Lee, G.M. Voelker, A. Wolman, W.A. Wong, J. Baer, B.N. Bershad, and H.M. Levy. The structure and performance of interpreters. *ACM SIGPLAN Notices*, 31(9):150–159, September 1996.

[Ros96] J. Roskind. Evolving the security model for Java from Navigator 2.x to Navigator 3.x., August 1996. Netscape Communications Corporation, Mountain View, CA.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[RT78] D.M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, 57(6 (part 2)):1905+, 1978.

[Rul98] J. Rule. *Dynamic HTML: the HTML developer's guide*. Addison-Wesley, Reading, MA, 1998.

[SA99] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[Sal74] J.H. Saltzer. Information protection and the control of sharing in the Multics system. *Communications of the ACM*, 17(7):388–402, July 1974.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[Sch90] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sch97] F.B. Schneider. *On Concurrent Programming*. Springer Verlag, 1997.

[Sch99a] F.B. Schneider, editor. *Trust in Cyberspace*. Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, National Research Council. National Academy Press, 1999.

[Sch99b] B. Schneier. Inside risks: The Trojan horse race. *Communications of the ACM*, 42(9):128, September 1999.

[Sch00] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000. Previous version published as Cornell University Technical Report TR98-1664, January, 1998.

[Sch03] F.B. Schneider. Least privilege and more. In *Computer Systems: Papers for Roger Needham*, pages 209–214, February 2003.

[SCK+92] R.L. Sites, A. Chernoff, M.B. Kirk, M.P. Marks, and S.G. Robinson. Binary translation. *Digital Technical Journal*, 4(4), 1992. Special Issue: Alpha AXP Architecture and Systems.

[SD02] K. Scott and J.W. Davidson. Safe virtual execution using software dynamic translation. In *18th Annual Computer Security Applications Conference*, December 2002.

[SE94] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report WRL Research Report 94/2, Digital Equipment Corporation, March 1994.

[SESS96] M.I. Seltzer, Y. Endo, C. Small, and K.A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 213–227, October 1996.

[SGB+98] E.G. Sirer, R. Grimm, B. Bershad, A. Gregory, and S. McDirmid. Distributed virtual machines: A system architecture for network computing. In *Proc. of 8th ACM SIGOPS European Workshop*, pages 13–16, September 1998.

[SGGB99] E.G. Sirer, R. Grimm, A.J. Gregory, and B.N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, pages 202–216, December 1999.

[Sho97] W.R. Shockley. Is the Reference Monitor concept fatally flawed? The case for the negative. In *Proc. of 1997 IEEE Symposium on Security and Privacy*, pages 6–7, May 1997.

[Sib96] W.O. Sibert. Malicious data and computer security. In *Proc. of 19th National Information Systems Security Conference*, October 1996.

[Sma97] C. Small. A tool for constructing safe extensible C++ systems. In *Proc. 3rd Conference on Object-Oriented Technologies and Systems*, June 1997.

[SR99] D.A. Solomon and M.E. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 3 edition, 1999.

[SRC84] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9):1278–1308, September 1975.

[SS84] M. Schaefer and R.R. Schell. Towards an understanding of extensible architectures for evaluated trusted computer system products. In *Proc. of 1984 IEEE Symposium on Security and Privacy*, pages 154–163, May 1984.

[SS96] C. Small and M.I. Seltzer. A comparison of OS extension technologies. In *Proc. 1996 USENIX Technical Conference*, pages 41–54, January 1996.

[Ste90] Guy L. Steele, Jr. *Common LISP: the language*. Digital Press, 2 edition, 1990.

[Ste91] D.F. Sterne. On the buzzword "security policy". In *Proc. of 1991 IEEE Symposium on Security and Privacy*, pages 219–230, May 1991.

[SV93] S.McCanne and V.Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of 1993 USENIX Winter Conference*, pages 259–270, January 1993.

[SVB+03] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proc. of 19th Symposium on Operating Systems Principles*, pages 15–28, October 2003.

[SVL01] J. Sugerman, G. Venkitachalam, and B-H. Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proc. of 2001 USENIX Annual Technical Symposium*, pages 1–14, June 2001.

[Tan76] A.S. Tanenbaum. In defense of program testing, or correctness proofs considered harmful. *ACM SIGPLAN Notices*, II(5):64–68, May 1976.

[Tan92] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992.

[Tas81] P.S. Tasker. Trusted computer systems. In *Proc. of 1981 IEEE Symposium on Security and Privacy*, pages 99–100, May 1981.

[Thi03] P. Thiemann. Program specialization for execution monitoring. *Journal of Functional Programming*, 13(3):573–600, May 2003.

[TW89] P. Terry and S. Wiseman. A 'new' security policy model. In *Proc. of 1989 IEEE Symposium on Security and Privacy*, pages 215–227, May 1989.

[Vig98] G. Vigna, editor. *Mobile Agents Security*. Lecture Notes in Computer Science. Springer-Verlag, 1998.

[Vis00] M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, Dept. of Computer Science, University of Pennsylvania, December 2000.

[Wag99] D. Wagner. Janus: An approach for confinement of untrusted applications. Master's thesis, Dept. of Computer Science, University of California, Berkeley, 1999. UCB Technical Report CSD-99-1056.

[Wal99a] D. Walker. A type system for expressive security policies. In *Proc. 1999 Federated Logic Conference Workshop on Run-time Result Verification*, June 1999.

[Wal99b] D.S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Dept. of Computer Science, Princeton University, January 1999.

[WF98] D.S. Wallach and E.W. Felten. Understanding Java stack inspection. In *Proc. of 1998 IEEE Symposium on Security and Privacy*, pages 52–63, May 1998.

[Wil92] P.R. Wilson. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management*, Saint-Malo (France), 1992. Springer-Verlag.

[WLAG93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault isolation. *Operating System Review*, 27(5), 1993.

[WN79] M.V. Wilkes and R.M. Needham, editors. *The Cambridge CAP computer and its operating system*. Operating and Programming System Series. Elsevier, North Holland, 1979.

[Wod10] P.G. Wodehouse. *Psmith in the City*. Adam & Charles Black, 1910.

[Woe94] J. Woehr. What's GNU? *Embedded Systems Programming*, 7(1):70–72, 74, January 1994.

[WZL03] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. of 2003 ACM International Conference on Functional Programming*, August 2003.

[YL96] F. Yellin and T. Lindholm. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[ZL97] A. Zakinthinos and E.S. Lee. A general theory of security properties. In *Proc. of 1997 IEEE Symposium on Security and Privacy*, pages 94–102, May 1997.