KTH Computer Science
and Communication

Inlined Reference Monitors: Certification, Concurrency and Tree Based Monitoring

ANDREAS LUNDBLAD

Doctoral Thesis
Stockholm, Sweden 2013

TRITA-CSC-A 2013:01
ISSN-1653-5723
ISRN-KTH/CSC/A–13/01-SE
ISBN 978-91-7501-654-2

KTH CSC TCS
SE-100 44 Stockholm
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie doktorsexamen i datalogi 15 mars 2013, 10:00 i F3, Kungl Tekniska Högskolan, Stockholm.

Tryck: E-print

Abstract

Reference monitor inlining is a technique for enforcing security policies by injecting security checks into the untrusted software in a style similar to aspect-oriented programming. The intention is that the injected code enforces compliance with the policy (security), without adding behavior (conservativity) or affecting existing policy compliant behavior (transparency).

This thesis consists of four papers which covers a range of topics including formalization of monitor inlining correctness properties, certification of inlined monitors, limitations in multithreaded settings and extensions using data-flow monitoring.

The first paper addresses the problem of having a potentially complex program rewriter as part of the trusted computing base. By means of proof-carrying code we show how the inliner can be replaced by a relatively simple proof-checker. This technique also enables the use of monitor inlining for quality assurance at development time, while minimizing the need for post-shipping code rewrites.

The second paper focuses on the issues associated with monitor inlining in a concurrent setting. Specifically, it discusses the problem of maintaining transparency when introducing locks for synchronizing monitor state reads and updates. Due to Java's relaxed memory model, it turns out to be impossible for a monitor to be entirely transparent without sacrificing the security property. To accommodate for this, the paper proposes a set of new correctness properties shown to be realistic and realizable.

The third paper also focuses on problems due to concurrency and identifies a class of *race-free* policies that precisely characterizes the set of inlineable policies. This is done by showing that inlining of a policy outside this class is either not secure or not transparent, and by exhibiting a concrete algorithm for inlining of policies inside the class which is secure, conservative, and transparent. The paper also discusses how certification in the style of proof-carrying code could be supported in multithreaded Java programs.

The fourth paper formalizes a new type of *data centric* runtime monitoring which combines monitor inlining with taint tracking. As opposed to ordinary techniques which focus on monitoring linear flows of events, the approach presented here relies on tree shaped traces. The paper describes how the approach can be efficiently implemented and presents a denotational semantics for a simple "while" language illustrating how the theoretical foundations is to be used in a practical setting.

Each paper is concluded by a practical evaluation of the theoretical results, based on a prototype implementation and case studies on real-world applications and policies.

Sammanfattning

Referensmonitorinvävning, eller monitorinvävning, är en teknik som används för att se till att en given säkerhetspolicy efterföljs under exekvering av potentiellt skadlig kod. Tekniken går ut på att bädda in en uppsättning säkerhetskontroller (en säkerhetsmonitor) i koden på ett sätt som kan jämföras med aspektorienterad programmering. Syftet med den invävda monitorn är att garantera att policyn efterföljs (säkerhet) utan att påverka ursprungsprogrammets beteende, såvida det följer policyn (transparans och konservativitet).

Denna avhandling innefattar fyra artiklar som tillsammans täcker in en rad ämnen rörande monitorinvävning. Bland annat diskuteras formalisering av korrekthetsegenskaper hos invävda monitorer, certifiering av invävda monitorer, begränsningar i multitrådade program och utökningar för hantering av dataflödesmonitorering.

Den första artikeln behandlar problemen associerade med att ha en potentiellt komplex programmodifierare som del i den säkerhetskritiska komponenten av ett datorsystem. Genom så kallad bevisbärande kod visar vi hur en monitorinvävare kan ersättas av en relativt enkel beviskontrollerare. Denna teknik möjliggör även användandet av monitorinvävning som hjälpmedel för programutvecklare och eliminerar behovet av programmodifikationer efter att programmet distribuerats.

Den andra artikeln fokuserar på problemen kring invävning av monitorer i multitrådade program. Artikeln diskuterar problemen kring att upprätthålla transparans trots införandet av lås för synkronisering av läsningar av och skrivningar till säkerhetstillståndet. På grund av Javas minnesmodell visar det sig dock omöjligt att bädda in en säkerhetsmonitor på ett säkert och transparent sätt. För att ackommodera för detta föreslås en ny uppsättning korrekthetsegenskaper som visas vara realistiska och realiserbara.

Den tredje artikeln fokuserar även den på problemen kring flertrådad exekvering och karaktäriserar en egenskap för en policy som är tillräcklig och nödvändig för att både säkerhet och transparens ska uppnås. Detta görs genom att visa att en policy utan egenskapen inte kan upprätthållas på ett säkert och transparent sätt, och genom att beskriva en implementation av en monitorinvävare som är säker och transparent för en policy som har egenskapen. Artikeln diskuterar också hur certifiering av säkerhetsmonitorer i flertrådade program kan realiseras genom bevisbärande kod.

Den fjärde artikeln beskriver en ny typ av datacentrisk säkerhetsmonitorering som kombinerar monitorinvävning med dataflödesanalys. Till skillnad mot existerande tekniker som fokuserar på linjära sekvenser av säkerhetskritiska händelser förlitar sig tekniken som presenteras här på trädformade händelsesekvenser. Artikeln beskriver hur tekniken kan implementeras på ett effektivt sätt med hjälp av abstraktion.

Varje artikel avslutas med en praktisk evaluering av de teoretiska resultaten baserat på en prototypimplementation och fallstudier av verkliga program och säkerhetsegenskaper.

Acknowledgements

First and foremost, I would like to take the opportunity to thank all those who have supported me and encouraged me during the past six years.

I owe my deepest gratitude to my adviser professor Mads Dam for all his help and guidance during my time as a graduate student. I want to thank him for his confidence in me and my work and for always finding the time and patience to stimulate me and keep me on track. Mads, with his deep knowledge of the subject and academical experience have been truly invaluable to me and I am honored and proud to have had Mads as my adviser.

I also want to thank Dilian Gurov for introducing me to the subject of formal methods. If it was not for his excellent teaching and inspiring course on semantics for programming languages, I would most likely never have pursued a doctoral degree in computer science to begin with.

I thank all my co-authors; Mads Dam, Frank Piessens, Bart Jacobs and Gurvan Le Guernic for all great discussions and fruitful collaboration.

My office mates, Fei Niu and Muddassar Azam Sindhu and my former ones Mika Cohen and Irem Aktug–I have really enjoyed your company. Thank you. Musard Balliu, Torbjörn Granlund, Karl Palmskog, Gunnar Kreitz, Oliver Schwartz, Siavash Soleimanifard, Pedro de Carvalho Gomes, Benjamin Greschbach, Emma Enström, thanks for providing such friendly and enjoyable atmosphere here at the department.

David Wennström, you are a better friend than I could ever wish for. I have shared more laughs and technically interesting discussions with you than with any one else during my graduate studies. Thank you.

A special thanks goes to my father-in-law Professor Björn Bergenståhl for all the discussions and advice on research in general.

I would like to thank mom, dad and my sister for all your warmth, encouragement and support the past years.

Tora–without your patience, your support and your love I would never have come this far. I simply can not thank you enough. I love you.

Finally, two very special persons have come into my life during the work on this thesis; Siri and Isabelle, my daughters. I dedicate this thesis to you.

Contents

Chapter 1

Introduction

During the summer of 2007 Dan Egerstad, a Swedish security consultant, carried out an experiment. Just like many other internet users around the globe he connected his computer to an anonymization network called The Onion Router (Tor) [125]. When a user connects to Tor, his or her traffic is no longer routed directly from source to destination. Instead the traffic takes random detours via the computers of other Tor users. When the data reaches its final destination—or if someone intercepts it in transit—it is virtually impossible to tell who the original sender is. The reason Dan connected to Tor was not however, to use the web anonymously; the purpose was to monitor the traffic being routed through his node in the network. In particular he wanted to highlight the fact that ordinary e-mail protocols are unencrypted and point at the risks you run by fetching your e-mail over an untrusted connection. After a couple of weeks Dan went through his network logs, and what he found was breathtaking. Usernames and passwords of thousands of email accounts had been logged. The accounts belonged to, among others, the Defense Research and Development Organization in India, foreign ministry of Iran, journalists working for large international newspapers, employees of multinational companies and numerous embassies belonging to Australia, Japan, Iran, India and Russia [60, 55]. What puzzled Dan at the time was how so many politicians, top diplomats and journalists had misunderstood the purpose and workings of Tor. Today he is convinced that the accounts that passed through his computer had already been compromised, and the reason he found them circulating in the Tor network was simply because the hackers in control of them did not want to reveal their identities [60].

Malicious code and hackers with bad intents have been around ever since multiuser systems emerged in the 1960s. Despite technological advancements the security challenges persist and as experiments like the one described above show, the field of computer security research is more relevant today than ever before. *Reference Monitoring* and *Security Policy enforcement* are central concepts in mitigating for instance injection attacks, trojan horses, access violations and other problems which for instance can lead to disclosure of credentials such as e-mail usernames and

passwords. This thesis discusses a specific approach to security policy enforcement called *reference monitor inlining* with emphasis on three aspects:

- How to deal with concurrency.
- How to certify the existence of an inlined reference monitor.
- How to naturally express and enforce policies with more sophisticated API-protocol constraints and data dependencies.

The rest of this introduction is laid out as follows. First a brief historical survey of the field of computer security research is presented, with emphasis on topics relevant for this thesis (Section 1.1). This is followed by an introduction to the concept of reference monitoring in general and reference monitor inlining in particular (Section 1.2). The motivation for studying the topics covered in this thesis is then presented (Section 1.3). Finally an overview of the thesis and the author contributions are given with brief descriptions of each included paper (Section 1.4).

1.1 Background

Computer security has its origins in the 1960s when multiuser systems were developed and put to use. The computers were operated by government agencies, universities and larger corporations. As the systems started to process confidential information such as military documents and unclassified but sensitive data such as personal information about citizens the need to protect the system from its users, and the users from each other emerged [56].

In the early 1970s two reports were published which can be viewed as the foundation of the subsequent research and as the start of computer security as a field of research in its own right. In February 1970 the report *Security Controls for Computer Systems* by W. Ware from the RAND Corporation [135] was published. This report summarized the technical foundations that the field of computer security had acquired by the end of the 1960s. As with most security research at the time, [73, 111, 137], it focused on how to protect classified information on multilevel resource-sharing computer systems used in the US defense sector. The report by Ware was followed by the *Computer Security Technology Planning Study* by J. P. Anderson, published in 1972. The study was conducted on behalf of the US Air Force and the intention was to

> "[...] develop a comprehensive plan for research and development leading to the satisfaction of requirements for multi-user open computer systems which process various levels of classified and unclassified information simultaneously through terminals in both secure and insecure areas. [...]"

The research programs triggered by the Anderson report led to, among other things, the development of a formal state transition model for enforcing access control in government and military applications. The first revision of the model

was developed by D. E. Bell and L. J. LaPadula in 1973, [10] and became known as the *Bell-LaPadula model*. Just as this theoretical model influenced the design and implementation of the operating systems at the time, the operating system engineers provided valuable feedback to the development of the theory. Bell and LaPadula refined the model and published the results the following years [77, 8, 11]. The core idea can be summarized in two mandatory and one discretionary access control rules [11]: (a) A subject at a given security level may not observe an object at a higher security level ("no read-up"), (b) a subject at a given security level must not write to any object at a lower security level ("no write-down"), and (c) every access should be governed by a discretionary access control matrix. The rules ensure the perseverance of the so called simple security (SS) property, the $\star$-property and the discretionary security (DS) property respectively. These three properties establishes total system security by inductively ensuring that the security is preserved from one state to the next.

During the late 1970s and the 1980s there was a concentrated effort in trying to realize an operating system with security as one of its primary design objectives [9]. Several attempts were made and among the most prominent ones we find the Kernelized Secure Operating System (KSOS) [47, 91], the Provably Secure Operating System (PSOS) [97, 45, 98], the Kernelized Virtual Machine (KVM) [110] and the Multics operating system [107, 108, 74, 136]. Multics had a significant impact in the field of computer security research. Despite being the first major operating system designed for security, it was however still broken into repeatedly [132]. One notable security issue caused the entire password file to be used as the message-of-the-day. The problem was caused by a bug which was triggered when multiple administrators opened a text editor at the same time [21]. This error was an early demonstration of the security problems associated with concurrency, which is one of the main topics of this thesis.

As the systems became more complex and as more systems were put to use the need to be able to evaluate and classify the systems in terms of security emerged. Steve Walker at the Office of Secretary of Defense formulated the Computer Security Initiative which resulted in, among other things, the publication *Trusted Computer Security Evaluation Criteria* (*The Orange Book*) initially published in 1983 and revised in 1985, [31]. The purpose of the document was to

> "[...] (a) provide users with a yardstick with which to assess the degree of trust that can be placed in computer systems for the secure processing of classified or other sensitive information; (b) to provide guidance to manufacturers as to what to build into their new, widely-available trusted commercial products in order to satisfy trust requirements for sensitive applications; and (c) to provide a basis for specifying security requirements in acquisition specifications. [...]"

The document defines four security divisions and seven security classes incrementally, i.e. the requirements of the lower classes are automatically inherited by higher ones. The requirements concern the security-relevant parts of a system, i.e. the

Division		Class	
A	Verified Protection	**A1**	Verified Design
B	Mandatory Protection	**B3**	Security Domains
		B2	Structured Protection
		B1	Labelled Security Protection
C	Discretionary Protection	**C2**	Controlled Access Protection
		C1	Discretionary Security Protection
D	Minimal Protection		

Table 1.1: The Orange Book security divisions and classes.

trusted computing base (TCB). The security divisions range from A to D as shown in Table 1.1. The D-class is basically reserved for systems that fail to meet the requirements for a higher division (i.e. "no security"). On the other end of the scale we have the A1-systems which, while functionally equivalent to B3-systems, require a formal model of the security policy, a formal top level specification and a consistency proof between the model and the specification. The policies in the Orange Book, discretionary access control and mandatory access control based on a lattice of security labels, could for instance be satisfied by systems implementing the Bell-LaPadula model, but other models have also been used in Orange Book evaluations. The standard approach of security policy enforcement is to have a *reference monitor* (a term introduced in detail in Section 1.2) which verifies that subjects are authorized to access the objects they request.

Given the strict requirements on analysis of the TCB in the higher classes such as A1, systems with a complex TCB tend to fall into the lower evaluation classes. Keeping the TCB simple (which allows for a more comprehensive analysis) is thus of high importance. How to utilize inlined reference monitoring as a security enforcement mechanism without having to include a complex monitor inliner in the TCB is another topic of this thesis.

By the end of the 1980s a computer no longer required a dedicated room and staff to operate and when the computer was placed on a desk in someone's office, it effectively became a single user machine. In a single user operating system such as the mass-market CP/M, MS-DOS or AmigaOS the aspects of multi-level and multi-user security became meaningless. According to some experts, this triggered a degrade of computer security, according to others the issues were not due to the single-user operating systems per see, but rather due to the fact that they were eventually put to use outside their intended environment [56].

During the 1980s PC users also experienced the first computer viruses, worms

and trojan horses. The first virus believed to go outside of the computer system on which it was developed was *Elk Cloner*, written by Richard Skrenta in 1982 [70]. This virus spread by infecting the boot sector of the floppy disks storing the Apple II operating system. Apart from that, it did no harm. Viruses continued to evolve during the late 1980s and many operating systems were affected including MS-DOS, AmigaOS and ProDOS. During late 1990s the first *macro viruses* (viruses embedded in documents in the form of macro code) started to spread. The first macro virus, *Concept*, appeared in July 1995 [114] and infected Microsoft Word documents. Another famous example of a macro virus that spread during the 1990s was the Melissa virus which spread through Microsoft Outlook e-mail clients. As will be discussed in Section 1.3, the technique of enforcing security policies by inlining a reference monitor into a program can be particularly effective against this type of viruses.

The prerequisites for viruses to spread is (a) the technological platform, i.e. a computing device that can accept and run 3rd party code, and (b) a critical mass of users [51]. During the 2000s these conditions were met for PDAs and smartphones running operating systems such as PalmOS, Symbian and Windows CE. One of the earliest PDA malware was a trojan horse called *Palm.Liberty.A*. Installing it on a PalmOS device caused other applications to be deleted [138]. The spread of the trojan horse was limited since it relied upon victims downloading it and installing it themselves. It did not take long however, until the malware started to copy itself from device to device through infrared interfaces and through Bluetooth [51, 15, 69].

Today the way malware is spread to smartphones resembles the way it is spread to desktop computers: Users download and install programs from untrusted sources over the Internet [46]. The software distribution channels and security mechanisms on the other hand differs. Desktop systems typically allow the user to download and install software from arbitrary servers and webpages. The system is then continuously monitored by antivirus software which scans the file system for malware. On a smartphone the user typically installs software from within a centralized application store. Each downloaded application requests a set of permissions which the user grants before the installation or during the first launch of the program. The operating system then ensures that the software does not exceed its granted permissions by means of a reference monitor. The first approach, which can be compared to a blacklisting of disallowed behavior, has the disadvantage that there is no way to exhaustively list current and future malicious software. The second approach, which analogously can be seen as a white listing of allowed behavior, has the disadvantage that many innocent tasks often require a combination of permissions that could be used maliciously. How to tackle the latter problem is the topic of the last paper of this thesis.

1.2 Reference Monitors

The concept of *reference monitors* (or *reference validation mechanisms*) was first introduced in the Computer Security Technology Planning Study published in 1972 [3]. The definition from this publication states that the function of a reference monitor is to *"validate all references (to programs, data, peripherals, etc.) made by programs in execution against those authorized for the subject (user, etc.)."* The report then lists three requirements that must be met by a reference validation mechanism:

1. The reference validation mechanism must be tamper proof.
2. The reference validation mechanism must always be invoked.
3. The reference validation mechanism must be small enough to be subject to analysis and tests to assure that it is correct. [3]

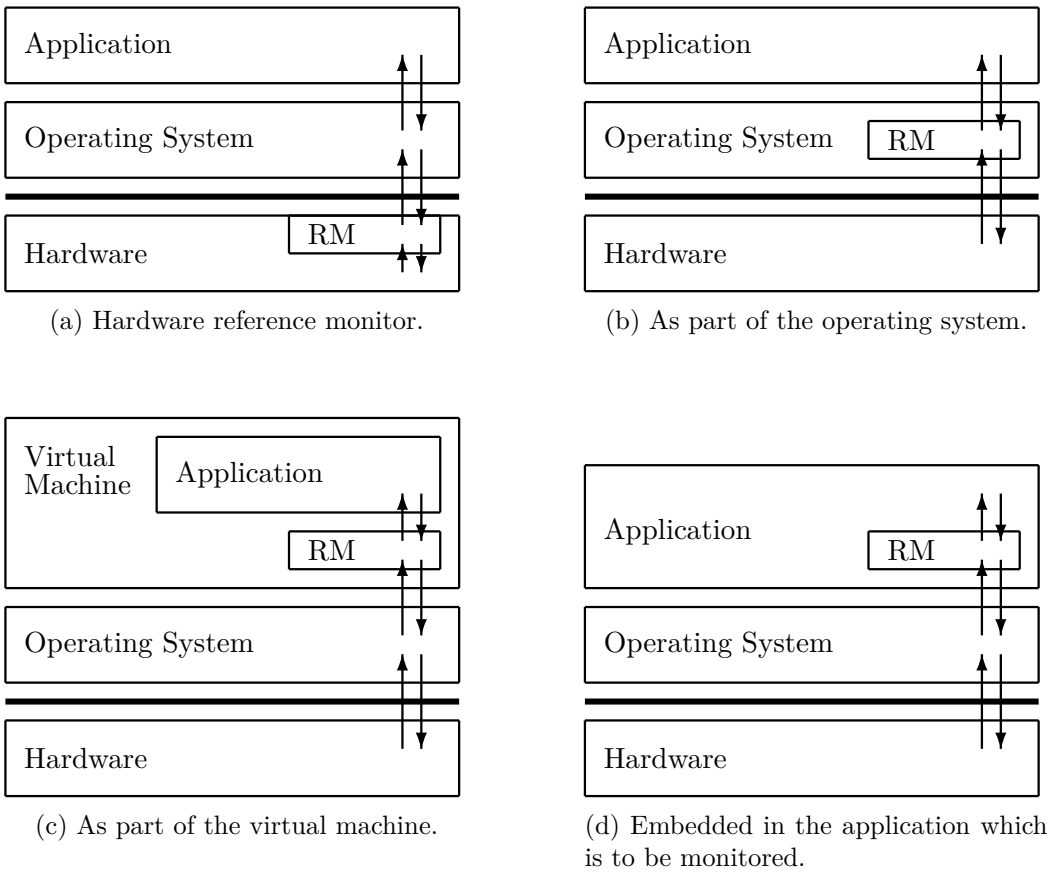
The first property says that there should be no way to (programmatically or manually) alter the security mechanism. The rationale behind this requirement is to provide a guarantee of the integrity of the mechanism. The second property (which is commonly referred to as *complete mediation*) states that the reference monitor must be consulted upon each security relevant action (SRA), regardless of when and how it is performed. Finally, the third property states that the mechanism must be amenable to formal analysis, which is a property relevant also for the TCB as a whole.

While the three security properties are still important and accurate, the definition of a reference monitor has been generalized slightly. The Orange Book [31] defines a reference monitor as follows:

> *Reference Monitor Concept* - An access control concept that refers to an abstract machine that mediates all accesses to objects by subjects.

Since the concept of a reference monitor is quite general it encompasses most runtime security enforcement in a system. In a layered system design, reference monitors can in principle be placed on any level [56]. On the lowest level of a system a reference monitor is typically implemented in hardware and used as a core mechanism for protecting the integrity of the operating system and for controlling access to memory [56, 103] (Figure 1.1a). A reference monitor can also be implemented in software as part of the operating system to for instance enforce a file access policy (Figure 1.1b). For code that is interpreted such as scripts and bytecode, a reference monitor could be implemented in for instance the Javascript engine of a web browser or as a security manager of a Java virtual machine (Figure 1.1c). Finally, a reference monitor could be embedded, or *inlined*, into the program which is to be monitored (Figure 1.1d). This thesis focuses on the latter approach.

Each design choice has its advantages and disadvantages. The lower the level is at which the reference monitor operates, the fewer primitives (and ways to invoke



(a) Hardware reference monitor.

(b) As part of the operating system.

(c) As part of the virtual machine.

(d) Embedded in the application which is to be monitored.

Figure 1.1: Reference Monitor implementation strategies.

these) it has to consider. This makes it easier to argue that all accesses performed in the system are indeed observed by the monitor, i.e. that the reference monitor follows the principle of complete mediation. Not all policies however can be suitably expressed at such low level of abstraction. A security policy for a script in a web page for instance is usually expressed in terms of DOM accesses. In such case a higher level reference monitor is more appropriate. This will be discussed further in Section 1.3.

1.2.1 Reference Monitor Inlining

The idea of embedding the reference monitor into the program which is to be monitored and, in effect, create a self-monitoring program was first explored by F. B. Schneider and Ú. Erlingsson in 1990 [41]. The classical approach to monitor inlining relies on having a program rewriter, or *inliner* as part of the TCB. Whenever a program is downloaded, installed or launched for the first time, the inliner rewrites the program so that the program itself checks that every security relevant instruc-

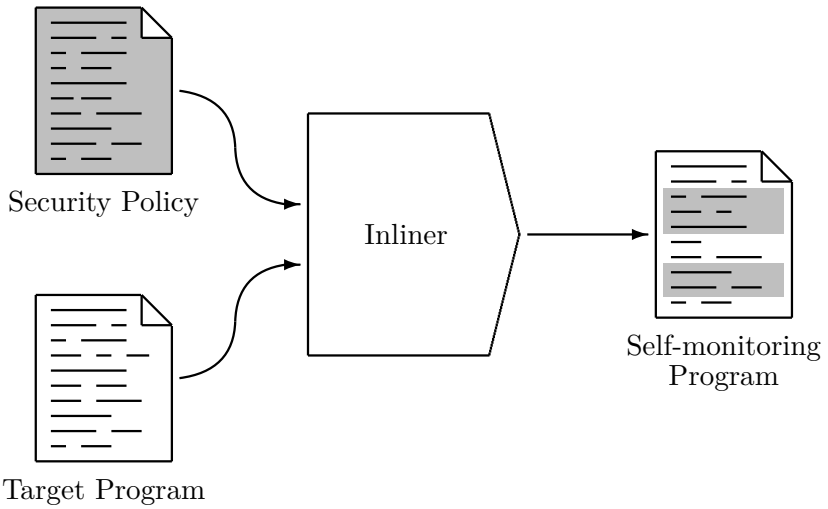


Figure 1.2: Monitor inlining overview.

tion is executed only after it has been checked and found not to be violating the security policy. The inliner does this by parsing a description of a policy, compiling it into snippets of code and then weaving these snippets into appropriate places in the code of the target application. Figure 1.2 illustrates the idea. An example of a reference monitor inlining follows.

Example 1 (Reference Monitor Inlining)**.** *Given a program which calls a method sendSMS (Figure 1.3a) and a policy stating that at most five such calls may be performed (Figure 1.3b), an inliner would do the following (Figure 1.3c):*

1. *Add a counter for the number of sendSMS-calls, that has been performed (*). This counter would represent the* monitor state *(also referred to as the* security state*).*

2. *Add code in connection to each call to sendSMS that*

   a) *terminates the execution if the SMS counter has reached the maximum value (**)*

   b) *increments the counter after the call has been performed (***)*

The monitoring code in the example above implements what is called a *truncation monitor* since it terminates the execution whenever the program is about to violate the policy. While there exists other approaches to avoiding a policy violation (for instance throwing an exception [128] or inserting a remedial action such as showing an error dialog to the user [83]), we will in this thesis focus on monitors that halt the execution.

```
...
sendSMS()
...
```

(a) Target program excerpt.

```
SECURITY STATE smsesSent = 0

BEFORE sendSMS()
    ASSERT smsesSent < 5

AFTER sendSMS()
    smsesSent = smsesSent + 1
```

(b) Send at most 5 SMSes policy.

```
...
int smsesSent = 0                    // Security state (global variable)   (*)
...

...

IF (smsesSent ≥ 5)                   // Security state check               (**)
    exit()

sendSMS()                            // Original method call

smsesSent = smsesSent + 1            // Security state update              (***)
...
```

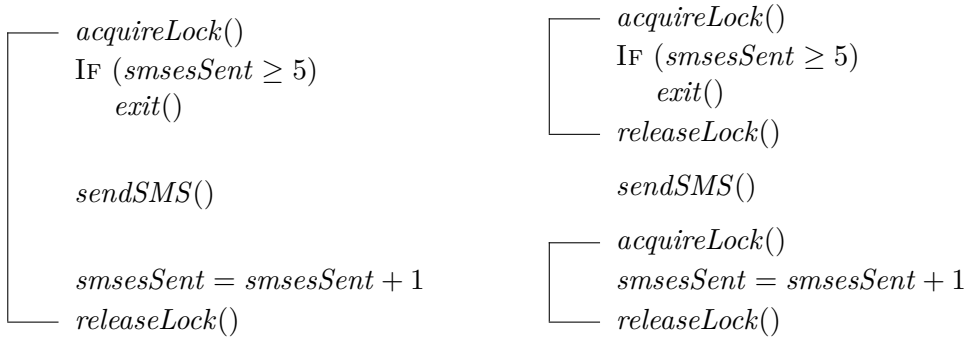
(c) Resulting self-monitoring program.

Figure 1.3: Inlining example

If the application in Example 1 would have had multiple calls to *sendSMS*, the inliner would have been required to either insert monitoring code for each one of them, or to create a secure wrapper method for *sendSMS*, and reroute calls to the wrapper method instead. To modify the actual implementation of *sendSMS* and add the monitoring code inside that method is however not always an option since the security relevant method in question may be part of a shared library and it may not be appropriate to enforce the same policy for all clients of this library. Restricting the modifications to the client code also allows programs with inlined monitors to be deployed in a standard unmodified runtime environment, since the self-monitoring application is executed just as any other application. An inliner which only modifies client code is referred to as a *client-side* inliner and it is this type of inliner that is the focus of this thesis.

Thread 1:	Thread 2:
IF ($smsesSent \geq 5$)	
$exit()$	
	IF ($smsesSent \geq 5$)
	$exit()$
$sendSMS()$	
$smsesSent = smsesSent + 1$	
	$sendSMS()$ *(possible violation!)*
	$smsesSent = smsesSent + 1$

Figure 1.4: Problematic interleaving of inlined code.



(a) Blocking Inlining

(b) Non-blocking Inlining

Figure 1.5: Blocking vs non-blocking inlining schemes.

Concurrency

In a single threaded setting, the inlined monitoring code in Example 1 is perfectly secure. In a multithreaded setting however, this is not the case. Consider what could potentially happen if two threads executed the inlined code simultaneously as shown in Figure 1.4. Clearly, this execution would violate the policy if four SMSes had already been sent. To avoid this type of interleaving and to guarantee policy adherence in a multithreaded setting, the inlined reference monitor could use a lock which it acquired before executing the inlined code, and released after the monitor code has completed, as shown in Figure 1.5a. This would rule out the problematic interleaving described above. An inliner that generates monitors according to this scheme is referred to as a *blocking inliner*, since it blocks other threads from executing SRAs in parallel. The main issue is of course that the SRAs are serialized with potentially large performance impacts (or in rare cases

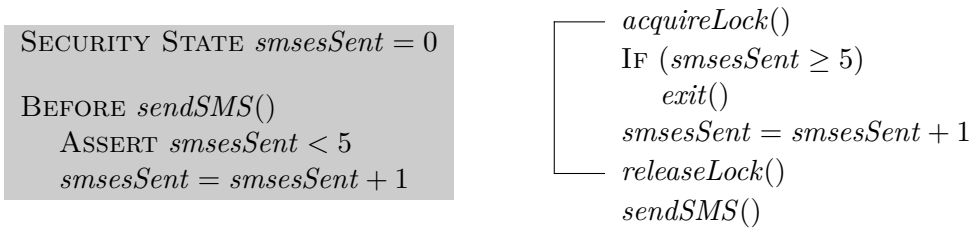


Figure 1.6: A race free version of the policy in Figure 1.3b and a secure non-blocking monitor inlining.

even deadlocks) as a result. The implications of enforcing policies using a blocking inliner are discussed in Chapter 3.

In an attempt to remedy this, one might try to release the lock temporarily under the duration of the API-call, as shown in Figure 1.5b. This would be the result of what we refer to as a *non-blocking* inlining. For this particular example however, this defeats the purpose of the lock, since the problematic interleaving described in Figure 1.4 is again a valid scheduling.

The policy, as it is expressed in Figure 1.3b, turns out to be impossible to enforce (without serializing all SRAs) in a multithreaded setting. To enforce the policy in a multithreaded setting, the policy needs to be reformulated and inlined as described in Figure 1.6. As opposed to the original policy, the policy in Figure 1.6 is *race free*. The exact characterization of race free policies (i.e. policies that can be securely enforced by a non-blocking inliner) is one of the main contributions of the paper presented in Chapter 4.

Inlining Correctness Properties

The three correctness properties mentioned in the Anderson report [3] concern security, i.e. they are intended to establish that the systems security policy is properly enforced by the reference monitor. When implementing the mechanism by program rewriting, another correctness aspect comes into play, namely the aspect of *preserving* the original behavior of the program. (An inliner that inserts a call to *exit*() before the first instruction ensures that any security policy is enforced, but is arguably not correctly implemented.) The notion of preserving the behavior of the target program is formally divided into *conservativity* and *transparency*. An inliner that does not *add* any behavior is conservative and an inliner that does not *remove* any behavior (apart from policy violating behavior) is transparent [82].

As briefly demonstrated in Section 1.2.1, there is a class of policies (the non-race-free policies) that can only be enforced using a blocking inliner. Since a blocking inliner excludes certain schedulings of the original program, this type of inliner is however not transparent for all policies and multithreaded programs. To characterize the correct behavior for blocking inliners the paper presented in Chapter 3

refines the notion of conservativity and proposes a property called *strong conservativity* which captures the idea of terminating the application only when the policy would have been violated.

1.2.2 Linear vs Tree based Monitoring

The classical approach to security monitoring (which we henceforth refer to as *linear monitoring*) builds on a model in which a program performs a linear sequence of security critical function calls. These sequences are observed by monitors defined in terms of deterministic finite automata [25, 39, 64]—which will be discussed in the first, second and third paper of this thesis— edit automata [84], linear temporal logic [71, 104, 67, 115, 141, 140] or context free grammars [92, 116].

Tree based monitoring on the other hand (introduced in detail in the fourth paper of this thesis), relies on a different model in which the actions of a program are described in terms of *trees* of function calls. If for example x is a result of a call to $f()$ and y is the result of a call to $g()$ and the program performs the call $h(x, y)$, the monitor will observe the action $h(f(), g())$. Since the actions of the program are described by trees, policies can be conveniently modeled as tree automata. This approach caters for a more natural way of expressing policies that involve data dependencies and more sophisticated API-protocol constraints.

An example illustrating the difference between linear monitoring and tree based monitoring follows.

Example 2. *Assume that we have a simple SQL-sanitization policy and the following security relevant functions readInput, sanitize, concat and execQuery. Given the following program,*

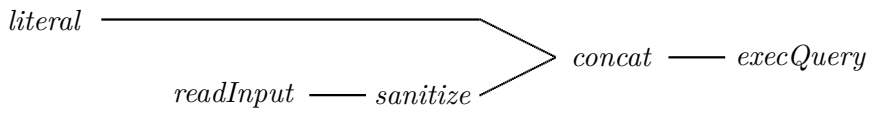
$q :=$ `"SELECT password FROM Users WHERE user="`
$x := readInput()$
IF $flipCoin()$ THEN
    $x := sanitize(x)$
$q := concat(q, x)$
$execQuery(x)$

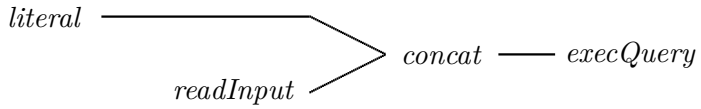
a linear monitoring model would yield one of the following two observable traces:

$readInput(), concat($`"SELECT..."`, `"John"`$), execQuery($`"SELECT...John"`$)$

$readInput(), sanitize($`"John"`$), concat($`"SELECT..."`, `"John"`$), execQuery($`"SELECT...John"`$)$

and a tree based monitoring model would yield one of the following observable traces:





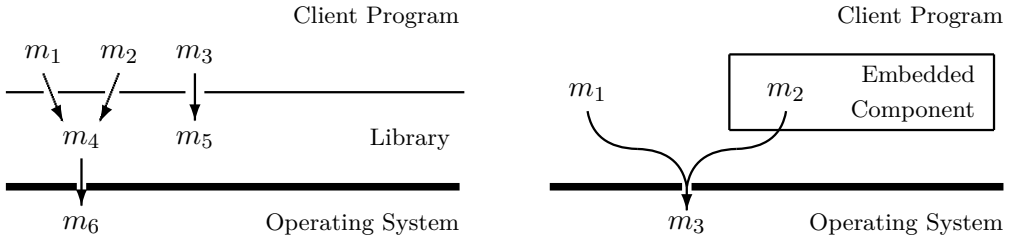
As illustrated by the example above, the tree based monitoring approach puts the arguments given to the security relevant functions in a better context. As opposed to the linear trace, it is clear whether or not the argument to *execQuery* is properly sanitized in the tree shaped trace. A drawback of the tree based approach is that it requires a slightly more complex implementation as the monitoring state is associated with the individual values in the execution which are spread out in memory. Another issue is due to the fact that the approach requires a mechanism for tracking origins of data. To prove soundness of the approach is thus as difficult as showing non-interference for any information-flow framework.

1.3 Research Issues and Motivation

In more complex software systems large parts of the code base come from internal or external libraries. These libraries are in turn not necessarily written from scratch, but instead often rely on other libraries. The reason for this type of approach is to allow each component in the system to focus on the business logic and the problem it is intended to solve. The effect is that a larger software system spans over several levels of abstraction and that the distance between the interface which the core business logic components interacts with and the operating system interface may be far apart logically and code wise. To express and enforce policies on the operating system interface level is under such circumstances unnatural and impractical at best. Firstly not all actions that could potentially be viewed as security relevant generate a system call to be monitored in which case such policy would not be supported. Secondly, even if each potentially security relevant action triggers a system call a policy expressing a business level constraint may require an over-approximation if expressed in terms of system level primitives. See Figure 1.7a for an illustration.

The problems are similar—if not worse—in extensible systems where external components are loaded and executed in runtime. An embedded application (such as a document macro, a stored procedure in a database, some 3rd party plugin or some content embedded in a webpage) should typically be executed with a stricter security policy than the program they are embedded in (a word processor, DBMS or a web browser). Since a policy enforcement mechanism operating at the system-level is not able to differentiate between system calls performed by the application itself and system calls performed by the embedded component, the enforcement mechanism would, in order to be sound, effectively have to restrict the entire application. Figure 1.7b illustrates this situation.

Apart from enforcing the policy at a better level of abstraction, an inlined reference monitor can also be implemented in a higher-level language (HLL). For a document macro, the monitor could be realized by inserting snippets of macro scripts and for an applet embedded in a web page, it could be realized in terms of



(a) A reference monitor implemented at the operating system interface can neither differentiate between actions m_1 and m_2 nor observe the action m_3.

(b) Constraining actions of embedded components may require constraining actions of the host application.

Figure 1.7: Problems with enforcing business level policies on OS interface level.

extra bytecode instructions. The advantages of using an HLL are many:

1. To securely embed a reference monitor into an application it is important to make sure that all control flow paths are considered. In particular jumps directly to security relevant instructions must be rerouted to a policy decision point. The analysis of the control flow of a program written in a structured language is trivial compared to a program with arbitrary and global jumps. Even in unstructured HLL such as Java byte code, there is no risk of for instance unpredicted jumps due to buffer overflow attacks.

2. An inlined reference monitor is more concise and to the point if expressed in an HLL. The higher level of abstraction caters for a simpler rewriter (i.e. a smaller TCB) which is easier to reason formally about and to prove correct.

3. Features of higher-level languages such as for instance type safety also provide strong guarantees that can be used to ensure that a secured application cannot compromise its IRM.

The reasons stated above motivate the study of inlined reference monitoring in general. The remainder of this section motivates the study of each specific topic discussed in the thesis.

Certification A drawback with the classical approach to reference monitor inlining where the consumer inlines the policy prior to execution is that the TCB needs to include a potentially large and complex program rewriter. Since the level of assurance of a system is inversely proportional to the size and complexity of the TCB, studying how to reduce or eliminate the inliner from the TCB is of high importance. If the benefits of inlining the reference monitor into the program are to be kept however, this requires the monitor to be embedded by an inliner that is outside the trust boundary. In such case the consumer needs some form of guarantee

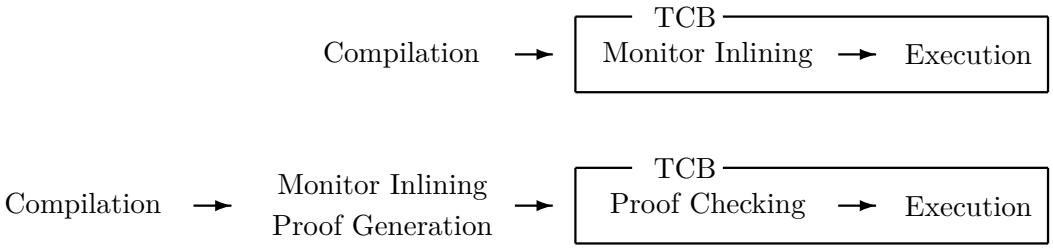


Figure 1.8: Monitor inlining as part of the TCB vs Proof-carrying code with untrusted monitor inlining.

of the fact that a reference monitor is properly embedded in the program. One way of solving this is by a technique called *proof-carrying code* (PCC). The concept of proof-carrying code was first described by G. Necula and P. Lee. in 1996 [96] and exploits the fact that, while *proving* that a program complies with a policy may be hard, *verifying that an existing proof is correct* is easy. The motivating example used in [96] discusses packet filters and demonstrates how the technique can be used to ensure adherence of a memory safety policy. In our scenario, the purpose of the proofs is to convey the fact that a given program has a security monitor with certain properties correctly embedded. The main benefit of the approach is that the inliner is replaced by a relatively simple proof checker, thus reducing the size and complexity of the TCB. Figure 1.8 illustrates the difference.

In addition to having a smaller TCB, this approach also enables the inlining to be performed by the developer instead of the consumer. There are several advantages with such approach:

- The developer has a better insight in the structure of the code and can thus guide the monitor inlining process to optimize for speed and/or code size.
- The developer can test and debug the precise code that will be executed by the consumer.
- A proof checker is not only smaller and less complex to analyze, it typically demands less resources in runtime, than a rewriter. This can be a big advantage when targeting mobile devices with limited resources.

The main drawback of letting the developer perform the inlining is of course that the decision of which policy to enforce must be made prior to distributing the program to the end users.

Concurrency In the early 2000s there was a sharp turn in the trend toward faster CPU clock speeds [122] and after several decades of steady increase in the speed at which computers execute sequential programs we now seem to have reached the limit. In a foreseeable future, applications which are not designed with concurrency

in mind will thus not run faster than they do today. (In fact they might even run slower, considering that individual cores run at lower clock speeds to reduce power consumption [123]). As a result we will see more programs taking advantage of multicore architectures which motivates the study of the security implications of executing multithreaded programs in general. As most interesting policies judge whether to accept or reject a security relevant action based on the history of the execution the monitor typically maintains a *security state*. This poses a fundamental challenge in terms of synchronization as soon as two or more threads perform security relevant actions. Important questions arise, such as how are the classical correctness properties affected, what type of policies can be enforced securely through client-side inlining and how can we leverage the certification mechanisms to deal with multithreaded applications.

Tree Based Monitoring Earlier work on security enforcement through IRMs has focused on linear monitoring. This approach is well suited for enforcing most types of temporal constraints, such as *"don't send after read"* or *"access a resource only between calls to acquire and release"*. Our experience however, shows that many useful policies that are common in practice fall outside of this class, most notably due to the lack of support for expressing data-dependencies. By assuming a more data-centric view, as opposed to the classical control flow centric view, where each piece of data is associated with its own security state describing which functions it originates from, allows us to naturally express policies such as

- *Arguments to some query-method must be either constants, outputs of a sanitize function, or concatenations of any such values*
- *Only send location data if it has been properly encrypted*
- *Don't send SMSes to numbers provided by an external data source*

all of which are hard (or even impossible) to express and enforce using linear monitoring techniques.

1.4 This Thesis

Apart from this introduction and the concluding chapter, the thesis consists of four papers. Each paper has been revised to include more elaborate proofs, provide clearer explanations and reduce overlapping material.

The first paper (presented in Chapter 2) focuses on how to certify that a program is self-monitoring by means of proof-carrying code. The results apply to single threaded programs. The second paper (presented in Chapter 3) turns to the issues related to concurrency and describes in detail what can (and cannot) be achieved with a blocking inlining scheme. The third paper (presented in Chapter 4) extends the study to non-blocking inliners and discusses how to adapt the certification mechanism to multithreaded programs. While the study of the first three

papers assumes a standard program and policy model, the fourth paper (presented in Chapter 5) describes a new type of inlining based on a program model with support for data-tracking and a policy model based on tree-automata. In this setting there is no global security state and consequently no security relevant data-races to worry about in a concurrent execution model.

Each paper follows the same structure: First a motivation of the research presented is provided. The program and policy model are then presented, followed by the definitions, theorems and proofs of the theoretical results. Each paper then describes a prototype implementation and a set of case studies intended to evaluate the practicality of the results. Although all prototype implementations targets Java bytecode, the theoretical results are fundamental and does not rely on any semantical properties specific to the JVM (except when reasoning formally about the correctness of the prototype implementation).

1.4.1 Included Papers

A summary of the results and contributions presented in the included papers follow.

Paper I: A Proof-Carrying Code Framework for Inlined Reference Monitors in Sequential Java Bytecode

The contribution of this paper is a proof-carrying code framework for inlined reference monitoring. In the scenario envisaged in this paper, the monitor inlining is performed by the developer, i.e. outside trust boundary of the consumer. A description of the workflow follows:

1. The program is compiled by the developer.

2. The developer pins down precisely what resources the application needs and formulates an as restrictive policy as possible. (This policy is referred to as the *contract*).

3. The developer inlines a monitor enforcing this contract and generates a proof of adherence that shows that a monitor is indeed properly embedded.

4. The developer bundles the program, contract and adherence proof in a package which is shipped to the consumer.

5. The consumer checks that the bundled contract is compatible (i.e. more restrictive) than the consumer policy.

6. The consumer verifies that the proof indeed shows that the there is an embedded monitor enforcing the contract.

7. The consumer safely executes the program.

The framework is designed on top of a weakest precondition calculus and works at the granularity of methods. Each method has its own pre- and post-conditions and a notion of method local validity is defined. The first theorem of the paper shows that local validity implies (global) validity, i.e. if each method adheres to its pre- and post-conditions, the program as a whole adheres to the specified contract. The paper also includes an example inliner and an algorithm to generate adherence proofs. By providing theorems stating that (a) the existence of a (valid) adherence proof implies contract adherence and (b) that the proof generation algorithm works for arbitrary monitors embedded by the proposed inliner, the paper shows that the inliner presented is secure.

Finally the paper describes a prototype implementation. The implementation consists of two parts: one part for the developer (an inliner and proof generator) and one part for the consumer (a Java ME based proof checker). Both parts are fully automatic and, as shown in the benchmarks, scale well in real world applications.

The original version of this paper was co-authored with M. Dam and is available online as a technical report [28].

Paper II: Provably Correct Inline Monitoring for Multithreaded Java-like Programs

As described in Section 1.2.1 there is a fundamental problem associated with classical client-side inlining in a multithreaded context. To guarantee a secure enforcement of the policy, the order in which the snippets of inlined code is executed must accurately reflect the order in which the security relevant actions are executed. One way of doing this is to use a blocking inliner, which is the approach examined in this paper.

The first contribution of the paper is the definition of *strong conservativity* which is a relaxation of ordinary conservativity, intended to capture the correct inlining behavior is in a multithreaded setting. The paper continues by describing an example inlining scheme which is shown to be strongly conservative.

Finally, the paper concludes by presenting the results from four different case studies. For each case study, a security policy is applied to an off-the-shelf Java ME application. The effectiveness of the embedded monitor is tested and relevant benchmarks are recorded. As demonstrated, the prototype scales well and can be applied to large applications.

The original version of this paper was co-authored with M. Dam, B. Jacobs and F. Piessens and published in Journal of Computer Security, 2010 [26].

Paper III: Security Monitor Inlining and Certification for Multithreaded Java

This paper focuses on the same problems as the previous paper, namely the problems related to client-side inlining in a multithreaded context, but with focus on non-blocking inlining.

The first contribution of this paper is a characterization of so called race free policies. The set of race free policies turns out to be the maximal set of transparently enforceable policies. This is shown in the paper by proving that (a) any inlining of a policy outside this class is either not secure or not transparent, and (b) by exhibiting a concrete inliner for policies inside the class which is secure, conservative and transparent.

The inliner is implemented for Java and applied to five applications and security policies (none of which have been used in prior work). As shown, the prototype scales well even for policies with 200+ clauses and application binaries on over a megabyte. Finally, the paper revisits the topic of certification and shows how proof-carrying code could be supported in the context of multithreading.

The original version of this paper was co-authored with M. Dam, B. Jacobs and F. Piessens and accepted for publication in Mathematical Structures in Computer Science, 2012 [22]. It is a journal version of *Security monitor inlining for multi-threaded Java* published in the proceedings of the 23rd European Conference on Object-Oriented Programming in 2009, [23].

Paper IV: TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies

Traditional runtime monitoring (such as the techniques discussed in Paper I–III) handles policies that restrict the control flow of a program, or more precisely, the interleaved sequence of security relevant actions performed by the program. This paper proposes a technique which extends the expressiveness of the policy language to also reason about data dependencies. As shown in the paper, this caters for a much more natural representation of many real world policies.

The first contribution of the paper is a theoretical formalization of the monitoring framework. The formalization consists of a program model based on λ-calculus which includes external (and observable) function calls and a policy model based on tree automata. The second contribution is a hierarchical classification of policy types. The paper identifies three policy classes (prefix-closed, local and subtree-closed) and describes their properties and relations under call-by-value reduction and arbitrary reduction strategies. As a third contribution we describe a denotational semantics of a simple imperative language, and show how a naive and straight forward encoding in our calculus induces the expected monitoring properties. This forms the bases of the last contribution, which is a full implementation of the framework running on top of the Android platform. The implementation is evaluated in five different case studies which include applications on over 100,000 lines of code and real world malware.

The original version of this paper was co-authored with M. Dam and G. Le Guernic and was published in the proceedings of the 19th ACM conference on Computer and communications security, CCS, 2012 [27].

1.4.2 Author Contributions

In this section I, Andreas Lundblad, give account of the contributions I have made in the papers included in this thesis.

When I started my graduate studies in March 2007 the breeding ground for the first paper had already been laid out. Together with my adviser Mads Dam, I developed a transition semantics for a program model and verification condition generator for the proof system. In the development of the final version of the paper, I contributed with writing, development of the proofs and carrying out the case studies. During 2008 through 2011 I collaborated with Mads Dam, Bart Jacobs and Frank Piessens and together we published the second and third paper in this thesis. The theoretical formalization, including the proofs, were to a large extent developed in a close collaboration between Jacobs and myself during one visit by Jacobs to KTH and two visits by me to K. U. Leuven. Apart from contributing to the theoretical parts of the papers together with Jacobs, I developed the prototype implementation and performed the case studies. During 2012 I, together with Mads Dam and Gurvan Le Guernic published the fourth paper in this thesis. I (with a background in reference monitoring) and Le Guernic (with a background in information flow) hatched the idea together. While Le Guernic contributed with some initial work on the λ-calculus model I was the driving force behind most parts of the writing, proof development, prototype implementation and case studies.

Chapter 2

A Proof-Carrying Code Framework for Inlined Reference Monitors in Sequential Java Bytecode

Mads Dam, Andreas Lundblad

KTH, Royal Institute of Technology, Sweden
{mfd, landreas}@kth.se

Abstract

We propose a light-weight approach for certification of monitor inlining for sequential Java bytecode using proof-carrying code. The goal is to enable the use of monitoring for quality assurance at development time, while minimizing the need for post-shipping code rewrites as well as changes to the end-host TCB. Standard automaton-based security policies express constraints on allowed API call/return sequences. Proofs are represented as JML-style program annotations. This is adequate in our case as all proofs generated in our framework are recognized in time polynomial in the size of the program. Policy adherence is proved by comparing the transitions of an inlined monitor with those of a trusted "ghost" monitor represented using JML-style annotations. At time of receiving a program with proof annotations, it is sufficient for the receiver to plug in its own trusted ghost monitor and check the resulting verification conditions, to verify that inlining has been performed correctly, of the correct policy. We have proved correctness of the approach at the Java bytecode level. An implementation, including an application loader running on a mobile device, is available, and we conclude by giving benchmarks for two sample applications.

2.1 Introduction

Program monitoring [82, 75, 17] is a well-established technique for software quality assurance, used for a wide range of purposes such as performance monitoring, protocol compliance checking, access control, and general security policy enforcement. The conceptual model is simple: Monitorable events by a client program are intercepted and routed to a decision point where the appropriate action can be taken, depending on policy state such as access control lists, or on application history. This basic setup can be implemented in a huge variety of ways. In this paper our focus is monitor inlining [43]. In this approach, monitor functionality is weaved into client code in AOP style, with three main benefits:

- Extensions to the TCB needed for managing execution of the client, intercepting and routing events, and policy decision and enforcement are to a large extent eliminated.
- Overhead for marshaling and demarshaling policy information between the various decision and enforcement points in the system is eliminated.
- Moreover, there is no need to modify and maintain a custom API or Virtual Machine.

This, however, presupposes that the user can trust that inlining has been performed correctly. This is not a problem if the inliner is known to be correct, and if inlining is performed within the users jurisdiction. But it could be of interest to make inlining available as a quality assurance tool to third parties (such as developers or operators) as well. In this paper we examine if proof-carrying code can be used to this effect in the context of Java and mobile applications, to enable richer, history-dependent access control than what is allowed by the current, static sandboxing regime.

Our approach is as follows: We assume that J2ME applications are equipped with *contracts* that express the provider commitments on allowed sequences of API calls performed by the application. Contracts are given as security automata in the style of Schneider [112] in a simple contract specification language ConSpec [2]. The contract is compiled into bytecode and inlined into the application code as in PoET/PSLang [42], and a proof is generated asserting that the inlined program adheres to the contract, producing in the end a self-certifying code "bundle" consisting of the application code, the contract, and an embedded proof object.

Upon reception the remote device first determines whether the received bundle should be accepted for execution, by comparing the received contract with the device policy. This test uses a simulation or language containment test, and is explored in detail by K. Naliuka et al. [12]. It then verifies that code adheres to the contract by checking the correctness of the accompanying proof.

The contribution of this paper is the efficient representation, generation and checking of proof objects. The key idea is to compare the effects of the inlined,

untrusted, monitor with a "ghost" monitor which implements the intended contract. A ghost monitor is a virtual monitor which is never actually executed, and which is represented using program annotations. Such a ghost monitor is readily available by simply interpreting the statements of the ConSpec contract as monitor updates performed before and after security relevant method calls. No JVM compilation is required at this point, since these updates are present solely for proof verification purposes.

The states of the two monitors are compared statically through a *monitor invariant*, expressing that the state of the embedded monitor is in synchrony with that of the ghost monitor. This monitor invariant is then inserted as an assertion at each security relevant method call. The assertions for the remaining program points could then in principle be computed using a weakest precondition (WP) calculus. Unfortunately, there is no guarantee that such an approach would be feasible. However, it turns out that it is sufficient to perform the WP computations for the inlined code snippets and not for the client code, under some critical assumptions:

- The inlined code appears as contiguous subsequences of the entire instruction sequences in the inlined methods.
- Control transfers in and out of these contiguous code snippets are allowed only when the monitor invariant is guaranteed to hold.
- The embedded monitor state is represented in such a way that a simple syntactic check suffices to determine if some non-inlined instruction can have an effect on its value.

The last constraint can be handled, in particular, by implementing the embedded monitor state as a static member of a final security state class. The important consequence is that instructions that do not appear in the inlined snippets, and do not include `putstatic` instructions to the security state field, may be annotated with the monitor invariant to obtain a fully annotated program. This means that a simple syntactic check is sufficient to eliminate costly WP checks in almost all cases and allows a very open-ended treatment of the JVM instruction set.

The resulting annotations are locally valid in the sense that method pre- and post-conditions match, and that each program point annotation follows from successor point annotations by elementary reasoning. This allows us to robustly and efficiently generate and check assertions using a standard verification condition (VC) approach, as indicated in Figure 2.1.

Our approach is general enough to handle a wide range of inliners. The developer (who has a better insight in the application in question) is free to tweak the inlining process for his specific application and could for instance optimize for speed in certain security relevant call sites, and for code size elsewhere.

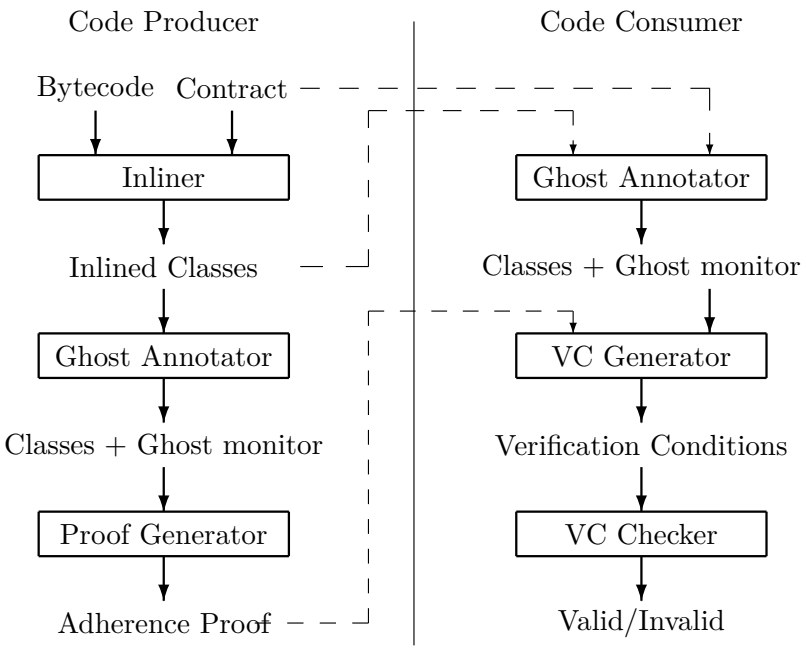


Figure 2.1: The architecture of our PCC implementation.

2.1.1 Related Work

Our approach adopts the Security-by-Contract (SxC) paradigm (cf. [12, 94, 33, 75, 17]) which has been explored and developed mainly within the S^{3}MS project [106].

Monitor inlining has been considered by a number of authors, cf. [43, 42, 40, 1, 128]. Erlingsson and Schneider [42] represents security automata directly as Java code snippets, making the resulting code difficult to reason about. The ConSpec contract specification language used here is for tractability restricted to API calls and (normal or exceptional) returns, and uses an independent expression syntax. This corresponds roughly to the call/return fragment of PSLang which includes all policies expressible using Java stack inspection [43].

Edit automata [83, 82] are examples of security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations. This approach has been fully implemented for Java by J. Ligatti et al. in the Polymer tool [7] which is closely related to Naccio [44] and PoET/PSLang [42].

Certified reference monitors has been explored by a number of authors, mainly through type systems, e.g. in [117, 6, 133, 65, 29], but more recently also through model checking and abstract interpretation [119, 118]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al. [65]. The Mobile system uses a simple library extension to Java bytecode to help managing updates to the security state. The use of linear types allows a type system to

localize security-relevant actions to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance. Mobile enforces per-object policies, whereas the policies enforced in our work (as in most work on IRM enforcement) are per session. Since Mobile leaves security state tests and updates as primitives, it is quite likely that Mobile could be adapted, at least to some form of per session policies. On the other hand, to handle per-object policies our approach would need to be extended to track object references. Finally, it is worth noting that Mobile relies on a specific inlining strategy, whereas our approach, as mentioned in the previous section, is less sensitive to this.

In [119, 118] Sridhar et al. explores the idea of certifying inlined reference monitors for ActionScript using model-checking and abstract interpretations. The approach is not tied to a specific inlining strategy and is general enough to handle different inlining techniques including non-trivial optimizations of inlined code. Although the certification process is efficient, the analysis, however, has to be carried out by the consumer.

For background on proof-carrying code we refer to [95]. Our approach is based on simple Floyd-like program point annotations in the style of Bannwarth and Müller [4], and method specifications extended by pre- and post-conditions in the style of JML [78]. Recent work related to proof-carrying code for the JVM include [5], all of which has been developed in the scope of the Mobius project.

An alternative to inlined reference monitoring and proof-carrying code, is to produce binaries that are structurally simple enough for the consumer to analyze himself. This is currently explored by B. Chen et al. in the Native Client project [139] which handles untrusted x86 native code. This is done through a customized compile chain that targets a subset of the x86 instruction set, which in effect puts the application in a sandbox. When applicable it has a few advantages in terms of runtime overhead, as it eliminates the monitoring altogether, but is constrained in terms of application and policy complexity.

Overview of the Paper

The JVM machine model is presented in Section 2.2. In Section 2.3 the state assertion language is introduced, and in Section 2.4 we address method and program annotations and give the conditions for (local and global) validity used in the paper. We briefly describe the ConSpec language and (our version of) security automata in Section 2.5. The example inlining algorithm is described briefly in Section 2.6. Section 2.7 introduces the ghost monitor, and Section 2.8, then, presents the main results of the paper, namely the algorithms for proof generation and proof recognition, including soundness proofs. Finally, Section 2.9 reports briefly on our prototype implementation, and we conclude by discussing some open issues and directions for future work in Section 2.10.

Java Bytecode Programs	
$Prg : (c \to Class, c_{main})$	Programs
$c \in String$	Class identifiers
$Class ::= (m \to M, f^*)$	Class definitions
$m \in String$	Method identifiers
$M ::= (\iota^+, H^*)$	Method definitions
$\iota \in Insn$	Instructions
$f \in String$	Field identifiers
$H ::= (\ell_b, \ell_e, \ell_t, c)$	Exception handler
$\ell \in \mathbb{N}$	Program labels

JVM Configurations	
$C ::= (h, S)$	Configurations
$S \in R^*$	Activation Record Stack
$h : r \cup (c \times f) \to Obj$	Heap
$Obj : f \to Val$	Object
$r \in o \cup \{\text{NULL}\}$	References
$o \in \mathbb{N}$	Location
$Val ::= r \mid v$	Values
$v \in int \cup float \cup boolean$	Primitive values
$R ::= (c.m, pc, s, l) \mid (o)$	Activation record
$pc \in \mathbb{N}$	Program counter
$s \in Val^*$	Operand stack
$l : \mathbb{N} \to Val$	Local variable store

Table 2.1: JVM Programs and configurations.

2.2 A Single Threaded Program Model [1]

The study of the research presented in this chapter is set in the context of single-threaded Java bytecode. We assume that the reader is somewhat familiar with Java bytecode and the JVM. In this section we give an overview of our program model and discuss the semantics of the monitorable API calls.

Table 2.1 provides an overview of the structure of bytecode programs and JVM configurations as well as the semantics of transition relation, $\to$, for key instructions. A few simplifications have been made in the presentation. In particular we disregard static initializers, and to ease notation a little we ignore issues concerning overloading. We also disregard from reflection and the (now deprecated) `jsr/ret` instructions since they render the analysis unfeasible.

[1] This section has been rewritten to be consistent with (and reused in) Chapter 3 and 4.

2.2.1 Programs

A program consists of a mapping from (fully qualified) class names, ranged over by c, to class definitions and specifies which class contains the main method. A class definition declares a set of fields and maps method identifiers, ranged over by m, to method definitions. A method definition is a pair of an instruction array and an exception handler array. An exception handler (b, e, t, c) catches exceptions of type c (and its subtypes) raised by instructions in the range $[b, e)$ and transfers control to address t, if the handler is the topmost handler in the exception handler array that handles the instruction for the given type.

2.2.2 Configurations

A *configuration* of the JVM is a pair $C = (h, R)$ of a heap, h, and a stack, S, of activation records. The heap maps locations to objects. A reference is either a location o, or the value NULL. Objects are finite maps of non-static fields to values. Static fields are identified with field references of the form $c.f$. To handle those, heaps are extended to assign values to static fields.

For normal execution, the activation record at the top of the execution stack has the shape (M, pc, s, l), where M is the currently executing method, pc is the program counter, $s \in Val^*$ is the operand stack and l is the local variable store. Except for API calls the transition relation $\rightarrow$ on JVM configurations is standard. A configuration $(h, (M, pc, s, l) :: S)$ is *calling*, if $M[pc]$ is an invoke instruction, and it is *returning normally*, if $M[pc]$ is a return instruction. For exceptional configurations the top frame has the form (o) where o is the location of an exceptional object, i.e. an object of type *Throwable*. We say that C is *returning exceptionally* if the current method has no exception handler covering the current program counter and exception type.

2.2.3 Types

The details of the Java type system are essentially irrelevant for the presentation of the results in this paper. It suffices to assume that the class declarations induce a class hierarchy, denoted by $<:$, and that the typing assertion $h \vdash v : c$ holds if v is some location o mapped to an object of type c in the heap h. Typing preserves the subclass relation, in the sense that if $h \vdash v : c$ and $c <: c'$ then $h \vdash v : c'$ as well.

2.2.4 API Method Calls

We are interested in security policies as constraints on the usage of external (API) methods. To this end we assume a fixed API, as a set of classes disjoint from that of the client program, for which we have access only to the signature, but not the implementation of its methods. We therefore represent API method activation records specially. When an API method is called in some thread a special API method stack frame is pushed onto the call stack. The execution can then proceed either by re-

turning or throwing an exception. When the call returns, an arbitrary return value of appropriate type is pushed onto the caller's operand stack; alternatively, when it throws an exception, an arbitrary, but correctly typed exceptional activation record is placed on the call stack. The heap may have changed during the call, but by the fact that we disregard from reflection, objects of the classes from the client program will remain unchanged. Since this model makes no assumptions about the behavior of API methods, our results hold for all (correctly typed) API implementations. This semantics does not, however, make any provisions for call-backs.

To refer to API calls and returns we use labeled transitions. Transition labels, or *actions*, α come in four variants to reflect the act of invoking an external method (referred to as a *pre-action*), returning from an external method normally or exceptionally (referred to as a *normal* or *exceptional post-action*), or performing an internal, not directly observable computation step. Actions have one of the following shapes:

- $(c.m, o, v)^{\uparrow}$ represents the invocation of API method $c.m$ on object o with arguments v.
- $(c.m, o, v, r)^{\downarrow}$ represents the normal return of $c.m$ with return value r.
- $(c.m, o, v, t)^{\Downarrow}$ represents the exceptional return of $c.m$ with exception object (of class *Throwable*) t.
- τ represents an internal computation step.

We write $C \xrightarrow{\alpha} C'$ if either $\alpha = \tau$ and $C \to C'$, or $\alpha \neq \tau$ and C' results from C by the action α according to the above non-deterministic semantics.

2.2.5 Transition Semantics

We here present a transition semantics of JVM instructions used in proofs. We assume that the configurations are type safe, in the sense that heap contents match the types of corresponding references, and that arguments and return / exceptional values for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions (cf. [80]). We only present the rules for the bytecode instructions mentioned in the paper. The rules for the other bytecode instructions are similar and straightforward.

Notation

Besides self-evident notation for function updates, array lookups etc. the transition rules uses the following auxiliary operations and predicates:

- $handler(H, h, o, pc)$ returns the proper target label given an exception handler array H, heap h, throwable o and program counter pc in the standard way:

$$handler(\epsilon, h, o, pc) = \bot$$
$$handler((b, e, t, c) \cdot H', h, o, pc) = \begin{cases} t & \text{if } b \leq pc < e \text{ and } h \vdash o : c \\ handler(H', h, o, pc) & \text{otherwise} \end{cases}$$

  The function is overloaded to also accept a method, $M = (I, H)$ as first argument, in which case it the function simply uses the exception handler, H of that method.

- Stack frames have one of three shapes (M, pc, s, l), (o) where o is throwable in the current heap, and $(\square)$ used for API calls (see Section 2.2.4).

Local Variables and Stack Transitions

$$\frac{S \to S'}{(h, S) \to (h, S')}$$

$$\frac{M[pc] = \texttt{aload}\ n}{\begin{array}{c}(M, pc, s, l) :: S \\ \to (M, pc + 1, l(n) :: s, l) :: S\end{array}} \qquad \frac{M[pc] = \texttt{astore}\ n}{\begin{array}{c}(M, pc, v :: s, l) :: S \\ \to (M, pc + 1, s, l[n \mapsto v]) :: S\end{array}}$$

$$\frac{M[pc] = \texttt{athrow}}{\begin{array}{c}(M, pc, o :: s, l) :: S \\ \to (o) :: (M, pc + 1, o :: s, l) :: S\end{array}} \qquad \frac{M[pc] = \texttt{goto}\ L}{\begin{array}{c}(M, pc, s, l) :: S \\ \to (M, L, s, l) :: S\end{array}}$$

$$\frac{M[pc] = \texttt{iconst\_}n}{\begin{array}{c}(M, pc, s, l) :: S \\ \to (M, pc + 1, n :: s, l) :: S\end{array}} \qquad \frac{M[pc] = \texttt{ldc}\ c}{\begin{array}{c}(M, pc, s, l) :: S \\ \to (M, pc + 1, c :: s, l) :: S\end{array}}$$

$$\frac{M[pc] = \texttt{ifeq}\ L \qquad n = 0}{\begin{array}{c}(M, pc, n :: s, l) :: S \\ \to (M, L, s, l) :: S\end{array}} \qquad \frac{M[pc] = \texttt{ifeq}\ L \qquad n \neq 0}{\begin{array}{c}(M, pc, n :: s, l) :: S \\ \to (M, pc + 1, s, l) :: S\end{array}}$$

$$\frac{M[pc] = \texttt{instanceof}\ c \quad \neg(\mathrm{s}_0 <: c)}{\begin{array}{c}(M, pc, v :: s, l) :: S \\ \to (M, pc + 1, \textsc{f} :: s, l) :: S\end{array}} \qquad \frac{M[pc] = \texttt{instanceof}\ c \quad \mathrm{s}_0 <: c}{\begin{array}{c}(M, pc, v :: s, l) :: S \\ \to (M, pc + 1, \textsc{t} :: s, l) :: S\end{array}}$$

Heap transitions

$$\frac{M[pc] = \texttt{putstatic}\ c.f}{\begin{array}{c}(h, (M, pc, v :: s, l) :: S) \\ \to (h[c.f \mapsto v], (M, pc + 1, s, l) :: S)\end{array}} \qquad \frac{M[pc] = \texttt{getstatic}\ c.f}{\begin{array}{c}(h, (M, pc, s, l) :: S) \\ \to (h, (M, pc + 1, h(c.f) :: s, l) :: S)\end{array}}$$

Exceptional Transitions

$$\frac{pc' = handler(M,h,o,pc) \quad pc' \neq \bot}{(h,(o) :: (M,pc,s,l) :: S) \rightarrow (h,(M,pc',s,l) :: S)} \qquad \frac{handler(M,h,o,pc) = \bot}{(h,(o) :: (M,pc,s,l) :: S) \rightarrow (h,(o) :: S)}$$

API calls

As discussed in Section 2.2.4 API calls are treated specially. To model the fact API methods can affect the heap, but only the fields of the classes defined by the API we define an equivalence relation over heaps, $\simeq_{API}$ and the API-transition rules as follows.

Definition 1. *Let $\simeq_{API}$ be an equivalens relation over heaps such that $h \simeq_{API} h'$ holds if and only if $h(c.f) = h'(c.f)$ for each $c \notin API$ and for each field f in c.*

$$\frac{M[pc] = \texttt{invokevirtual}\ c.m \quad c.m \in API}{(h,(M,pc,s,l) :: S) \rightarrow (h,(\Box) :: (M,pc+1,s,l) :: S)} \qquad \frac{h \simeq_{API} h'}{(h,(\Box) :: (M,pc,s,l) :: S) \rightarrow (h',(\Box) :: (M,pc,s,l) :: S)}$$

$$\frac{-}{(h,(\Box) :: (M,pc,s,l) :: S) \rightarrow (h,(o) :: (M,pc+1,s,l) :: S)} \qquad \frac{-}{(h,(\Box) :: (M,pc,s,l) :: S) \rightarrow (h,(M,pc,v :: s,l) :: S)}$$

The rules above only deal with invocation of API methods. Other invocations (client code calling client code) are standard, and we do not spell out the rule here.

2.3 Assertions

Annotations are given in a language similar to the one described by F. Y. Bannwart and P. Müller in [4]. The syntax of assertions a and (partial) expressions e are given in the following BNF grammar:

$$e ::= v \mid e.f \mid c.f \mid \mathrm{s}_i \mid \mathrm{l}_i \mid e \circ e \mid e \rightarrow e | e \mid (e,e) \mid \bot$$
$$a ::= \mathrm{T} \mid \mathrm{F} \mid e\ r\ e \mid a \wedge a \mid \neg a \mid e : c$$

where $i \in \omega$. The semantics, as mappings $[\![e]\!]C$ and $[\![a]\!]C$ is given in Figure 2.2. The operations $\circ$ and r are generic binary operators and relation symbols, respectively, with Kleene equality. The expression s_i refers to the i'th element of the operand stack, and l_i refers to the i'th local variable; (e_1, e_2) is pairing and T and F represent true and false respectively. A *heap assertion* is an assertion that does not reference the stack or any of the local variables. Disjunction ($\vee$) and implication ($\Rightarrow$) are defined as usual. We let $\mathrm{IF}(a_0, a_1, a_2)$ denote the conditional expression ($a_0 \Rightarrow$

$$
\begin{array}{ll}
[\![e.f]\!](h,S) = h([\![e]\!](h,S)).f & [\![e : c]\!](h,S) = \begin{cases} \textsc{t} & \text{if } h \vdash [\![e]\!](h,S) : c \\ \textsc{f} & \text{otherwise} \end{cases} \\
[\![c.f]\!](h,S) = h(c.f) & [\![e_1 \to e_2 \mid e_3]\!]C = \begin{cases} [\![e_2]\!]C\ , & \text{if } [\![e_1]\!]C = \textsc{t} \\ [\![e_3]\!]C, & \text{otherwise} \end{cases} \\
[\![e_1 \circ e_2]\!]C = [\![e_1]\!]C \circ [\![e_2]\!]C & [\![\mathrm{s}_i]\!](h,(M,pc,s,l) :: S) = s_i \\
[\![(e_1,e_2)]\!]C = ([\![e_1]\!]C, [\![e_2]\!]C) & [\![\mathrm{l}_i]\!](h,(M,pc,s,l) :: S) = l_i \\
[\![a_1 \wedge a_2]\!]C = [\![a_1]\!]C \wedge [\![a_2]\!]C & [\![e_1\ r\ e_2]\!]C = [\![e_1]\!]C\ r\ [\![e_2]\!]C \\
[\![\neg a]\!]C = \overline{[\![a]\!]C} & [\![\textsc{t}]\!]C = \textsc{t} \\
[\![\bot]\!]C = \bot & [\![\textsc{f}]\!]C = \textsc{f}
\end{array}
$$

Figure 2.2: Semantics of expressions and assertions

$a_1) \wedge (\neg a_0 \Rightarrow a_2)$ and $\textsc{select}(A_1, A_2, a_{else})$ the generalized conditional expression $\textsc{if}(A_{1,0}, A_{2,0}, \textsc{if}(A_{1,1}, A_{2,1}, \ldots, \textsc{if}(A_{1,n}, A_{2,n}, a_{else}) \ldots))$.

2.4 Extended Method Definitions

In this section we extend the syntactical definition of a method by an array of program point assertions and by invariants at method entry and (normal or exceptional) return.

Definition 2 (Extended Method Definition)**.** *An* extended method definition *is a tuple* $(I, H, A, pre, post)$ *in which* (I, H) *is a method definition,* A *is an array of assertions such that* $|I| = |A|$ *and* pre *and* $post$ *are heap assertions. An* extended program *is a program with extended methods. We let the array of assertions associated with a method* M *be denoted by* A^M*.*

For extended programs, the notions of transition and execution are not affected by the presence of assertions. An extended program is said to be *valid*, if all annotations are validated by their corresponding configurations in each execution starting in a configuration satisfying the initial precondition. The notion of validity is defined below.

Definition 3 (Configuration Validity)**.** *A (normal) configuration* $C = (h, (c.m, pc, s, r) :: S)$ *is* valid *if* $[\![A^{c.m}_{pc}]\!]C$ *holds.*

For an extended program, validity is defined as follows.

I_L	$WP_{(I,H,A,pre,post)}(L)$
`aload` n	$\mathit{unshift}(A_{L+1}[\mathrm{l}_n/\mathrm{s}_0])$
`astore` n	$(\mathit{shift}(A_{L+1})) \wedge \mathrm{s}_0 = \mathrm{l}_n$
`athrow`	$\textsc{select}((\mathrm{s}_0 : c \wedge b \leq L < e)_{(b,e,L',c)\in H}, (A_{L'})_{(b,e,L',c)\in H}, \mathit{post})$
`dup`	$\mathit{unshift}(A_{L+1}[\mathrm{s}_1/\mathrm{s}_0])$
`getfield` f	$\mathit{unshift}(A_{L+1}[\mathrm{s}_0.f/\mathrm{s}_0])$
`getstatic` $c.f$	$\mathit{unshift}(A_{L+1}[c.f/\mathrm{s}_0])$
`goto` L'	$A_{L'}$
`iconst_`n	$\mathit{unshift}(A_{L+1}[n/\mathrm{s}_0])$
`if_icmpeq` L'	$\textsc{if}(\mathrm{s}_0 = \mathrm{s}_1, \mathit{shift}^2(A_{L'}), \mathit{shift}^2(A_{L+1}))$
`ifeq` L'	$\textsc{if}(\mathrm{s}_0 = 0, \mathit{shift}(A_{L'}), \mathit{shift}(A_{L+1}))$
`instanceof` c	$A_{L+1}[\mathrm{s}_0 : c/\mathrm{s}_0]$
`invokevirtual` $c.m$	$\textsc{select}((\mathrm{s}_0 : c \wedge b \leq L < e)_{(b,e,L',c)\in H}, (A_{L'})_{(b,e,L',c)\in H}, \mathit{post}) \wedge (\bigwedge_{c'\in \mathit{defs}(c.m)} \mathit{pre}_{c'.m})$
`invokestatic` *System.exit*	T
`ldc` v	$\mathit{unshift}(A_{L+1}[v/\mathrm{s}_0])$
`putstatic` $c.f$	$\mathit{shift}(A_{L+1})[\mathrm{s}_0/c.f]$
`return`	post

Figure 2.3: Specification of the WP_M function

Definition 4 (Extended Program Validity)**.** *An extended program Prg is* valid *if for each execution $E = C_0C_1 \ldots C_k$ of Prg, such that $[\![pre_{main}]\!]C_0$ holds, all normal configurations in E are valid.*

We reduce this form of validity to a more tractable form of *local validity*. Our notion of local validity is modular in methods and more appropriate for proof-carrying code. The local validity is defined in terms of verification conditions generated by a WP-calculus. Basically it requires the precondition to imply the assertion of the first instruction, and that each assertion must be strong enough to ensure validity of any successor assertion. (As we shall see however, the process is much simplified as weakest preconditions need only be computed for the instructions added by the inliner.)

The WP_M function is specified for the instructions that are relevant for the presentation of this paper in Table 2.3. The definition uses the auxiliary functions *shift* and *unshift* which increments, resp. decrements, each stack index by one and $\mathit{defs}(c.m)$ which denotes the set of all classes c' such that $c <: c'$ and c' defines m.

The account of dynamic call resolution in Table 2.3 is crude, but the details are unimportant since, in this paper, pre- and post-conditions are always identical and common to all methods. For now we state the following lemma regarding variable and stack transitions.

Lemma 1 (WP soundness)**.** *Let* $M = (I, H, A, pre, post)$*. If* $C = (h, (M, pc, s, l) :: S)$*,* $C' = (h', (M, pc', s', l') :: S)$ *and* $C \to C'$*, then* $[\![WP_M(pc)]\!] C \Rightarrow [\![A_{pc'}]\!] C'$

Proof. We split the proof into one case per instruction. Since all cases are similar and fairly straight forward we give the details only for a the most interesting instructions.

- `aload` n: By the definition of $\to$ and by the definition of WP_M we need to show:

$$[\![\mathit{unshift}(A_{pc'}[\mathrm{l}_n/\mathrm{s}_0])]\!] C \Rightarrow [\![A_{pc'}]\!](h', (M, pc', l(n) :: s, l) :: S)$$

  This is shown by structural induction on $A_{pc'}$. The base cases T and F are trivial and $a_1 \wedge a_2$ follow immediately from the induction hypothesis. The cases $e_1\ r\ e_2$ and $e : c$ require the following auxiliary lemma:

$$[\![\mathit{unshift}(e[\mathrm{l}_n/\mathrm{s}_0])]\!] C = [\![e]\!](h', (M, pc', l(n) :: s, l) :: S)$$

  This is in turn shown by structural induction on e. The most interesting case is when $e = \mathrm{s}_i$. If $i = 0$ we have $[\![\mathit{unshift}(\mathrm{s}_0[\mathrm{l}_n/\mathrm{s}_0])]\!] C = [\![\mathit{unshift}(\mathrm{l}_n)]\!] C = [\![\mathrm{l}_n]\!] C = l(n) = [\![\mathrm{s}_0]\!]((M, pc', l(n) :: s, l) :: S)$. If $i > 0$ we have the following $[\![\mathit{unshift}(\mathrm{s}_i[\mathrm{l}_n/\mathrm{s}_0])]\!] C = [\![\mathit{unshift}(\mathrm{s}_i)]\!] C = [\![\mathrm{s}_{i-1}]\!] C = s_{i-1} = [\![\mathrm{s}_i]\!]((M, pc, l(n) :: s, l) :: S)$. The other base cases (v, l_i) are trivial and the inductive cases follow immediately from the induction hypotheses.

- `putstatic` $c.f$ By the definition of WP and $\to$ we need to show that

$$[\![\mathit{shift}(A_{pc'})[\mathrm{s}_0/c.f]]\!] C \Rightarrow [\![A_{pc'}]\!](h[c.f \mapsto s_0], (M, pc', s', l') :: S)$$

  Just as in the previous case we proceed by structural induction on $A_{pc'}$. The base cases hold by inspection and $a_1 \wedge a_2$ follow immediately from the induction hypothesis. For the cases involving an expression, we need to show:

$$[\![\mathit{shift}(e)[\mathrm{s}_0/c.f]]\!] C = [\![e]\!](h[c.f \mapsto s_0], (M, pc', s', l') :: S)$$

  For $e = \mathrm{s}_i$, we have $[\![\mathit{shift}(\mathrm{s}_i)[\mathrm{s}_0/c.f]]\!] C = [\![\mathrm{s}_{i+1}]\!] C$. Since $s' = s_1 s_2 \ldots$ we know that $[\![\mathrm{s}_i]\!] C' = s_{i+1}$. For $e = c.f$, we have $[\![\mathit{shift}(c.f)[\mathrm{s}_0/c.f]]\!] C = [\![\mathrm{s}_0]\!] C = [\![c.f]\!](h[c.f \mapsto \mathrm{s}_0], (M, pc', s', l') :: S)$. The remaining cases for e are trivial.

- `instanceof` c For this case we need to show that

$$[\![A_{pc'}[\mathrm{s}_0 : c/\mathrm{s}_0]]\!] C \Rightarrow [\![A_{pc'}]\!](h', (M, pc', b :: s, l') :: S)$$

  where b is T if $s_0 : c$ and F otherwise. We proceed by structural induction on $A_{pc'}$. The cases involving an expression rely on, and follow immediately by, the following

$$[\![e[\mathrm{s}_0 : c/\mathrm{s}_0]]\!] C = [\![e]\!](h', (M, pc', b :: s, l') :: S)$$

  where b is T if $s_0 : c$ and F otherwise. Again the result follows from structural induction on e. The remaining cases for $A_{pc'}$ are trivial or follow immediately from the induction hypothesis.

□

For a method to be locally valid the assertion of the first instruction must be a consequence of the methods precondition and each assertion must ensure that, after executing the corresponding instruction, the resulting configuration should satisfy the assertion of the successor instructions.

Definition 5 (Local Validity)**.** *An extended method* $M = (I, H, A, pre, post)$ *is* locally valid*, if the following verification conditions hold.*

1. $pre \Rightarrow A_0$*,*
2. $A_L \Rightarrow WP_M(L)$ *for all* $0 \leq L < |I|$*, and*
3. $\bigvee_{c'.m \in defs(c.m)} post_{c'.m} \Rightarrow A_L$ *for all* L *succeeding an invoke of* $c.m$*.*

An extended program is locally valid if all its methods are locally valid and the precondition of the main method holds in an initial configuration.

We note that local validity implies validity, as expected.

Theorem 1 (Local Validity Implies Validity)**.** *For any extended program Prg, if Prg is locally valid then Prg is valid.*

Proof. Given a locally valid program *Prg* and an arbitrary execution $E = C_0 C_1 \dots C_k$ of *Prg* we need to, according to Definition 4, show that all normal configurations of E are valid according to Definition 3. The proof proceeds by induction over the length of E. The base case, $E = C_0$ holds since (by the definition of local validity) pre_{main} holds in the initial configuration, and since $pre_{main} \Rightarrow A_0^{main}$. For the inductive step, we assume that all normal configurations in $E = C_0 \dots C_{n-1}$ are valid and show that if $C_{n-1} \rightarrow C_n$ and C_n is normal, then C_n is also valid. We assume that C_n is valid, and split the proof into the following cases:

- If C_{n-1} is normal we have the following subcases: (a) If the transition from C_{n-1} to C_n is an invoke instruction, it follows from the definition of *WP* and the conditions 1 and 2 of local validity. (b) If it is a return instruction it follows from the definition of *WP* and the conditions 2 and 3 of local validity. (c) For the other instructions it follows directly from Lemma 1.
- If C_{n-1} is exceptional, we have the following subcases: (a) If the exception thrown in C_{n-1} is caught locally in the method, C_n is valid by the definition of *WP* and by condition 2 of local validity. (b) If C_{n-1} is an exceptional return, we let C_j denote the last normal configuration in $C_0 \dots C_{n-2}$. We note that the current instruction of C_n is a successor of the method call performed by C_j. By the definition of *WP* (for `invokevirtual`) and by condition 2 and 3 of local validity, C_n must be valid.

□

```
SECURITY STATE String lastApproved = "";

AFTER file = GUI.fileSendQuery()
    PERFORM true → {lastApproved = file; }

EXCEPTIONAL GUI.fileSendQuery()
    PERFORM false → {}

BEFORE Bluetooth.obexSend(String file)
    PERFORM file = lastApproved → {}
```

Figure 2.4: A security specification example written in ConSpec.

2.5 Security Specifications

We consider security specifications written in a policy specification language ConSpec [2], similar to PSlang [42], but more constrained, to be amenable to analysis. An example specification is given in Figure 2.4. The syntax is intended to be largely self-explanatory: The specification in Figure 2.4 states that the program can only send files using the Bluetooth Obex protocol upon direct request by the user. No exception may arise during evaluation of the user query.

A ConSpec policy specifies in which state and with what arguments an API method may be invoked. If the policy has one or more constraints on a method, the method is *security relevant*. In the example there are two security relevant methods, *GUI.fileSendQuery* and *Bluetooth.obexSend*. The specification expresses constraints in terms BEFORE, AFTER and EXCEPTIONAL clauses. Each clause is a guarded command where the guards are side-effect free boolean expressions, and the assignment updates the security state. Guards may involve constants, method call parameters, object fields, and values returned by accessor or test methods that are guaranteed to be side-effect free and terminating. Guards are evaluated top to bottom in order to obtain a deterministic semantics. If no clause guard holds, the policy is violated. In return clauses the guards must be exhaustive.

2.5.1 Security Automata

A ConSpec contract determines a security automaton (Q, Σ, δ, q_0) where Q is a countable (not necessarily finite) set of states, Σ is the alphabet of security relevant actions, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \to Q$ is the transition function. We assume a special error state $\perp \in Q$ and view all states in Q except $\perp$ as accepting. We require that security automata are strict in the sense that $\delta(\perp, \alpha) = \perp$.

Executions produce security relevant actions in the expected manner. A calling configuration generates a pre-action determined by the called method and the current arguments (top n operand stack values for an n-ary method). A returning configuration then gives rise to a normal post-action determined by the identifier of

the returning method and the return value (top operand stack value). For sake of simplicity we assume that all API methods return a value. An exceptionally returning configuration generates an exceptional post-action determined by the method identifier of the returning method. The security relevant actions (the security relevant trace) of an execution E is denoted by $SRT(E)$ and formally defined below.

Definition 6 (Security Relevant Trace)**.** *The security relevant trace, $SRT(E)$, of an execution E is defined as*

$$
\begin{aligned}
SRT(E) &= SRT(E, \epsilon) \\
SRT(\epsilon, \epsilon) &= \epsilon \\
SRT(CE, \gamma) &= \begin{cases}
(c'.m', \mathbf{v})^{\uparrow} SRT(E, \mathbf{v} :: \gamma) \\
\quad \textit{if } C = (h, (c.m, pc, \mathbf{v} :: s, l) :: S) \textit{ is calling } c'.m' \\
(c.m, \mathbf{v}, r)^{\downarrow} SRT(E, \gamma') \\
\quad \textit{if } C = (h, (c.m, pc, r :: s, l) :: S) \textit{ is returning and } \gamma = \mathbf{v} :: \gamma' \\
(c.m, \mathbf{v})^{\Downarrow} SRT(E, \gamma') \\
\quad \textit{if } C = (h, (o) :: S) \textit{ is returning exceptionally and } \gamma = \mathbf{v} :: \gamma' \\
SRT(E, \gamma) \qquad \textit{otherwise}
\end{cases}
\end{aligned}
$$

We generally identify a ConSpec contract with its set of security relevant traces, i.e. the language recognized by its corresponding security automaton. A program is said to adhere to a contract if all its security relevant traces are accepted by the contract.

Definition 7 (Contract Adherence)**.** *The program Prg* adheres to contract $\mathcal{C}$ *if for all executions E of Prg, $SRT(E) \in \mathcal{C}$.*

For simplicity we assume (without loss of generality) that ConSpec policies initialize the security state variables to the default Java values.

2.6 Example Inlining Algorithm

In this section we give an algorithm for monitor inlining (from now on referred to as an inlining algorithm, or simply an inliner) in the style of Erlingsson [43]. As previously mentioned, the developer is free to decide what inlining strategy to use, so the algorithm presented here serves merely as an example and does for instance not include any optimizations.

The inliner traverses the instructions and replaces each invoke instruction with a block of monitoring code. This block of code first stores the method arguments in local variables for use in post-actions. Then the class hierarchy is traversed bottom up for virtual call resolution, and when a match is found the relevant clauses,

guards, and updates are enacted. For post-actions the main difference is in exception handling; exceptions are rerouted for clause evaluation, and then rethrown.

Our inliner lets the state of the embedded security monitor be represented by a static field *ms* of a final security state class, named to avoid clashes with classes in the target program. This choice of representation relies on the following fact of JVM execution and allows for our open-ended treatment of large parts of the instruction set.

Lemma 2. *Suppose f is a static field of class c. If $C = (h, (M, pc, s, r) :: S) \to C'$ and $M[pc] \neq$ `putstatic` $c.f$, then $[\![c.f]\!]C = [\![c.f]\!]C'$.*

Proof. By inspection of the semantics (as defined in the JVM specification [86]) of each instruction in the JVM instruction set. □

In other words, the only instruction which can affect the value stored in a static field f of a class c is an explicit assignment to $c.f$. In particular, the assumption ensures that instructions originating from the target program are unable to affect the embedded monitor state.

Each `invokevirtual` $c.m$ instruction is replaced by a block of inlined code that evaluates which concrete method is being invoked, then checks and updates the security state accordingly. We assume for simplicity that no instructions in a block of inlined code other than `athrow` will raise exceptions. The code is easily adapted at the cost of some additional complexity to take runtime exceptions, such as JVM errors, violating this assumption into account.

Figure 2.5 shows a schematic policy for a method $m : int \to int$ defined in class c and overridden in a subclass d. The policy has event clauses for BEFORE, AFTER and EXCEPTIONAL cases for each definition of m, each with two guards and two statement lists.

SECURITY STATE $int\ ms = 0$;

BEFORE $c.m(int\ a)$ PERFORM $cb_{g_1} \to \{ cb_{s_1} \} \mid cb_{g_2} \to \{ cb_{s_2} \}$
AFTER $r = c.m(int\ a)$ PERFORM $ca_{g_1} \to \{ ca_{s_1} \} \mid ca_{g_2} \to \{ ca_{s_2} \}$
EXCEPTIONAL $c.m(int\ a)$ PERFORM $ce_{g_1} \to \{ ce_{s_1} \} \mid ce_{g_2} \to \{ ce_{s_2} \}$

BEFORE $d.m(int\ a)$ PERFORM $db_{g_1} \to \{ db_{s_1} \} \mid db_{g_2} \to \{ db_{s_2} \}$
AFTER $r = d.m(int\ a)$ PERFORM $da_{g_1} \to \{ da_{s_1} \} \mid da_{g_2} \to \{ da_{s_2} \}$
EXCEPTIONAL $d.m(int\ a)$ PERFORM $de_{g_1} \to \{ de_{s_1} \} \mid de_{g_2} \to \{ de_{s_2} \}$

Figure 2.5: Schematic ConSpec policy

Figure 2.6 gives the inlining details for the policy schema in Figure 2.5. In the figure, each *`[EVALUATE g]`* section transforms a configuration $(h, (M, pc, s, r) :: S)$ to $(h, (M, pc', v :: s, r) :: S)$ where v is 0 or 1 if the guard g is false or true respectively. An *`[EXECUTE stmts]`* transforms the configuration $(h, (M, pc, s, r) ::$

S) to $(h[[\![stmts]\!](ms)/ms], (M,\ pc', s, r) :: S)$. (The remaining invoke instructions (`invokestatic`, `invokeinterface` and `invokespecial`) are handled similarly.)

We refer to the method resulting from inlining a method M (program Prg) with a contract $\mathcal{C}$ as $\mathcal{I}(M, \mathcal{C})$ ($\mathcal{I}(Prg, \mathcal{C})$). The main correctness property we are after for inlined code is contract compliance:

Theorem 2 (Inliner Correctness)**.** *The inlined program $\mathcal{I}(Prg, \mathcal{C})$ adheres to $\mathcal{C}$.*

Proof. This follows from the fact that we are always able to generate a valid adherence proof (Theorem 4) and that the existence of such adherence proof ensures contact adherence (Theorem 3). (Both statements are proved in later sections.) □

2.7 The Ghost Monitor

Using auxiliary ghost constructs that do not affect the execution and whose sole purpose is to simplify the verification process is a well-known and common technique [79]. In this section we describe what we refer to as the *ghost monitor*.

The purpose of the ghost monitor is to keep track of what the embedded monitor state *should be* at key points in the execution. This provides a useful reference for verification. Moreover, since the ghost monitor writes only to special ghost variables that are invisible to the client program, and since it is incapable of blocking, it does not in fact have any observable effect on the client program.

The ghost monitor uses special assignments which we refer to as *ghost updates*: Guarded multi-assignment commands used for updating the state of the ghost monitor and for storing method call arguments and dynamic class identities in temporary variables. A ghost update has the shape $\langle x^g := e \rangle$ where x^g is a tuple of ghost variables, special variables used only by the ghost monitor, and e is an expression of matching type. Typically, e is a conditional of similar shape as the policy expressions, and e may mention security state ghost variables as well as other ghost variables holding security relevant call parameters. Given the post-condition A_{L+1}, the weakest precondition for the ghost instruction $\langle x^g := e \rangle$ at label L is $WP_M(L) = A_{L+1}[e_1/x_1^g, \ldots, e_n/x_n^g]$.

The ghost updates are embedded right before and after each security relevant invoke instruction as well as in an exception handler catching any exception (*Throwable*) thrown by the invoke instruction and nothing else. Note that the existence of such an exception handler is easily checked, and that the code delivered by our inliner always has exception handlers of this form. The details are presented in Figure 2.7. A method M with ghost updates embedded, corresponding to the security automaton of a contract $\mathcal{C}$ is denoted by $\mathcal{I}^g(M, \mathcal{C})$.

We let $\mathcal{I}^g(M, \mathcal{C})$ denote the method in which a ghost monitor corresponding to contract $\mathcal{C}$ has been embedded into M. The key property of the ghost monitor is that the trace of ghost monitor states in an execution E, is the same as the states visited by the security automaton, given $SRT(E)$ as input. This is easily be shown by an induction over the length of E.

Label	Instruction
tArgs:	`astore` r_a
	`astore` r_t
	`aload` r_t
	`aload` r_a
dbChk:	`aload` r_t
	`instanceof d`
	`ifeq cbChk`
dbGrd1:	*[EVALUATE* db_{g_1}*]*
	`ifeq dbGrd2`
	[EXECUTE db_{s_1}*]*
	`goto BEnd`
dbGrd2:	*[EVALUATE* db_{g_2}*]*
	`ifeq dBFail`
	[EXECUTE db_{s_2}*]*
	`goto BEnd`
dBFail:	`iconst_1`
	`inv_static Sys.exit`
cbChk:	`aload` r_t
	`instanceof c`
	`ifeq BEnd`
cbGrd1:	*[EVALUATE* cb_{g_1}*]*
	`ifeq cbGrd2`
	[EXECUTE cb_{s_1}*]*
	`goto BEnd`
cbGrd2:	*[EVALUATE* cb_{g_2}*]*
	`ifeq cbFail`
	[EXECUTE cb_{s_2}*]*
	`goto BEnd`
cbFail:	`iconst_1`
	`inv_static Sys.exit`
BEnd:	`invokevirtual c.m`
	`goto hdlEnd`
hdlStrt:	`aload` r_t
	`instanceof d`
	`ifeq ceChk`
deGrd1:	*[EVALUATE* de_{g_1}*]*
	`ifeq deGrd2`
	[EXECUTE de_{s_1}*]*
	`goto EEnd`
deGrd2:	*[EVALUATE* de_{g_2}*]*
	`ifeq deFail`
	[EXECUTE de_{s_2}*]*
	`goto EEnd`
deFail:	`iconst_1`
	`inv_static Sys.exit`
ceChk:	`aload` r_t
	`instanceof c`
	`ifeq EEnd`
ceGrd1:	*[EVALUATE* ce_{g_1}*]*
	`ifeq ceGrd2`
	[EXECUTE ce_{s_1}*]*
	`goto EEnd`
ceGrd2:	*[EVALUATE* ce_{g_2}*]*
	`ifeq ceFail`
	[EXECUTE ce_{s_2}*]*
	`goto EEnd`
ceFail:	`iconst_1`
	`inv_static Sys.exit`
EEnd:	`athrow`
hdlEnd:	`aload` r_t
	`instanceof d`
	`ifeq caChk`
daGrd1:	*[EVALUATE* da_{g_1}*]*
	`ifeq daGrd2`
	[EXECUTE da_{s_1}*]*
	`goto AEnd`
daGrd2:	*[EVALUATE* da_{g_2}*]*
	`ifeq daFail`
	[EXECUTE da_{s_2}*]*
	`goto AEnd`
daFail:	`iconst_1`
	`inv_static Sys.exit`
caChk:	`aload` r_t
	`instanceof c`
	`ifeq AEnd`
caGrd1:	*[EVALUATE* ca_{g_1}*]*
	`ifeq caGrd2`
	[EXECUTE ca_{s_1}*]*
	`goto AEnd`
caGrd2:	*[EVALUATE* ca_{g_2}*]*
	`ifeq caFail`
	[EXECUTE ca_{s_2}*]*
	`goto AEnd`
caFail:	`iconst_1`
	`inv_static Sys.exit`
AEnd:	

Figure 2.6: Schematic inlining of policy in Figure 2.5

$$
\begin{array}{rl}
L\colon & \langle (t^g, args_1^g, \ldots, args_n^g) := (s_n, \ldots, s_0) \rangle \\
& \langle ms^g := t^g : c^k \to \delta(ms^g, (c^k.m, args^g)^{\uparrow}) \\
& \qquad\qquad \vdots \\
& \qquad\quad \mid\ t^g : c^1 \to \delta(ms^g, (c^1.m, args^g)^{\uparrow}) \\
& \qquad\quad \mid\ ms^g \rangle \\
& \texttt{invokevirtual c.m} \\
& \langle ms^g := t^g : c^k \to \delta(ms^g, (c^k.m, args^g, s_0)^{\downarrow}) \\
& \qquad\qquad \vdots \\
& \qquad\quad \mid\ t^g : c^1 \to \delta(ms^g, (c^1.m, args^g, s_0)^{\downarrow}) \\
& \qquad\quad \mid\ ms^g \rangle \\
& \vdots \\
L_{HStart}\colon & \langle ms^g := t^g : c^k \to \delta(ms^g, (c^k.m, args^g)^{\Downarrow}) \\
& \qquad\qquad \vdots \\
& \qquad\quad \mid\ t^g : c^1 \to \delta(ms^g, (c^1.m, args^g)^{\Downarrow}) \\
& \qquad\quad \mid\ ms^g \rangle
\end{array}
$$

Figure 2.7: Ghost updates induced by security automaton (Q, Σ, δ, q_0) for an invocation of $c.m$, where t^g is the target object, $args^g$ represents the arguments and $c^1 <: \ldots <: c^k$ denote all API-classes defining or overriding m.

Lemma 3. *Let $E = C_0 \ldots C_k$ be an execution of $\mathcal{I}^g(Prg, \mathcal{C})$ and ms_i^g denote the ghost monitor state in configuration C_i. If for all $0 \leq i \leq k$, $ms_i^g \neq \bot$, then $SRT(E) \in \mathcal{C}$.*

Proof. The result follows from the fact that the ghost monitor state accurately reflects the security relevant trace of the execution and that all states of the security automaton, (except $\bot$) are accepting. □

2.8 Contract Adherence Proofs

The key idea of a contract adherence proof is to show that the embedded monitor state ms of the program $\mathcal{I}^g(Prg, \mathcal{C})$ and the ghost monitor state ms^g are in agreement at certain program points. These points certainly need to include all potentially security relevant call and return sites. But, since we aim for a procedural analysis, and to cater for virtual call resolution actually all call and return sites are included.

In fact, this is all that is needed, and hence:

Definition 8 (Adherence Proof)**.** *An adherence proof for program Prg and contract $\mathcal{C}$ assigns to each method $M = (I, H)$ in $\mathcal{I}^g(Prg, \mathcal{C})$ an assertion array A such that the extended method $(I, H, A, ms = ms^g, ms = ms^g)$ is locally valid.*

Such an account has two main benefits which are heavily exploited below:

- It leaves the choice of a particular proof generation strategy open.
- It opens for a lightweight approach to on-device proof checking, by performing the local validity check on a program with a locally produced ghost monitor.

Theorem 3 (Adherence Proof Soundness)**.** *If an adherence proof exists for a program Prg and contract $\mathcal{C}$, then Prg adheres to $\mathcal{C}$.*

Proof. Assume Π is an adherence proof for a program *Prg* and a contract $\mathcal{C}$. By Theorem 1 we know that the corresponding extended program for $\mathcal{I}^g(Prg, \mathcal{C})$ is globally valid. This implies that $ms = ms^g$ at each configuration that is calling (or returning from) a security relevant configuration. Furthermore, since the $\perp$ value is an artificial "error" value of the security automaton with no Java counterpart, we know that if $ms = ms^g$, then $ms^g \neq \perp$. Thus, by Lemma 3, $SRT(E) \in \mathcal{C}$ and therefore *Prg* adheres to $\mathcal{C}$. □

2.8.1 Example Proof Generation

The process of generating contract adherence proofs is closely related to the process of embedding the reference monitor, thus the inlining and proof generation is preferably done by the same agent. This section describes how proofs may be generated for code produced by the example inliner presented in Section 2.6.

The monitor invariant, $ms = ms^g$ is set as each methods pre- and post-condition. The assertion for each specific instruction is generated differently, according to whether the instruction appears as part of an inlined block or not. Instructions inside the inlined block affect the processing of the embedded state, method call arguments etc. For this reason these instructions need detailed analysis using the *WP* function. Instructions outside the inlined blocks on the other hand, allow a more robust treatment, as they are only required to preserve the monitor invariant which they do due to Lemma 2. The critical property of the annotation function is the following:

Lemma 4. *Given a method $M = (I, H)$ of $\mathcal{I}^g(Prg, \mathcal{C})$ and a set IL labeling the inlined instructions in I, an array A of assertions can be computed such that the extended method $(I, H, A, ms = ms^g, ms = ms^g)$ is locally valid.*

Proof. Figure 2.9 shows the construction for a call of a method $m : int \rightarrow int$ in class c, under the schematic contract shown in Figure 2.8. We assume that an exception thrown by the invoked method is matched by an exception handler table entry on the form $(30, 32, 34, any)$. For brevity we let σ_{bef}, σ_{aft} and σ_{exc} denote the appropriate substitution for the effect of updating ms according to the before, after and exceptional clause of $c.m$ respectively. For instance, if bef_s denotes $ms = ms \times x; ms = ms - 5$, then σ_{bef} is $[(ms \cdot x) - 5/ms]$. □

The array is constructed by annotating the return instructions with the post-condition, and then in a breadth first manner, annotate the preceding instructions

SECURITY STATE $int\ ms = 0$

BEFORE $c.m(int\ a)$ PERFORM $bef_g \rightarrow \{bef_s\}$
AFTER $r = c.m(int\ a)$ PERFORM $aft_g \rightarrow \{aft_s\}$
EXCEPTIONAL $c.m(int\ a)$ PERFORM $exc_g \rightarrow \{exc_s\}$

Figure 2.8: Schema contract for the proof of Lemma 4.

using the *WP* function in case of inlined instructions and by using the monitor invariant in other cases.

Theorem 4 (Proof Generation)**.** *For each program Prg and contract $\mathcal{C}$ there is an algorithm, polynomial in $|Prg|+|\mathcal{C}|$, which produces an adherence proof of $\mathcal{I}(Prg, \mathcal{C})$.*

Proof. The algorithm described above treats each method in isolation. The breadth first traversal of the instructions takes time linearly proportional to the size of the instruction array plus the number of ghost updates. The resulting adherence proof is correct by construction. □

As an example Figure 2.11 illustrates a generated proof for a part of a program which has been inlined to comply with the policy in Figure 2.10.

2.8.2 Proof Recognition

Checking the validity of contract adherence proofs involves verifying local validity, which in general is undecidable. However, the problem is much simplified in our setup, since proofs apply to programs that have already been inlined with code corresponding from ConSpec clauses. Since the ConSpec syntax rules out for instance loops, the verification conditions that need to be discharged are predictable in the sense that they follow a certain pattern.

Theorem 5 (Efficient Recognition)**.** *The class of polynomial-time recognizable adherence proofs includes all adherence proofs generated from inlined programs using the algorithm of Theorem 4.*

Proof. To verify the validity of a given adherence proof we look at the requirements of Definition 8. Verifying that the pre- and post-conditions equal the monitor invariant is a simple syntactic check and can be done in time linearly proportional to the number of methods in the program.

For the requirement of local validity, it is sufficient to check that the verification conditions from Definition 5 can be rewritten to $\top$ in time polynomial in the size of the instruction array. The interesting verification conditions are those of the form $A_L \Rightarrow WP_M(L)$ where L is the label of the first instruction in an inlined block. A_L is, in this case, of the form $ms = ms^g \wedge a_0^g = a_0 = s_0 \wedge \ldots \wedge a_m^g = a_m = s_m$ and $WP_M(L)$ is of the form

Lbl	Instruction	Lbl	Instruction
	$ms = ms^g$		// Exceptional
	(non-inlined instruction)		
		34:	$ms = ms^g \wedge a = a^g \wedge t = t^g$
	// Inlined code start		$\langle ms^g := t^g : c \rightarrow \delta(ms^g, (c.m, a^g)^{\Downarrow}) \vert ms^g \rangle$
	$ms = ms^g$	38:	IF$(t : c, A_{40}, A_{42})$
	`astore a`		`aload t`
	`astore t`		`instanceof c`
	`aload t`		`ifeq 42`
	`aload a`		
		40:	IF$(\text{exc}_g, ms\sigma_{exc}(a) = ms^g, A_{41})$
	// Before		[Evaluate exc_g]
26:	IF$(t : c, A_{28}, A_{30})$		`ifeq 41`
	`aload t`		[Perform exc_s]
	`instanceof c`		`goto 42`
	`ifeq 30`		
		41:	T
28:	IF$(\text{bef}_g, \text{IF}(s_1 : c, \text{IF}(\text{bef}_g,$		`iconst_1`
	$ms\sigma_{bef}(a) = ms^g\sigma_{bef}(s_0), ms\sigma_{bef} = \bot),$		`invokestatic` *System.exit*
	$ms = ms^g) \wedge a = s_0 \wedge t = s_1, \text{T})$		
	[Evaluate bef_g]	42:	$ms = ms^g$
	`ifeq 29`		`athrow`
	[Perform bef_s]		
	`goto 30`		// After
		43:	IF$(t : c, A_{44}, A_{46})$
29:	T		`aload t`
	`iconst_1`		`instanceof c`
	`invokestatic` *System.exit*		`ifeq 46`
30:	IF$(s_1 : c, \text{IF}(\text{bef}_g, ms = ms^g\sigma_{bef}(s_0), ms = \bot),$	44:	IF$(\text{aft}_g, ms\sigma_{aft}(r, a) = ms^g, \text{T})$
	$ms = ms^g) \wedge a = s_0 \wedge t = s_1$		[Evaluate aft_g]
	$\langle (t^g, a^g) := (s_1, s_0) \rangle$		`ifeq 45`
	$\langle ms^g := t^g : c \rightarrow \delta(ms^g, (c.m, a^g)^{\uparrow}) \mid ms^g \rangle$		[Perform aft_s]
			`goto 46`
	$ms = ms^g \wedge a = a^g \wedge t = t^g$		
	`invokevirtual c.m(int) : int`	45:	T
			`iconst_1`
32:	$ms = ms^g \wedge a = a^g \wedge t = t^g$		`invokestatic` *System.exit*
	$\langle r^g := s_0 \rangle$		
	$\langle ms^g := t^g : c \rightarrow \delta(ms^g, (c.m, a^g, r^g)^{\downarrow}) \mid ms^g \rangle$		// Inlining code end
	$A_{43}[r/\text{s}_0]$	46:	$ms = ms^g$
	`astore r`		(non-inlined instruction)
	`aload r`		
	A_{43}		
	`goto 43`		

Figure 2.9: Schematic annotation for contract displayed Figure 2.8

SECURITY STATE *boolean haveRead* = *false*;

BEFORE *javax.microedition.rms.RecordStore.openRecordStore*(
String name, *boolean createIfNecessary*)
PERFORM *true* → {*haveRead* = *true*; }

BEFORE *javax.microedition.io.Connector.openDataOutputStream*(*String url*)
PERFORM *haveRead* == *false* → {}

Figure 2.10: A ConSpec specification which disallows the program from sending data over the network after accessing phone memory.

$$
\begin{array}{ll}
 & \vdots \\
 & 40{:}\{\Psi\} \\
 & \texttt{aload\_1} \\
\text{inlined} \left\{ \right. & 41{:}\{\texttt{IF}(0 \neq \texttt{SS.haveRead}, \texttt{T}, \texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead}))\} \\
 & \texttt{astore\_3} \\
 & 42{:}\{\texttt{IF}(0 \neq \texttt{SS.haveRead}, \texttt{T}, \texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead}))\} \\
 & \texttt{getstatic SS.haveRead} \\
 & 45{:}\{\texttt{IF}(0 \neq \mathrm{s}_0, \texttt{T}, \texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead}))\} \\
 & \texttt{iconst\_0} \\
 & 46{:}\{\texttt{IF}(\mathrm{s}_0 \neq \mathrm{s}_1, \texttt{T}, \texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead}))\} \\
 & \texttt{if\_icmpne 52} \\
 & 49{:}\{\texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead})\} \\
 & \texttt{goto 56} \\
 & 52{:}\{\texttt{T}\} \\
 & \texttt{iconst\_m1} \\
 & 55{:}\{\texttt{T}\} \\
 & \texttt{invokestatic System.exit} \\
 & 56{:}\{\texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead})\} \\
 & \texttt{aload\_3} \\
 & \{\texttt{IF}(\mathit{haveRead}^g = \texttt{F}, \Psi, \bot = \texttt{SS.haveRead})\} \\
 & \langle \mathit{haveRead}^g := \mathit{haveRead}^g = \texttt{F} \rightarrow \mathit{haveRead}^g \rangle \\
 & 71{:}\{\Psi\} \\
 & \texttt{invokestatic Connector.openDataOutputStream} \\
 & 74{:}\{\Psi\} \\
 & \texttt{astore\_2} \\
 & \vdots
\end{array}
$$

Figure 2.11: Generated assertions for inlining of `Connector.openDataOutput-Stream` where Ψ denotes the monitor invariant.

$$
\begin{array}{l}
\textsc{select}((t : c^n \wedge t^g : c^n, \ldots, t : c^1 \wedge t^g : c^1), \\
\quad (\textsc{select}((c^n.m_{G_1} \wedge c^n.m^g_{G_1}, \ldots, c^n.m_{G_i} \wedge c^n.m^g_{G_i}), \\
\qquad (c^n.m_{f_1}(ms, \mathbf{a}) = c^n.m^g_{f_1}(ms^g, \mathbf{a}^g), \ldots, \\
\qquad\quad c^n.m_{f_i}(ms, \mathbf{a}) = c^n.m^g_{f_i}(ms^g, \mathbf{a}^g)), \textsc{t}), \\
\qquad \vdots \quad \vdots \\
\qquad \textsc{select}((c^1.m_{G_1} \wedge c^1.m^g_{G_1}, \ldots, c^1.m_{G_j} \wedge c^1.m^g_{G_j}), \\
\qquad\quad c^1.m_{f_1}(ms, \mathbf{a}) = c^1.m^g_{f_1}(ms^g, \mathbf{a}^g), \ldots, \\
\qquad\quad c^1.m_{f_j}(ms, \mathbf{a}) = c^1.m^g_{f_j}(ms^g, \mathbf{a}^g)), \textsc{t})), \\
\quad ms = ms^g)
\end{array}
$$

The verification condition can then be rewritten and simplified by iterated applications of the rule $x = y \Rightarrow \phi \longrightarrow \phi[z/x][z/y]$ where x and y are instantiated with real variables and ghost counterparts respectively and where z is free for x and y in ϕ. These rewrites can be performed in time proportional to the length of the formula and does not increase the size of the expression since x, y and z are atomic. The result can then be rewritten to $\textsc{t}$ using the rules $(\psi \Rightarrow \phi) \wedge (\neg\psi \Rightarrow \phi) \longrightarrow \phi$ and $\phi = \phi \longrightarrow \textsc{t}$ in time polynomial in the size of the formula.

All other verification conditions ($pre_M \Rightarrow A_0$ and $A_L \Rightarrow WP_M(L)$ for all labels L except those of the first instructions in an inlined block) are trivial as their antecedents and succeedents are identical. □

2.9 Implementation and Evaluation

A full implementation of the framework, including a Java SE proof generator, a Java ME client, instructions and examples is available at `www.csc.kth.se/~landreas/irm_pcc`. Both the on- and the off-device software utilize a parser generated by CUP / JFlex [68, 76] and the ASM bytecode engineering library [99]. The implementation has been evaluated in two non-trivial case studies described below.

2.9.1 MobileJam

MobileJam was developed in the context of the S^{3}MS project to serve as a general case study. It is a GPS-based traffic jam reporter which utilizes the Yahoo Maps API. The policy in this case study restricts the connectivity by disallowing connections to any domains other than `local.yahooapis.com`.

2.9.2 Snake

The *Snake*-application used in this case study is a slightly modified version of a simple, off-the-shelf, game of snake. In addition to maintaining a local highscore list on file, it allows the user to submit his or her scores to an online highscore list. The policy used in the case study prevents data-leakage by preventing any network-writes after reading from local files.

	MobileJam	Snake
Security Relevant Invokes	4	2
Original Size	428.0 kb	43.7 kb
Size increase for IRM	4.8 kb	1.1 kb
Size increase for Proofs	20.6 kb	2.6 kb
Inlining	10.1 s	8.6 s
Proof Generation	4.7 s	0.8 s
Proof Recognition	98 ms	117 ms

Table 2.2: Benchmarks for the two case studies.

The application is intentionally designed to allow the user to break the policy at runtime (by first reading the local highscore list, and then sending a score over the network), so that the effectiveness of the inlined monitor can be demonstrated.

2.9.3 Statistics

Table 2.2 summarizes overhead for inlining, proof generation and load-time proof recognition. Inlining and proof generation was performed on an Intel Core 2 CPU at 1.83 GHz with 2 Gb memory and proof recognition was performed on a Sony-Ericsson W810i. The implementation is to be considered a prototype, and very few optimizations in terms of e.g. proof size have been implemented.

2.10 Conclusions

We have demonstrated the feasibility of a proof-carrying approach to certified monitor inlining at the level of practical Java bytecode, including exceptions and inheritance. This answers partially a question raised by K. W. Hamlen et al. [65].

We have proved correctness of our approach in the sense of soundness: Contract adherence proofs are sufficient to ensure compliance. We also obtain partial completeness results, namely that proofs for inlined programs can always be generated, and such proofs are guaranteed to be recognized at program loading time. Other properties are also interesting such as transparency [109], roughly, that all adherent behavior is preserved by the inliner. This type of property is, however, more relevant for the specific inliner, and not so much for the certification mechanism, and consequently not addressed here (but see e.g. [82, 128, 26, 23] for results in this direction).

The approach is efficient: Proofs are small and recognized easily, by a simple proof checker. An interesting feature of our approach is that detailed modeling of bytecode instructions is needed only for instructions appearing in the inlined code snippets. For other instructions a simple conditional invariance property on static fields suffices. This means, in particular, that our approach is easy to adapt to future versions of the Java virtual machine, needing only a check that the static

field invariance is maintained. Worth pointing out also is that the enforcement architecture can be realized in a way which is backwards compatible, in the sense that PCC-aware client programs can be executed without modification in a PCC-unaware host environment.

The approach to proof recognition described in this paper relies on an automatic term rewriting algorithm. As shown in Theorem 5 this is indeed sufficient for our inlining algorithm and our proof generation strategy. If the framework were to be extended to support for instance more sophisticated policies or inlining algorithms with optimized output, the verification conditions would no longer be as predictable and easy to discharge. In such case a proper proof system would have to be defined, and a corresponding proof checker implemented in the consumer TCB.

It is possible to extend our framework to multi-threading by protecting security relevant updates with locks, either locking the entire inlined block or releasing the lock during the security relevant call itself for increased parallelism. For proof generation the main upshot is that assertions must be stable under interference by other threads. This requires the ability to protect fields, such as those in the security state class, with locks by only allowing updates of these fields when the lock has been acquired. The validity of an assertion may then only depend on fields protected by locks that has been acquired at that point in the code.

Chapter 3

Provably Correct Inline Monitoring for Multithreaded Java-like Programs

Mads Dam[1], Bart Jacobs[2], Andreas Lundblad[1], Frank Piessens[2]

[1]KTH, Royal Institute of Technology, Sweden
{mfd, landreas}@kth.se

[2]Katholieke Universiteit Leuven, Belgium
{bartj,frank}@cs.kuleuven.be

Abstract

Inline reference monitoring is a powerful technique to enforce security policies on untrusted programs. The security-by-contract paradigm proposed by the EU FP6 S^{3}MS project uses policies, monitoring, and monitor inlining to secure third-party applications running on mobile devices. The focus of this paper is on multi-threaded Java bytecode. An important consideration is that inlining should interfere with the client program only when mandated by the security policy. In a multi-threaded setting, however, this requirement turns out to be problematic. Generally, inliners use locks to control access to shared resources such as an embedded monitor state. This will interfere with application program non-determinism due to Java's relaxed memory consistency model, and rule out the transparency property, that all policy-adherent behavior of an application program is preserved under inlining. In its place we propose a notion of strong conservativity, to formalize the property that the inliner can terminate the client program only when the policy is about to be violated. An example inlining algorithm is given and proved to be strongly conservative. Finally, benchmarks are given for four example applications studied in the S^{3}MS project.

3.1 Introduction

Program monitoring is a well-established and efficient approach to prevent potentially misbehaving software clients from causing harm, for instance by violating

system integrity properties, or by accessing data to which the client is not entitled. Potentially dangerous actions by a client program are intercepted and routed to a policy decision point (PDP) in order to determine whether the actions should be allowed to proceed or not. In turn, these decisions are routed to a policy enforcement point (PEP), responsible for ensuring that only policy-compliant actions are executed.

The Security of Software and Services for Mobile Systems (S^3MS) project has investigated the use of such program monitors for ensuring the security of communicating mobile applications. This paper focuses on one of the key scientific results of the S^3MS project: the design and implementation of inlined reference monitors in multithreaded Java.

The idea of monitor inlining is to push policy decision and enforcement functionality into the client programs themselves, by embedding a security state into the client program, and using code rewriting to ensure this embedded state is correctly queried and updated at the appropriate points. When applicable, such an approach has a number of advantages:

- Overhead for marshalling and demarshalling policy information between the various decision and enforcement points in the system is eliminated.

- All information needed for policy enforcement is directly available to the PDP and the PEP.

- Extensions to the trusted computing base (TCB) needed for policy enforcement are localized to the client code.

- By proving the inliner correct, in the sense that it enforces the policy correctly, and that it interferes with program execution only when necessary, the need for extensions (trust) can to a large extent be eliminated.

The starting point for much previous work on monitor inlining has been security automata in the style of Schneider [112]. The PoET/PSLang toolset by Erlingsson [40] implements monitor inlining for Java. That work represents security automata directly in terms of Java code snippets, making it difficult to formally prove correctness properties of the approach. As an alternative we propose to use a dedicated policy specification language ConSpec [2], similar to PSLang, but more constrained in order to allow for a decidable containment problem. The ConSpec language, in particular, is designed to monitor only accesses to some specific API, determined by the application program under consideration.

Formal correctness of inlining for the case of sequential bytecode has been examined in [1] for Java, and in [129] for .NET. In particular, [1] shows how to generate bytecode level specification annotations under rather modest assumptions on the inliner, by fixing control points immediately before and after each method call at which the embedded state must be correctly updated.

Other recent work on monitoring and monitor inlining includes work on edit automata [7, 83, 82], security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations. Type-based approaches for security policy enforcement have been considered by a number of authors, e.g. [117, 133, 29, 48]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al [65]. The Mobile system uses a simple library extension to Java bytecode to help managing updates to the security state. The use of linear types allows a type system to localize security-relevant actions to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance.

Our contribution is to propose correctness criteria for monitor inlining in the case of multi-threaded bytecode programs, and to formally prove correctness for an example inliner. In particular we address the implications of relaxed memory consistency models in intermediate bytecode languages such as JVML and MSIL. This turns out to be non-trivial, since locks introduced by the inliner to control access to shared resources such as the embedded security state will in general interfere with application program nondeterminism, and rule out the transparency property [82], that all policy-adherent behavior of an application program is preserved under inlining. In its place we propose a notion of strong conservativity, to formalize the property that any complete trace of an inlined program is either a policy-compliant complete trace of the uninlined program, or a partial trace of the uninlined program truncated at the point of the policy violation.

The paper is structured as follows. In Section 3.2 we survey the S^{3}MS project context, and briefly introduce the ConSpec language. In Section 3.3 we present those parts of a model for multi-threaded Java bytecode execution needed to understand the rest of the paper, in particular the concepts of legal execution and observable trace. Section 3.4 briefly introduces security automata, to pin down the key concept of policy compliance. Section 3.5 present the main results of the paper: Correctness criteria, example inliner, and the correctness proof. Section 3.6 gives benchmark results for four sample mobile applications, and Section 3.7 concludes.

3.2 Security by Contract

The key objective of the S^{3}MS project [105] is the creation of a framework and technological solutions for trusted deployment and execution of communicating mobile applications in heterogeneous environments. A contract-based security mechanism lies at the core of the framework [36, 32].

Application contracts specify the security behavior of mobile applications, and can be matched with *device policies* specifying acceptable behavior of applications on the device.

This section provides a brief summary of the *security-by-contract* (SxC) paradigm developed in the S^{3}MS project. We start by analyzing the requirements for a se-

curity architecture for mobile applications and services, and go on to discuss how the SxC paradigm fulfills these requirements. Then we discuss how monitor inlining fits in this picture, and show that the contribution of this paper—provably correct monitor inlining for multithreaded Java—is an essential ingredient of SxC.

3.2.1 Security for mobile applications and services

Mobile phones and personal digital assistants have evolved over the past years to become general purpose computation platforms. Many of these devices support downloading third party applications built on for instance the .NET Compact Framework or Java Micro Edition. However, supporting applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications on a phone can lead to denial of service, loss of money, leaking of confidential information on the device and so forth.

Current devices already provide certain countermeasures against these threats, with support for sandboxing and code signing. The key idea is that unsigned code is severely limited in what it can do on the device, i.e. it runs in a strict sandbox. Code that is signed by a trusted party can break out of the sandbox. The device has a keystore that contains the public keys of trusted parties.

This security model has a number of drawbacks. First, it is not flexible: applications either run in a restricted sandbox, or have full power. Many interesting types of applications cannot run in a sandbox. Examples of case studies considered in the S^{3}MS project include:

- Multiplayer games, where communication between the players and/or a game server is essential.
- A traffic jam reporter, that interacts with the GPS device and that sends and receives traffic information to and from a server.
- Social networking applications, where users can track the location of their friends on their mobile device.

None of these case studies can function in a sandbox. On the other hand, the risk of giving full power to third party applications is substantial.

A second disadvantage of the current security model is that no precise meaning is associated with the signatures of trusted third parties: a signature either means that the application comes from the software factory of the signatory or that the signatory vouches for the software, but there is no clear definition of what guarantees it offers. Hence, device owners trust the third party both for (a) appropriate vetting of applications, and (b) using a suitable notion of good behavior. Incidents [101] show that the current security model is inappropriate.

3.2.2 Application contracts and policies

The SxC paradigm addresses the shortcomings of the current mobile device security model.

A key ingredient is the notion of an *application security contract*. Such a contract specifies the security behavior of the application. Technically, a contract is a security automaton in the sense of Schneider [112], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton. Other models of contracts have been proposed in the domain of security policies, notably versions of temporal logic, [71, 104, 67, 115].

Mobile devices are equipped with a *security policy*, a security automaton that specifies the behavior that is considered acceptable by the device owner. The key task of the S^{3}MS device run-time environment is to ensure that all applications will comply with the device security policy. To achieve this, the run-time can make use of the contract associated with the application (if it has one), and of a variety of policy enforcement technologies:

- Monitor inlining, a program rewriting technique to ensure that a program complies with a given policy.
- Contract-policy matching [90], the process of checking whether the security behavior specified in a contract is a subset of the allowed security behavior specified in a policy.
- Explicit run-time monitoring for compliance with policies.

All these enforcement technologies can run on-device. Some of them (matching and inlining) can also be provided as a web service that the device can call during the installation of an application on the device.

An application contract is a statement about the behavior of an application, and there is no a priori guarantee that this statement is correct. Testing and static analysis can be used at development time to increase confidence in the contract. In addition, monitor inlining of the *contract* at development time can provide strong assurance of compliance.

If the device makes security decisions based on the contract (for instance when it uses contract-policy matching), then there is a clear need to transfer these development-time guarantees to the device that will eventually execute the application. Without a secure transfer of these guarantees, it would be easy for an attacker to modify either the application or the contract. Two key technologies support this transfer:

1. A cryptographic signature by a trusted third party can vouch for application-contract compliance. Note the difference with the use of signatures in the traditional mobile device security model. In the S^{3}MS approach, a signature has a clear semantics: the third party claims that the application respects

the supplied contract [37]. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user.

2. Proof-carrying-code techniques can be used, to enable verification on the mobile device of contract compliance proofs constructed by the program developer (as described in the paper presented in Chapter 2).

3.2.3 Example: Mobile 2-player Chess

As an example application (also used as a case study in the S^{3}MS project), we consider a two-player chess game running on the .NET Compact Framework. This application supports standalone games (where two players play chess against each other on the same device), as well as games between two devices communicating over either a TCP/IP network, or using text messages (SMS's). Chess games rarely take more than 70 moves per player to finish, and the chess program enforces a hard upper limit of 100 moves. As a consequence, the program's contract can specify hard upper bounds on its use of communication resources. One move either takes 20 bytes of TCP traffic, or 1 SMS. Hence, one run of the program will consume at most 2000 bytes of network traffic, and send at most 100 SMS messages. The contract in Figure. 3.1 specifies this.

The contract is expressed in the ConSpec policy language [2]. A ConSpec specification tells when and with what arguments an API method may be invoked. If the specification has one or more constraints on a method, the method is said to be a *security relevant method* (SRM).

The first part of the contract declares the security state. This security state contains a definition of all the variables that will be used in the contract, and defines the set of states of the corresponding security automaton. In the example contract, two state variables maintain (1) the number of bytes that have already been sent over the network, and (2) the number of SMS messages that have been sent.

The security state declaration is followed by one or more clauses. Each clause represents a rule on a security-relevant API method call. These rules can be evaluated before the method is called, after the method is called, or when an exception occurs. A clause definition consists of the BEFORE, AFTER or EXCEPTIONAL keyword, the signature of the method on which the rule is defined, and a list of guard/update blocks. The guard is a boolean expression that is evaluated when a rule is being processed. The guard may mention variables from the security state declaration, arguments given in the method call and the return value (if it is part of an after clause). If the guard evaluates to true, the corresponding update block is executed. All state changes that should occur can be incorporated in this update block. When a guard evaluates to true, the evaluation of the following guards (and consequently the potential execution of their corresponding update blocks) is skipped.

SECURITY STATE
int bytesSent = 0;
int smsSent = 0;

BEFORE *System.Net.Sockets.Socket.Send*(*byte*[] *array*)
PERFORM *array.Length* == 20 && *bytesSent* + *array.Length* ≤ 2000 → {}

AFTER *int sent* = *System.Net.Sockets.Socket.Send*(*byte*[] *array*)
PERFORM *true* → {*bytesSent*+ = *sent*; }

BEFORE *Microsoft.WindowsMobile.PocketOutlook.SmsMessage.Send*()
PERFORM *smsSent* ≤ 100 → {}

AFTER *Microsoft.WindowsMobile.PocketOutlook.SmsMessage.Send*()
PERFORM *true* → {*smsSent*+ = 1; }

Figure 3.1: A ConSpec contract for the chess game.

SECURITY STATE *int bytesSent* = 0;

BEFORE *System.Net.Sockets.Socket.Send*(*byte*[] *array*)
PERFORM *bytesSent* + *array.Length* ≤ 10000 → {}

AFTER *int sent* = *System.Net.Sockets.Socket.Send*(*byte*[] *array*)
PERFORM *true* → {*bytesSent*+ = *sent*; }

Figure 3.2: An example device policy.

If none of the guards evaluates to true, this means the contract does not allow the method call. For example, in Figure 3.1, if the current state of the policy has *bytesSent* = 2000, then a call to the *Send* method with an array of length 20 will fail all the guards.

Note that the contract can be quite specific about the behavior of the application. For instance, the example contract specifies explicitly that the application will only send messages consisting of 20 bytes over the TCP/IP network. The contract also encodes the upper bound of 100 moves enforced by the application.

The contract in Figure 3.1 matches with a device policy that limits network traffic to (for instance) 10 kilobytes. Such a policy is shown in Figure 3.2. Note the differences between the contract and the policy: while both are written in ConSpec, and both semantically correspond to security automata, the device policy for instance does not make any assumptions about the size of messages sent (beyond the fact that the total size of traffic is limited to 10 kilobytes).

For the remainder of the paper we focus on inlining of policies in multi-threaded Java bytecode. But, the techniques are equally applicable to contracts (instead of policies) and to .NET (instead of Java) [34].

3.3 A Multithreaded Program Model [1]

This paper focuses on multithreaded Java bytecode. The program model used in this chapter inherits most of the definitions from the previous chapter. Issues regarding inheritance and dynamic binding are ignored in this paper since it has already been addressed in previous work and since it is orthogonal to the challenges introduced by adding concurrency.

Multithreaded programs are specified the same way as single threaded programs are, except for the addition of two instructions: `monitorenter` and `monitorexit`. The main difference in the program model lies in the JVM runtime configuration. Whereas the configuration of a single threaded program has a single activation record stack, a configuration of a multithreaded program has one activation record stack per thread. We model this as a map, Θ, from thread identifiers, $tid \in \mathbb{N}$ to activation record stacks, S. The synchronization mechanism provided by the JVM specifies that each object may be used as a semaphore (acquired and released through `monitorenter` resp. `monitorexit`) held by at most one thread. This is modeled by a partial map, $\Lambda : o \to tid$, mapping objects to their owning threads. A configurations thus looks as follows: (h, Λ, Θ).

3.3.1 Executions and Traces

An *execution* of a program *Prg* is, just as in the single threaded case, a finite (or infinite) sequence of configurations $E = C_0C_1 \ldots (C_k)$ with the additional constraint that E is compatible with the happens-before relation as defined by the Java Language Specification (JLS3) [59]. (The implications of this constraint will be elaborated upon in Section 3.3.2 below.) The initial configuration consists of a single thread with a single, normal activation record with an empty stack, no values for local variables, with the main method of *Prg* as its current method and with $pc = 0$.

Since we are interested in inliners that are independent of implementation details concerning e.g. scheduling, memory management and error handling we do not make any distinctions between executions that are allowed by the JLS3 memory model and executions that are possible for an actual implementation.

Observable actions are extended to include the identifier of the thread that performed the action, i.e. a before action for example, has the shape $(tid, c.m, o, v)^{\uparrow}$. The *trace* of E, $\omega(E)$, is the sequence $\alpha_0\alpha_1 \ldots$ with τ actions removed, and $\mathcal{T}(Prg) = \{\omega(E) \mid E \text{ is an execution of } Prg\}$. In this paper we restrict attention to traces T that are realizable, in the sense that $T = \omega(E)$ for some execution E.

3.3.2 Field Accesses and Legal Executions

In this paper, we wish to reason about the behavior of arbitrary multithreaded programs. Therefore, we cannot assume that the programs we consider are correctly

[1] This section has been rewritten to avoid overlap with Section 2.2.

synchronized. This complicates our execution semantics, because non-correctly-synchronized programs may exhibit non-sequentially-consistent executions (Chapter 17 of JLS3 [59]). An execution is sequentially consistent if there is a total order on the field accesses in the execution such that each read of a field yields the value written by the most recent preceding write of that field in this total order. In order to ensure that our semantics captures all possible executions of a program, the transition relation $\rightarrow$ does not constrain the value yielded by a field read; specifically, it does not imply that this value is the value in the heap for that field. However, JLS3 does provide some guarantees, even for non-correctly-synchronized programs. Therefore, below we will consider only *legal executions*. A legal execution is an execution which satisfies both the transition relation $\rightarrow$ and the memory consistency constraints of JLS3.

The *happens-before order* [59] is a partial order on the transitions in an execution. It consists of the program order (ordering of two actions performed by the same thread) and the synchronizes-with order (order induced by synchronization constructs), and the transitive closure of the union of these.

An important guarantee provided by JLS3 that we rely on in this paper, is that if in some legal execution a given field is protected by a given lock, then each read of that field yields the value written by the most recent preceding write of that field. We say that a given field is protected by a given lock in a given execution, if whenever a thread accesses the field, it holds the lock.

3.3.3 Transition Semantics

The transition semantics is similar to the single threaded case. The contextual rule for local variable and stack transitions changes from

$$\frac{S \rightarrow S'}{(h, S) \rightarrow (h, S')} \qquad \text{to} \qquad \frac{\Theta(tid) = S \quad S \rightarrow S'}{(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto S'])}$$

To support thread creation there is a distinguished API method, *Thread.start* which has, besides the standard effect of an API call discussed in Section 2.2.4, an additional side effect of creating a new thread in the configuration:

$$\frac{\begin{array}{c}\Theta(tid) = (M, pc, o :: s, l) :: S \qquad fresh(tid') \\ M[pc] = \texttt{invokevirtual}\ \mathit{Thread.start} \qquad h \vdash o : c\end{array}}{(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (\square) :: \Theta(tid)] \cup (tid' \mapsto (c.run, 0, \epsilon, \epsilon)))}$$

The rules for the locking instructions, `monitorenter` and `monitorexit` update the lock map as expected:

$$\frac{\begin{array}{c}\Theta(tid) = (M, pc, v :: s, l) :: S \\ M[pc] = \texttt{monitorenter} \qquad \Lambda(v) \in \{\bot, tid\}\end{array}}{(h, \Lambda, \Theta) \rightarrow (h, \Lambda[v \mapsto tid], \Theta[tid \mapsto (M, pc + 1, s, l) :: S])}$$

$$\frac{\Theta(tid) = (M, pc, v :: s, l) :: S \qquad M[pc] = \texttt{monitorexit} \qquad \Lambda(v) = tid}{(h, \Lambda, \Theta) \rightarrow (h, \Lambda[v \mapsto \bot], \Theta[tid \mapsto (M, pc + 1, s, l) :: S])}$$

As discussed in Section 3.3.2, field reads return an arbitrary value, and these rules should be complemented with the Java memory model constraints. In particular, a read of a field is guaranteed to always yields the value last written to that field as long as there is a synchronizes-with relation between the read and the write. This is however a constraint of the execution rather than the transition between two configurations, and thus not expressed as a constraint in the rule below.

$$\frac{\Theta(tid) = (M, pc, v :: s, l) :: S \qquad M[pc] = \texttt{putstatic}\ c.f}{(h, \Lambda, \Theta) \rightarrow (h[c.f \mapsto v], \Lambda, \Theta[tid \mapsto (M, pc + 1, s, l) :: S])}$$

$$\frac{\Theta(tid) = (M, pc, s, l) :: S \qquad M[pc] = \texttt{getstatic}\ c.f}{(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (M, pc + 1, v :: s, l) :: S])}$$

3.4 Security Automata

ConSpec policies are formalized in terms of security automata. The notion of security automata was introduced by Schneider [112]. In this paper we view a *security automaton* as an automaton $\mathcal{A} = (Q, \Omega, \delta, q_0)$ where Q is a countable (not necessarily finite) set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Omega \rightharpoonup Q$ is a (partial) transition function, where $\Omega = \Omega^{\uparrow} \cup \Omega^{\downarrow} \cup \Omega^{\Downarrow}$ is the set of observable actions. All states $q \in Q$ are viewed as accepting.

Notation 1. *For a security automaton $\mathcal{A} = (Q, \Omega, \delta, q_0)$, $q \xrightarrow{\alpha} q'$ abbreviates the condition $q' = \delta(q, \alpha)$.*

A security automaton can be derived from a ConSpec policy in the obvious manner. We refer to [1] for details. We assume after clauses of the ConSpec policy to be exhaustive such that an after action can never fail, but it can update the security state.

Definition 9 (Policy Adherence)**.** *The program Prg adheres to security policy $\mathcal{P}_{\mathcal{A}}$, if for all executions E of Prg, $\omega(E) \in \mathcal{P}_{\mathcal{A}}$.*

3.5 Inlining

By *inlining* we refer to the procedure of compiling a contract into a JVML based reference monitor and embedding this monitor into a target program. Formally, an

inliner is a function $\mathcal{I}$ which for each policy $\mathcal{P}$ and program *Prg* produces an inlined program $\mathcal{I}(\mathcal{P}, Prg)$. The intention is that the inserted code enforces compliance with the policy, and otherwise interferes with the execution of the client program as little as possible.

In this section, we first look at various correctness properties for inliners. Then, we introduce the design of our inliner and we prove its correctness.

3.5.1 Inlining Correctness Properties

We first look at the traditional correctness properties for inliners: security, conservativity, and transparency. Then, we introduce a number of new correctness properties that deal with complications caused by the setting of multithreaded Java-like programs: strong conservativity, relative strong conservativity, and weak transparency.

For an inliner whose only expected functionality is to intercept and abort execution of an underlying client program in case of policy violation there are three correctness properties of fundamental interest (cf. [82] for the case of edit automata). Namely, the inliner should enforce policy adherence (security), it should not add new behavior (conservativity), and it should not remove policy-adherent behavior (transparency). More formally:

Definition 10 (Inliner Correctness Properties)**.** *An inliner $\mathcal{I}$ is:*

- Secure *if, for every program Prg, every trace of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$ adheres to $\mathcal{P}$, i.e. $\mathcal{T}(\mathcal{I}(\mathcal{P}, Prg)) \subseteq \mathcal{P}$.*
- Conservative *if, for every program Prg, every trace of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$ is a trace of P, i.e. $\mathcal{T}(\mathcal{I}(\mathcal{P}, P)) \subseteq \mathcal{T}(P)$.*
- Transparent*, if every adherent trace of the client program is also a trace of the inlined program, i.e. if $\mathcal{T}(P) \cap \mathcal{P} \subseteq \mathcal{T}(\mathcal{I}(\mathcal{P}, P))$.*

Recall from Section 3.3.1 that the set of traces $\mathcal{T}(P)$ of a program P is the set of the sequences T of observable actions (i.e. API calls and normal and exceptional returns from API calls) such that there is a (partial or complete) execution of the program whose observable trace is T.

Unfortunately, in case the client program is not well-synchronized, transparency is infeasible in general, because it is not possible to perform inlining without introducing extra synchronization and consequently eliminating certain executions. To illustrate this, consider the program of Figure 3.3. This program is not well-synchronized, since there are data races on fields *beforeA* and *afterA*. Specifically, threads 1 and 2 do not synchronize their accesses of these fields. In the presence of data races, the semantics of Java allow field accesses to appear out of order (this is necessary to allow the JIT compiler, which compiles bytecode to machine code, and the hardware to perform important optimizations). In the example, suppose the body of method *sra* is a simple field assignment. In that case, the JIT compiler can inline this method and then reorder the field accesses, since they are independent.

Thread 1:	$beforeA = 1;$		Thread 2:	$x = afterA;$	
	$sra();$	(A)		$sra();$	(B)
	$afterA = 1;$			$y = beforeA;$	
				IF $(x == 1$ && $y == 0)$	
				$sra();$	(C)

Figure 3.3: Transparency counterexample.

This is why an execution where x gets the value 1 and y gets the value 0 is a legal execution. As a result, the program has a trace with three *sra*() calls.

Now, consider the inlined version of this program. In general, the inlined code needs to access the security state; since multiple security-relevant calls may occur concurrently, these accesses must be synchronized. This means that in general, the inliner inserts synchronization constructs before and after each *sra*() call. As a result, the JIT compiler is no longer allowed to move the accesses of *beforeA* and *afterA* across the *sra*() calls, and the execution where x equals 1 and y equals 0 is no longer legal. Therefore, the inlined program does not have a trace with three *sra*() calls, which means that the inliner is not transparent.

For this reason, the transparency property is only really meaningful for well-synchronized programs. For this restricted case, however, transparency still serves as a useful correctness check: An inliner which is transparent for well-synchronized clients (and, which is secure and conservative) must necessarily exploit race conditions to interfere in an undesirable way with a client program. However, to allow also for programs that are ill-synchronized we look for alternative correctness criteria.

Definition 11. *The* truncation $trunc_{\mathcal{P}}(T)$ *of a trace* T *under a policy* $\mathcal{P}$ *is the greatest prefix of* T *that adheres to* $\mathcal{P}$*.*

Thus, if T adheres to $\mathcal{P}$, $trunc_{\mathcal{P}}(T) = T$, and otherwise T is of the form $\alpha_0 \cdots \alpha_n$ such that, for some $i : 0 \leq i < n$, $\alpha_0 \cdots \alpha_i \in \mathcal{P}$ and $\alpha_0 \cdots \alpha_{i+1} \notin \mathcal{P}$.

Definition 12 (Strong Conservativity)**.** *An inliner* $\mathcal{I}$ *for a given policy* $\mathcal{P}$ *is* strongly conservative *if, for each program Prg, every complete trace (every trace which is not a prefix of any other trace) of the inlined program* $\mathcal{I}(\mathcal{P}, Prg)$ *is the truncation of a complete trace of Prg under* $\mathcal{P}$*:*

$$\mathcal{T}_c(\mathcal{I}(\mathcal{P}, Prg)) \subseteq trunc_{\mathcal{P}}(\mathcal{T}_c(Prg))$$

Example 3. *An abstract version of the program in Figure 3.3 might have traces* AB*,* BA*,* ABC *and* BAC*, all complete and all in* $\mathcal{P}$*. Suppose the set of complete traces of* $\mathcal{I}(\mathcal{P}, Prg)$ *is* $\{AB, BA\}$*. The inliner* $\mathcal{I}$ *is strongly conservative (for this particular program), but not transparent. As another example suppose* Prg' *accepts the traces* A*,* AB*,* AC*,* ABC *such that* $A, AB \in \mathcal{P}$*,* $AC, ABC \notin \mathcal{P}$*, and* AC*,* ABC*, but not* A*,* AB*, are complete. Suppose the only trace of* $\mathcal{I}(\mathcal{P}, Prg')$ *is* A *(so*

A is complete). Again, $\mathcal{I}$ is strongly conservative (for the program Prg') but not transparent.

Proposition 1. *An inliner which is strongly conservative is secure and conservative.*

Proof. Let T be an arbitrary trace of $\mathcal{I}(\mathcal{P}, Prg)$. Pick some $T' \in \mathcal{T}_c(\mathcal{I}(\mathcal{P}, Prg))$ such that T is a prefix of T'. By strong conservativity we know that $T' \in trunc_{\mathcal{P}}(\mathcal{T}_c(Prg))$. By the definition of $trunc_{\mathcal{P}}$ we know that $T' \in \mathcal{P}$ and that $T' \in \mathcal{T}_c(Prg)$. Since each policy is prefix-closed, we have $T \in \mathcal{I}(\mathcal{P}, Prg)$ (the inliner is secure) and since each set of traces emitted by a program is prefix-closed, we have $T \in \mathcal{T}(Prg)$ (the inliner is conservative). □

Strong conservativity implies that the inliner does not add new termination or deadlock behavior. But, in a threaded setting inliners typically use locks to access shared resources, in particular the security state. This may constrain the order of actions. In particular, as is the case in this paper, if the security state is locked across the entire security-relevant call, each such call must be completed before a new security-relevant call can take place. But this may not be compatible with constraints induced by the API, as the following example shows.

Example 4. *Consider an API A with a* barrier *method m that allows two threads to synchronize as follows: When one thread calls m, the thread blocks until the other thread calls m as well. Suppose this method is considered to be security-relevant, and the inliner, to protect its state, acquires a global lock while performing each security-relevant call. This inliner is strongly conservative: The notion of complete trace simply does not take constraints induced by the API into account. On the other hand a client program may consist of two threads, each calling m and then terminating. The inlined version will have one complete trace where one of the threads enters m and then blocks. An uninlined complete trace will contain two calls and returns of m. Thus the inlined complete trace will not be the truncation of an uninlined one at the point of policy violation.*

So the definition of strong conservativity needs to be amended to take such order-inducing API calls into account. Note that the JVM semantics of API calls given in Section 2.2 of Chapter 2 does not do this.

Definition 13 (Relative Strong Conservativity)**.** *An inliner $\mathcal{I}$ is* strongly conservative relative to the API *A, if for each policy $\mathcal{P}$ and each program Prg,*

$$\omega(exec_A^c(\mathcal{I}(\mathcal{P}, Prg))) \subseteq trunc_{\mathcal{P}}(\omega(exec_A^c(Prg)))$$

where $exec_A^c(Prg)$ denotes the set of complete executions of Prg in the API A.

An implication of this definition is that, if in some execution E in some API the inliner kicks in and blocks an SRA α, then there will be an execution of the uninlined program which after the trace of E executes α. The condition does not

guarantee, however, that E without the inliner next would have performed α. This is a consequence of our strictly observational definition of strong conservativity; if more precision is needed, one needs to take internal intermediate states into account, e.g. using bisimulation-based techniques.

As we noted above, inliners generally cannot be transparent for ill-synchronized programs. In fact, some reasonable inliners are not transparent even for well-synchronized programs, because they force the start action and the return action of a security-relevant call to occur atomically, for instance by locking (as we do in this paper).

In that case there may be client program traces with nonatomic API calls and returns that cannot be realized after inlining, only because of execution constraints induced by the inliner. However, these inliners may still be transparent after *canonicalization* of the traces with respect to a set of *atomic* methods:

Definition 14. *A method m is* atomic *in a trace T if, for every normal or exceptional return action from m performed by a thread t in T, no observable action by t intervenes between this return action and the corresponding call action.*

Consider for instance methods m and m' with call and return actions $call_m(t)$, $ret_m(t)$, etc, performed by thread t. Then m is atomic in the traces $call_m(t)ret_m(t)$ $call_{m'}(t)$ and $call_m(t)call_{m'}(t')ret_m(t)$ (with $t' \neq t$) but not in the trace $call_m(t)$ $call_{m'}(t)ret_m(t)$. Notice that "m is atomic in T" is equivalent to stating that "m does not perform callbacks in T".

Definition 15. *Let an API A be given. The* canonicalization *of the trace T with respect to A is the trace $canon_A(T)$ obtained by moving each normal or exceptional return action from a method m in A in T right after the corresponding call action.*

The following is an immediate consequence of our assumptions on the JLS3 execution model in Section 3.3.2:

Proposition 2. *Suppose all methods of API A are atomic in all traces of Prg. If T is a trace of Prg so is $canon_A(T)$.*

Proposition 2 presupposes the "order-oblivious" API semantics of Section 2.2.5, as order-inducing API calls may prevent the shuffling around of return actions needed for the proof.

For inliners that force atomicity of API calls a suitable weakening of the transparency conditions restricts attention to canonic traces in the following way.

Definition 16. *An inliner $\mathcal{I}$ is* weakly transparent *relative to an API A, if for every policy $\mathcal{P}$, every program Prg, and every trace T of Prg that adheres to $\mathcal{P}$, the canonicalization of T equals the canonicalization of some trace of $\mathcal{I}(\mathcal{P}, Prg)$, i.e.*

$$canon_A(\omega(exec_A(Prg)) \cap \mathcal{P}) \subseteq canon_A(\omega(exec_A(\mathcal{I}(\mathcal{P}, Prg))))$$

Notice that weak transparency only makes sense for policies that are closed under canonicalization.

3.5.2 Example Inliner

In order to enforce a policy through inlining, it is convenient to be able to statically decide whether a given event clause applies to a given call instruction. Therefore, in this example inliner, we impose the restriction on policies that they should have simple call matching. We say a policy has simple call matching if for any security-relevant method $c.m$, an `invokevirtual` $d.m$ call is bound at run time to method $c.m$ if and only if $d = c$. We deal with the full inheritance problem in earlier work [28] (Chapter 2 of this thesis).

For simplicity, we also require that the initial values for the security state variables specified by the policy are the default initial values for their corresponding Java types.

The inliner we propose replaces each instruction L : `invokevirtual` $c.m$ where $c.m$ is security-relevant by JVML code corresponding to the pseudo code in Figure 3.4. The replacement is referred to as a block of inlined code.

The inliner locks the security state and stores arguments to the virtual call for use in event handler code. Each piece of event code evaluates guards by reference to the security state and the stored arguments, and updates the state according to the matching clause, or exits, if no matching clause is found. Before passing control to the API method, the original arguments are restored, and immediately upon return the return value on the operand stack is stored in a local variable. On normal return, after successful completion of the normal return event handler code the security state is unlocked and the inlined code fragment is exited. On exceptional return the exception is instead rethrown.

We now prove two central lemmas regarding the inlining strategy described in Figure 3.4 which we will be referring to when proving the overall correctness of the inliner.

Lemma 5. *The following three properties hold for any block of inlined code:*

1. *Whenever control of a thread transfers into a section of inlined code, it does so at label L.*

2. *Whenever the control of a thread exits a section of inlined code, it does so at label excReleased or done.*

3. *Assuming the state of the IRM (the fields of the SecState class) equals the state of the security automaton before entering a section of inlined code, the inlined code will update the IRM state the same way as the observable actions associated with the security relevant method call will update the state of the security automaton.*

Proof. For (1) we note that L was originally the label of the invoke instruction, so any jumps in the original program which had the invoke instruction as its destination will be rerouted to L. Since all other labels in the inlined block are "fresh" there will be no jump from any non-inlined instruction, to an inlined instruction.

Inlined Label	Instruction	Inlined Label	Instruction
L:	`ldc` *SecState*	*afterG*$_j$:	`[eval after` G_j`]`
	`monitorenter`		`ifeq` *exit*
			`[after update` j`]`
	`astore` 0		
	⋮	*afterEnd*:	`aload` n
	`astore` $n-1$		
			`ldc` *SecState*
beforeG$_1$:	`[eval before` G_1`]`		`monitorexit`
	`ifeq` *beforeG*$_2$		
	`[before update 1]`	*afterReleased*:	`goto` *done*
	`goto beforeEnd`		
	⋮	*excG*$_1$:	`[eval exceptional` G_1`]`
			`ifeq` *exceptionalG*$_2$
beforeG$_i$:	`[eval before` G_i`]`		`[exceptional update 1]`
	`ifeq` *exit*		`goto` *excEnd*
	`[before update` i`]`		
			⋮
beforeEnd:	`aload` $n-1$	*exceptinoalG*$_k$:	`[eval exceptional` G_k`]`
	⋮		`ifeq` *exit*
	`aload` 0		`[exceptional update` k`]`
invoke:	`invokevirtual` *c.m*	*excEnd*:	`ldc` *SecState*
			`monitorexit`
invokeDone:	`astore` n		
		excReleased:	`athrow`
afterG$_1$:	`[eval after` G_1`]`		
	`ifeq` *afterG*$_2$	*exit*:	`iconst` -1
	`[after update 1]`		`invokestatic` *System.exit*
	`goto` *afterEnd*		
	⋮	*done*:	

Extra entries in exception handler array:

From	*To*	*Target*	*Type*
invoke	*invokeDone*	*excG*$_1$	*any*
L	*excReleased*	*exit*	*any*
exit	*done*	*exit*	*any*

Figure 3.4: The inlining replacement of L: `invokevirtual` *c.m*.

Label	IRM state	$\mathcal{P}_\mathcal{A}$ state
L	s	q
$\vdots$	$\vdots$	$\vdots$
beforeG₁	s	q
invoke	$\delta(s, \alpha_b)$	q
invokeDone	$\delta(s, \alpha_b)$	$\delta(q, \alpha_b)$
afterG₁	$\delta(s, \alpha_b)$	$\delta(\delta(q, \alpha_b), \alpha_a)$
afterReleased	$\delta(\delta(s, \alpha_b), \alpha_a)$	$\delta(\delta(q, \alpha_b), \alpha_a)$
excG₁	$\delta(s, \alpha_b)$	$\delta(\delta(q, \alpha_b), \alpha_e)$
excReleased	$\delta(\delta(s, \alpha_b), \alpha_e)$	$\delta(\delta(q, \alpha_b), \alpha_e)$
done	$\delta(\delta(s, \alpha_b), \alpha_a)$	$\delta(\delta(q, \alpha_b), \alpha_a)$

Table 3.1: IRM state and security automaton state during execution of a block of inlined code. (The actions α_b, α_a and α_e represents $(tid, c.m, v)^{\uparrow}$, $(tid, c.m, v, r)^{\downarrow}$, $(tid, c.m, v, t)^{\Downarrow}$ respectively.)

Item (2) holds by inspection of the control flow of the inlined block of code.

For (3) we reason as follows. The IRM state and automaton state are unchanged up until *beforeG₁*. The instructions ranging from *beforeG₁* through *beforeEnd* will evaluate the guards and clauses from top to bottom according to the policy semantics and will update the IRM state from s to $\delta(s, \alpha_b)$ (where α_b represents the before action $(tid, c.m, v)^{\uparrow}$). The `aload` and `astore` instructions guarantees that the same arguments are used in the IRM state update as the observable action. The invocation of the security relevant method will then update the security automaton state the same way. Same reasoning applies for the after clauses and exceptional clauses except that in these cases the IRM state and security automaton state will be updated in the opposite order. The state of the IRM and the security automaton at each label is summarized in Table 3.1. The result follows from the fact that the IRM state and security automaton state are equal at *done* and at *excReleased* which are the only possible exit points according to (2). □

Lemma 6. *All modifications to the IRM state and all policy automaton transitions are performed under protection of the SecState.class lock.*

Proof. The identifier of the security state class is chosen not to conflict with any identifiers of the original program, so no instructions of the original program can affect the security state. The result follows from item 1 and 2 of Lemma 5 and the fact that the lock is acquired at label L and released right before *done* and *excReleased*. □

The two main complications which we had to address when designing this inliner are the possibility of internal exceptions, and the interaction of our locking strategy

with API-induced ordering constraints.

The Java Virtual Machine Specification [86] allows a JVM to throw an *InternalError* or *UnknownError* exception at any time whatsoever. This means that, for instance if the JIT compiler runs out of memory, it can throw such an internal exception instead of having to terminate the entire program. Whereas internal exceptions are useful for JVM implementers, they cause complications for the design of our inliner. Specifically, for security, we must maintain the property that whenever no block of inlined code is being executed, the current security state corresponds to the trace of security-relevant actions performed previously during the execution. If an internal exception were to cause control to exit a block of inlined code prematurely, this property would be violated. Therefore, we catch all exceptions that occur anywhere in the inlined code and, when any exception is thrown by any instruction other than the security-relevant call, we exit the program. Notice that this is secure and conservative but not strongly conservative, since we exit at a place where the original program does not exit. Below, we prove strong conservativity of our inliner under the assumption that the JVM is error-free, i.e. it never throws an internal exception.

The other complication is caused by our choice of locking strategy. Since the program may perform multiple security-relevant calls concurrently, accesses to the security state by the inlined code must be synchronized. We do so by protecting the security state using a lock. There are essentially two ways to do so: acquire the lock for the entire duration of the inlined code (*strong synchronization*), or acquire it once when processing the before action, release it before performing the security-relevant call, and then acquire it again for processing the after or exceptional action (*weak synchronization*, analogous to the behavior of the PoET/PSLang inliner [40]). In this paper, we adopted strong synchronization; it has the advantage that both actions associated with a given security-relevant call (i.e. the before action and the after or exceptional action) always occur together, whereas in the case of weak synchronization, the actions from multiple security-relevant calls may be interleaved, leading to a less intuitive policy semantics. A downside of strong synchronization, however, is that it is not applicable in the case where security-relevant methods have synchronization behavior themselves, as discussed above. Indeed, in that case, strong synchronization may introduce deadlocks that did not exist in the original program. Therefore, below, we prove strong conservativity under the assumption that security-relevant methods are *non-blocking*. Furthermore, strong synchronization is not appropriate when the security-relevant methods include long-running operations that benefit from concurrent execution.

We now proceed to state and prove two correctness theorems for our inliner. The first is general, and applies to both ill-synchronized and well-synchronized programs. The second additionally states weak transparency for well-synchronized programs.

Definition 17 (Non-blocking Method)**.** *A method c.m in API A is* non-blocking*, if for all programs Prg and all executions* $E \in exec_A(Prg)$ *either:*

1. *E is infinite, or*

2. *E is terminating, or*

3. *E is deadlocked with final configuration C, and no thread in C is inside c.m.*

Theorem 6. *Let $\mathcal{I}$ be the inliner of Figure 3.4.*

1. *$\mathcal{I}$ is secure*

2. *$\mathcal{I}$ is conservative.*

3. *For an error-free JVM, and relative to an API for which each SRM is non-blocking, the inliner $\mathcal{I}$ is strongly conservative.*

Proof. For security, let $E = C_0 \dots (C_k)$ where $C_i = (h_i, \Lambda_i, \Theta_i)$ be an execution of $\mathcal{I}(\mathcal{P}, Prg)$. Furthermore let $sec(C_i)$ denote the state of the IRM in configuration C_i (the tuple of the values of the static fields in *SecState* as defined by h_i) and let q_i be the state of $\mathcal{P}_\mathcal{A}$ after reading $\omega(C_0 \dots C_i)$. We now show the following invariant: If a thread *tid* holds the lock of the security state in C_i, that is if $\Lambda_i(SecState.class) = tid$ (abbreviated as $locked_i$), then $sec(C_i)$ and q_i have the values specified in Table 3.1 (denoted by $sec_{inl}(pc)$ and $q_{inl}(pc)$) where *pc* is the current program counter of $\Theta_i(tid)$, otherwise (i.e. if $\neg locked_i$ holds) $sec(C_i) = q_i$. Put formally,

$$I_i \equiv (locked_i \Rightarrow sec(C_i) = sec_{inl}(pc) \wedge q_i = q_{inl}(pc)) \wedge (\neg locked_i \Rightarrow sec(C_i) = q_i)$$

This invariant is shown to hold throughout the execution by induction over the length of E. For the base case, $E = C_0$ it holds since the policy state is assumed to be initialized with the default values of the corresponding Java types, which is also what the fields of the *SecState* class holds in h_0. For the inductive step we need to show that if $C_{n-1} \to C_n$ and I_{n-1} holds, then I_n holds. We split this step into four cases:

1. If $\neg locked_{n-1}$ and $\neg locked_n$ (the transition is performed outside an inlined block of code) then by the induction hypothesis $sec(C_{n-1}) = q_{n-1}$. Since no SRA occurs outside of a locked region of code we have $q_{n-1} = q_n$ and since all modifications to the IRM state is performed under the lock (Lemma 6) we have $sec(C_{n-1}) = sec(C_n)$ thus $sec(C_n) = q_n$.

2. If $\neg locked_{n-1}$ and $locked_n$ we have a transition from original code to inlined code. Since such transition neither affects the state of the IRM nor the policy automaton state, we have, by the induction hypothesis, $sec(C_n) = q_n$ which is in accordance with the values in Table 3.1 for the first label of the inlined code. That is $sec(C_i) = sec_{inl}(pc)$ and $q_i = q_{inl}(pc)$ for the *pc* of the thread that acquired the security lock.

3. If $locked_{n-1}$ and $locked_n$ then the transition is either performed by the thread holding the security lock, or by a thread not holding the security lock. In the former case, the thread updates the values according to Table 3.1 (as shown in

Lemma 5) and in the latter case the values (and the *pc* of the thread holding the security lock) are unchanged for the same reason as described in case (1). In both cases I_n hold.

4. If $locked_{n-1}$ and $\neg locked_n$ the thread holding the security lock just released it. As can be seen in Lemma 5 we have $sec(C_i) = q_i$ at each `monitorexit`, thus I_n holds.

By the fact that the IRM state cannot hold the undefined value $\perp$ and by the fact that the policy automaton state either equals the IRM state or holds a value according to Table 3.1 we know that $q_i \neq \perp$ for all $i = 0 \ldots k$, i.e. $\omega(E) \in \mathcal{P}$.

For conservativity we need to show that each possible trace of $\mathcal{I}(\mathcal{P}, Prg)$ is also a possible trace of *Prg*. Since inlined code never affects the values of the variables in the original program, and since the inliner only adds synchronization, each execution of $\mathcal{I}(\mathcal{P}, Prg)$ can be simulated by *Prg* in a way such that all observable actions are preserved. How this is done is described in detail in the proof of Theorem 11 which deals with the slightly more general case of a non-blocking inliner.

We now turn to the third item. Assume an error-free JVM and let $\mathcal{P}$ and *Prg* be given, and assume that the API A is non-blocking with respect to the SRMs of the policy. Consider an execution $E \in exec_A(\mathcal{I}(\mathcal{P}, Prg))$, and let $T = \omega(E)$. There are three cases: Either (1) E is infinite, (2) E is terminating, or (3) E is deadlocked.

(1) We claim it is possible to extract from E another execution E' which replaces each complete execution of an inlined block with the execution of the single invokevirtual instruction for which the block was inserted, and which replaces each partial execution with either nothing or the invokevirtual instruction, depending on whether the instruction concerned is eventually executed in E or not (note that we do not assume fairness so it is possible for a thread from some point onwards never to be scheduled again). Note that this replacement can be done in parallel, since *SecState.class* locks all accesses to the security state.

To see how this is done let E have the shape $C_0 \cdots C_n \cdots C_m \cdots$ such that $C_i = (h_i, \Lambda_i, \Theta_i)$, and such that, for some *tid*, $\Theta_n(tid) = (M_n, pc_n, s_n, r_n) :: S$, $\Theta_m(tid) = (M_n, pc_m, s_m, r_m) :: S$, pc_n points to label L in Figure 3.4, pc_m points to label *done*, and $L \leq pc_i \leq done$ for all $i \in [n, m]$. This situation corresponds to the normal, complete execution of the inlined block in 3.4. Now transform each configuration C_i as follows:

- If $\Lambda_i(SecState.class)$ is set, unset it.
- Whenever pc_i is less than the pc of the invokevirtual instruction, replace s_i by s_n, and otherwise replace s_i by s_m.
- Remove all register values inserted by the inliner from all r_i.

A similar construction is applied to exceptional, complete executions. Since virtual machine errors are disregarded, only the invokevirtual instruction and

the rethrow instruction can raise exceptions. The transformation of exceptional thread configurations is as above, except that the entire frame is replaced, instead of just the operand stack and part of registers. Partial executions are handled in the obvious way. The claim, now, is that the execution thus obtained is an infinite execution of the inlined program with all inlined instructions replaced by noop's and the exception tables restored accordingly. A further transformation step eliminates the noop's and restores the exception tables completely, thus obtaining an execution of the original program. It is clear that the execution remains infinite under this transformation as well. This completes the case.

(2) Assume then that E is terminating. We claim that we can extract an execution E' for the uninlined program which is terminating as well, and such that $\mathcal{T}(E) = \mathit{trunc}_{\mathcal{P}}(\mathcal{T}(E'))$. If E terminates because of a call to *System.exit* by an inlined block for a call of a security-relevant method $c.m$ with target o and arguments v in a thread *tid*, then this can happen only because either all before guards have been evaluated to false, or all after guards have. The latter cannot happen since the disjunction of the guards is a tautology, and since the guards are evaluated correctly on the call parameters. The former can happen only if the trace $\mathcal{T}(E)(\mathit{tid}, c.m, o, v)$ is policy violating. In this case we can eliminate all inlined blocks from E, as above, and reroute control flow at the end of (the transformed) E to the invokevirtual instruction, execution of which was prevented by the exception. In this way we obtain a prefix of E' which can be completed to satisfy the requirements of the statement.

(3) The final case is where E is deadlocked. This can only be the case if each live thread in the final configuration, say C_k, is waiting at a lock. The lock can be either *SecState.class*, or another lock set either from a client instruction, or from an API method. In the latter case, the method call is not inlined, since otherwise the method would be non-blocking. If all locks are set from a client instruction or a non-inlined API call then we extract from E an uninlined complete execution with the same trace, as above. Finally, if a thread is waiting at a security state lock then it must be waiting at the initial `monitorenter` instruction of some inlined block. But that can only be the case if some other thread is deadlocked inside an inlined block, which is impossible, as it would then be deadlocked inside a non-blocking SRM.

□

Lemma 7. *Consider a set of methods $m \in M$. If the methods in M are non-blocking, then M is atomic in any trace of any program.*

Proof. By contradiction. Suppose there is a program *Prg* and a trace T of *Prg* such that some method $m \in M$ performs a callback in T. Then *Prg* can be modified such that it deadlocks inside the callback. It follows that m is not non-blocking. □

Theorem 7. *Relative to an API for which each SRM is non-blocking, $\mathcal{I}$ is weakly transparent for well-synchronized programs and policies that are closed under canonicalization.*

Proof. Consider a policy $\mathcal{P}$ that is closed under canonicalization, and a well-synchronized program *Prg*. Further consider a trace T of *Prg* that adheres to $\mathcal{P}$. We need to prove that there is a trace of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$ whose canonicalization equals the canonicalization of T. Since each SRM is non-blocking, the SRMs are atomic in T. Choose an execution E of *Prg*. Then, let E' be the sequence of configurations obtained by moving each normal or exceptional SRM return transition in E right after the corresponding call transition. Then E' is an execution of *Prg* and its trace is $canon_A(T)$, the canonicalization of T; this is always true because the SRMs are non-blocking. Now, further transform E' by inserting the inlined code prolog operations before each SRM call transition, and by inserting the inlined code epilog operations after each SRM return transition. The resulting sequence of configurations E'' is a legal execution of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$, because *Prg* is well-synchronized and therefore the extra synchronization has no influence on existing field accesses, and because $canon_A(T)$ adheres to $\mathcal{P}$. It follows that $canon_A(T)$ is a trace of $\mathcal{I}(\mathcal{P}, Prg)$. Since $canon_A$ is idempotent, $canon_A(canon_A(T))$ equals $canon_A(T)$ and we have proven the theorem. □

3.6 Case Studies and Benchmarks

The inlining algorithm described above has been implemented in Java on top of the bytecode engineering library, ASM [99]. We present some results and benchmarks of this inliner in four case studies. All case studies comprise a JavaME application and a relevant security policy and are available at `http://www.csc.kth.se/~landreas/inlining`.

ImageExchange (IE) ImageExchange is a combined server/client application that allows users to exchange images over a Bluetooth connection. The user may choose to act as a server and publish selected images, or as a client and download published images.

The policy in this case study restricts the program to only send the file that was last approved by the user. We adapt the Bluetooth and user interface API's slightly to allow this policy to be conveniently formulated.

Snake (SN) This is a classic game of snake in which the player may submit current score to a server over a network connection.

The policy prevents data from being sent over the network after reading from phone memory.

MobileJam (MJ) The MobileJam application is a Bluetooth GPS based traffic jam reporter which utilizes the online Yahoo! Maps API.

	IE	SN	MJ	BN
Security Relevant Invokes	2	2	4	2
Original Size of Binaries	35.2 kb	23.2 kb	196.2 kb	81.8 kb
Inlining Duration	0.56 s	0.48 s	1.80 s	1.25 s
Size increase (inlining):	1.1 %	1.1 %	0.2 %	0.3 %

Table 3.2: Benchmarks for the case studies. Inlining was performed with an Intel Core 2 CPU at 1.83 GHz with 2 Gb memory.

The policy prevents the application from connecting to any URLs other than those starting with `http://local.yahooapis.com`.

BatallaNaval (BN) BatallaNaval is a multiplayer battleship game that communicates through SMS messages.

In this case the policy restricts the number of sent SMS's to a constant.

The applications originate from the case studies of the S^{3}MS project. All policies were successfully enforced by our inliner. The benchmarks for the case studies are summarized in Table 3.2.

3.6.1 Inlining Overhead

To determine the runtime overhead impact of inlining, a program that invoked an empty dummy SRM in a loop was constructed. The execution time of this loop was then measured before and after inlining. The inlining caused the execution time to increase from 407 ms to 1358 ms when the loop iterated 10^6 times on a Sony-Ericsson W810i. This indicates that the overhead in this experiment was 951 nanoseconds per security-relevant call. This suggests that even program that performs many security-relevant calls can be inlined with a close to negligible performance impact. The sample policy used mentioned the dummy SRM in one BEFORE and one AFTER clause with two guards each.

Note, however, that the above experiment did not measure the performance impact resulting from the loss of parallelism due to the serialization of security-relevant calls. Clearly, this impact is highly dependent on the specific application and its inputs.

3.7 Conclusions

We have surveyed the security-by-contract paradigm for mobile application security proposed by the EU FP6 project S^{3}MS. A main technical component of this framework is monitoring and monitor inlining, and as the technical contribution of this paper we have discussed inlining correctness criteria suitable for multi-threaded

bytecode in the style of Java and .NET, and used the criteria to prove correctness for a concrete inlining algorithm.

The inliner we examine is blocking in the sense that the embedded security state is locked across the security-relevant call, thus preventing concurrent accesses to those methods. This may cause serious performance degradation, in particular for methods involving I/O. Indeed, Erlingsson's original inliner [40] avoids this problem by unlocking just at the point of executing the call itself. This, however, is sound only for policies that are *race free*, in the sense of being insensitive to the sequencing of concurrent actions. This topic is addressed in the next chapter where correctness of a non-blocking inliner is proven, but for a restricted policy language. In the present setting one can alleviate the problem to some extent by splitting the security state into disjoint components that are locked separately.

A number of extensions of this work merit attention. First, we do not address inheritance in this paper. This has been considered for the case of sequential Java in [28] (Chapter 2), and multi-threading is not likely to add significant complications. Security automata as we consider here are allowed to be infinite state. This poses no problems for inlining, and it is very useful to correlate actions as in the IE application considered above. (But, contract-policy matching becomes undecidable, for obvious reasons.) We do not allow the heap to be used in policy guards; whereas this would be useful, allowing it creates significant theoretical and practical problems which merit further investigation.

3.8 Acknowledgements

We acknowledge the members of the S^{3}MS consortium, and reviewers, for many valuable discussions on topics related to security policies, monitoring, and inlining. Special thanks go to Fabio Massacci for his competent leadership of the S^{3}MS consortium, and to our colleagues Gurov and Aktug at KTH and Desmet, Philippaerts and Vanoverberghe at K.U.Leuven.

The work on Dam, Lundblad, and Piessens was partially supported by the S^{3}MS project. Additionally, Dam and Lundblad received support through grants 2003-6108 and 2007-6436 from the Swedish Research Council. Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO). Jacobs and Piessens were partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy.

Chapter 4

Security Monitor Inlining and Certification for Multithreaded Java

Mads Dam[1], Bart Jacobs[2], Andreas Lundblad[1], Frank Piessens[2]

[1]KTH, Royal Institute of Technology, Sweden
{mfd, landreas}@kth.se

[2]Katholieke Universiteit Leuven, Belgium
{bartj,frank}@cs.kuleuven.be

Abstract

Security monitor inlining is a technique for security policy enforcement whereby monitor functionality is injected into application code in the style of aspect-oriented programming. The intention is that the injected code enforces compliance with the policy (security), and otherwise interferes with the application as little as possible (conservativity and transparency). Such inliners are said to be correct. For sequential Java-like languages, inlining is well understood, and several provably correct inliners have been proposed. For multithreaded Java one difficulty is the need to maintain a shared monitor state. We show that this problem introduces fundamental limitations in the type of security policies that can be correctly enforced by inlining. A class of race free policies is identified that characterizes the inlineable policies by showing that inlining of a policy outside this class is either not secure or not transparent, and by exhibiting a concrete inliner for policies inside the class which is secure, conservative for arbitrary programs and transparent for well-synchronized programs. The inliner is implemented for Java and applied to a number of practical security policies. Finally, we discuss how certification in the style of Proof-Carrying Code could be supported for inlined programs by using annotations to reduce a potentially complex verification problem for multithreaded Java bytecode to sequential verification of just the inlined code snippets.

4.1 Introduction

Security monitoring, cf. [112, 82], is a technique for security policy enforcement, widely used for access control, authorization, and general security policy enforcement in computers and networked systems. The conceptual model is simple: Security relevant events by a program such as requests to read a certain file, or opening a connection to a given host, are intercepted and routed to a decision point where the appropriate action can be taken, depending on policy state such as access control lists, or on history or other contextual information. This basic setup can be implemented in many different ways, at different levels of granularity. Two approaches of fundamental interest are known, respectively, as execution monitoring (EM) and inlined reference monitoring (IRM) (cf. [66]). In EM [112, 131], monitors perform the event interception and control explicitly, typically by an agent external to the program being executed. Using IRM, cf. [43], the enforcement agent modifies the program prior to execution in order to guarantee policy compliance, for instance by weaving monitor functionality into the application code in an aspect oriented style. Upon encountering a program event which may be relevant to the security policy currently being enforced—such as an API call—the inlined code will typically retrieve both the program state and the security state to determine if the program event should be allowed to go ahead, and if not, terminate execution. Under the assumption that the external monitor is only given capabilities available to an IRM, execution monitoring and inlining enforce the same policies [66]. (In this paper security policies are viewed as sets of traces of observable, security relevant events. If we consider broader classes of policies for e.g. information flow, program rewriting can enforce strictly more policies [66].) But if the external monitor has stronger capabilities, for instance the capability to perform type-unsafe operations, external execution monitoring can be more powerful. Our first contribution is to show that such an effect arises in a multithreaded setting. The fact that an inlined monitor can only influence the scheduler indirectly—by means of the synchronization primitives offered by the programming language—has the consequence that certain policies cannot be enforced securely and transparently by an inlined reference monitor. In support of this statement we give a simple example of a policy which an inliner is either unable to enforce securely, or else the inliner will need to affect scheduling by locking in a way that can result in loss of transparency, performance degradation and, possibly, deadlocks. On the other hand, the policy is easily enforced by an execution monitor which at each computation step can inspect the global execution state.

In spite of this, inlining remains an attractive implementation strategy in many applications. We identify a class of *race free policies*, and show that this class characterizes the policies which can be enforced correctly by inlining in well-synchronized multithreaded Java programs. We argue that the set of race free policies is in fact the largest class that is meaningful in a multithreaded setting. Even if many inliners for multithreaded Java-like languages exist for non-race free policies [40, 7, 65], these inliners must necessarily sacrifice either security or transparency (even for

well-synchronized programs), and anyhow these policies are, in a multithreaded setting, likely to *not* express what the policy writer intended.

The characterization result is proved in two steps: First we show that no inliner exists which can enforce a non-race free policy both securely and transparently without taking implementation specific details of the API, scheduler or JVM into account. We then exhibit a concrete inliner and prove that it correctly enforces all race free policies for well-synchronized programs.

A potential weakness of inlining is that there is a priori no way for a consumer of an inlined piece of code to tell that inlining has been performed correctly. This makes it hard to use IRM as a general software quality improvement tool. Also, it generally forces inlining and execution to take place under the same jurisdiction. To address this problem we turn to certification. For sequential code, certification can be done using Proof-Carrying Code (PCC) [95]. In this case a code producer essentially ships along with the code a correctness proof, which can be efficiently validated at the time the code is invoked by the code consumer. For multithreaded programs, however, the construction of general purpose program logics and verification condition generators is a significant research challenge. We bypass this problem by restricting attention to multithreaded Java bytecode produced using the IRM presented earlier. This allows us to produce security certificates for race free ConSpec policies by combining existing program verification techniques for sequential Java with a small number of syntactic checks on the received code. Certificates are presented as bytecode augmented with a reference ("ghost") monitor. This allows the code consumer to validate certificates against a local, trusted policy by checking the certificate with the monitor suitably replaced. The main result is a soundness result, that if a certificate exists for a program with a given policy, then the program is secure, i.e. the policy is guaranteed not to be violated.

4.1.1 Related Work

Our approach adopts the Security-by-Contract (SxC) paradigm (cf. [12, 94, 33, 75, 17]) which has been explored and developed mainly within the S^3MS project [106].

Monitor inlining has been considered by a large number of authors, for a wide range of languages, mainly sequential ones, cf. [35, 43, 42, 40, 1, 129, 66, 62, 118]. Several authors [62, 17, 7] have exploited the similarities between inlining and AOP style aspect weaving. Erlingsson and Schneider [42] represent security automata directly as Java code snippets. This makes the resulting code difficult to reason about. The ConSpec policy specification language used here [2] is for tractability restricted to API calls and (normal or exceptional) returns, and uses an independent expression syntax. This corresponds roughly to the call/return fragment of PSLang which includes all policies expressible using Java stack inspection [43].

Aktug et al. [1] formalized the analysis of inlined reference monitors and showed how to systematically generate correctness proofs for the ConSpec language, but restricted to sequential Java. Chudnov and Naumann [19] propose a provably correct inliner for an information flow monitor. They prove security and transparency,

but again restricted to a sequential programming language.

Edit automata [83, 82] are examples of security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations. This approach has been fully implemented for Java by Bauer and Ligatti in the Polymer tool [7] which is closely related to Naccio [44] and PoET/PSLang [42].

Certified reference monitors has been explored by a number of authors, mainly through type systems, e.g. in [117, 6, 133, 65, 29], but more recently also through model checking and abstract interpretation [119, 118].

The type-based Mobile system [65] uses a simple bytecode extension to help managing updates to the security state. The use of linear types allows security-relevant actions to be localized to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance. Mobile enforces per-object policies, whereas the policies enforced in our work (as in most work on IRM enforcement) are per session. Since Mobile leaves security state tests and updates as primitives, it is quite possible that Mobile could be adapted, at least to some forms of per session policies. As we show in the present paper, however, the synchronization needed to maintain a shared security state will have non-trivial effects. In particular the locking regime suggested in [65] forces mutually exclusive access to security-relevant calls (it is *blocking*, in the terminology used below), potentially resulting in deadlocks.

In [119, 118] Sridhar and Hamlen explore the idea of certifying inlined reference monitors for ActionScript using model-checking and abstract interpretations. The approach can handle a limited range of inlining strategies including non-trivial optimizations of inlined code. It is, however, restricted to sequential code and to non-recursive programs. Although the certification process is efficient, the analysis has to be carried out by the consumer.

The impact of multithreading has so far had limited systematic attention in the literature. There are essentially two different strategies, depending on whether or not the inliner is meant to block access to the shared security state during security relevant events such as API method calls. In the present paper we focus attention on the non-blocking strategy, which is the most relevant case in practice. In an earlier paper [26] we have examined the blocking strategy. In that case transparency is generally lost, as the inliner may introduce synchronization constraints that rule out correct executions that would otherwise have been possible. However, the blocking inlining strategy is not acceptable in practice as it may cause uncontrollable performance degradation and deadlock which motivates our attention to the non-blocking case in this paper.

The present paper is an extended and completely rewritten version of [23]. In that paper the main results concerning inlineability and race free policies were presented. This version contains a more thorough and self-contained presentation of the policy framework, rewritten and restructured proofs, and a completely rewritten presentation of the inliner. New material is the sections on case studies and evaluation, and on certification.

Overview of the Paper

The rest of this paper is structured as follows: We start by describing the syntax and semantics of the security policies we consider in the paper (Section 4.3). We then define the notion of correct (secure, transparent and conservative) reference monitor inlining (Section 4.4) and show that these correctness criteria cannot be met for the programs and policies previously presented (Section 4.5). An alternative, weaker correctness criterion, is presented (Section 4.6) together with an inlining algorithm that satisfies this criterion (Section 4.7). We then report on our experience with our implementation in five case studies (Section 4.8). Finally we present an approach for certifying an inlined reference monitor (Section 4.9) and present our conclusions and future work (Section 4.10).

4.2 Program Model

The content of this section has been covered in the corresponding sections of Chapter 2 and 3, and is therefore excluded from the version of the paper presented here.

4.3 Security Policies

We study security policies in terms of allowed sequences of API method invocations and returns, as in a number of previous works, cf. [42, 7, 2, 129, 1, 26]. Our work is based on a slight extension of the ConSpec policy specification language [2]. We briefly present our dialect of ConSpec here for completeness.

ConSpec is similar to Erlingsson's PSlang [42], but for tractability it describes conditionals and state updates in a small purpose-built expression language instead of the object language (Java, for PSLang) itself. ConSpec policies represent security automata by providing a representation of a security state together with a set of clauses describing how the security state is affected by the occurrence of a control transfer action between the client code and the API. A control transfer can be either an API method invocation, or a return action, either normal or exceptional. ConSpec proper allows for both per-object, per-session, and per-multisession policies. In this paper we work exclusively with per-session policies which is the case most interesting in practice.

4.3.1 ConSpec Policy Syntax

A ConSpec policy $\mathcal{P}$ consists of a security state declaration of the shape

$$\text{SECURITY STATE } type_1\ s_1, \ldots, type_n\ s_n; \tag{4.1}$$

together with a list of rules. For simplicity, we require that the initial values for the security state variables are the default initial values for their corresponding Java types.

A *rule* defines how the security automaton reacts to an API method call of a given signature. Rules have the following general shape:

$$\begin{aligned} &\mathit{modifier}\ [\mathit{type}\ y\ =]\ c.m(\mathit{type}_1\ x_1, \ldots, \mathit{type}_n\ x_n)\ [\text{ON}\ z] \\ &\quad \text{PERFORM}\ G_1 \rightarrow \{F_1\} \\ &\qquad\qquad \vdots \\ &\qquad\qquad G_m \rightarrow \{F_m\} \\ &\qquad\qquad [\text{ELSE}\ \{F\}] \end{aligned} \tag{4.2}$$

where *modifier* is either BEFORE, AFTER or EXCEPTIONAL, type, $\mathit{type}_1, \ldots, \mathit{type}_n$ are the return and argument types of $c.m$ and G_i and F_i are guards and update statements respectively. BEFORE rules refer to pre-actions, and AFTER and EXCEPTIONAL rules to normal and exceptional post-actions respectively. The method signature following the event modifier specifies the method that the rule applies to. If the policy has a rule defined for a method (of a given signature, of a given modifier type), the method is said to be *security relevant* and we refer to invocations and returns of this method as *security relevant actions*. For instance, if a BEFORE rule for method $c.m$ of a given signature is present then invocations of $c.m$ of that signature are security relevant, but if no AFTER rule is present, normal returns are not regarded as security relevant. There is at most one rule per method defined for each of the three event modifiers. The return value specification is absent for BEFORE rules. Each *clause* of the shape $G_i \rightarrow \{F_i\}$, or the clause ELSE $\{F\}$ expresses a (conditional) update of the security state in the obvious way. The ELSE clause is syntactic sugar for a clause with a constantly true guard. The callee qualifier ON z and the ELSE clause are both optional except for AFTER and EXCEPTIONAL rules for which the ELSE clause is required. Hence a policy can never forbid a return from an API method.

The syntax of the guards G_i and update expressions, F_i and F are only described by example in this paper. Additional examples are given in Section 4.5. The syntactical details are not critical. The only requirements are that expressions are side-effect free and that the expressions allow verification conditions to be efficiently generated. Currently this is an unchecked obligation of the policy-writer but can of course be enforced by restricting the use of methods to an allowed subset of API methods. Guards and update expressions may refer to the state variables, argument and return value variables and the callee variable. Guards are evaluated top to bottom, in order to obtain a deterministic semantics. For the first guard that evaluates to true, the corresponding update expression is executed. If no guard evaluates to true (and no ELSE clause is present) the rule is not allowed to fire. This indicates a security violation and program execution must be terminated.

Example 5. *The policy in Figure 4.1 states that the program has to ask the user for permission each time it intends to send a file over Bluetooth. The specification has two security relevant methods, JOptionPane.showConfirmDialog and*

```
SECURITY STATE String requestorUrl, requestedFile;

BEFORE BluetoothToolkit.sendFile(String destUrl, String file)
  PERFORM
    requestedFile.equals(file) &&requestorUrl.equals(destUrl) → { }

AFTER int reply = JOptionPane.showConfirmDialog(String query)
  PERFORM
    reply > 0 && goodFileQuery(query) → {
      requestedFile = queryFile(query);
      requestorUrl = queryRequestor(query);
    } ELSE { }
```

Figure 4.1: A security specification example written in ConSpec.

```
SECURITY STATE Set initialized = new HashSet();

BEFORE C.initialize()
  PERFORM
    !initialized.contains( Thread.currentThread()) → {
      initialized.add( Thread.currentThread());
    }
```

Figure 4.2: Accessing the current thread identifier in ConSpec.

BluetoothToolkit.sendFile. The specification uses the following three helper functions which we leave undefined:

- *goodFileQuery(query) returns true iff query is a well formulated file send query, for instance because it matches a predefined pattern.*
- *queryRequestor(query) and queryFile(query) returns the requester and file substrings of query respectively.*

Example 6. *The policy in Figure 4.2 expresses that C.initialize can only be invoked once for each thread.*

4.3.2 ConSpec Semantics

A ConSpec policy $\mathcal{P}$ specifies a deterministic automaton (Q, Σ, δ, q_0), explained below, which observes an execution of some client program and changes state, and potentially aborts, according to the policy specification. The details are straightforward. Assume an execution $E = C_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{n-1}} C_n$. The initial state q_0 is obtained by initializing the security state of $\mathcal{P}$ to its default, using, if necessary, a local heap. The alphabet Σ is the set of observable actions. The state space Q is

the set of all type safe assignments to the security state variables. Having reached the i'th configuration of E with automaton state q, if $\alpha_i = \tau$ or if the action is not security relevant (of the given modifier type) the $i+1$:th state is q as well. Otherwise the relevant rule is extracted, variables are bound as indicated above, a matching guard clause is identified, and the first matching update is enacted to compute a new automaton state, and if no matching guard is found, $\omega(E)$ is rejected. If $\omega(E)$ is not rejected it is accepted, and if the traces of all executions of a program *Prg* are accepted by (the automaton determined by) $\mathcal{P}$, *Prg* is said to *adhere to* $\mathcal{P}$.

4.4 Reference Monitor Inlining

A reference monitor inliner (for short just inliner) is a function $\mathcal{I}$ that for each policy $\mathcal{P}$ and program *Prg* produces a program $\mathcal{I}(Prg, \mathcal{P})$ with embedded policy checking functionality.

Our program model makes a clear distinction between the (untrusted) program, and the (trusted) API that it interacts with, and inliners are limited to rewriting the program. This may seem to limit the applicability of our model, as some existing inliners do rewrite the Java Platform API implementation. However, the reader should keep in mind that what we call the API in our model does not necessarily have to map on the Java Platform API. Any inliner has to make a choice as to what part of the system can be rewritten and what remains unchanged. In our model, this is what defines the boundary between program and API. Existing inliners make different choices as to where they draw this boundary: some can rewrite all Java bytecode (including Java Platform API methods that are themselves implemented in Java). For such inliners the API of our model covers only the natively implemented methods. Other inliners will only rewrite application classes and leave the entire Java Platform API untouched. For such inliners the API of our model covers the entire Java Platform API. If an inliner were also to rewrite the native method implementations, then our model is not directly applicable, since we only model Java bytecode. But a similar model where the program consists of assembly code and the API consists of system calls could be built and would reveal the same limitations as the one we discuss in this paper: the limitations are fundamental.

One assumption that does limit the applicability of the model is the fact that we assume that API method invocations and returns are good abstractions of the security relevant actions that policies want to talk about. In other words, the limitations on enforceable policies that we identify in this paper are only applicable to policies that talk about API method invocations and returns, where the API is defined as above: the boundary of the part of the system that can be rewritten. The implementation of an API method is trusted to achieve exactly the effect that the policy writer wants to talk about. Hence we do not consider calls from within the implementation of an API method to other API methods.

Another consequence of the model is that an inliner can never prevent an API method from returning: inlined code can only be executed after the call has re-

turned. This is why post-actions are required to always be enabled in ConSpec.

4.4.1 Inlining Correctness Properties

There are three correctness properties of fundamental interest (cf. [82, 66]), namely security, conservativity and transparency.

Security, arguably the most important property of an inliner, states that all possible traces of the inlined program should be compliant with the policy provided to the inliner.

Definition 18 (Security)**.** *An inliner $\mathcal{I}$ is* secure *if, for every program Prg and policy $\mathcal{P}$, every trace of the inlined program $\mathcal{I}(Prg, \mathcal{P})$ adheres to $\mathcal{P}$, i.e.*

$$\mathcal{T}(\mathcal{I}(Prg, \mathcal{P})) \subseteq \mathcal{P}.$$

Transparency states that the policy adherent behavior of the client program should be preserved by the inliner.

Definition 19 (Transparency)**.** *An inliner $\mathcal{I}$ is* transparent*, if for every policy $\mathcal{P}$ and program Prg, each trace of Prg that adheres to $\mathcal{P}$ is also a trace of the inlined program, i.e.*

$$\mathcal{T}(Prg) \cap \mathcal{P} \subseteq \mathcal{T}(\mathcal{I}(Prg, \mathcal{P})).$$

Conservativity states that no behavior should be added to the original program.

Definition 20 (Conservativity)**.** *An inliner $\mathcal{I}$ is* conservative *if, for every program Prg and policy $\mathcal{P}$, every trace of the inlined program $\mathcal{I}(Prg, \mathcal{P})$ is a trace of Prg, i.e.*

$$\mathcal{T}(\mathcal{I}(Prg, \mathcal{P})) \subseteq \mathcal{T}(Prg).$$

Other correctness properties have been proposed, such as the concept of *strong conservativity*, which was used in [26]. This correctness criteria refines the notion of conservativity and forbids arbitrary truncation of the traces. Since this is mostly useful for the case of a blocking inliner to account for the necessary loss of transparency, cf. [26], we do not discuss it further in this paper.

4.5 Limitations of Inlining in a Multithreaded Setting

In this section, we show that the traditional correctness criteria for inlined monitors are too strong in a multithreaded setting. While it is possible to securely and transparently enforce any policy specified as explained in Section 4.3 by an *external* monitor implemented as part of the JVM, it is impossible to do this with an inlined monitor without taking specificities of the API implementation and/or virtual machine into account.

One of the key differences between an external monitor and a monitor inlined in the client program is the ability to affect the behavior of a thread executing within

```
SECURITY STATE
    boolean ok = false;

BEFORE c.m()
    PERFORM
        true → { ok = true; }

BEFORE c.n()
    PERFORM
        ok == true → { }
```

(a) Not enforceable by inlining.

```
SECURITY STATE
    boolean ok = false;

AFTER c.m()
    PERFORM
        true → {ok = true; }

BEFOREc.n()
    PERFORM
        ok == true → { }
```

(b) Enforceable by inlining.

Figure 4.3: Policy enforceability through monitor inlining.

```
class SomeClass {
    public static void main(String[]args){
        new Thread() { public void run() { c.m(); }}.start();
        c.n();
    }}
```

Figure 4.4: A program invoking $c.m$ and $c.n$ in a non-deterministic order.

an API-method. As opposed to an external reference monitor, an inlined reference monitor cannot in general control the scheduling of such a thread, and this affects the enforceability of certain policies.

Consider the policy in Figure 4.3a. This policy states that $c.n$ may only be called when *ok* has been set to true, that is, after $c.m$ has been called (but not necessarily returned). So the trace $T_1 = (tid, c.m, o, v)^{\uparrow}, (tid', c.n, o', v')^{\uparrow}$ is allowed by the policy, but the trace $T_2 = (tid', c.n, o', v')^{\uparrow}, (tid, c.m, o, v)^{\uparrow}$ is not. Now consider a program whose traces include both T_1 and T_2, for instance the one shown in Figure 4.4. For an inliner to exclude trace T_2 from this program (but keep the trace T_1), it could either exploit some implementation-dependent knowledge of the virtual machine, or else it would have to introduce a happens-before relation between $(tid, c.m, o, v)^{\uparrow}$ and $(tid', c.n, o', v')^{\uparrow}$. In the latter case we note that there is no way such a happens-before relation can be enforced by the inliner since, by convention, after the call has been made, the control lies within the API method which is not to be altered. In terms of the formal semantics of API calls given in Section 2.2.4 the former case is also ruled out. To lift this to practical virtual machines let us say that a correctness property is *uniform* if it holds for all API implementations, including the fully nondeterministic one from Chapter 2. Using the API semantics from Chapter 2, the inlined program will either have both traces

T_1 and T_2 (in which case the inliner is not secure) or it will have neither of the two traces (in which case the inliner is not transparent). We have thus shown:

Theorem 8. *No inliner can be both uniformly transparent and uniformly secure for the policy $\mathcal{P}$ in Figure 4.3a.* □

Evidently, an inliner could "over-approximate" and guard the entire call to $c.m$ by a lock and let the monitor release the lock after $c.m$ has returned, but in that case the monitor would be enforcing the stronger policy shown in Figure 4.3b and prevent some traces that are allowable by the policy in Figure 4.3a.

4.6 Race Free Policies

Generalizing from the example in Figure 4.3a, the key issue is that no client program (not even after inlining) can arbitrarily constrain the set of observable traces. Given a certain trace of observable actions, in general there will be permutations of that trace that are also possible traces of the client program no matter what synchronization efforts the client performs. These permutations that are always possible are captured by the notion of *client-order preserving* permutations.

Definition 21 (Client-order Preserving Permutation)**.** *A permutation $\pi(T)$ of a trace T of observable actions is* client-order preserving *if, for all i and j such that $i < j$ and (a) T_i and T_j take place on the same thread, or (b) T_i and T_j correspond to a post- resp. pre-action, then $\pi(i) < \pi(j)$.*

The intuition is the following: the client can control pre-actions, and can only observe post-actions. If a pre-action takes place somewhere after a post-action, the client *could* have synchronized to ensure this ordering. The client cannot perform such synchronization for concurrent pre-actions or concurrent post-actions.

If a policy accepts a given trace, but rejects a client-order preserving permutation of the trace, then that policy is not securely and transparently enforceable by inlining a monitor in the client code. This is captured by the following definition:

Definition 22. *A policy is* race free *iff, for any trace T and any client-order preserving permutation T' of T, if T is allowed, then T' is allowed.*

As an example, the policy in Figure 4.1 is race free. As a broader class of examples consider the class of policies where the security state is a set of permissions, pre-actions require a permission to be present in this set and cause the permission to be removed, and post-actions restore the permission. Such policies are race free. This can be checked for instance by using Proposition 4 below.

We show further that the class of race free policies is a lower bound on the class of policies enforceable by inlining by constructing an inliner that is secure and conservative (for arbitrary programs), and transparent (for well-synchronized programs) for this class of policies.

The following theorem shows that the bound is tight.

Theorem 9. *No inliner can be uniformly secure and uniformly transparent for a non-race free policy, and no inliner can be transparent for ill-synchronized programs.*

Proof. For the first part, let $\mathcal{P}$ be a non-race free policy. It suffices to show that $\mathcal{P}$ is not enforceable for the fully non-deterministic semantics of Section 2.2.4. By definition there is a trace T of some program *Prg* which $\mathcal{P}$ accepts and a client-order preserving permutation T' of T which $\mathcal{P}$ rejects. Now for an inliner, $\mathcal{I}$, to be transparent, $\mathcal{I}(Prg, \mathcal{P})$ has to admit the trace T. But, since a client-order preserving permutation respects the happens-before relations stipulated by any program, $\mathcal{I}(Prg, \mathcal{P})$ must also admit the trace T', which means that $\mathcal{I}$ is not secure.

For the second part we refer to Section 3.5.1 in the previous chapter. □

A policy for which there exists a (uniformly) secure, transparent and conservative inliner is said to be (uniformly) *inlineable*. The corollary below follows immediately.

Corollary 1. *The set of uniformly inlineable policies is a subset of the set of race free policies.*

Proof. Let $\mathcal{P}$ be an arbitrary uniformly inlineable policy. By definition there exists a uniformly secure and transparent inliner for $\mathcal{P}$, thus by Theorem 9, $\mathcal{P}$ must be race free. □

An interesting question is how to decide if a policy is race free. Using Lipton's moverness terminology [87] we obtain the following:

Proposition 3. *It is a necessary and sufficient condition for race freedom that all pre- and post-actions occurring in different threads are right- resp. left-movers, in the set of allowed observable traces. (I.e., if a trace T is allowed, then swapping a pair of consecutive actions α_1,α_2 in different threads where α_1 is a pre-action or α_2 is a post-action yields an allowed trace.)*

Proof. Such swappings generate the client-order preserving permutations. □

In particular, if such swappings always have the same effect on the policy state, we know the policy is race free:

Proposition 4. *The following is a sufficient condition for race freedom. For any state q_1 of the security automaton corresponding to a given policy, and for any pre-action α_1 and post-action α_2 with different thread identifiers, if $\delta(\delta(q_1, \alpha_1), \alpha_2) = q_2$ then $\delta(\delta(q_1, \alpha_2), \alpha_1) = q_2$.*

Proof. These conditions imply the conditions from Proposition 3. □

Sufficient syntactical criteria for the conditions of Proposition 4 are easily identified. For example, for the common case where the security state is a set of permissions, a sufficient requirement is that pre-actions only consume permissions from the set, and post-actions only add permissions.

4.6.1 Relevance of non-race free policies

Are there interesting or practically relevant policies that are not race free? A policy that is not race free imposes constraints not only on the client program, but also on the API implementation and/or the scheduler. Hence, we argue that such policies do not make sense. Even if an enforcement mechanism (such as an external execution monitor) could enforce the policy, the result of the enforcement is most likely not in line with what the policy writer intended to express. Policies impose constraints on API method invocations because of the effects (such as writing a file, reading from the network, activating a device, ...) that these API implementations have. A policy such as the one in Figure 4.3a intends to specify that initiation of one effect should come after the initiation of another effect. But without further information about the API implementations and the operation of the scheduler, there is no guarantee that enforcing this ordering on the API invocations will also enforce this ordering on the actual effects.

In other words, the race in the policy that makes it impossible for an inliner to enforce the policy, also makes it impossible to interpret method invocations soundly as initiations of effects.

Hence, a policy that is not race free either indicates a bug in the policy (for instance, the policy writer intended to specify the policy in Figure 4.3b instead of the policy in Figure 4.3a – an easy mistake to make as in the single-threaded setting both policies are equivalent) or it is an indication of a misunderstanding of the policy writer (for instance the policy writer considers the start of the API method invocation as a synonym of the start of the effect the API method implements). Jones and Hamlen [72] make a similar observation for a different class of policies that is hard to enforce with inlining.

As a consequence, the practicality of inlining as an enforcement mechanism is not at stake, and detection of races in policies is useful as a technique to detect bugs in policies.

4.7 Race free Policies are Inlineable

In this section we show that race free policies can be enforced by IRM, by giving an inlining scheme that is secure and conservative (for arbitrary programs) and transparent (for well-synchronized programs) for race free policies. From this point onward we restrict attention to the API semantics from Section 2.2.4 in order to eliminate from consideration pathological virtual machines that may introduce implementation-dependent errors or, e.g., manipulate the scheduler in non-standard ways. For sequential Java a correct inlining scheme is already known to exist. In this section we show that the race free policies is the maximal set of policies for which correct inlining in well-synchronized programs is possible.

The state of the IRM might possibly be updated by several threads concurrently. The updates to this state must therefore be protected by a global lock. A key design choice is whether to keep holding this lock during the API call, or to temporarily

release the lock during the call and reacquire it after the call has returned. In the former case we say that the inliner is *blocking*, and in the latter we say it is *non-blocking*.

The first choice (locking across calls) is easier to prove secure, as there is a strong guarantee that the updates to the security state happen in the correct order. The implications of this design choice was examined in [26] (Chapter 3). The problem is that a blocking inliner can introduce deadlocks in the inlined program (even if the target program is well-synchronized) and it is thus not transparent. Consider for instance an API with a barrier method B that allows two threads to synchronize as follows: When one thread calls B, the thread blocks until the other thread calls B as well. Suppose this method is considered to be security-relevant, and the inliner, to protect its state, acquires a global lock while performing each security-relevant call. For a client program that consists of two threads, each calling B and then terminating, the inliner will introduce a deadlock, as one thread blocks in B while the other thread blocks on the global lock introduced by the inliner.

Even if it does not lead to deadlock, acquiring a global lock across a potentially blocking method call can cause serious performance penalties. For this reason, our algorithm releases the lock before calling an API method. In fact, our algorithm ensures that the global lock is only held for very short periods of time.

It is worth emphasizing that the novelty in this section is not the inlining algorithm itself: The algorithm is similar to existing algorithms developed in the sequential setting and the locking strategy is relatively straightforward. The contribution, rather, is the proof that the notion of race free policies gives an exact characterization of the class of policies enforceable on multithreaded Java-like programs by a non-blocking inlining scheme.

4.7.1 Inlining Algorithm

In order to enforce a policy through inlining, it is convenient to be able to statically decide whether a given policy clause applies to a given call instruction. Therefore we impose the restriction on programs that they should have *simple call matching*, namely that for all security-relevant methods $c.m$, an `invokevirtual` $d.m$ call is bound at run time to method $c.m$ if and only if $d = c$. Essentially, this means that we ignore all issues concerning inheritance and dynamic binding. These concerns are orthogonal to the results of this paper, and it has been described elsewhere how to deal with them [129, 1].

The inliner, $\mathcal{I}_{Ex}$, takes a policy with security state definition and event rules of the shapes (4.1) and (4.2) (see Section 4.3) and applies it to a Java bytecode program. The inliner uses static fields s_i of type $type_i$ of an auxiliary class *SecState* to store the shared security state, as in the ConSpec security state declaration (4.1). (In general a unique name needs to be chosen for the security class itself, to allow the inliner to be iteratively applied). We assume for simplicity that rules are present for each of the three rule types BEFORE, AFTER and EXCEPTIONAL, and we use $G_{i,t}$, F_t, $F_{i,t}$, $t \in \{b, a, e\}$ to indicate the corresponding guard and update blocks

Inlined label	Instruction	Inlined label	Instruction
L:	*lock* `SecState`		`ifeq` *afterElse*
	store arguments		$[eval(F_{m,a})]$
	store callee		`goto` *afterEnd*
beforeG$_1$:	$[eval(G_{1,b})]$	*afterElse*:	$[eval(F_a)]$
	`ifeq` *beforeG*$_2$	*afterEnd*:	*restore return value*
	$[eval(F_{1,b})]$		*unlock* `SecState`
	`goto` *beforeEnd*		`goto` *done*
	$\vdots$	*excG*$_1$:	*lock* `SecState`
beforeG$_m$:	$[eval(G_{m,b})]$		*store exception*
	`ifeq` *beforeElse*		$[eval(G_{1,e})]$
	$[eval(F_{m,b})]$		`ifeq` *excG*$_{2,e}$
	`goto` *beforeEnd*		$[eval(F_{1,e})]$
beforeElse:	$[eval(F_b)]$		`goto` *excEnd*
beforeEnd:	*restore callee*		$\vdots$
	restore arguments	*excG*$_m$:	$[eval(G_{m,e})]$
	unlock `SecState`		`ifeq` *excElse*
invoke:	`invokevirtual` $c.m$		$[eval(F_{m,e})]$
invokeDone:	*lock* `SecState`		`goto` *excEnd*
	store return value	*excElse*:	$[eval(F_e)]$
afterG$_1$:	$[eval(G_{1,a})]$	*excEnd*:	*restore exception*
	`ifeq` *afterG*$_2$		*unlock* `SecState`
	$[eval(F_{1,a})]$	*excReleased*:	`athrow`
	`goto` *afterEnd*	*exit*:	`iconst` -1
	$\vdots$		`invokestatic` *System.exit*
afterG$_m$:	$[eval(G_{m,a})]$	*done*:	

Figure 4.5: The inlining replacement of L: `invokevirtual` $c.m$.

in (4.2). The compilation of guard clauses and update blocks into bytecode is well understood and we simply assume that they are compiled into basic blocks denoted by $eval(G_{i,t})$, $eval(F_t)$ and $eval(F_{i,t})$ that behave as required. In particular, the callee is extracted from the top of the stack, arguments from stack elements $1, \ldots, n$, security state variables from corresponding fields of the *SecState* class, and the calling thread identifier is extracted using *Thread.currentThread*. The inliner then replaces each instruction L : `invokevirtual` $c.m$ of arity n where $c.m$ is security-relevant by bytecode implementing the pseudo-code in Figure 4.5. The inliner locks the security state by acquiring the lock associated with the *SecState* class, and stores callee and arguments to the method call for use in event handler code using fresh local variables. The security state lock is taken by executing first `ldc SecState` and then entering the monitor. The use of a static class for the security state makes it easy to determine statically that locks taken or released outside the inlined code snippets do not affect the security state lock. The lock is released just prior to invocation of the inlined call, and retaken after return. Each piece of event code

From	*To*	*Target*	*Type*
invoke	*invokeDone*	$excG_1$	*any*
L	*excReleased*	*exit*	*any*
exit	*done*	*exit*	*any*

Figure 4.6: Exception handler array modifications

evaluates guards by reference to the security state and the stored arguments, and updates the state according to the matching clause, or exits, if no matching clause is found. Thus, if F_b (i.e. the ELSE-clause) is absent the block at *beforeEnd* is replaced by a jump to *exit*.

If no BEFORE rule is present, evaluation of the BEFORE guards and update clauses is evidently not performed. Arguments and callee are still stored in local variables and restored before the method is called, as arguments and callee may be needed for evaluating an AFTER or EXCEPTIONAL rule.

The exception handler array is modified by adding the entries in Figure 4.6 and adding $done - L - 1$ to all offsets above L in the original handler. Exceptions emanating from the call to $c.m$ are routed to the inlined handler at $excG_1$. After processing of EXCEPTIONAL events the security state is unlocked and the exception rethrown. Exceptions caused by inlined instructions are routed to *exit*.

One complication is the possibility of internal exceptions. The Java Virtual Machine Specification [86] allows a JVM to throw an *InternalError* or *UnknownError* exception at any time whatsoever. This means that when the JVM attempts to, for instance, compile a piece of bytecode but does not have sufficient memory to complete the operation, it can throw an internal exception instead of having to terminate the entire program. Whereas internal exceptions are useful for JVM implementers, they cause complications for the design of our inliner. Specifically, for security, we must maintain the property that whenever no block of inlined code is being executed, the current security state matches the trace of security-relevant actions performed previously during the execution. If an internal exception were to cause control to exit a block of inlined code prematurely, this property would be violated. Therefore, we catch all exceptions that occur anywhere in the inlined code and, when any exception is thrown by any instruction other than the security-relevant call, we exit the program. Notice that this is secure and conservative, since we exit at a place where the original program does not exit. But in pathological cases (such as a JVM which chooses to randomly abort execution whenever a static class `SecState` is defined) transparency may fail. For this reason we assume below that the JVM is error-free, i.e. it never throws internal exceptions.

4.7.2 Correctness

We first prove security, i.e. that for each program *Prg* and race free policy $\mathcal{P}$, $\mathcal{T}(\mathcal{I}_{Ex}(\mathcal{P}, Prg)) \subseteq \mathcal{P}$. The basic insight is that race freedom ensures that actions and monitor updates are sufficiently synchronized so that security is not violated. To see this we need to compare the observable actions of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ with the corresponding *monitor actions*, i.e. actions of the inlined code manipulating the inlined security state. We use the notation $\mathrm{mon}(\alpha)$ for the monitor action corresponding to the observable action α. The monitor action $\mathrm{mon}(\alpha)$ occurs at step $i \in [0, n-1]$ of the execution $E = C_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{n-1}} C_n$, if the instruction scheduled for execution at configuration C_i is `monitorexit`, corresponding to one of the unlocking events in Figure 4.5 for the action α. We refer to the points in E at which the monitor actions occur, as *monitor commit points*.

Depending on which case applies we talk of the monitor action $\mathrm{mon}(\alpha)$ as a monitor pre-, normal monitor post-, or exception monitor post-action. Then the *extended trace of* E, $\tau_e(E)$, lists all *extended actions*—that is, non-τ actions and monitor actions—of E in sequence, and the *monitor trace* of E, $\tau_m(E)$, projects from $\tau_e(E)$ the monitor actions only. Let β range over extended actions.

Pick now an execution E of an inlined program $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$, and let $\tau_e(E) = \beta_0, \ldots, \beta_{n-1}$. Say that E is *serial* if in $\tau_e(E)$ there is a bijective correspondence between actions and monitor actions, and if each pre-action α is immediately preceded by the corresponding monitor action $\mathrm{mon}(\alpha)$, and each post-action α' is immediately succeeded by its corresponding monitor action $\mathrm{mon}(\alpha')$.

We first observe that monitor traces are just traces of the corresponding security automaton:

Proposition 5. *Let E be an execution of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$. Then $\tau_m(E) \in \mathcal{P}$.*

Proof. The locking regime ensures that all monitor actions, hence automaton state updates, are happens-before related. Since each thread updates the automaton state according to the transition relation, the result follows. □

Lemma 8. *Assume that $\mathcal{P}$ is race free. For any execution E of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ there exists a serial execution E' such that $\tau(E) = \tau(E')$.*

Proof. Let E of length n be given as above. Note first that, by the happens-before constraints, the bijective correspondence must be such that pre-actions are preceded by their corresponding monitor actions, and vice versa for post-actions. We construct the execution E' by induction on the length m of the longest serial prefix of $\tau_e(E)$. If $n = m$ we are done so assume $m < n$. Say that β_{m-1} is produced by thread t. Note first that β_{m-1} can be either a pre-action or a monitor post-action as E' is serial, and that β_m can be either a post-action or a monitor pre-action. For the latter point assume for a contradiction that β_m is a pre-action. Then β_m must be produced by a thread $t' \neq t$, by the control structure of the inlining algorithm, Figure 4.5. The last action in $\tau_e(E')$ by thread t' must be a monitor pre-action $\beta_l = \mathrm{mon}(\beta_m)$ for $0 \leq l < m-1$ and, as each action records the tid, $\beta_k \neq \beta_m$

for any $l < k < m - 1$. But then the extended trace $\beta_0, \ldots, \beta_{m-1}$ is not serial, a contradiction. The case where β_m is a monitor post-action is similar.

Now, if β_m is a post-action, say, then thread t is at one of the control points *invokeDone* or $excG_1$. Either $\text{mon}(\beta_m) = \beta_{m'}$ for some $m' > m$ or else thread t does not produce any extended actions in $\tau_e(E')$ after m. In the latter case it is possible to schedule $\text{mon}(\beta_m)$ directly, as the guards for post-actions are exhaustive. In the former case we need to also argue that all extended actions β_k for $m \leq k$ and $k \neq m'$ remain schedulable, even after scheduling $\text{mon}(\beta_m)$ right after β_m. But this follows from the left-moverness of monitor post-actions with respect to both monitor actions, Proposition 3, and non-monitor actions on different threads.

If on the other hand β_m is a monitor pre-action $\text{mon}(\alpha)$. If $\beta_{m+1} = \alpha$ we are done. Otherwise β_{m+1} is a monitor action or non-monitor action of another thread, and regardless which, by rescheduling, β_m can be moved right until it is left adjacent to α. But this case can only apply a finite number of times at the end of which E' can be extended. This completes the proof. □

Inliner security is now an easy consequence.

Theorem 10 (Inliner Security)**.** *If $\mathcal{P}$ is race free then $\mathcal{I}_{Ex}$ is secure, i.e. $\mathcal{T}(\mathcal{I}_{Ex}(\mathcal{P}, Prg)) \subseteq \mathcal{P}$.*

Proof. Pick any execution E of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$. Use Lemma 8 to convert E to an execution E' with the property that $\tau(E) = \tau(E') = \tau_m(E') \in \mathcal{P}$ by Proposition 5 and since E' is serial. □

For conservativity, our proof is based on the observation that there is a strong correspondence between executions of an inlined program, and executions of the underlying program before inlining: An execution of the original program can be obtained by removing all transitions which are due to inlined instructions, all local variables and mappings in the heap that are due to inlined code and all mappings in the lock map which are due to inlined acquierings of the security state lock.

Theorem 11. *$\mathcal{I}_{Ex}$ is conservative, i.e. $\mathcal{T}(\mathcal{I}_{Ex}(\mathcal{P}, Prg)) \subseteq \mathcal{T}(Prg)$.*

Proof. Given an execution $E = C_0 \ldots (C_k)$ of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ we show that there exists an execution $E' = C'_0 \ldots (C'_l)$ of *Prg* such that $\omega(E) = \omega(E')$. We do this by showing that $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ simulates *Prg* in a way such that all observable actions are preserved. The simulation relation is defined over configurations as follows: $(h, \Lambda, \Theta) \sim (h', \Lambda', \Theta')$ if

1. $h'(SecState.f) = \bot$ for each field f in *SecState* and $h'(x) = h(x)$ otherwise.
2. $\Lambda'(SecState.class) = \bot$ and $\Lambda'(tid) = \Lambda(tid)$ otherwise.
3. $dom(\Theta') = dom(\Theta)$ and for each $tid \in dom(\Theta')$ there is a one-to-one mapping of activation records between $\Theta'(tid)$ and $\Theta(tid)$ such that for each activation record $(M', pc', s', l') \in \Theta'(tid)$ there is a corresponding activation record $(M, pc, s, l) \in \Theta(tid)$ such that

a) $M = M'$,

b) pc' equals L if pc points at an instruction in range $[L, done)$ in Figure 4.5, and pc otherwise,

c) s' equals s with any values pushed by inlined instructions removed, and

d) l' equals l with any variables introduced by inlined code removed.

First we note that $C_0 \sim C'_0$ (trivial check). We now show that

1. if $C_a \xrightarrow{\alpha} C_b$ and $C_a \sim C'_c$ for some C'_c then there exists a C'_d such that $C'_c \rightarrow^* C'_d$ and such that $C_b \sim C'_d$, and
2. the simulated transitions ($C'_c \rightarrow^* C'_d$) emits the same observable action (α) as the original transition

as depicted below:

$$\begin{array}{ccccc} & & C_a & \sim & C'_c \\ \mathcal{I}_{Ex}(\mathcal{P}, Prg) & & \downarrow \alpha & & \downarrow \alpha \\ & & & & \downarrow * \\ & & C_b & \sim & C'_d \end{array} \qquad Prg$$

Assume $C_a \xrightarrow{\alpha} C_b$ and $C_a \sim C'_c$ for some C'_c. If $C_a \xrightarrow{\alpha} C_b$ represents an execution of an inlined instruction (if the program counter of C_a points at an instruction in range $(L, done)$ of the code in Figure 4.5) we let $C'_d = C'_c$. This is valid because no inlined instruction affects any non-inlined variables, and because the program counter of C_b maps back to the same program counter as C'_c (thus $C'_c \sim C_b$) and because no inlined instruction emits an observable action (thus $\alpha = \tau$). If $C_a \rightarrow C_b$ is due to an instruction of the original program, then there exists a C'_d such that $C'_c \xrightarrow{\alpha} C'_d$ and such that $C_b \sim C'_d$ since both C_a and C'_c refers to the same instruction and since no non-inlined instruction relies on the state of any variables added by the inliner.

We have now shown that each execution of the inlined program E has a corresponding execution of the original program E' such that $\omega(E) = \omega(E')$ i.e. the inliner conservative. □

For transparency we need to show the opposite: That each execution of the original program has a corresponding execution of the inlined program. This is a slightly more delicate task due to the relaxed memory consistency model. As described in Section 3.5.1, some executions of ill-synchronized programs can exhibit traces which are possible only due to certain instruction reorderings. While such instruction reorderings may be allowed by the memory model in the original program, it may not be allowed in the inlined program due to the added synchronization actions (`monitorenter` and `monitorexit`). For this reason, we restrict attention to well-synchronized programs in the theorem below.

Theorem 12. *The inliner $\mathcal{I}_{Ex}$ is transparent for well-synchronized programs, i.e. $\mathcal{T}(Prg) \cap \mathcal{P} \subseteq \mathcal{T}(\mathcal{I}(Prg, \mathcal{P}))$ for any well-synchronized program Prg.*

Proof. We show that for each policy adherent execution of the original program $E = C_0 \ldots (C_k)$ there is an execution $E' = C'_0 \ldots (C'_l)$ (with the same trace) of the inlined program. We do this by showing that each transition of E can be simulated by one or more transitions in E'. Let $\sim$ be a relation over configurations defined as follows: $(h, \Lambda, \Theta) \sim (h', \Lambda', \Theta')$ if

1. h and h' agrees on all non-inlined variables
2. $\Lambda = \Lambda'$
3. $dom(\Theta') = dom(\Theta)$ and each activation record in Θ equals the corresponding activation record in Θ' with the *pc* offset to the closest preceding non-inlined instruction and the local variables added by the inliner removed.

We note that $C_0 \sim C'_0$ (trivial check) and show the following:

1. If $C_a \xrightarrow{\alpha} C_b$ and $C_a \sim C'_c$ for some C'_c, then there exists a C'_d such that $C'_c \rightarrow^* C'_d$ and such that $C_b \sim C'_d$, and
2. the simulated transitions ($C'_c \rightarrow^* C'_d$) emits the same observable action (α) as the original transition.

Assume $C_a \xrightarrow{\alpha} C_b$ and $C_a \sim C'_c$ for some C'_c.

If $C_a \xrightarrow{\alpha} C_b$ invokes a security relevant method $c.m$, we let $C'_c \rightarrow^* C'_d$ represent the transitions of the inlined program corresponding to the before clauses of $c.m$ followed by the same invoke transition as the original program. By the fact E adheres to $\mathcal{P}$ we know that the transitions will not halt the execution, and by the fact that no inlined instruction emits any observable actions, the only observable action of $C'_c \rightarrow^* C'_d$ will due to the last invoke which will emit the same observable action, α, as the original transition. By the fact that each block of inlined code starts by acquiring the lock of the *SecState* class and ends by releasing it, we know that no thread holds the lock in C'_c and that the lock maps are equal in C_b and C'_d. The crucial part is now to show the remaining constraints for $C_b \sim C'_d$, i.e. that the state of all non-inlined variables agrees in C_b and C'_d. We first note that no inlined instruction affects the state of a variable of the original program. We now turn to the issue of potential instruction reordering of the original program. By the fact that *Prg* is well-synchronized the Java Memory Model guarantees that E will appear to be sequentially consistent ([59] Section 17.4.3). This implies that there is a total order of the actions in E which is consistent with the program order of *Prg*. This in turn guarantees that no instruction reorderings affects the execution. This is all we need to guarantee that C'_d can be chosen in a way such that $C_b \sim C'_d$.

Similarly, if $C_a \xrightarrow{\alpha} C_b$ returns (normally or exceptionally) from a security relevant method $c.m$, we let $C'_c \rightarrow^* C'_d$ represent same return transition followed by the a

sequence of transitions corresponding to the execution of the instructions which have been inlined for the after or exceptional clauses of $c.m$. By the same reasoning as above, we know that $C'_c \rightarrow^* C'_d$ emits the same observable action, α, as the original transition and that $C_b \sim C'_d$.

If $C_a \xrightarrow{\alpha} C_b$ represents any other transition we know, by $C_a \sim C'_c$ that $C'_c \xrightarrow{\alpha} C'_d$, and that $C_b \sim C'_d$.

We have thus shown that each execution E of Prg has a corresponding execution E' of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ such that $\omega(E) = \omega(E')$ which implies that $\mathcal{I}_{Ex}$ is transparent. □

Corollary 2. *The set of race free policies is the maximal set of inlineable policies.*

Proof. Since $\mathcal{I}_{Ex}$ is secure, and conservative (for arbitrary programs) and transparent (for well-synchronized programs) for all race free policies, we know that any race free policies is by definition inlineable. The result then follows from Corollary 1. □

4.8 Case Studies

We have implemented an inliner that parses policies written in ConSpec and performs inlining according to the algorithm described in Section 4.7.1. This inliner has been evaluated in five case studies of varying characteristics. Case study descriptions and results are provided below. For detailed descriptions and case study applications and policies, we refer to the web page [88].

4.8.1 Case Study 1: Session Management

It is common for web applications to allow users to login from one network and then access the web page using the same session ID but with a different IP address from another network. Provided that the session ID is kept secret this poses no security problems. However, the session can be *hijacked* due to for instance predictable session IDs, session sniffing or cross-site scripting attacks [100].

In this case study we examine a simple online banking application implemented using the Winstone Servlet Container and the HyperSQL DBMS. Users may login though an HTML form, transfer money and logout. The session management is handled by the classes provided by the standard Servlet API.

To eliminate one source of session hijacking attacks the policy in this case study forbids a session ID from being used from multiple IP addresses. It does this by a) associating every fresh session ID with the IP address performing the request, and b) rejecting requests referring a known session ID performed from IP addresses not equal to the associated one.

The policy is implemented using a hash map for storing the IP to session ID association, and monitors (and restricts) all invocations of the *HttpServlet.service* method.

4.8.2 Case Study 2: HTTP Authentication

In this case study we look at the HTTP authentication mechanism [50]. This allows a user to provide credentials as part of an HTTP request. On top of this the Servlet API provides a security framework based on user roles. The access control of this setup is on the level of HTTP-commands, such as GET and POST. This is however too coarse-grained for some applications.

The application in this case study is the same as in the previous case study, but here we focus on the administration interface of the web application. This part is protected by HTTP authentication and supports two roles: *Secretaries* and *administrators*. The intention is that secretaries should be allowed to *query* the database whereas administrators are allowed to also *update* the database.

The policy enforces this by making sure the application calls *HttpServletRequest.isUserInRole* and that only users in the secretary role may invoke *java.sql.Statement.executeQuery* and only users in the administrator role may invoke *java.sql.Statement.executeUpdate*. Since these rules only apply for the administrative part of the web application the policy is implemented to check requests only if *request.getRequestURI*().*startsWith*("/*admin*") returns true. Furthermore, to prevent interference of multiple simultaneous requests, the policy state is stored in *ThreadLocal* variables.

4.8.3 Case Study 3: Browser Redirection

Following the example of Sridhar and Hamlen [118] we examined an ad applet that, when being clicked on, redirects the browser to a new URL. The policy in this case states that the applet is only allowed to redirect the browser to URLs within the same domain as which the applet was loaded from.

The policy enforces this by asserting that URLs passed to *AppletContext.showDocument* have the same host as the host returned by *Applet.getDocumentBase*().

4.8.4 Case Study 4: Cash Desk System

In this case study we monitor the behavior of a concurrent model of a cash desk system. The application stems from an ABS model that was developed for the HATS project [16]. The policy keeps track of the number of sales in progress (by monitoring invocations of *newSaleStarted*() and *saleFinished*()) and asserts that the number of ongoing sales is positive.

4.8.5 Case Study 5: Swing API Usage

The classes in the Java Swing API are not thread safe and once the user interface has been realized (*Window.show*(), *Window.pack*() or *Window.setVisible*(*true*) has been called) the classes may be accessed only through the event dispatch thread (EDT). This constraint is sometimes tricky to adhere to as it is hard to foresee all flows of a program and whether or not some code will be executed on the EDT or not.

	ConSpec clauses	Size before inlining (kB)	Size after inlining (kB)	Size increase (%)	Security relevant calls	Inlining time (s)	Runtime Overhead (%)
CS1 (Sessions)	1	532.7	533.1	0.08	1	2.47	0.44
CS2 (HTTP Auth.)	4	532.7	535.6	0.54	12	2.66	0.87
CS3 (Redirection)	2	27.5	28.2	2.41	1	0.18	n/a
CS4 (Cash Desk)	2	652.9	654.0	0.17	2	2.52	n/a
CS5 (Swing)	249	1888.6	2140.7	13.35	1038	26.68	11.27

Table 4.1: Quantative results of the case studies.

In this case study we monitor the usage of the Swing API in a large (68 kloc), off-the-shelf, drawing program called JPicEdt (version 1.4.1_03) [102]. The inlined monitor has two states: realized and not realized and the policy states that once realized, a Swing method may only be called if *EventQueue.isDispatchThread*() return true.

This case study demonstrates how the inliner can be useful, not only in a security critical setting, but also during testing and development. The inlined reference monitor revealed three violations of the policy and by letting the monitor print the stack trace upon a violation we managed to locate and patch the errors.

4.8.6 Results

A summary of the case studies is given in Table 4.1. Benchmarks were performed on a computer with a 1.8 GHz dual core CPU and 2 GB memory. The runtime overhead due to inlining was measured for the web application case studies (CS1 and CS2) and for the Swing case study (CS 5). The runtime overhead for the web application was based on a roughly one minute long stress test and for the Swing application we measured the startup time (the time required to construct the user interface).

4.9 Certification

Monitoring is essentially a tool for quality assurance: By monitoring program execution we are able to observe actions taken by a program and intervene if a state of affairs is discovered which we for some reason are unhappy with. By inlining we can make this tool available for developers as well, for instance to enforce richer, history-dependent access control than what is allowed in the current, static sandboxing regime.

However, the code consumer may not necessarily trust the developer (code producer) to enforce the consumer's security policy. Moreover, different consumers may want to enforce different security policies. In this section we turn to the issue of *certification*, that is, we ask for an algorithm, a *checker*, by which the recipient of a piece of code can convince herself that the application is secure. To support efficient verification, the code producer can ship additional metadata with the code, for instance (elements of) a proof, following the idea of Proof-Carrying Code (PCC) [95]. This metadata will be called a *certificate*, not to be confused with the concept with the same name used in public-key cryptography.

The scenario we want to support is the following (a classic PCC scenario):

1. A code producer develops an application, and ensures that it complies with the *producer policy* by inlining a corresponding monitor. This producer policy is developed with the intention that it will cover all the security concerns of potential consumers of the application, but of course these consumers do not necessarily trust the producer for this.

2. Various code consumers want to run the application. Before doing so, each consumer will check that the code complies with his or her *consumer policy*. (Each consumer may have a different policy.)

3. In order to help a consumer with this check, the producer ships a *certificate* together with the code. The certificate will contain a proof of the fact that the code complies with the producer policy.

4. The code consumer uses a *checking* algorithm which checks if the application complies with his consumer policy. This checking algorithm takes as (untrusted) input the application code and the certificate.

We outline an approach for building a checker that can verify the security property of IRMs inlined using techniques similar to the algorithm we discussed in this paper. The contribution of this section is that we show that, for this inlining approach, a checker for multithreaded Java programs can be built using established program verification techniques based on sequential Java.

4.9.1 Assumptions about the inlined code

The checking algorithm in this section is designed for a class of inliners that (1) are non-blocking, i.e. they do not lock the security state across security relevant API calls, and (2) use one global lock to protect the inlined security state.

More concretely, let us assume that the security state is kept in static fields of a designated *SecState* class, and that the *SecState* class object is used to lock the security state. The actual inlined code then operates in phases:

1. A *neutral* phase (N), where the *SecState* lock is not held. If all threads are in this N state, then the inlined security state is in sync with the history of security relevant actions encountered so far.

2. A *locked before* phase (LB), where the inliner is updating its state in anticipation of an upcoming security relevant call.

3. An *unlocked before* phase (UB), where things might be happening between the inlined check and the actual call. The inlined security state has been updated already, but the actual security relevant action has not yet happened.

4. A *calling* phase (C) where the actual security relevant call is executing.

5. An *unlocked after* phase (UA), where things might be happening between the (normal) return of the call, and the inlined security state update.

6. A *locked after* phase (LA), where the inliner is updating its state in response to a successfully returned security relevant call.

7. Similar *unlocked exceptional* and *locked exceptional* phases, to deal with exceptional returns of the security relevant method invocation. These are similar to the UA and LA phases, and we do not discuss them further in this section. Extending the results in this section to deal with exceptional returns of security relevant calls is straightforward.

Notice that, with the inliner of Figure 4.5, it appears that no instructions are actually executed during the UB and UA phases. This is, however, not entirely accurate: When the inliner is applied iteratively, say twice in succession, the instructions executed in the locked phases of the second inlining will appear as instructions in the unlocked phases for the first inlining. In fact, we can allow arbitrary code to be present in the unlocked phases, as long as it does not interfere with the inlined state. This allows a wider class of inliners to be supported than the one introduced above. One such example is briefly discussed in the conclusions.

A key part of the checking algorithm is to recognize these phases. Once the phases are recognized, an approach similar to the one taken in [1] for sequential Java can be enacted.

To assist the checker in identifying the phases, the certificate contains the following information: For each bytecode instruction in the program that performs a security relevant method invocation, the code producer should include in the certificate a tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$, where $c'.m'$ is the name of the method containing the call, and the other elements of the tuple are labels in the method body of $c'.m'$:

- L_{lb} indicates where the LB phase starts,

- L_{ub} indicates where the LB phase ends and the UB phase starts,

- L_{call} indicates where the calling phase C starts and ends. Recall that in our semantics, API calls happen in two steps. The first step initiates the calling phase, and the second step ends it, and starts the UA phase.

- L_{la} indicates where the UA phase ends and the LA phase starts.
- Finally, L_n indicates where the LA phase ends and the inliner returns to the neutral phase.

A first part of the checking algorithm verifies, based on the above information, whether the code complies with the assumptions we make about the inlining process. The example inliner $\mathcal{I}_{Ex}$ that we proposed in Section 4.7 will pass this check.

Check 1. *For each tuple,* $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$*, in the certificate, perform the following checks:*

- *The* L_{lb} *and* L_{la} *labels point to a* `ldc SecState` *instruction, followed by a* `monitorenter`*.*
- *The* L_{ub} *and* L_n *labels point to a* `monitorexit` *instruction preceded by a* `ldc SecState`*.*
- *The labels* L_{lb}*,* L_{ub}*,* L_{call}*,* L_{la}*,* L_n *occur in this order in the method body of* $c'.m'$*.*
- *Construct the control-flow-graph (CFG) for the method body of* $c'.m'$*, and check that:*
  - *The only way to enter the block between* L_{lb} *and* L_n *is by entering through* L_{lb}*. (No jumps over blocks of inlined code or into the middle of inlined code)*
  - *Each path in the CFG that passes through* L_{lb} *also passes through* L_{ub}*,* L_{call}*,* L_{la}*, and* L_n*, or leads to* `System.exit()`*.*

In addition, to make sure that the global security state (stored in static fields of the `SecState` *class) is only accessed under the* `SecState` *lock, perform the following checks:*

- *No other* `ldc SecState` *instructions occur anywhere in the program. This makes sure the SecState class object is only used for acquiring or releasing a lock, and no other aliases to the object are created.*
- `putstatic` *and* `getstatic` *for fields of the SecState class only occur between* L_{lb} *and* L_{ub}*, and between* L_{la} *and* L_n *labels.*

These checks allow us to reason about the actual inlined security state sequentially (because all accesses to that state happen under a single lock). Moreover, any invariant on the security state that is true in the initial state and maintained by each block of code that holds the SecState lock will be true at each program point where the SecState lock is not held.

These two observations will be crucial in designing the second step of the checker. For this second step, the checker will inline a reference automaton used for verification purposes, henceforth referred to as a "ghost reference monitor", or ghost IRM for short. We first describe this ghost IRM and how it is inlined by the checker.

4.9.2 The Ghost Reference Monitor

The ghost IRM is implemented by inserting special purpose assignments called *ghost instructions* into the program. The ghost instructions are essentially ConSpec rules, lightly compiled to evaluate guards and updates using the JVM stack and heap, together with a set of auxiliary *ghost variables* used to represent the state of the ghost IRM, and to store intermediate values, e.g. across method calls. Programs containing ghost instructions are called *augmented programs*.

A ghost instruction has the shape

$$\langle x^g := a_1 \to e_1 \mid \ldots \mid a_n \to e_n \rangle$$

where x^g is a vector of *ghost variables*, a_i are guard assertions and e_i are expression vectors of the same type and dimension as x^g. The instruction assigns the first expression whose guard holds, to the left hand side variable, similar to the way ConSpec rules are evaluated. If no guards hold, the instruction *fails* and the execution is said to be *incorrect*. The guards a_i and expressions e_i may refer to ghost variables, actual variables, the stack, and they may extract callee and thread id as described above.

Example 7. *The ghost instruction below could be used to express that an execution is incorrect if the invoke instruction is executed with true as argument more than 10 times.*

$$\begin{array}{l} \ldots \\ \langle x^g := \mathrm{s}_0 \wedge x^g < 10 \to x^g + 1 \mid \neg \mathrm{s}_0 \to x^g \rangle \\ \mathit{invoke}\ c.m \\ \ldots \end{array}$$

Ghost variables can be global or local. This scope will be notationally clarified by the superscripts x^g and x^{gl}, respectively.

An execution of an augmented program is a sequence of augmented configurations which in turn are regular configurations augmented with a ghost variable valuation. An augmented program is said to be *correct* if all of its executions are correct.

4.9.3 Ghost Inlining

The ghost inliner augments clients with ghost instructions to maintain various types of state information. This includes the ghost IRM state, intermediate data used only by the ghost IRM, and information to assist the checker in relating the ghost IRM state and the actual IRM state.

Identifier	Purpose
ms^g	A global vector representing the ghost security state, i.e. a type correct assignment to the security state variables as in Section 4.3.
$status^{gl}$	A local variable ranging over ready, meaning that the action trace is in sync with the ghost IRM, or before_$c.m$, return_$c.m$, indicating that the ghost IRM is one pre- or post-action out of sync.
arg^{gl}, o^{gl}, tid^{gl}, r^{gl}	Local variables to hold the arguments of security relevant calls during the call (they may be referenced in an after-clause), resp. calling thread, callee, and return value.

Table 4.2: Variables introduced by ghost inliner.

The code consumer will perform the ghost inlining algorithm, using the following inputs:

- The consumer policy, from which the ghost IRM state, and the implementation of the ghost IRM state transitions can be computed.
- The code and the certificate.

The ghost inliner introduces the variables listed in Table 4.2, and it implements the ghost IRM by inserting blocks of ghost instructions according to the following scheme. For each $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$ tuple in the certificate for a call to security relevant method $c.m$, do the following:

1. Insert in $c'.m'$ before label $L_{ub} - 1$:

$$\begin{aligned}
&\langle tid^{gl} := Thread.currentThread() \rangle \\
&\langle o^{gl} := \mathrm{s}_0 \rangle \\
&\langle arg^{gl} := (\mathrm{s}_1, \ldots, \mathrm{s}_n) \rangle \\
&\langle ms^g := status^g = \mathsf{ready} \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl})^{\uparrow}) \rangle \\
&\langle status^{gl} := before_{c.m} \rangle
\end{aligned}$$

   If $c.m$ is security relevant but lacks a BEFORE clause, the ghost security state ms^g is not updated, but the other assignments are still performed.

2. Insert in $c'.m'$ before label L_{call}:

$$\begin{aligned}
\langle status^{gl} := {} & status^{gl} = before_{c.m} \\
& \wedge \ o^{gl} = \mathrm{s}_0 \ \wedge arg^{gl} = (\mathrm{s}_1, \ldots, \mathrm{s}_n) \rightarrow \mathsf{ready} \rangle
\end{aligned}$$

Arbitrary non-inlined code
L_{lb}: `ldc SecState`
`monitorenter`
Actual IRM BEFORE code
$\langle arg^{gl} := (\mathrm{s}_1, \ldots, \mathrm{s}_n)\rangle$
$\langle ms^{g} := status^{gl} = \mathsf{ready} \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl})^{\uparrow})\rangle$
$\langle status^{gl} := \mathrm{before\_}c.m\rangle$
`ldc SecState`
L_{ub}: `monitorexit`
Arbitrary code
$\langle status^{gl} := status^{gl} = \mathrm{before\_}c.m \wedge arg^{gl} = (\mathrm{s}_1, \ldots, \mathrm{s}_n) \rightarrow \mathsf{ready}\rangle$
L_{call}: `invokevirtual` $c.m$
$\langle r^{gl} := \mathrm{s}_0\rangle$
$\langle status^{gl} := status^{gl} = \mathsf{ready} \rightarrow \texttt{return\_}c.m\rangle$
Arbitrary code
L_{la}: `ldc SecState`
`monitorenter`
Actual IRM AFTER code
$\langle ms^{g} := status^{gl} = \texttt{return\_}c.m \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl}, r^{gl})^{\downarrow})\rangle$
$\langle status^{gl} := \mathsf{ready}\rangle$
`ldc SecState`
L_n: `monitorexit`
Arbitrary non-inlined code

Figure 4.7: Schematic summary of ghost inlining for `invokevirtual` $c.m$. Current thread *tid* and callee s_0 has been omitted for brevity.

3. Insert in $c'.m'$ after label L_{call}:

$$\langle r^{gl} := \mathrm{s}_0\rangle$$
$$\langle status^{gl} := status^{gl} = \mathsf{ready} \rightarrow return_{c.m}\rangle$$

4. Insert in $c'.m'$ before label $L_n - 1$:

$$\langle ms^{g} := status^{gl} = return_{c.m} \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl}, r^{gl})^{\downarrow})\rangle$$
$$\langle status^{gl} := \mathsf{ready}\rangle$$

We refer to ghost instruction blocks inserted according to condition i above as a *block of type* i.

A schematic summary of the treatment of a security relevant invoke is illustrated in Figure 4.7. Correctness is proved by an extension of the inliner security argument of Section 4.7. In analogy with Proposition 5 we first show that the ghost inliner is sound in the sense that traces of the ghost monitor are allowed by the policy, and we then show security through a serialization property similar to Lemma 8.

Let $\mathcal{I}^g(\mathcal{P}, Prg)$ be the result of ghost inlining Prg with respect to policy $\mathcal{P}$ and Prg's certificate. Similar to Section 4.7 we compare the observable actions of Prg with *ghost actions* α^g of $\mathcal{I}^g(\mathcal{P}, Prg)$. The *ghost extended trace* of an execution E, $\tau_{ge}(E)$ is the sequence of observable actions and ghost actions of E, and the ghost trace of E, $\tau_g(E)$, projects from $\tau_{ge}(E)$ the ghost actions only.

Proposition 6. *Let E be a legal execution of $\mathcal{I}^g(\mathcal{P}, Prg)$. Then $\tau_g(E) \in \mathcal{P}$.*

Proof. Let $\tau_g(E) = {\alpha_0}^g \cdots {\alpha_n}^g$ be the ghost trace of E. In the context of E, say that a block of type 1 *justifies* a block of type 2 or 4, if the values assigned to ghost variables o^{gl}, arg^{gl} in the type 1 block are the values used in the block of type 2 or 4. For the case of a type 2 block the value of $status^{gl}$ also needs to match the value assigned in the type 1 block. Similarly say that a block of type 4 *confirms* a block of type 3, if the values assigned to r^{gl}, $status^{gl}$ in the type 3 block are those used in the type 4 block.

If α_n is a pre-action then a block of type 1 justifying ${\alpha_n}^g$ happens before ${\alpha_n}^g$ and after ${\alpha_{n-1}}^g$. Since the prefix of $\tau_g(E)$ not including ${\alpha_n}^g$ is in $\mathcal{P}$, so is $\tau_g(E)$. For this argument to work out we need to observe that, if ${\alpha_{n-1}}^g$ is a block of type 3 then that block is confirmed by a block of type 4 before control is transferred to the block of type 1 justifying ${\alpha_n}^g$. The case of α_n a post-action is virtually identical and left to the reader. □

With Proposition 6 in place the security proof is essentially complete, as the proof of serialization can follow that of Lemma 8 line for line.

As a result we obtain the correlate of the Inliner Security Theorem, now transferred to the ghost inliner:

Theorem 13 (Ghost Security)**.** *If $\mathcal{P}$ is race free, and Prg is a correct program, then $\mathcal{T}(\mathcal{I}^g(\mathcal{P}, Prg)) \subseteq \mathcal{P}$* □

4.9.4 The checker

The checker algorithm should check that a given program (with certificate) satisfies a code consumer policy. To achieve this, the checker first performs Check 1 from Section 4.9.1. Then the checker augments a ghost IRM based on the consumer policy. Building on Theorem 13, the only remaining thing the checker needs to do is verify that the resulting program is *correct*, i.e. that none of the inlined ghost instructions fail.

Checking that an arbitrary program with inlined ghost instructions is correct is a hard problem, as hard as verifying full functional correctness of multithreaded Java code. However, with the assumptions we made about the actual inlining process, and given the concrete ghost inlining algorithm, checking correctness can be substantially simplified. In particular, we show in this section that verification of correctness can be done using sequential reasoning only. We assume that we are given as an oracle a proof checker for a standard sequential bytecode program logic

(for instance the logic proposed by Bannwart and Müller [4]). In order to ensure that sequential verification is sound in our multithreaded setting, we rewrite the bytecode before sending it to the sequential verifier. In a multithreaded setting, reads from the heap are not necessarily stable. The only two parts of the state that we can reason about sequentially are local variables and the global security state (while the SecState lock is being held). We encode this by replacing all other reads from the heap by method calls to a method *randomValue*() of appropriate return type. This ensures that the verifier knows nothing about values read from the heap. Whenever we send blocks of bytecode (and corresponding proofs) to the verification oracle, we preprocess these blocks of bytecode to (1) remove all the locking/unlocking instructions, and (2) to replace reads from the heap (except reads of the fields of SecState in the LB or LA phase) with calls to such a *randomValue*() method of the appropriate type.

To support this second part of the checking algorithm, the code producer should include additional information in the certificate.

First, the code producer should provide an invariant $I(ms, ms^g)$ that relates the actual inlined security state ms to the ghost inlined security state ms^g. This invariant can be through of as a *simulation* relation between the states of the actual security automaton and the ghost automaton. Obviously, $I(ms, ms^g)$ is only allowed to refer to ghost security state variables and to static fields of the SecState class.

Second, the certificate provided by the code producer should contain some proofs checkable by the sequential program verification oracle, as detailed below.

Check 2. *For each tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$ in the certificate for a security relevant call to c.m, the checker performs the following verifications:*

- *For the locked before block B (the code between the acquiring of the SecState lock at L_{lb} and releasing of that lock at L_{ub}), check that the certificate contains a valid proof that the following code:*

$$\langle ms^g := \delta((tid^{gl}, c.m, s_0, (s_1, \ldots, s_n))^{\uparrow})\rangle; B$$

  maintains the invariant $I(ms, ms^g)$, and does not fail when started from a state where this invariant is true.

- *For the full inlined block F (the code between the acquiring of the SecState lock at L_{la} and releasing of that lock at L_n), check that the certificate contains a valid proof that F maintains the invariant $I(ms, ms^g)$, and does not fail when started from a state where this invariant is true.*

Finally, check that $I(ms, ms^g)$ holds for the default initial values for all ghost and actual security state variables.

Lemma 9. *If a program passes the checker, then, in any execution of the program, the invariant $I(ms, ms^g)$ holds whenever the SecState lock is not being held by any thread.*

Proof. By contradiction. Assume there is an execution that violates this property. Identify the first step in the execution where the property fails. This cannot be the first step of the execution, as Check 2 checks that $I(ms, ms^g)$ holds in the initial state. Since changes to the variables mentioned in the invariant can only be done under the SecState lock (Check 1), the first step where the property fails must be a step where the SecState lock is being released. Because of Check 1, the lock can only be released by an instruction that is labeled L_{ub} or L_n. Let us consider the case L_n (the other case is similar), and let us call the thread that performs this monitorexit t. Select from the execution all steps from the thread t. Since t reaches L_n, and because of the control flow checks in Check 1, one of these execution steps must execute the instruction at L_{lb}. Consider the last step of thread t that executes the instruction at L_{lb}, and remove from the execution all steps before that one. The resulting execution is a single-threaded execution of the full inlined block F verified in Check 2 to maintain the invariant. Moreover, the execution starts in a state where the invariant holds (because we have selected the first step in the execution where the property fails). If our sequential verification oracle is sound, this cannot happen. □

We can now show that the checker is secure: if all the checks succeed, the program being checked is secure.

Theorem 14. *A program that passes the checker is secure.*

Proof. By Theorem 13 it suffices to prove that the ghost inlined program can never fail. We prove this by contradiction. Assume there is an execution of the program that fails, i.e. that leads to one of the guards in the ghost statements evaluating to false. We show that from this execution, we can construct a failing single-threaded execution of one of the blocks of code that have been verified not to fail by the sequential verification oracle.

Let the thread identifier of the thread where the failure happens be t.

Consider all steps of thread t leading to the failure of a ghost statement. Because of the CFG check in Check 1, and since thread t reaches one of the ghost inlined instructions, thread t must have executed the instruction at label L_{lb}. Select the latest execution by thread t of that instruction, and remove all steps before that step. The remaining execution is a single threaded execution of the full inlined block verified not to fail during Check 2. Contradiction. □

4.9.5 Creating certificates for the example inliner

Finally, we show that a code producer that uses the concrete inliner $\mathcal{I}_{Ex}$ that we proposed in Section 4.7 can easily produce a certificate that the resulting program complies with the inlined policy. Certificates contain three parts:

- For each security relevant `invokevirtual` bytecode instruction at a label L_{call} in method $c'.m'$, a certificate contains the tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$

marking the beginning and ending of the different phases of the inliner. Computing these for $\mathcal{I}_{Ex}$ is trivial.

- An invariant $I(ms, ms^g)$ that relates ghost security state to actual security state. To certify that an inlined program complies with the inlined policy, this invariant is just the identity.

- For each security relevant `invokevirtual` bytecode instruction, the certificate contains two sequential correctness proofs, one for the locked before block B, and one for the full inlined block F. It is an easy exercise to verify that the code blocks produced by our inliner are valid. Given an oracle for constructing proofs of valid programs in sequential Java, we can complete the certificate with this third part.

Theorem 15. *A program inlined with our inliner and with a certificate constructed as above will pass the checker.* □

To summarize, we have shown that our inliner is able to inline a reference monitor in a way such that it is statically decidable whether or not the resulting program adheres to the given (race free) policy. This is what Hamlen et al refers to as $\mathcal{P}$-verifiability [120]. Thus, put another way, we have shown that the set of race free policies are $\mathcal{P}$-verifiable.

4.9.6 Discussion

The checker developed in this section is, to the best of our knowledge, the first one that can certify compliance with security automata for multithreaded Java bytecode. The certification approaches proposed by other authors (and discussed in Section 4.1.1) focus on sequential programs only, or on blocking inliners for multithreaded programs. While our checker can only handle programs that have been generated by an inliner that complies with the assumptions we outlined in Section 4.9.1 (it will reject any other program as possibly insecure), this is a significant step forward. However, further improvements are possible.

Most importantly, one of the key motivations for Proof-Carrying Code is that it can reduce the Trusted Computing Base (TCB). Security only relies on correctness of the verifier, not on the (possibly complicated) techniques used by the code producer to construct the code and the proof. In many PCC approaches, the verifier is just a proof checker for proofs in a simple program logic. The checker we proposed in this paper is significantly more complicated than that. The main reason for this is that there is no existing program logic for multithreaded Java bytecode. Designing such program logics (and proving them sound) is an important avenue for future work.

What we did show in this section is how, for the class of inliners that we support, the issues related to multithreading can be handled separately using a relatively simple syntactic check (Check 1). Given a suitable program logic, it is likely that

the insight reported in this section could be used to construct security proofs in that logic for programs that are inlined with such an inliner. Then, security could be verified using just a proof checker for a program logic.

Even though we have not yet reached that stage, our checker is still significantly simpler than the inliner: ghost inlining is done at a higher level of abstraction, and avoids many of the intricate bytecode rewriting tasks that the real inliner has to deal with, including things such as updating jumps, recomputing switch tables, updating exception handling tables, and so forth.

4.10 Conclusions and Future Work

Inlining is a powerful and practical technique to enforce security policies. Several inlining implementations exist, also for multithreaded programs. The study of correctness and security of inlining algorithms is important, and has received a substantial amount of attention the past few years. But, these efforts have focused on inlining in a sequential setting. This paper shows that inlining in a multithreaded setting brings a number of additional challenges. Not all policies can be enforced by inlining in a manner which is both secure and transparent. Fortunately, these non-enforceable policies do not appear very important in practice: They are policies that constrain not just the program, but also the API or the scheduler. We have identified a class of so-called race free policies which characterizes exactly those policies that can be enforced by inlining in a secure and transparent fashion on multithreaded Java bytecode. This result is quite general: It relies mainly on the ability of policies to distinguish between entries to and exits from some set of API procedures, and very little on the specificities of the Java threading model. We have shown that the approach is useful in practice by applying it in several realistic application scenarios, and we have shown how certification of inlining in the multithreaded setting can be reduced to standard verification condition checking for sequential Java.

A number of extensions of this work merit attention. We discuss three issues: Inheritance, iterated inlining, and callbacks.

Inheritance, first, is relatively straightforward: In order to evaluate the correct event clause, runtime checks on the type of the callee object would be interleaved with the checks of the guards. This is spelled out for the sequential setting in [129] for C#. We do not expect any issues to carry this over to the multithreaded setting.

For iterated inlining there are two options:

1. The ConSpec policies are merged before inlining. This can be achieved by using a syntactic cross product construction for policies, $\mathcal{I}(Prg, \prod_i \mathcal{P}_i)$.

2. Alternatively, the monitors can be nested by inlining one policy at a time: $\mathcal{I}(\ldots\mathcal{I}(\mathcal{I}(Prg, \mathcal{P}_1), \mathcal{P}_2), \ldots \mathcal{P}_n)$.

If the example inliner, $\mathcal{I}_{Ex}$, is used, the certification approach described above is general enough to easily certify the fully inlined program from certificates for each

policy $\mathcal{P}_i$ by itself. If a different inliner is used however, the second approach needs a different treatment in general. One common strategy, for instance, is to create a wrapper method for each security relevant method, place the policy code in the wrapper method and replace the security relevant calls, with calls to the wrapper methods. The reason for this is that, except for the last inlining step, the inlined policy code will no longer reside in the same method as the security relevant call. To handle this one can either:

- Do the analysis from the first inlined BEFORE-instruction, to the last inlined AFTER / EXCEPTIONAL instruction globally. (This is obviously not tractable in general, but for simple wrapper methods it would not pose any problems.)
- Perform a simple renaming of security relevant methods, so that the outer policies consider the new wrapper methods to be security relevant instead.

Callbacks can be accommodated as well, but with more significant changes. First, the notion of event must be changed, to include not only calls from the client program to the API and return, but also from the API to the client program. This affects not only the program model but also the policy language. The negative results will remain valid, but the inlining algorithm must be amended to inline pre- and post checks in each public client method.

Finally, we believe that our study of the impact of multithreading on program rewriting in the context of monitor inlining is a first step towards a formal treatment of more general aspect implementation techniques in a multithreaded setting. Indeed, our policy language is a domain-specific aspect language, and our inliner is a simple aspect weaver.

Acknowledgements

Thanks to Irem Aktug, Dilian Gurov and Dries Vanoverberghe for useful discussions on many topics related to monitor inlining. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund K.U.Leuven, the IWT, and by the European Commission under the FP6 and FP7 programs.

Chapter 5

TreeDroid: A Tree Automaton Based Approach to Enforcing Data Processing Policies

Mads Dam, Gurvan Le Guernic, Andreas Lundblad

KTH, Royal Institute of Technology, Sweden
{mfd, gurvan, landreas}@kth.se

Abstract

Current approaches to security policy monitoring are based on linear control flow constraints such as *`runQuery` may be evaluated only after `sanitize`*. However, realistic security policies must be able to conveniently capture data flow constraints as well. An example is a policy stating that *arguments to the function `runQuery` must be either constants, outputs of a function `sanitize`, or concatenations of any such values.*

We present a novel approach to security policy monitoring that uses tree automata to capture constraints on the way data is processed along an execution. We present a λ-calculus based model of the framework, investigate some of the models meta-properties, and show how it can be implemented using labels corresponding to automaton states to reflect the computational histories of each data item. We show how a standard denotational semantics induces the expected monitoring regime on a simple "while" language. Finally we implement the framework for the Dalvik VM using TaintDroid as the underlying data flow tracking mechanism, and evaluate its functionality and performance on five case studies.

5.1 Introduction

Today 95% of all mobile devices run Android, Symbian, iOS or RIM [53]. All those operating systems share the same security model for third party applications.

When a new application is installed (or launched for the first time) the operating system asks the user if he or she grants the application a set of permissions. Such permissions typically allow the application to access internet, the GPS hardware, address book data, camera, etc. Unfortunately the model is quite crude. Most useful and innocent tasks require a combination of permissions which could just as well be used maliciously [46, 130]. Many applications request the Internet access permission, for example in order to display ads, together with permissions for other phone resources, which can then potentially be remotely accessed and controlled. For this reason it is of high importance to study techniques, such as the one proposed in this paper, which allow policies to be expressed at a finer level of granularity.

Our proposal is to use bottom-up tree automata to track how an application processes data at runtime. The approach monitors how each data item in an execution has been computed and prevents certain function calls from being made based on this information. This data-centric approach to runtime monitoring allows for a wide range of policies to be expressed, including API usage policies restricting which methods may be applied to what arguments and data flow policies stating how data must have been processed before being passed to certain functions.

The policies in this framework are different from the ones handled by existing techniques. For example, as opposed to existing runtime monitoring techniques which handle policies expressing temporal properties such as "*f may be invoked after g has been invoked but not vice versa*" our approach handles policies such as "*f may be applied to the result of g but not vice versa*". A more concrete example of a policy which is naturally expressed in our framework (but difficult or impossible to express in others) could for instance state that `sanitize` accepts any string as argument, while the function `runQuery` only accepts string constants, strings returned by `sanitize` or concatenations of such strings.

The approach described in this paper differs from traditional runtime monitoring on three key points. The approach is (a) data centric, (b) based on tree shaped traces and (c) relies on richer observable actions. (a) Standard techniques [39, 64, 49, 134] are control flow oriented: They monitor the linear flow of events as they occur at system/thread/object level. By contrast, our technique is data oriented, allowing different flows of data to be monitored in isolation, even if they are arbitrarily interleaved in the application. (b) Existing runtime monitoring frameworks are typically based on deterministic finite automata (DFA) [25, 39, 64], edit automata [84], LTL [104, 67, 115, 71, 141, 140], context free grammars [92], or a variation thereof [85, 54, 116], all of which rely on a model of *linear* traces. Since our approach focuses on how data is processed, i.e. how functions are combined rather than in what order actions are performed, traces manipulated in our framework are tree shaped. (c) Similarly to work by others [13, 64, 63, 92, 84], we let the function calls be the actions observable by the monitor. However, the fact that monitoring is performed at the data level allows the observable actions to depend on the computational history of each arguments in a manner which is impossible or inconvenient using the existing frameworks.

5.1.1 Contributions

The first contribution of the paper is a theoretical formalization of the framework using λ-calculus. It includes a program model which records computational histories of data, and a policy model which accepts or rejects certain computations. As our second contribution we identify three policy classes and establish their relationships. As a third contribution, the paper describes a solution allowing an efficient implementation of the approach by using so called *labels* which are to be seen as abstractions of computational histories. We show how policies, which are semantically defined in terms of bottom-up tree automata, can be enforced using data labels corresponding to the automaton states. As a validation of our framework we then show how a standard denotational semantics induces the expected monitoring regime on a simple imperative language. The final contribution is an implementation of the framework for Java programs run on top of the Android platform. The implementation relies on *taint tracking* for its underlying data flow mechanism and on *monitor inlining* for policy enforcement. The practicality of the approach is demonstrated in five different case studies.

5.1.2 Related Work

The theoretical part of the paper is related to the work on labeled λ-calculus which was initially proposed by Lévy [81]. A labeled λ-calculus associates labels with subterms in order to track how they affect the reduction. Gandhe, Venkatesh and Amitabha [52] use this as a theoretical basis for analysis of certain aspects of functional programs. Specifically, they define a notion of *need* and show how to use the calculus to identify to what extent an argument is needed to reduce a function application to its head normal form. This involves tracking computations and origins of subterms just as required by our framework. However, the existing labeled λ-calculi do not reflect the exact semantics of practical data flow tracking techniques such as the taint tracking mechanism on which our framework relies, which is why the calculus presented in this paper differs from existing ones.

Taint analysis is a well-known technique for tracking direct data flows and has been studied extensively over the years. Our framework is built upon the *TaintDroid* taint analysis framework [38]. TaintDroid targets Android applications and is based on an extension of the Dalvik VM. The extension allows for simultaneous real-time tracking of data coming from multiple different sources with the relatively small runtime overhead of 14%.

The inlining algorithm presented in the paper builds upon the algorithm described by Dam et al [25, 24] with important differences regarding the representation and manipulation of automaton states.

Several papers describe *static* approaches for checking and enforcing policies related to the ones handled in our framework. These approaches usually rely on some form of type system and typically focus on checking API protocols. The programming language concept of *typestates* is one example of such an approach. Typestate

is a refinement of the concept of a type: whereas the type of an object determines the set of operations *ever* permitted on the object, a typestate determines the subset of these operations which is permitted in a particular context. The idea was introduced by Strom and Yemini [121] and has recently been developed further by DeLine and Fändrich [30] and by Bierhoff and Aldrich [13, 14]. When compared to our approach, a typestate could be seen as the compile-time counterpart of a label. However, just as the runtime type of an object is more precise than its static type, our dynamic labels are more precise than typestates. The higher precision available at runtime allows us to avoid many of the problems that static program analysis faces due to, for instance, aliasing and concurrency. Furthermore, even if typestates were tracked and inspected in runtime, our notion of label is more general than typestates, since labels are not bound to a specific type and since labels can propagate from one object to another.

Outline

The paper is divided into two parts. The first part describes a formalization of the approach which starts by presenting program model based on λ-calculus (Section 5.2) and defining what a policy is (Section 4.3). This is followed by a description of the notion of labels (Section 5.4) and a section on how to express policies (Section 5.5). The theoretical part of the paper is concluded by a discussion on the applicability to imperative languages and gives an encoding for a While language (Section 5.6). The second part of the paper describes an implementation and evaluates the approach practically by exploring five case studies with varying characteristics (Section 5.7 and 5.8). Concluding remarks and directions for future work complete this paper (Section 5.9).

5.2 A Calculus with API Functions

This section presents the calculus used as theoretical foundation. The calculus is an untyped λ-calculus extended with constants, ranged over by $c \in \mathsf{C}$, n:ary function symbols, ranged over by $f^n \in \mathsf{F}$ and choice, written $(t = t)\ t\ t$. Applying f^n on $c_1, \ldots, c_n$ yields a value in C atomically and without side-effects. The semantics of functions is externally defined and written simply as $[\![f^n(c_1, \ldots, c_n)]\!]$. The calculus grammar follows:

$$
\begin{aligned}
t &::= v \mid t\ t \mid (t = t)\ t\ t \\
v &::= x \mid c \mid \lambda x\,.\,t \mid f^n\ c_1 \ldots c_m \qquad \text{where } m < n
\end{aligned}
$$

The standard transition relation $\rightharpoonup$ is identical to the extended transition relation $\rightarrow$ of Figure 5.1 with the $\hat{\tau}$-related annotations removed.

We regard terms as programs, and finite (resp. infinite) sequences of reductions, written $t_0 \rightharpoonup t_1 \rightharpoonup \ldots \rightharpoonup t_n$ (resp. $t_0 \rightharpoonup t_1 \rightharpoonup \ldots$), as *runs* or *executions*. For brevity, we write $t_0 \rightharpoonup t_1 \rightharpoonup \ldots (\rightharpoonup t_n)$ when our reasoning applies to both finite and infinite executions.

Example 8. *Provided* $[\![\texttt{userInput}()]\!]$ *yields some string* s, $[\![\texttt{flipCoin}()]\!]$ *yields either* hd *or* tl, $[\![\texttt{sanitize}(s)]\!]$ *yields* s', *and* $[\![\texttt{exec}(c)]\!]$ *yields* r_c, *the following program:*

$$\texttt{exec}\ (\lambda x\,.\ ((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ \texttt{userInput})$$

executes as follows whenever $[\![\texttt{flipCoin}()]\!]$ *yields* hd*:*

$\rightharpoonup\ \texttt{exec}\ (\lambda x.((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ s)$
$\rightharpoonup\ \texttt{exec}\ ((\texttt{flipCoin} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup\ \texttt{exec}\ ((\mathsf{hd} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup\ \texttt{exec}\ (\texttt{sanitize}\ s)$
$\rightharpoonup\ \texttt{exec}\ s'$
$\rightharpoonup\ \mathsf{r\_}s'$

This execution is safe as the input is sanitized before being executed. If $[\![\texttt{flipCoin}()]\!]$ *yields* tl*, the execution proceeds as follows:*

$\ldots$
$\rightharpoonup\ \texttt{exec}\ ((\mathsf{tl} = \mathsf{hd})\ (\texttt{sanitize}\ s)\ s)$
$\rightharpoonup\ \texttt{exec}\ s$
$\rightharpoonup\ \mathsf{r\_}s$

This execution is unsafe. As the user input is not sanitized, the user can execute any "bad" command.

Example 8 emphasizes a first distinction between static verification and our dynamic approach. Static techniques reject the whole program as it contains at least one bad execution. Our approach rejects executions where $[\![\texttt{flipCoin}()]\!]$ yielded tl, but accepts the others.

5.2.1 Observable Actions

An *observable action* is an action performed by the program, and observed by the execution monitor (and possibly rejected). As in similar work [24, 25, 64, 63, 39], the observable actions are the calls to external functions. However, the novelty is the fact that not only function identifier and argument values are taken into account, but also the history of how the arguments were computed.

To keep track of the history of the computations the resulting constants are annotated with a *function application tree* (FAT), $\hat{\tau} ::= f(\tau_1, \ldots, \tau_n)$ where $\tau ::= \hat{\tau} \mid c$[1]. A FAT is intended to capture the full history of function applications producing a constant. For example, $\mathtt{f}(\mathtt{g}()){:}1$ means that 1 is the result of applying $\mathtt{f}(\cdot)$ to $\mathtt{g}()$. Each time a function f^n is called with some arguments, $\tau_1{:}c_1, \ldots, \tau_n{:}c_n$, a new tree is constructed: $f(\tau_1, \ldots, \tau_n)$. This newly created tree serves both as the annotation of the resulting constant, and as the descriptor of the observable action that took place. The reduction of t to t' is written $t \xrightarrow{\hat{\tau}} t'$ if it generates the observation $\hat{\tau}$,

[1] As opposed to previous chapters, τ denotes an observable action and not a silent action.

$$\frac{t_1 \xrightarrow{\hat{\tau}} t_1'}{t_1\ t_2 \xrightarrow{\hat{\tau}} t_1'\ t_2}\ \text{T-AppL} \qquad \frac{c_1 = c_2}{(\tau_1{:}c_1 = \tau_2{:}c_2)\ t_1\ t_2 \xrightarrow{\epsilon} t_1}\ \text{T-CondT}$$

$$\frac{t_2 \xrightarrow{\hat{\tau}} t_2'}{v_1\ t_2 \xrightarrow{\hat{\tau}} v_1\ t_2'}\ \text{T-AppR} \qquad \frac{t_1 \xrightarrow{\hat{\tau}} t_1'}{(t_1 = t_2)\ t_3\ t_4 \xrightarrow{\hat{\tau}} (t_1' = t_2)\ t_3\ t_4}\ \text{T-CondL}$$

$$\frac{-}{(\lambda x\,.\,t)\ v \xrightarrow{\epsilon} t[v/x]}\ \text{T-AppAbs} \qquad \frac{c_1 \neq c_2}{(\tau_1{:}c_1 = \tau_2{:}c_2)\ t_1\ t_2 \xrightarrow{\epsilon} t_2}\ \text{T-CondF}$$

$$\frac{t_2 \xrightarrow{\hat{\tau}} t_2'}{(\tau_1{:}c_1 = t_2)\ t_3\ t_4 \xrightarrow{\hat{\tau}} (\tau_1{:}c_1 = t_2')\ t_3\ t_4}\ \text{T-CondR}$$

$$\frac{-}{f^n\ \tau_1{:}c_1\ \ldots\ \tau_n{:}c_n \xrightarrow{f(\tau_1,\ldots,\tau_n)} f(\tau_1,\ldots,\tau_n){:}[\![f^n(c_1,\ldots,c_n)]\!]}\ \text{T-AppFun}$$

Figure 5.1: Reduction rules with history annotations and observable actions.

and $t \xrightarrow{\epsilon} t'$ otherwise. The extended semantics with annotations is described in Figure 5.1. An unannotated reduction sequence can always be annotated to form a corresponding annotated reduction sequence, and vice versa. In other words, $\rightharpoonup$ and $\rightarrow$ are bisimilar. Depending on the context, $[\hat{\tau}]$ denotes $\hat{\tau}$ or ϵ. If the generated observation is not relevant, the reduction is written $t \rightarrow t'$.

Theorem 16. *Let $\sim$ be a relation between unannotated terms and annotated terms such that $t \sim t'$ holds whenever t equals the term formed by replacing all annotated constants $\tau{:}c$ in t' with c. If $t_0 \sim t_0'$ then either both t_0 and t_0' are in their normal forms, or $t_0 \rightarrow t_1$ and $t_0' \xrightarrow{\tau} t_1'$, for some (unique) t_1 and t_1' such that $t_1 \sim t_1'$.*

Proof. This follows from the fact that each Rule for $\rightarrow$ has a corresponding T-Rule for $\xrightarrow{\tau}$ and the fact that the annotations do not affect which rule is chosen. Since both reduction relations are deterministic, t_1 and t_1' are unique. □

Definition 23 (Observable Trace: $\omega \circ \mathcal{T}$)**.** *Given an execution $e = t_0 \rightharpoonup \ldots (\rightharpoonup t_n)$, $\mathcal{T}(e)$ is the annotated reduction sequence $t_0' \xrightarrow{[\hat{\tau}_1]} \ldots (\xrightarrow{[\hat{\tau}_n]} t_n')$ where t_i' equals t_i with every constant annotated; in particular, every constant of t_0' is annotated $c : c$. Furthermore $\omega(\mathcal{T}(e))$ denotes the* observable trace *of e, which is the sequence of observable actions $[\hat{\tau}_1], \ldots (, [\hat{\tau}_n])$ with the silent actions ϵ filtered out.*

Example 9. *Let the first execution in Example 8 be denoted by e. The annotated execution $\mathcal{T}(e)$ looks as follows:*

$$\texttt{exec}\ (\lambda x.((\texttt{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\texttt{sanitize}\ x)\ x)\ \texttt{userInput})$$

$$
\begin{aligned}
&\xrightarrow{\mathsf{userInput}()} \; \mathit{exec}\ (\lambda x.((\mathit{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\mathit{sanitize}\ x)\ x)\ \mathsf{userInput}(){:}s)\\
&\xrightarrow{\epsilon} \; \mathit{exec}\ ((\mathit{flipCoin} = \mathsf{hd}{:}\mathsf{hd})\ (\mathit{sanitize}\ \mathsf{userInput}(){:}s)\ \mathsf{userInput}(){:}s)\\
&\xrightarrow{\mathsf{flipCoin}()} \; \mathit{exec}\ ((\mathsf{flipCoin}(){:}\mathsf{hd} = \mathsf{hd}{:}\mathsf{hd})\ (\mathit{sanitize}\ \mathsf{userInput}(){:}s)\ \mathsf{userInput}(){:}s)\\
&\xrightarrow{\epsilon} \; \mathit{exec}\ (\mathit{sanitize}\ (\mathsf{userInput}(){:}s))\\
&\xrightarrow{\mathsf{sanitize}(\mathsf{userInput}())} \; \mathit{exec}\ (\mathsf{sanitize}(\mathsf{userInput}()){:}s')\\
&\xrightarrow{\mathsf{exec}(\mathsf{sanitize}(\mathsf{userInput}()))} \; \mathsf{exec}\ (\mathsf{sanitize}(\mathsf{userInput}())){:}\mathsf{r\_}s'
\end{aligned}
$$

The observable trace of e, $\omega(\mathcal{T}(e))$, *is:* userInput(), flipCoin(), sanitize(userInput()), exec(sanitize(userInput())).

The calculus ensures that the annotation of any constant fully captures how that value was computed, and filters out unrelated processing. In fact, if each constant c in a term t is annotated with c itself and $t \rightarrow^* \hat{\tau}{:}\,c'$ then $\hat{\tau}$ alone can be used to recover the computation resulting in the constant c'. This property is formalized in Theorem 17.

Theorem 17. *Given a term* t *in which all constants have the shape* $c{:}\,c$, *if* $t \rightarrow^* \tau{:}\,c'$ *then* $term(\tau) \rightarrow^* \tau : c'$ *where* $term(f(\tau_1, \ldots, \tau_n)) = f^n\ term(\tau_1)\ \ldots\ term(\tau_n)$ *and* $term(c) = c{:}\,c$.

Proof. We start by showing that if, for all annotated constants $\tau : c$ in a term t, $term(\tau) \rightarrow^* \tau : c$ holds and $t \rightarrow t'$ then, for all annotated constants $\tau' : c'$ in t', $term(\tau') \rightarrow \tau' : c'$ holds . This is shown by induction on the derivation tree of $t \rightarrow t'$. All cases except T-AppFun are trivial as no other rule introduces a new constant. For the T-AppFun case we need to show that $term(f(\tau_1, \ldots, \tau_n)) \rightarrow^* term(f(\tau_1, \ldots, \tau_n)) : [\![f^n(c_1, \ldots, c_n)]\!]$. We first note that $term(f(\tau_1, \ldots, \tau_n)) = f^n\ term(\tau_1)\ \ldots\ term(\tau_n)$. Since we know that $term(\tau_i) \rightarrow^* \tau_i{:}\,c_i$ for all annotated constants in t, we have $f^n term(\tau_1) \ldots term(\tau_n) \rightarrow^* f^n\,\tau_1{:}\,c_1\ \ldots\ \tau_n{:}\,c_n \rightarrow f(\tau_1, \ldots, \tau_n){:}\,[\![f^n(c_1, \ldots, c_n)]\!]$.

The result now follows from the fact that all constants in t are on the form $c{:}\,c$ and $term(c) \rightarrow^* c{:}\,c$. □

Note, however, that the trace of a reduction $t \rightarrow^* \tau{:}\,c$ may not be equal to the trace of $term(\tau) \rightarrow^* \tau{:}\,c$ since some computations may be discarded in the former reduction. In Example 9 for instance, `exec (sanitize (userInput()))` is sufficient to retrieve the core processing resulting in $\mathsf{r\_}s'$ (from which the `flipCoin` related code has been filtered out).

Discussion regarding branch sensitivity The choice of tracking direct function applications and not decisions regarding branching (i.e. tracking direct data flows and not indirect ones) is deliberate but not a fundamental requirement of the approach. The calculus could in principle be adapted to take branching decisions (explicit indirect flows) into account simply by (1) annotating terms with a context describing which computations the current computations depends upon and

(2) update this context based on τ_1 and τ_2 in the T-CondT and T-CondF rules. However, from a theoretical point of view, the policies we currently have in mind are strongly related to data processing (i.e. *what* is actually computed rather than *under what conditions* something is computed) and can be conveniently enforced using existing taint tracking mechanisms (i.e. mechanisms tracking only direct flows). Moreover, from a practical point of view, the absence of efficient dynamic data tracking mechanisms for a commercial-level system handling indirect flows, on top of which to implement our approach, reinforces this choice.

5.3 Policies

In our setting, a policy $\mathcal{P}$ specifies which computations (nesting of function applications) are allowed to be performed. Since each new function application is recorded in the form of an observable action, policies can be conveniently expressed as a predicate over traces, i.e. sequences of observable actions. This nomenclature is standard in monitoring, [85, 1, 64]. A reduction sequence, e, is said to be *accepted* by $\mathcal{P}$ if and only if $\mathcal{P}(\omega(\mathcal{T}(e)))$ holds and *rejected* otherwise. In this paper, we do not consider arbitrary policies. Some policies can not even be enforced in practice. The class of policies considered include only the ones that are *local* and *subtree closed*.

Definition 24 (Local Policy)**.** *A policy, $\mathcal{P}$, is said to be* local *if a predicate P exists such that $P(c)$ holds for all $c \in \mathsf{C}$ and $\mathcal{P}(\hat{\tau}_1, \ldots (, \hat{\tau}_n))$ holds iff $\forall_{0 \leq i\,(<n)}\, P(\hat{\tau}_i)$ holds.*

This property allows us to focus on stateless policies stating *which* computations may be performed rather than *when* they may be performed and alleviates the need of a global monitor state. This constraint is however not fundamental and can be relaxed by instead stating that the set of accepted traces should be prefix-closed (i.e. that if a trace is accepted then so should all its prefixes). This would allow policies to express temporal properties of the observable traces, such as *"f may not be evaluated until g has been evaluated"* and would arguably be more suitable when, for instance, dealing with functions with side-effects. Such class of policies has however been studied in depth already, [24, 25, 1] and we see no incompatibility with those studies and the results in this paper.

Definition 25 (Subtree Closed Policy)**.** *A local policy is* subtree closed *if the set of observable actions for which P (Definition 24) holds is subtree closed.*

This property rules out policies that for instance accept the evaluation of $g(f())$ but rejects the evaluation of $f()$. As discussed in Section 5.3.1 such policies are not meaningful in languages with call-by-value semantics like Java, since $f()$ indeed needs to be evaluated in order to evaluate $g(f())$ (in a call-by-name setting however, $f()$ does not necessarily need to be evaluated).

Example 10. *A typical example of a policy which is local and subtree closed could for instance express that* `sanitize` *accepts any string, while* `exec` *only accepts results from the* `sanitize` *function (i.e. all strings must be sanitized before they are passed to* `exec`*). With such a policy, the evaluation of the term of Example 8 is accepted if* `flipCoin` *returns* hd *and rejected otherwise.*

Before providing a syntax and accompanying semantics for defining policies (Section 5.5), the paper examines the relation between the different policy classes (Section 5.3.1) and introduces the notion of labels (Section 5.4).

5.3.1 A Hierarchy of Policy Classes

This section discusses the relation between prefix-closed (PC), local (Loc) and subtree-closed (SC) policies, both in general and under the assumption of a call-by-value semantics (CBV).

The hierarchy can be summarized as follows: Subtree-closed policies are by definition also local. Local policies are not necessarily subtree closed (Theorem 18) except for CBV semantics (Theorem 19). Local policies are prefix-closed (Theorem 20) but prefix-closed policies are not necessarily local, not even when assuming a CBV semantics (Theorem 21). Figure 5.2 depicts the hierarchy in a Venn diagram.

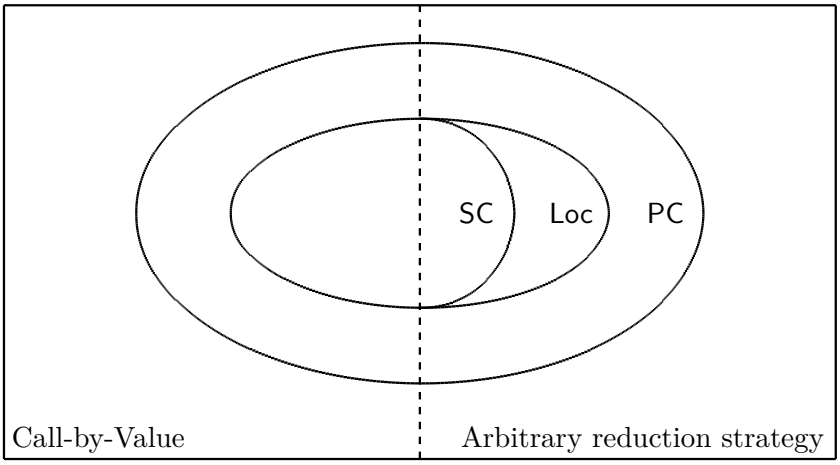


Figure 5.2: Relation between policy classes

As a general CBV reduction relation, the paper use the $\rightharpoonup$-relation presented in Section 5.2.

Lemma 10. *In CBV semantics, if* $\hat{\tau}_1, \ldots, \hat{\tau}_n, f^i(\tau_j, \ldots, \tau_k)$ *is a prefix of a trace of an execution* $\omega(\mathcal{T}(t_0 \rightharpoonup \ldots (\rightharpoonup t_m)))$*, then* $(\{\tau_j, \ldots, \tau_k\} \setminus \mathsf{C}) \subseteq \{\hat{\tau}_1, \ldots, \hat{\tau}_n\}$.

Proof. Any argument of f^i must have the form $\tau_l : c$. Either c comes from the original term, in which case $\tau_j \in \mathsf{C}$ or c is the result of a function application prior to $f^i(\tau_j, \ldots, \tau_k)$ in which case $\tau_j \in \{\hat{\tau}_1, \ldots, \hat{\tau}_n\}$. □

Theorem 18 ($\mathsf{Loc} \not\subseteq \mathsf{SC}$)**.** *Local policies are not necessarily subtree closed.*

Proof. Any policy accepting all permutations of a non-subtree-closed set of observable actions is in Loc but not in SC. The policy that accepts the trace $g(f())$, but not the trace $f()$, is an example of such policy. (Note that this policy hold in a call-by-name setting if the result of $f()$ is not needed when evaluating $g(f())$.) □

By assuming a CBV semantics, the counterexample in the above proof is ruled out. In fact no counter example can be constructed for the above theorem under such semantics.

Theorem 19 (CBV $\Rightarrow$ ($\mathsf{Loc} = \mathsf{SC}$))**.** *In a CBV semantics, any local policy is subtree-closed.*

Proof. Let P be the predicate (Definition 24) of a local policy $\mathcal{P}$. In CBV, any $\hat{\tau}_j = f^j(\tau_k, \ldots, \tau_l)$ in a trace $\hat{\tau}_1, \ldots (, \hat{\tau}_n)$ accepted by $\mathcal{P}$ is such that every $\tau_m \in \tau_k, \ldots, \tau_l$ is either in C (and $P(\tau_m)$ by Definition 24) or, by Lemma 10, in $\hat{\tau}_1, \ldots, \hat{\tau}_{j-1}$ (and $P(\tau_m)$ by Definition 24). Therefore, CBV $\Rightarrow$ ($\mathsf{Loc} \subseteq \mathsf{SC}$) and Definition 25 concludes. □

Theorem 20 ($\mathsf{Loc} \subseteq \mathsf{PC}$)**.** *A local policy is prefix-closed.*

Proof. Let P be the predicate (Definition 24) of a local policy $\mathcal{P}$. If $\mathcal{P}$ accepts some trace ω then P must hold for all observable actions in ω. Since P holds for all observable actions in each prefix of ω, $\mathcal{P}$ accepts all prefixes of ω which makes $\mathcal{P}$ prefix-closed. □

Theorem 21 ($\mathsf{PC} \not\subseteq \mathsf{Loc}$)**.** *Prefix-closed policies are not necessarily local, not even when assuming CBV.*

Proof. The policy that accepts $f()$ and $f(), g()$, but not $g()$, is prefix-closed but not local. □

5.4 Labels

Manipulating FATs at runtime for enforcement is not efficient in practice. For instance, if a policy requires an argument to be the result of an even number of applications of `toggle`, the FATs could grow indefinitely, despite the fact that a boolean value would suffice to maintain and describe the relevant computational history. To circumvent this problem, *labels* are introduced to replace FATs. A label can be seen as an abstraction of a FAT.

Labels (denoted by α, β, ...) range over a set L of ground labels closed under $\oplus$. A special label $\mathsf{L}_0 \in \mathsf{L}$ denotes the default label and is used for constants present in the initial term. By convention, L_f denotes the label associated to f^n (L_f can be the same as L_g). The deterministic $\oplus$ operator is used to compute the label of the result of a function application. $\oplus$ can be seen as an abstraction of the FAT constructor. The pair $\langle \mathsf{L}, \oplus \rangle$ is referred to as a *labeling scheme*. It has the nice

$$\frac{t_1 \to_\oplus t_1'}{t_1\ t_2 \to_\oplus t_1'\ t_2}\ \text{L-AppL} \qquad \frac{-}{(\alpha_1 : c = \alpha_2 : c)\ t_1\ t_2 \to_\oplus t_1}\ \text{L-CondT}$$

$$\frac{t_2 \to_\oplus t_2'}{v_1\ t_2 \to_\oplus v_1\ t_2'}\ \text{L-AppR} \qquad \frac{t_1 \to_\oplus t_1'}{(t_1 = t_2)\ t_3\ t_4 \to_\oplus (t_1' = t_2)\ t_3\ t_4}\ \text{L-CondL}$$

$$\frac{-}{(\lambda x.t_1)\ v \to_\oplus t_1[v/x]}\ \text{L-AppAbs} \qquad \frac{c_1 \neq c_2}{(\alpha_1 : c_1 = \alpha_2 : c_2)\ t_1\ t_2 \to_\oplus t_2}\ \text{L-CondF}$$

$$\frac{t_2 \to_\oplus t_2'}{(\alpha_1 : c = t_2)\ t_3\ t_4 \to_\oplus (\alpha_1 : c = t_2')\ t_3\ t_4}\ \text{L-CondR}$$

$$\frac{\gamma = \mathsf{L}_f \oplus \alpha_1 \oplus \cdots \oplus \alpha_n}{f^n\ \alpha_1 : c_1\ \ldots\ \alpha_n : c_n \to_\oplus \gamma : [\![f^n(c_1, \ldots, c_n)]\!]}\ \text{L-AppFun}$$

Figure 5.3: Reduction rules with labels.

property to be instantiable to reflect the taint tracking policy of TaintDroid, on which our implementation is based.

For enforcement purposes, programs are evaluated using a new semantics $\to_\oplus$ manipulating labels similar to the one of Figure 5.1 where observable actions are removed and tree-annotated constants $\tau_\mathsf{i} : c_i$ are replaced by label-annotated constants $\alpha_i : c_i$. The new set of reduction rules is presented in Figure 5.3.

Just as in the case with FATs, the labels do not affect the resulting terms and every annotated reduction sequence has a corresponding unannotated reduction sequence. *strip*(t) is t with the label of every constant removed and *init*(t) is the term t in which each unlabeled constant c is replaced by $\mathsf{L_0} : c$.

Proposition 7. *If* $t \to_\oplus^n t'$ *then* *strip*$(t) \rightharpoonup^n$ *strip*(t').

The reverse is true for every term only if $\oplus$ is a total function. It is a potential property of an execution.

Definition 26 (Valid Labeling)**.** *An execution* $t_0 \rightharpoonup^n t_n$ *has a* valid $\langle \mathsf{L}, \oplus \rangle$-labeling *iff there exists* $t_1', \ldots, t_n'$ *such that* *init*$(t_0) \to_\oplus^n t_n'$ *where* $t_i =$ *strip*(t_i') *for* $i \in [1, n]$.

By allowing $\oplus$ to be a partially defined function certain sequences of reductions are ruled out due to the premise of the L-AppFun rule. This can be (and is) exploited as an enforcement mechanism as shown in the following definition.

Definition 27 ($\langle \mathsf{L}, \oplus \rangle$ Enforcement)**.** $\langle \mathsf{L}, \oplus \rangle$ enforces *a policy* $\mathcal{P}$ *if all executions with valid* $\langle \mathsf{L}, \oplus \rangle$*-labelings are accepted by* $\mathcal{P}$*. If the reverse also holds, i.e. all executions accepted by* $\mathcal{P}$ *have a valid* $\langle \mathsf{L}, \oplus \rangle$*-labeling,* $\langle \mathsf{L}, \oplus \rangle$ *is said to* precisely enforce $\mathcal{P}$.

An example of how a labeling scheme enforces a *sanitize before executing*-policy is presented in Example 12 in Section 5.6.

When reasoning about the correctness of the labeling scheme we need a way to tell which label a certain FAT corresponds to. For this purpose we define the $R_{\mathsf{L},\oplus}$-function as follows:

Definition 28 ($R_{\mathsf{L},\oplus}$)**.** *$R_{\mathsf{L},\oplus}(\tau)$ is defined as follows:*

$$R_{\mathsf{L},\oplus}(c) = \mathsf{L}_0$$
$$R_{\mathsf{L},\oplus}(f(\tau_1, \ldots, \tau_n)) = \mathsf{L}_f \oplus R_{\mathsf{L},\oplus}(\tau_1) \oplus \ldots \oplus R_{\mathsf{L},\oplus}(\tau_n)$$

Lemma 11 (Labels abstract FATs)**.** *For any t_0:*

$$t_o \rightarrow^n \tau{:}c \ \wedge \ R_{\mathsf{L},\oplus}(\tau) \in \mathsf{L} \ \Leftrightarrow \ \mathit{init}(t_0) \rightarrow^n_{\oplus} \mathsf{R}_{\mathsf{L},\oplus}(\tau){:}c$$

Proof. By induction on the length of the derivation. The only interesting case, T-AppFun, follows directly from the semantical definitions and Definition 28. □

An execution is accepted iff, for any observable action $\hat{\tau}$ generated, $R_{\mathsf{L},\oplus}(\hat{\tau})$ is defined.

Theorem 22 (Accepted execution)**.** *Execution $e = t_0 \rightarrowtail^n t_n$ has a valid $\langle \mathsf{L}, \oplus \rangle$-labeling ($\mathit{init}(t_0) \rightarrow^n_{\oplus} t'_n$) iff, for any $\hat{\tau}$ in $\omega(\mathcal{T}(e))$, $R_{\mathsf{L},\oplus}(\hat{\tau})$ is defined ($R_{\mathsf{L},\oplus}(\hat{\tau}) \in \mathsf{L}$).*

Proof. Assuming $\rightarrow$ goes through, $\rightarrow_{\oplus}$ can only fail on L-AppFun, whose corresponding rule T-AppFun is the only one generating observable FATs. Hence, if $R_{\mathsf{L},\oplus}(\hat{\tau})$ is defined for any $\hat{\tau}$ in $\omega(\mathcal{T}(e))$ then Lemma 11 helps conclude that $\mathit{init}(t_0) \rightarrow^n_{\oplus} t'_n$. The other direction is proved by induction on the length of $\mathit{init}(t_0) \rightarrow^n_{\oplus} t'_n$ and by observing that, by Lemma 10 and induction hypothesis, for any subtree τ of the potentially newly generated FAT $\hat{\tau}_{n+1}$, $R_{\mathsf{L},\oplus}(\tau)$ is defined and, by Lemma 11, equal to the corresponding label in t'_n. Hence, $R_{\mathsf{L},\oplus}(\hat{\tau}_{n+1})$ is defined. □

5.5 Defining and Enforcing Policies

As suggested previously, security policies in this work are defined by a bottom-up *deterministic finite tree automaton* (DFTA) [20] that characterizes a set of FATs.

Definition 29 (DFTA [20])**.** *A (bottom-up)* deterministic finite tree automaton *over a ranked alphabet A is a tuple $\mathcal{A} = (Q, A, Q_f, \Delta)$ where Q is a set of (unary) states, $Q_f \subseteq Q$ is a set of accepting states and Δ is a partial function defined by a set of transition rules of the form $a(q_1(\mathfrak{t}_1), \ldots, q_n(\mathfrak{t}_n)) \rightarrowtriangle q(a(\mathfrak{t}_1, \ldots, \mathfrak{t}_n))$ where $n \geq 0$, $a \in A_n$, $q_1, \ldots, q_n \in Q$ and $\mathfrak{t}_1, \ldots, \mathfrak{t}_n$ are terms over A.*

A ground term $\mathfrak{t}$ of A is accepted by an automata $\mathcal{A}$ ($\mathfrak{t} \in \mathcal{L}(\mathcal{A})$) if $\mathfrak{t} \rightarrowtriangle^* q(\mathfrak{t})$ for some $q \in Q_f$.

Intuitively, an automaton accepts a term t iff every node in the tree t can be annotated with a state such that the root node is annotated with a final state, and the annotations are compatible with the transition rules Δ.

A policy is defined in terms of a DFTA over $\mathsf{F} \cup \mathsf{C}$, from now on referred to as a *security tree automaton* (STA).

Definition 30 (STA)**.** *A* security tree automaton *is a DFTA* $\mathcal{A} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$ *such that there exists* q_0 *in* Q *and for all* c *in* C*:* $c \rightarrowtriangle q_0(c) \in \Delta$*.*

$\mathcal{P}_\mathcal{A}$ denotes the policy defined by the STA $\mathcal{A}$ whose set of accepted traces is $\{[\hat{\tau}_0, \ldots (, \hat{\tau}_n)] \mid \forall i \in [0, n].\ \hat{\tau}_i \in \mathcal{L}(\mathcal{A})\}$.

Lemma 12. *A policy defined in terms of a STA is local.*

Proof. The predicate $P(\tau) \stackrel{\text{def}}{=} \tau \in (\mathsf{C} \cup \mathcal{L}(\mathcal{A}))$ is a valid candidate showing that the policy $\mathcal{P}_\mathcal{A}$ is local according to Definition 24. □

Additionally, as for ordinary security automata [113], all states of a STA are required to be accepting (this does not imply that all trees are accepted, since Δ is partial). This requirement is sufficient to ensure that STA policies are subtree closed.

Lemma 13. *The language of a STA is subtree closed.*

Proof. If $f(\tau_1, \ldots, \tau_n)$ is accepted, then there exist some states q, $q_1, \ldots, q_n$ such that $f(\tau_1, \ldots, \tau_n) \rightarrowtriangle^* f(q_1(\tau_1), \ldots, q_n(\tau_n)) \rightarrowtriangle q(f(\tau_1, \ldots, \tau_n))$. This means that, for all τ_i in $\tau_1, \ldots, \tau_n$, $\tau_i \rightarrowtriangle^* q_i(\tau_i)$ and since $Q_f = Q$, τ_i is also accepted. □

If there exists an injective function from the states of an STA $\mathcal{A}$ to the labels of a labeling scheme $\langle \mathsf{L}, \oplus \rangle$ and $\oplus$ simulates the transitions in Δ then $\langle \mathsf{L}, \oplus \rangle$ precisely enforces $\mathcal{P}_\mathcal{A}$.

Theorem 23 (Equivalent $\langle \mathsf{L}, \oplus \rangle$)**.** $\langle \mathsf{L}, \oplus \rangle$ *precisely enforces the policy described by the STA* $\mathcal{A} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$ *if there exists an injective function* $L : Q \to \mathsf{L}$ *such that, with* $\mathsf{L}_c = \mathsf{L}_0$ *for all* c *in* C*:*

$$L(q) = \mathsf{L}_a \oplus L(q_1) \oplus \ldots \oplus L(q_n) \iff a(q_1(\tau_1), \ldots, q_n(\tau_n)) \rightarrowtriangle q(a(\tau_1, \ldots, \tau_n)) \in \Delta \quad (5.1)$$

Proof. Following Definition 27, it is sufficient that for a given L the following holds for any execution $e = t_0 \rightharpoonup^n t_n$:

$$e \text{ has a valid } \langle \mathsf{L}, \oplus \rangle\text{-labeling} \iff e \text{ is accepted by } \mathcal{P}_\mathcal{A}$$

Since the policy is local and since all states in the policy automaton are accepting, by Lemma 22 and Definition 30, it is sufficient to show the following for each $\hat{\tau} \in \omega(\mathcal{T}(e))$:

$$R_{\mathsf{L},\oplus}(\hat{\tau}) \in \mathsf{L} \quad \Longleftrightarrow \quad \exists q.\ \hat{\tau} \rightarrowtriangle^* q(\hat{\tau})$$

which follows from the following holding for all τ:

$$\tau \rightarrowtriangle^* q(\tau) \ \Rightarrow\ R_{\mathsf{L},\oplus}(\tau) = L(q) \tag{5.2}$$

$$R_{\mathsf{L},\oplus}(\tau) = \alpha \ \Rightarrow\ \tau \rightarrowtriangle^* L^{-1}(\alpha)(\tau) \tag{5.3}$$

(5.2) and (5.3) follow by induction on τ by assuming (5.1). □

We now turn to the syntax of the language for describing security tree automata. We first give a concrete self-explanatory example and then generalize this into a proper definition.

Example 11. *A policy stating that the function* `exec` *only accepts strings returned by another function* `sanitize`*, or concatenations of any such strings is written as:*

$$\begin{array}{l}
\{\mathsf{unsanitized}, \mathsf{sanitized}\},\\
\{\mathtt{sanitize}(\alpha)\colon \quad \mathit{true} \to \mathit{sanitized}\\
\quad \mathtt{concat}(\alpha_1, \alpha_2)\colon \ \alpha_1 = \alpha_2 = \mathit{sanitized} \to \mathit{sanitized}\\
\qquad\qquad\qquad\quad \mathit{true} \to \mathit{unsanitized}\\
\quad \mathtt{exec}(\alpha)\colon \qquad \alpha = \mathit{sanitized} \to \mathit{sanitized}\}
\end{array}$$

Where the default label L_0 *equals* unsanitized*.*

The general syntax of a policy is defined as follows.

Definition 31 (Policy Syntax)**.** *Syntactically, a policy is expressed as follows*

$$\begin{array}{rl}
\{\mathsf{L}_0, \mathsf{L}_1, \ldots, \mathsf{L}_{n_\mathsf{L}}\},\\
\{f_1^{n_1}(\alpha_{11}, \ldots, \alpha_{1n_1})\colon & \mathit{guard}_{11} \to \mathit{expr}_{11}\\
& \cdots\\
& \mathit{guard}_{1g_1} \to \mathit{expr}_{1g_1},\\
\vdots & \\
f_k^{n_k}(\alpha_{k1}, \ldots, \alpha_{kn_k})\colon & \mathit{guard}_{k1} \to \mathit{expr}_{k1}\\
& \cdots\\
& \mathit{guard}_{kg_k} \to \mathit{expr}_{kg_k}\}
\end{array}$$

where each guard is a boolean expression and each expr is a label expression, both of which are composed from the declared label constants, the argument labels and simple tests such as equality.

Intuitively, when $f_i^{n_i}$ is invoked the formal parameters, $\alpha_{i1}, \ldots, \alpha_{in_i}$, are bound to the labels associated with the arguments. The return label is computed from the expression expr_{ij} corresponding to the first guard guard_{ij} that holds among $\mathit{guard}_{i1}, \ldots, \mathit{guard}_{ig_i}$. If no guard holds, the invocation is to be seen as a violation of the policy.

A formal translation into a security tree automaton follows.

Definition 32 (Policy Semantics)**.** *Given a policy $\mathcal{P}$ in the syntax of Definition 31, the corresponding security tree automaton, $\mathcal{A}_{\mathcal{P}} = (Q, \mathsf{F} \cup \mathsf{C}, Q, \Delta)$, is defined as follows: $Q = \{q_0, q_1, \ldots, q_{n_\mathsf{L}}\}$ and $\Delta = \{c \twoheadrightarrow q_0(c) \mid c \in \mathsf{C}\} \cup \bigcup \delta_{ij}$, where each δ_{ij} represents the set of automaton transitions corresponding to guard j in clause i:*

$$
\begin{aligned}
&\{\ f_i(q'_1(x_1), \ldots, q'_{n_i}(x_{n_i})) \twoheadrightarrow q'(f_i(x_1, \ldots, x_{n_i})) \\
&\mid\ q'_1 \ldots q'_{n_i} \in Q\ \wedge\ q' = [\![expr_{ij}[q_k/\mathsf{L}_k][q'_k/\alpha_{ik}]]\!] \\
&\quad \wedge\ [\![guard_{ij}[q_k/\mathsf{L}_k][q'_k/\alpha_{ik}]]\!] \\
&\quad \wedge \neg \bigvee_{1 \le g < j} [\![guard_{ig}[q_k/\mathsf{L}_k][q'_k/\alpha_{ik}]]\!]\ \}
\end{aligned}
$$

As mentioned in Section 5.4, the enforcement mechanism does not, for practical reasons, work directly on FATs but on labels. To enforce a policy, an equivalent labeling scheme is needed. For policies expressed in the syntax of Definition 31, such labeling scheme can be defined as follows.

Definition 33. *Given a policy $\mathcal{P}$ in the syntax of Definition 31, the labeling scheme $LS(\mathcal{P})$ is defined as $\langle \{\mathsf{L}_0, \mathsf{L}_1, \ldots, \mathsf{L}_{n_\mathsf{L}}\} \cup \{\mathsf{L}_{f_i}\}, \oplus \rangle$ where $\mathsf{L}_{f_i} \oplus \alpha_1 \oplus \cdots \oplus \alpha_{n_i}$ is:*

$$
\begin{aligned}
&\quad \textit{if } [\![guard_{i1}[\alpha_k/\alpha_{ik}]]\!] \ \textit{then } [\![expr_{i1}[\alpha_k/\alpha_{ik}]]\!] \\
&\textit{else if } \ldots \qquad\qquad\quad \textit{then } \ldots \\
&\textit{else if } [\![guard_{ig_i}[\alpha_k/\alpha_{ik}]]\!] \ \textit{then } [\![expr_{ig_i}[\alpha_k/\alpha_{ik}]]\!]
\end{aligned}
$$

We now show that $LS(\mathcal{P})$ indeed precisely enforces $\mathcal{P}$ according the semantics defined in Definition 32.

Theorem 24 (Correctness of LS)**.** *Given a policy $\mathcal{P}$ in the syntax of Definition 31, $LS(\mathcal{P})$ enforces $\mathcal{P}$ precisely.*

Proof. Follows directly from Theorem 23 with $L(q_i) = \mathsf{L}_i$ □

5.6 Labeled Imperative Language

A λ-calculus is a natural candidate for the core language since central concepts such as function applications are easily represented. Furthermore, the structure and potential data flow in a program is arguably clearer if written in the form of a λ-term than in a language with side effects. Nonetheless it is important to make sure that the calculus is general enough to serve as a foundation for other languages as well, such as Java which is the language of applications monitored by TreeDroid. This section introduces a While-language whose semantics track labels in a natural way. A straightforward encoding of the language in our calculus is then provided and shown to agree with the labeling semantics.

$$\frac{-}{\langle n, \sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_0 : n} \qquad \frac{\langle E, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}{\langle \mathrm{x}_k := E, \sigma\rangle \twoheadrightarrow_{\oplus} \sigma[\mathrm{x}_k \mapsto \alpha : n]}$$

$$\frac{-}{\langle \mathrm{x}_k, \sigma\rangle \twoheadrightarrow_{\oplus} \sigma(\mathrm{x}_k)} \qquad \frac{\langle E, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}{\langle \textbf{return } E, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha : n}$$

$$\frac{\langle S_1, \sigma\rangle \twoheadrightarrow_{\oplus} \sigma'}{\langle S_1; S_2, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S_2, \sigma'\rangle} \qquad \frac{\langle E_1, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n \quad \langle E_2, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n}{\langle \textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1, \sigma\rangle}$$

$$\frac{\langle S_1, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S_1'; S_2, \sigma'\rangle} \qquad \frac{\langle E_1, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \langle E_2, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n_2 \quad n_1 \neq n_2}{\langle \textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S_2, \sigma\rangle}$$

$$\frac{\langle E_1, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \langle E_2, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n_2 \quad n_1 \neq n_2}{\langle \textbf{ while } E_1 = E_2 \textbf{ do } S, \sigma\rangle \twoheadrightarrow_{\oplus} \sigma}$$

$$\frac{\langle E_1, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n_1 \quad \ldots \quad \langle E_m, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_m : n_m}{\langle f(E_1, \ldots, E_m), \sigma\rangle \twoheadrightarrow_{\oplus} \mathsf{L}_f \oplus \alpha_1 \oplus \cdots \oplus \alpha_m : [\![f(n_1, \ldots, n_m)]\!]}$$

$$\frac{\langle E_1, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_1 : n \quad \langle E_2, \sigma\rangle \twoheadrightarrow_{\oplus} \alpha_2 : n}{\langle \textbf{ while } E_1 = E_2 \textbf{ do } S, \sigma\rangle \twoheadrightarrow_{\oplus} \langle S; \textbf{ while } E_1 = E_2 \textbf{ do } S, \sigma\rangle}$$

Figure 5.4: Semantics of the While-language with Function Application Monitoring.

The language presented in this section is a simple imperative language with loops and the addition of labels, external function applications and return statement. The small-step operational semantics of the language is given in Figure 5.4. A configuration is represented as $\langle S, \sigma\rangle$ where S is the statement to be executed and σ the current store. The store maps variables to labeled values and the initial store, σ_0, maps each variable to L_0:0.

Example 12. *The program in this example is similar to the one of Example 8.*

$x_1 :=$ *userInput*();
IF *flipCoin() = 1* THEN
  $x_1 :=$ *sanitize*(x_1)
ELSE
$x_1 := x_1$;
RETURN *exec*(x_1)

With the labeling scheme, $\langle\{\mathsf{L}_0$, input, sanitized, result, $\mathsf{L}_{flipCoin}$, $\mathsf{L}_{sanitize}$, $\mathsf{L}_{userInput}$,

$\mathsf{L}_{exec}\}, \oplus\rangle$ *where*

$$\alpha_1 \oplus \alpha_2 \oplus \ldots \oplus \alpha_m = \begin{cases} \mathsf{L}_0 & \text{if } \alpha_1 = \mathsf{L}_{flipCoin} \\ \mathsf{input} & \text{if } \alpha_1 = \mathsf{L}_{userInput} \\ \mathsf{sanitized} & \text{if } \alpha_1 = \mathsf{L}_{sanitize} \\ \mathsf{result} & \text{if } \begin{cases} \alpha_1 = \mathsf{L}_{exec} \\ \alpha_2 = \mathsf{sanitized} \end{cases} \end{cases}$$

an execution in which `flipCoin` *yields 1 terminates:*

$$\begin{array}{ll} & \langle \mathrm{x}_1 := \texttt{userInput}(); \text{ if } \texttt{flipCoin}() = 1 \text{ then } \ldots,\ \sigma_0\rangle \\ \twoheadrightarrow_\oplus & \langle \text{if } \texttt{flipCoin}() = 1 \text{ then } \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle \\ \twoheadrightarrow_\oplus & \langle \mathrm{x}_1 := \texttt{sanitize}(\mathrm{x}_1); \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle \\ \twoheadrightarrow_\oplus & \langle \text{return } \texttt{exec}(\mathrm{x}_1),\ \sigma_0[x_1 \mapsto \mathsf{sanitized{:}7'}]\rangle \\ \twoheadrightarrow_\oplus & \mathsf{result{:}7''} \end{array}$$

and an execution in which `flipCoin` *yields 0 would get stuck:*

$$\begin{array}{ll} & \langle \mathrm{x}_1 := \texttt{userInput}(); \text{ if } \texttt{flipCoin}() = 1 \text{ then } \ldots,\ \sigma_0\rangle \\ \twoheadrightarrow_\oplus & \langle \text{if } \texttt{flipCoin}() = 1 \text{ then } \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle \\ \twoheadrightarrow_\oplus & \langle \mathrm{x}_1 := \mathrm{x}_1; \ldots,\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle \\ \twoheadrightarrow_\oplus & \langle \text{return } \texttt{exec}(\mathrm{x}_1),\ \sigma_0[x_1 \mapsto \mathsf{input{:}7}]\rangle \end{array}$$

since $\mathsf{L}_{exec} \oplus \mathsf{input}$ *is undefined.*

Figure 5.5 gives an encoding of While in the style of state transformers with $\mathsf{C} = \mathbb{N}$ and $\mathsf{F} = \{f_1, \ldots, f_k\}$.

Auxiliary definitions:

$$\begin{array}{rl} \mathit{get} = & \lambda s.\, \lambda x.\, s\ x \\ \mathit{set} = & \lambda s.\, \lambda x.\, \lambda v.\, \lambda k.\, (k = x)\ v\ (s\ k) \\ \mathit{fix} = & \lambda g.\, (\lambda x.\, g\ (\lambda y.\, x\ x\ y))\ (\lambda x.\, g\ (\lambda y.\, x\ x\ y)) \end{array}$$

Expressions:

$$\begin{array}{rl} [\![n]\!] = & \lambda s.\, \mathsf{L}_0 : n \\ [\![\mathrm{x}_k]\!] = & \lambda s.\, \mathit{get}\ s\ k \\ [\![f(E_1, \ldots, E_m)]\!] = & \lambda s.\, f\ ([\![E_1]\!]\ s)\ \ldots\ ([\![E_m]\!]\ s) \end{array}$$

Statements:

$$\begin{array}{rl} [\![S_1; S_2]\!] = & \lambda s.\, [\![S_2]\!]\ ([\![S_1]\!]\ s) \\ [\![\mathrm{x}_k := E]\!] = & \lambda s.\, \mathit{set}\ s\ k\ ([\![E]\!]\ s) \\ [\![\textbf{if } E_1 = E_2 \textbf{ then } S_1 \textbf{ else } S_2]\!] = & \lambda s.\, ([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ([\![S_1]\!]\ s)\ ([\![S_2]\!]\ s) \\ [\![\textbf{while } E_1 = E_2 \textbf{ do } S]\!] = & \mathit{fix}\ (\lambda w.\, \lambda s.\, ([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s) \\ [\![\textbf{return } E]\!] = & \lambda s.\, [\![E]\!]\ s \end{array}$$

Figure 5.5: Encoding of While in our λ-calculus

Definition 34. *A state is a term* $\lambda k.\, t$ *which reduces to a labeled constant when applied to a number:* $(\lambda k.\, t)\ n \to^* \alpha : c$. *A state* s *and a store* σ agree, $s \sim \sigma$, *iff* $s\ k \to^* \sigma(\mathrm{x}_k)$ *for all* k.

Initial state for evaluation is $s_0 = \lambda k.\, \mathsf{L_0} : \mathsf{0}$. The following lemmas show that $\twoheadrightarrow_\oplus$ evaluating P accepts the same executions as $\rightarrow_\oplus$ evaluating $[\![\mathrm{P}]\!]$.

Lemma 14 (Expression equivalence)**.** *If $s \sim \sigma$ then $\langle E, \sigma\rangle \twoheadrightarrow_\oplus \alpha : n \Leftrightarrow [\![E]\!]\ s \rightarrow_\oplus^* \alpha : n$.*

Proof. By structural induction on E. For $E \equiv n$, we have $\langle n, \sigma\rangle \twoheadrightarrow_\oplus \mathsf{L_0} : n$ and $[\![n]\!]\ s \rightarrow_\oplus \mathsf{L_0} : n$. For $E \equiv x_k$, we have $\langle \mathrm{x}_k, \sigma\rangle \twoheadrightarrow_\oplus \sigma(\mathrm{x}_k)$ and $[\![\mathrm{x}_k]\!]\ s \rightarrow_\oplus (\lambda s.get\ s\ k)\ s \rightarrow_\oplus get\ s\ k \rightarrow_\oplus s\ k$ where $s\ k$ reduces to $\sigma(\mathrm{x}_k)$ by Definition 34. For $E \equiv f(E_1, \ldots, E_m)$, $\langle f(E_1, \ldots, E_m), \sigma\rangle \twoheadrightarrow_\oplus \mathsf{L}_f \oplus \alpha_1 \oplus \ldots \oplus \alpha_m : [\![f(n_1, \ldots, n_m)]\!]$, assuming $\langle E_i, \sigma\rangle \twoheadrightarrow_\oplus \alpha_i : n_i$. By induction, $[\![f(E_1, \ldots, E_m]\!]\ s = \lambda s.f\ ([\![E_1]\!]s)\ \ldots\ ([\![E_m]\!]\ s)$ reduces to $f\ \alpha_1 : n_1\ \ldots\ \alpha_m : n_m$ and then $\mathsf{L}_f \oplus \alpha_1 \oplus \ldots \oplus \alpha_m : [\![f(n_1, \ldots, n_m)]\!]$. □

Definition 35. *The reduction relation, $\rightarrow^+$ is provided in Fig 5.6.*

$$\frac{[\![S_1]\!]\ s \rightarrow^+ [\![S_1']\!]\ s'}{[\![S_1; S_2]\!]\ s \rightarrow^+ [\![S_1'; S_2]\!]\ s'} \qquad \frac{[\![S_1]\!]\ s \rightarrow^+ s'}{[\![S_1; S_2]\!]\ s \rightarrow^+ [\![S_2]\!]\ s'}$$

$$\frac{[\![E]\!]\ s \rightarrow \alpha : n}{[\![x_k := E]\!]\ s \rightarrow^+ \lambda k'.(k' = k)\ \alpha : n\ (s\ k)}$$

$$\frac{[\![E_1]\!]\ s \rightarrow \alpha_1 : n \quad [\![E_2]\!]\ s \rightarrow \alpha_2 : n}{[\![\textbf{if } E_1 = E_2 \textbf{ then } S]\!]\ s \rightarrow^+ [\![S]\!]\ s}$$

$$\frac{[\![E_1]\!]\ s \rightarrow \alpha_1 : n_1 \quad [\![E_2]\!]\ s \rightarrow \alpha_2 : n_2 \quad n_1 \neq n_2}{[\![\textbf{if } E_1 = E_2 \textbf{ then } S]\!]\ s \rightarrow^+ s}$$

$$\frac{[\![E_1]\!]\ s \rightarrow \alpha_1 : n \quad [\![E_2]\!]\ s \rightarrow \alpha_2 : n}{[\![\ \textbf{while } E_1 = E_2\ \ \textbf{do } S]\!]\ s \rightarrow^+ [\![S;\ \textbf{while } E_1 = E_2\ \ \textbf{do } S]\!]\ s}$$

$$\frac{[\![E_1]\!]\ s \rightarrow \alpha_1 : n_1 \quad [\![E_2]\!]\ s \rightarrow \alpha_2 : n_2 \quad n_1 \neq n_2}{[\![\ \textbf{while } E_1 = E_2\ \ \textbf{do } S]\!]\ s \rightarrow^+ s}$$

Figure 5.6: $\rightarrow^+$ reduction relation

Lemma 15. *Each $\rightarrow^+$-reduction can be represented by one or more $\rightarrow$-reductions, i.e. $\rightarrow^+ \subseteq \rightarrow^*$.*

Proof. The conclusion of each $\rightarrow^+$-rule can be expanded into a sequence of $\rightarrow$ reductions assuming the premises of the rule holds. For brevity we only include the first **while**-rule where $[\![E_1]\!]\ s$ reduces to the same constant as $[\![E_2]\!]\ s$.

$$\begin{aligned} & [\![\ \textbf{while } E_1 = E_2\ \ \textbf{do } S]\!]\ s \\ = \ & (\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ s \end{aligned}$$

$\rightarrow_{\oplus} (\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s)\ s$
$\rightarrow_{\oplus} ([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s$
$\rightarrow^{*}_{\oplus} (\alpha_1 : n = \alpha_2 : n)\ ((\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s))\ s$
$\rightarrow_{\oplus} (\mathit{fix}\ (\lambda w.\lambda s.([\![E_1]\!]\ s = [\![E_2]\!]\ s)\ (w\ ([\![S]\!]\ s))\ s))\ ([\![S]\!]\ s)$
$= [\![S;\ \textbf{while}\ E_1 = E_2\ \ \textbf{do}\ S]\!]\ s$

□

Lemma 16. *If $\sigma \sim s$ we know that $\langle S, \sigma\rangle \rightarrow \langle S', \sigma'\rangle \Leftrightarrow [\![S]\!]\ s \rightarrow^{+} [\![S']\!]\ s'$ (or $\langle S, \sigma\rangle \rightarrow s' \Leftrightarrow [\![S]\!]\ s \rightarrow^{+} s'$) for some σ' and s' such that $\sigma' \sim s'$.*

Proof. By structural induction on S. The interesting case is the assignment statement, since it is the assignments that affect the store. (The other cases follow by the same reasoning as for the while-loop and the **while**-rule in Proof 15.)

We assume $\sigma \sim s$. By Lemma 14 we know that $\langle x_k := E, \sigma\rangle$ reduces to $\sigma[x_k \mapsto \alpha : n]$ if and only if $[\![x_k := E]\!]\ s = (\lambda s.\mathit{set}\ s\ k\ ([\![E]\!]\ s))\ s$ reduces to $\lambda l.(l = k)\ \alpha : n\ (s\ k)$ and since $\sigma \sim s$ we see that $\sigma[x_k \mapsto \alpha : n] \sim \lambda l.(l = k)\ \alpha : n\ (s\ k)$. □

Theorem 25. *For any While statement S, we have: $[\![S]\!]\ s_0 \rightarrow^{*}_{\oplus} \alpha : n$ if and only if $\langle S, \sigma_0\rangle \rightarrow_{\oplus} \alpha : n$*

Proof. By Lemma 15, the fact that $\rightarrow$ is deterministic and since $[\![S]\!]\ s$ always reduces to s' or $[\![S']\!]\ s'$ for some S' and s', we can write the reduction $[\![S]\!]\ s_0 \rightarrow^{*} \alpha : n$ on the form $[\![S]\!]\ s_0 \rightarrow^{+} [\![S_1]\!]\ s_1 \rightarrow^{+} \ldots \rightarrow^{+} [\![S_m]\!]\ s_m \rightarrow \alpha : n$ iff $\langle S, \sigma_0\rangle \rightarrow \alpha : n$. By Lemma 16 and the fact that $\sigma_0 \sim s_0$ we know that that $\sigma_m \sim s_m$. By case analysis we know that S_m must be a return statement and by Lemma 14 $[\![\textbf{return}\ E]\!]\ s_m$ reduces to the same term as $\langle \textbf{return E}, \sigma_m\rangle$. □

5.6.1 Supporting References

Apart from side-effects imperative languages commonly support references, i.e. values that point to other values. In such settings a few questions arises: Does it make sense to also let references have labels? If so, should two references referring to the same value be able to have different labels? When should the label of the reference be used, and when should the label of the value it refers to be used? The answer is that it depends on the situation. In some cases it does indeed make sense to have labels for references while in other cases it is more suitable to use the label of the referred value.

Assume for example that some processes share a set of files through file references provided by the system. Each process can ask the system either for read access or for write access to a file. The system makes sure that only one process can write to a file at any given point, while it allows for an arbitrary number of processes to read from a file. In this scenario it is the actual *handle* to the file that determines if it may be used for reading or writing. Thus the labels should be associated with the file references rather than the files directly.

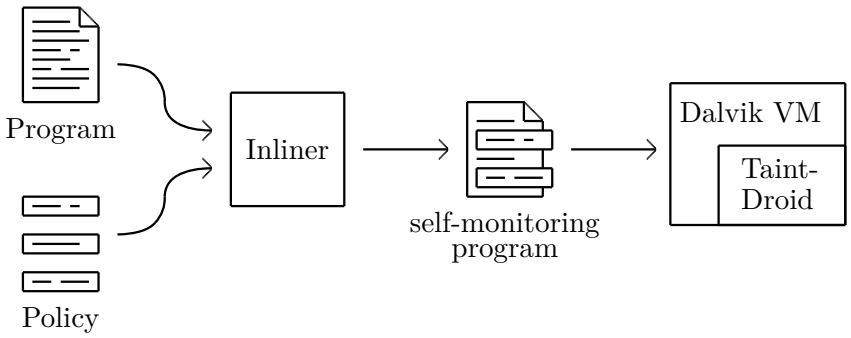


Figure 5.7: Overview of the implementation.

In an entirely different scenario you may want to express that a file may only be written to while being open, it makes little sense to associate labels with the file references, since after a file has been closed it may not be written to using any reference. In such situation it is more suitable to associate the label with the files.

5.7 Implementation

From a formal point of view, there is a large gap between the While-language and a real world language such as Java. Conceptually however, the semantics presented in the previous section outlines how the implementation of the labeling semantics works for Java. This section describes an implementation targeting Java (bytecode) and the Android platform which follows this outline. Implementing the framework involves solving two main tasks: tracking data flows and intercepting policy relevant function calls. In our implementation we solve the first task using *taint analysis* and the second task using *monitor inlining*. An overview of the framework is shown in Figure 5.7.

5.7.1 Tracking Data Flows using Taint Analysis

Taint analysis (also known as taint tracking) is a common technique for tracking data flows at runtime. The technique relies on (1) having points at which data is originally tainted (*taint sources*), (2) making sure that taints propagate along with every data flow (*taint propagation*) and (3) having points at which taints of output data is intercepted (*taint sinks*). In our work we rely on taint propagation for tracking data flows by letting taints represent labels. The notion of taint sources and sinks however are factored out and handled by the inlined monitor.

Taint analysis implementations targeting the JVM has been described by several authors [61, 127, 18, 38]. Our implementation uses the TaintDroid framework by Enck et al [38] which targets the Android Platform. TaintDroid is based on a modified version of the Dalvik VM which taints data coming from various privacy

related sources such as the GPS, camera, microphone etc. and monitors the taints of data being sent on the network.

Limitations due to Taint Analysis

In TaintDroid a taint is represented by a 32-bit word where each bit corresponds to one of the privacy related sources. If for instance the taint of a value v has bit 25 and bit 32 is set, then v contains data which potentially comes from the camera and GPS respectively. When data is copied from one location to another, the taint is copied along with it. If two values v_1 and v_2 are combined (added or concatenated for instance) the taint of the result is determined by the bitwise or of the taint of v_1 and the taint of v_2. While this approach makes sense when working with the type of privacy related policies which TaintDroid is intended for, it poses a limitation on what policies we are able to enforce in our framework. For a policy to be enforceable, when using TaintDroid as the underlying taint tracking mechanism, it must have the two properties described below.

Property 1. *If L is the set of labels in a policy $\mathcal{P}$, there need to exist an injective function F of type $\mathsf{L} \to 2^{\{1,\ldots,32\}}$ such that the range of F is closed under $\cup$.*

This property ensures that for any two labels L_1 and L_2 there exists a label L_3 such that $F(\mathsf{L}_1) \cup F(\mathsf{L}_2) = F(\mathsf{L}_3)$ or, put differently, the bitwise or-operation performed by TaintDroid always yields a taint representing a valid label.

The other property is regarding arithmetic operations. Such operations are encoded as external functions in the theory, but in practice do not correspond to observable actions.

Property 2. *Whenever an arithmetic operation is applied to two labeled constants $\mathsf{L}_1 : c_1$ and $\mathsf{L}_2 : c_2$ the policy must define the resulting label as $F^{-1}(F(\mathsf{L}_1) \cup F(\mathsf{L}_2))$ where F is the function described in Property 1.*

In terms of abstract algebra, Property 1 and 2 hold for a policy $\mathcal{P}$ iff there exists a monomorphism between the two algebras $(\mathsf{L}, \odot)$ and $(2^{\{1,\ldots,32\}}, \cup)$ where $\odot$ is the label operator for arithmetic operations induced by $\mathcal{P}$. Our implementation assumes that these properties hold for all policies given to the inliner, and does not have syntactical support for defining custom behavior for arithmetic operations.

Taint Propagation: Or vs And

Using bitwise *or* as taint propagation mechanism is suitable when handling confidentiality properties. When for example enforcing a policy such as *"do not send address book data on the network"*, a phone number should *maintain* its taint, even if it is manipulated. Conceptually however, our framework works just as well for integrity related properties, such as *"send text messages only to numbers from my address book"*. Enforcing such policies call for a bitwise *and* propagation mechanism since if a phone number is manipulated, it should *lose* its taint.

To support both types of policies, we split the taint words in two parts: bits 0-15 which are *or*:ed together when combined (referred to as *or*-flags), and bits 16-31 which are *and*:ed together when combined (referred to as *and*-flags). Rather than changing the actual propagation code in TaintDroid however, we simply changed the default taint from 32 zeros to 16 zeros followed by 16 ones and inverted the interpretation of the *and*-flags.

5.7.2 Intercepting Calls using Monitor Inlining

Whenever a program is about to call a function the monitor needs to check if the call is allowed by the policy or not. If it is not allowed the call should be prevented (for instance by terminating the execution) and if it is allowed the label of the result of the function call should be set according to the policy. This task could be handled by the VM. However, our implementation delegates the task to the program itself by inlining the monitor code into the program. This approach is known as monitor inlining [39] and has the advantage of not requiring extra support from the execution environment.

Inlining a monitor into `P` involves the following steps:

1. Parse a policy $\mathcal{P}$ given in the syntax specified in Definition 31.

2. Traverse `P`'s code and replace each call to a policy relevant function f_i with code that does the following

    a) Copy the arguments from the operand stack to local variables (as they may be needed after the call when evaluating the label expression in step 2e)

    b) Until a guard $guard_{ij}$ holds, evaluate the guards successively starting with $guard_{i1}$. If $guard_{ij}$ holds store j in a temporary variable x and go to 2d.

    c) Terminate the execution due to policy violation.

    d) Perform the original function call.

    e) Evaluate $expr_{1x}$ and assign the resulting label to the value returned by the function.

Steps 2b and 2e rely on code for accessing the labels of certain values. Since the labels are not accessible directly through bytecode instructions this requires interaction with TaintDroid. As TaintDroid was not originally designed to interact with client programs, a few minor modifications to TaintDroid were required (exposing some internal methods).

The inlining step is fully automatic and can conveniently be added to the compile chain, as it has been done by editing the build settings in the Eclipse IDE for example.

Limitations due to Client-Side Inlining

The general client-side inlining limitations have been explored previously [25, 24]. The main drawback in our setting is the lack of complete mediation [109] (function calls made internally by the runtime library are not observable).

The solution is to let the policy prevent internal computations from violating the policy, by restricting the calls performed by the client. For example, if `exec` is a policy relevant function, the policy must also restrict calls to functions that call `exec` internally, such as wrapper functions. This may require an over approximation of the intended policy.

Provided all internal policy violations are avoided by the above technique, as the resulting label would be overridden by the monitor when control returns to the client code regardless of the internal computations, the lack of complete mediation is irrelevant when calling functions which are explicitly mentioned in the policy. However, one cannot expect that a policy has a clause for each function in the Java API. To relate the behavior of our implementation to the theory of the framework, the semantics of calling a method not mentioned in the policy is considered to be the same as if the body of that method was recursively unfolded into the client code. In other words, the labels of values returned by internal function calls are determined solely by the rules for arithmetic operations (as described in Section 5.7.1).

5.7.3 Handling Impure Functions

For simplicity, the theoretical presentation of the framework is restricted to pure functions. As shown in Theorem 17, the value of an argument in such setting is fully determined by its FAT. For this reason the observable actions do not entail information regarding actual values of arguments. However, a language like Java depends heavily on impure functions. The case studies highlight the importance of being able to reason about argument values.

The modifications needed for proper handling of impure functions are however straightforward and do not affect the theorems presented in the paper. FATs (and thus observable actions) need to take argument values into account which is done by the following T-AppFun reduction:

$$f^n\ \tau_1 : c_1\ \ldots\ \tau_n : c_n \xrightarrow{f(\tau_1:c_1,\ldots,\tau_n:c_n)} f(\tau_1 : c_1, \ldots, \tau_n : c_n) : [\![f^n(c_1, \ldots, c_n)]\!]$$

Similarly $\oplus$ needs to operate on labeled constants instead of just labels and the L-AppFun needs to be written as

$$\frac{\gamma = \mathsf{L}_f \oplus \alpha_1 : c_1 \oplus \cdots \oplus \alpha_n : c_n}{f^n\ \alpha_1 : c_1\ \ldots\ \alpha_n : c_n \to \gamma : [\![f^n(c_1, \ldots, c_n)]\!]}$$

These modifications allow the policy guards and expressions to refer to the argument values in addition to the argument labels. Modifications to the definitions of the derived policy automaton and labeling scheme are straightforward.

5.8 Case Studies

This section evaluates the approach and the implementation in five case studies with varying characteristics. The webpage [89] contains full details including concrete policies.

5.8.1 Case Study 1: DroidLocator

Just as the popular application *Find My Phone* for iPhone, the *DroidLocator* application allows the user to locate a lost or stolen Android device through a web service. As opposed to Find My Phone and other similar services however, DroidLocator prevents server administrators and third parties from using the location data maliciously. It does so by *encrypting* the location data, based on a user-provided key, before uploading it to the server. When the user later retrieves the encrypted location data, he or she can decrypt it without revealing the location to anyone else.

Application

DroidLocator is a small application written by one of the paper's authors. It retrieves the location data from the GPS hardware, uses the `javax.crypto` package to encrypt it with a key retrieved through `EditText.getText`, and submits it to the server using the standard socket API.

Policy

The desired policy states that (A) the location may not be sent over the network unless it is encrypted and that (B) the encryption key needs to be provided by the user (retrieved through `EditText.getText` on an object with no prior calls to `EditText.setText`). Figure 5.8 shows the policy expressed in terms of the syntax in Definition 31. The formal semantics of this policy is provided by the STA, $\mathcal{A}_{DL}$, obtained from Figure 5.8 by Definition 32. Examples of FATs accepted and rejected by $\mathcal{A}_{DL}$ are found in Figure 5.9.

The labeling scheme used at runtime is obtained by following Definition 33 and adapting it to TreeDroid as described in Section 5.7. The resulting set of labels L is: $\{0^{16}1^{16}$ (L_0), $10^{15}1^{16}$ (sock), $010^{14}1^{16}$ (conf), $0010^{13}1^{16}$ (nonuser), $0^{17}1^{15}$ (userenc), $0^{16}101^{14}$ (userinp)$\}$, and the bitwise-or operator is used for $\oplus$.

The label specifying that data contains location information is encoded using an *or*-flag and the label specifying that data is encrypted is encoded using an *and*-flag.

Results

The behavior was unaffected by monitor inlining since the original application adhered to the policy. When the code was changed to use as encryption key a predefined string literal (such as, in Figure 5.9, the empty string), the execution was terminated before the location was uploaded.

$\{\mathsf{L}_0, \mathsf{sock}, \mathsf{conf}, \mathsf{userinp}, \mathsf{userenc}, \mathsf{nonuser}\}$,
$\{$`LocationManager.`
`getLastKnownLocation`(α): $true \to \mathsf{conf}$
`Location.toString`(α_{loc}): $true \to \alpha_{loc}$
`EditText.setText`$(\alpha_{et}, \alpha_{text})$: $true \to$ `nonuser`
`EditText.getText`(α_{et}): $\alpha_{et} \neq \mathsf{nonuser} \to \mathsf{userinp}$
$true \to \mathsf{L}_0$
`Editable.toString`(α): $true \to \alpha$
`SimpleCrypto.getRawKey`(α): $true \to \alpha$
`SimpleCrypto.encrypt`(α_k, α_m): $\alpha_k = \mathsf{user} \to \mathsf{userenc}$
$true \to \alpha_m$
`Socket.getOutputStream`(α): $true \to \mathsf{sock}$
`OutputStream.write`$(\alpha_{out}, \alpha_{val})$: $\alpha_{val} = \mathsf{L}_0 \to \mathsf{L}_0$
$\alpha_{val} = \mathsf{userenc} \to \mathsf{L}_0$
$\alpha_{out} \neq \mathsf{sock} \to \mathsf{L}_0\}$

Figure 5.8: Policy for DroidLocator case study.

5.8.2 Case Study 2: Sms2Group

An application, *Sms2Group*, requiring the SEND_SMS permission, allowing users to send SMS-messages to groups of contacts, is studied. The policy in this study restricts which numbers messages may be sent to.

Application

Sms2Group has been developed for the purpose of this study. The application allows the user to automate the task of sending text messages to a group of contacts. It relies on the group attribute in the contact book, fetched using the content provider API and uses the ordinary `SmsManager.sendTextMessage` method to send SMSes.

Policy

Messages are prevented from being sent to arbitrary numbers by ensuring that destination numbers (first argument of `sendTextMessage`) originate from the local address book. The label specifying that a value is a valid destination number is encoded using an *and*-flag which prevents attackers from using a modified address book number. This general policy naturally separates legitimate executions from malicious ones. Using traditional inlining techniques this type of policy would be expressed using a guard that scans the address book and checks that the destination number is present. There are two conceptual differences between these approaches. As opposed to a policy that relies on scanning the address book, our policy expresses that an SMS may not be sent to numbers with arbitrary origin *even* if the number is present in the address book. In this sense our policy is stricter. Another difference

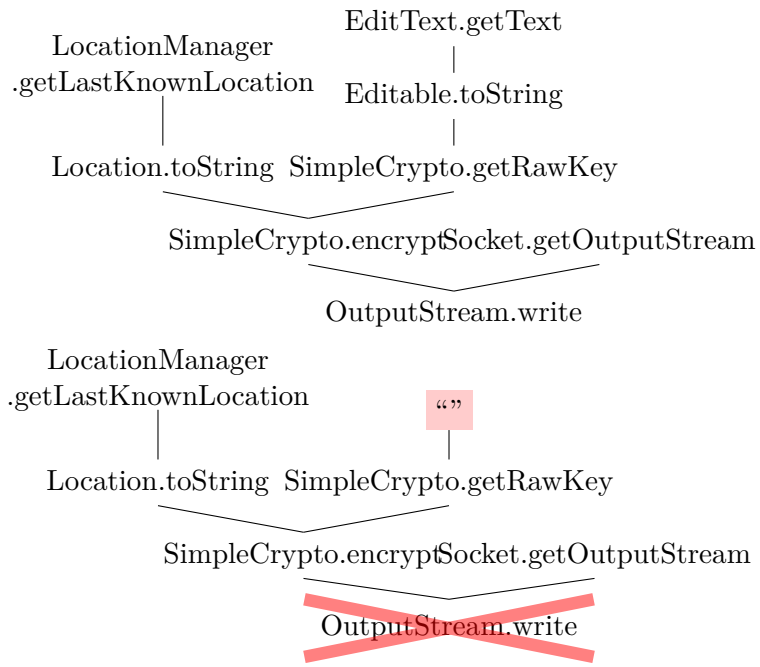


Figure 5.9: Accepted (top) and rejected (bottom) FATs for the DroidLocator policy.

is that a scan of the address book is typically a linear operation, whereas checking the taint of a value is a constant time operation.

Results

The inlining did not affect the functionality of the original program as it adheres to the policy. When changing the code so that the program attempts to send an SMS to a hard-coded number or a number from the address book concatenated with an arbitrary string the policy is violated and the program is terminated as expected.

5.8.3 Case Study 3: Bankdroid

This case study examines an internet banking application, *Bankdroid*, which allows users to review account information from several different banks. The application has many security concerns as the information it handles (balances, recent transactions, etc) is usually considered confidential. The main objective of the case study is to demonstrate how standard security policies can be applied *transparently on real world honest applications*, while still blocking dishonest variants of the same applications.

Application

Bankdroid (40k lines of code) is distributed through Google Play and is currently installed on 100.000+ devices [58]. It uses the Apache HttpClient library to communicate with the banks. To allow the policy to be expressed at the level of sockets (instead of at the level of the Apache HttpClient API), the library has been included in the client code base which adds another 60k lines of code.

Policy

The policy is a Chinese-Wall like policy which states that data received from host A may be sent back to host A but not to some other host B. As mentioned in the above paragraph, the policy is expressed at the level of sockets which makes it general and applicable to many other applications requiring Internet access.

Examples of accepted and rejected FATs is found in Figure 5.10.

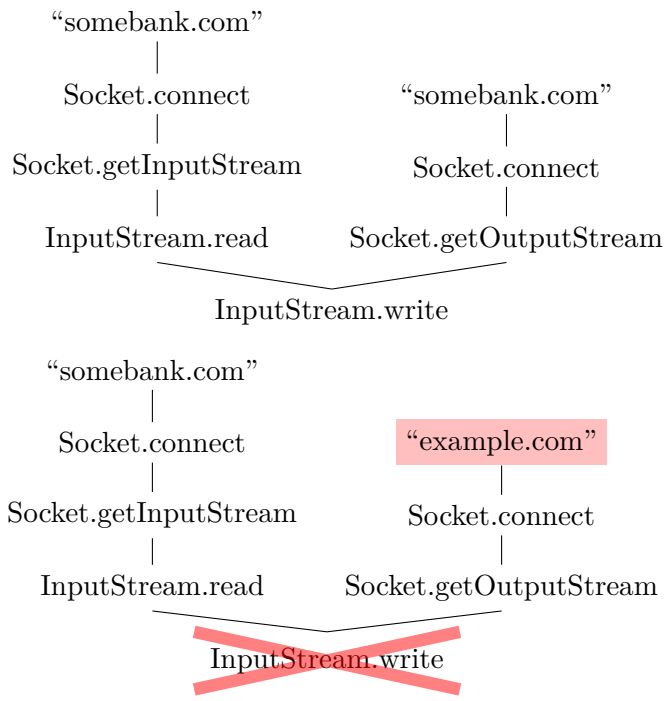


Figure 5.10: Accepted (top) and rejected (bottom) FATs for the Bankdroid policy.

Results

The application was modified to leak the current balance of each bank account to a host controlled by a potential attacker. The policy was then inlined in the modified application. When the leak was about to take place, the inlined code successfully terminated the execution.

5.8.4 Case Study 4: Auto Birthday SMS

Auto Birthday SMS is a application distributed on Google Play. It has over 10,000 installs [57] and allows the user to automatically send SMS-messages to friends on their birthdays. It is free of charge but displays ads which are retrieved over the network. It requires the INTERNET and SEND_SMS permissions. Applications requiring this combination of permissions are interesting to study since trojans sending premium-rate SMS messages are relatively common [46] and could potentially transform the phone into an SMS spamming bot. As demonstrated in this case study, TreeDroid is useful even for honest coders in order to harden their applications by inlining generic security policies.

Application

Application data, including numbers to send messages to, are stored in a SQLite database. The code turns out to be vulnerable to SQL-injection attacks which can be exploited by any application with permission to modify the address book data. The code calls `SQLiteDatabase.execSQL`, which updates the database, with an unsanitized query containing the name of a contact. The contact name should be sanitized by `DatabaseUtil.sqlEscapeString` before running the query.

Policy

The policy applied is a general sanitize-before-query policy stating that a query passed to `execSQL` must be a string literal, a result of `sqlEscapeString` or a concatenation of such strings. The label used for sanitized values is encoded using an *and*-flag to ensure that the concatenations of sanitized and unsanitized strings are considered sanitized. An example of an accepted FAT is found in Figure 5.11. Omitting the call to `sqlEscapeString` would result in a tree which would be rejected by the policy.

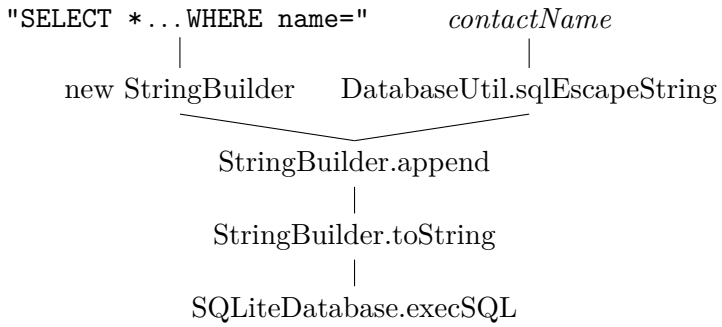


Figure 5.11: Accepted FAT for the SQL policy.

Results

The inlined code prevents the application from performing queries containing unsanitized arguments, such as raw contact names, in the SQL statements. The original application violates the policy upon certain user actions, in such cases execution is successfully terminated by the monitor.

5.8.5 Case Study 5: Lovetrap

Lovetrap is a real world SMS-trojan detected by Symantec in July 2011 [124]. Among other bad behaviors, it sends premium rate SMS messages (which is the focus of this study). This case study demonstrates the efficiency of TreeDroid on real world attacks.

Application

Lovetrap, which looks like a regular game, starts a service which downloads a list of numbers and messages which it repeatedly tries to send by SMS.

Policy

The policy from case study 1 is reused without modification, which is an indication of the policy genericness.

Results

By locally redirecting requests going to the host of the attacker to our own server, we managed to supervise the actions of the trojan. The monitor inlining at the bytecode level proceeds as expected without special tweaking. After inlining, the trojans service is terminated immediately and therefore no longer able to send SMS messages as intended.

5.8.6 Statistics

Case studies statistics have been collected in Table 5.12. For applications where we have access to the source code, business logic execution time has been measured. In Bankdroid we measured the time it takes to update the accounts, for DroidLocator we measured the time it takes to encrypt and upload the location, and for Sms2Group we measured the time it takes to collect the group information and send the SMS. Taint tracking runtime overhead has been estimated by TaintDroids authors to about 14 % on a Google Nexus One [38]. Our measurements (significantly higher, as expected since they are performed using Dalvik in debug mode on an Android emulator) are included for comparison with the runtime overhead due to the inlined code. The bytecode size overhead in the Auto Birthday SMS study is due to the fact that the relatively common operation of concatenating strings is considered policy relevant.

	DroidLocator	Sms2Group	Bankdroid+HttpClient	Auto Birthday SMS	Lovetrap
Lines of Java source code:	330	240	101079	N/A	N/A
Size bytecode before inlining:	16.8 kB	17.0 kB	2.6 MB	193.9 kB	55.0 kB
Size increase due to inlining:	24.9 %	35.2 %	0.395 %	43.2 %	5.30 %
Inlining duration:	178 ms	190 ms	2740 ms	870 ms	210 ms
Policy relevant method calls:	11	15	213	359	1
Number of policy clauses:	9	4	14	5	4
Average total execution time:	142 ms	884 ms	9780 ms	N/A	N/A
Overhead due to TaintDroid:	38.9 %	53.3 %	28.0 %	N/A	N/A
Overhead due to inlined code:	45.4 %	16.9 %	19.5 %	N/A	N/A
Downloads on Google Play:	N/A	N/A	>100,000	>10,000	N/A

Figure 5.12: Statistics from the case studies.

5.9 Conclusions and Future Work

The paper presents a new monitoring technique using tree automata to track and enforce data processing constraints in a novel way. Many security properties, which were either difficult or impossible to express using existing techniques, can be treated. The approach is theoretically well-founded and practical as demonstrated by the various case studies.

Usability could be further increased by using techniques that give a formal semantics to textual policies [93, 126]. It would allow application authors to provide usage description of required permissions (which is a recommended good practice) that are both user-readable and from which enforceable formal policies could be extracted.

The focus on direct flows that can be tracked by taint analysis is not due to a fundamental limitation of the approach. A possible direction for future work would be to extend the program model, notion of observable actions and policy semantics to support indirect flows (decisions influencing data processing). It should be noted, however, that no practical approaches currently exists that can provide a comprehensive protection against covert flows anyway, and so it is far from clear that the added quality of protection offered by such an extension really motivates the additional complexity and runtime overhead.

Another direction for future work is to extend the implementation to support complete mediation. This could be done by either (a) allowing the inliner to rewrite relevant parts of the runtime library or by (b) solving the task of monitoring function calls using some other technique than inlining. Since our implementation separates policy and mechanism (TaintDroid is unaware of the policy being enforced, and the inliner works independently of the underlying data tracking mechanism) it is

flexible enough to be extended in either way.

Concurrency poses no problems if the order of policy relevant actions does not matter (as for local policies) since each thread can be monitored in isolation. For non-local policies, however, where the order of the actions does matter, the monitor has to synchronize the threads to exclude schedulings that yield illegal executions. This requires inlined code to be executed atomically together with policy relevant actions. This is problematic for a client-side inliner due to the fact that there is no way to acquire a lock before calling a method, and releasing it immediately when control has passed into the API method. The solution is either to release the lock after the policy relevant method has completed, i.e. use a *blocking inliner* [25] or restrict attention to so-called *race free* policies [22].

Leveraging tree automata theory allows for reuse of existing algorithms such as automata containment and minimization. Exploring these techniques further is left as future work.

Finally, our approach could very well be used in conjunction with existing control-flow bound techniques which would allow policies to express properties such as "*if the authenticate method returned true for credentials that has been provided by the user, queries do not have to be sanitized*". Some techniques for linear monitoring can also naturally be applied directly to tree based monitoring. Translating the idea of using edit automata instead of ordinary word automata into the context of tree automata would for instance allow us to express policies such as "*whenever an unsanitized query is about to be evaluated, sanitize it first*".

5.9.1 Acknowledgments

We would like to express our appreciation to Tomas Andréasson for his work on the inliner tool and assistance in carrying out the case studies.

Chapter 6

Concluding Remarks

The following research topics have been addressed in this thesis:

- How to support monitor inlining as a security mechanism without including an inliner in the trusted computing base.
- How to correctly (securely, transparently and conservatively) implement monitor inlining for multhithreaded programs.
- What type of policies can be enforced correctly in a multithreaded setting without locking across security relevant actions.
- How to lift the IRM certification to multithreaded applications.
- How to express and enforce policies that constrain more complex API-protocols and data dependencies.

6.1 Summary of Results

As shown in the paper presented in Chapter 2, it is possible to use proof-carrying code to certify that a correct monitor inlining has been performed. This allows us to reduce the TCB by replacing the inliner with a relatively simple proof checker. The approach is proven sound and shown to work in practice by means of a prototype implementation and evaluation in two case studies.

How to correctly implement reference monitor inlining as a security enforcement mechanism in a multithreaded program is answered in the paper presented in Chapter 3. This paper presents a secure but somewhat crude approach to inlining based on a serialization of all security relevant actions (blocking inlining). This approach is not fully transparent, but without assuming certain properties of the given policies and/or programs, it is not possible to solve the problem in general. To characterize what is to be considered correct inlining of policies that are not possible to enforce transparently, the paper proposes the notion of strong conservativity (and a few variations of this property) and shows that the presented inlining algorithm is strongly conservative.

A prototype based on the presented inlining scheme was developed and the practicality and effectiveness evaluated in four case studies.

The paper presented in Chapter 4 addresses the question of what policies can be enforced without serializing all security relevant actions. This class of policies is referred to as race-free policies and include all policies that do not require that a call takes place before any other event, or any return takes place after another event. Policies that fall outside of this class are shown not to be inlineable, and for the policies that are race-free an example inliner is provided shown to be correct. The results imply that the class of race-free policies is the maximal set of inlineable policies. The paper also presents the results from a new set of case studies based on web applications and Swing-based GUI applications (which are inherently multithreaded).

Furthermore the paper shows how IRM certification can be lifted to multithreaded applications. This is done by including, in the certificate, information regarding exactly where the security lock is being acquired and released. Once this is established, the proof checker can basically fall back on checking the inlined instructions sequentially.

Finally, the paper presented in Chapter 5 addresses the question of how to enforce data-centric policies that express constraints involving data-dependencies. It does so by presenting the concept of tree-based monitoring. In tree-based monitoring the observable actions entails, apart from the method name and argument values, the history of observable actions involved in the computations of the argument values. Constraints are expressed in terms of what arguments can be applied to which methods using a tree automata based policy model. A description of how the idea can be effectively realized through an abstraction of the function application trees is presented, and implemented in a prototype targeting the Android platform. The prototype is successfully evaluated in five case studies which includes real world applications and malware.

6.2 Future Work

Before the techniques presented in this thesis gets a wider adoption a few issues needs to be addressed.

Tools and Integration

The implementations described in this thesis are prototypes used to evaluate the theoretical results and to demonstrate a proof of concept. The tools are not as polished as one may wish and lacks adequate documentation. For a wider adoption of the techniques, this would need to be addressed. An Eclipse plugin which provides a user friendly policy editor and adds the inlining step to the compile chain would be a step in the right direction.

Furthermore the techniques needs some form of device integration. Adding an inlining mechanism (or proof-checker) to the application manager, such as the

AppStore on iOS or the Google Play application in Android would be one user friendly solution.

IRM Optimizations

The naive implementations presented in this thesis put no effort in trying to optimize the IRM. This means that programs that call security relevant methods in a policy adherent way still suffers from the code-size and runtime overheads induced by the IRM. This is something that could be addressed in many practical situations by an optimizing inliner. Even if the inliner cannot eliminate checks all together, it may be able to move snippets of inlined code out of loops etc. This is a venue for future work, closely related to compiler research.

Policy Containment

In the scenario presented in the first paper the application contract is inlined by the developer. When the application is downloaded or installed by the end user, the TCB needs to verify that the application contract is indeed compatible with the device policy. This process involves a DFA containment check, which is tractable only for DFAs of limited sizes. This issue has been discussed and partially addressed in [12], but requires further research and better tool support before the technique as a whole can be widely adopted.

Monitoring API

In the work presented in this thesis, an inliner is regarded as non-transparent and/or non-conservative (i.e. “incorrect”) if it affects the policy-adherent executions of the target program. This is a good starting point and base line for research on security enforcement through monitor inlining. From a practical point of view, this is not always a desirable property, since it rules out all interaction between the client program and the security mechanism. In a practical setting it would for instance be more user friendly if the application could query the IRM for its current state and dynamically adapt to the device policy.

Bibliography

[1] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. *The Journal of Logic and Algebraic Programming*, 78(5):304 – 339, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[2] I. Aktug and K. Naliuka. ConSpec – A formal language for policy specification. *Science of Computer Programming*, 74(1-2):2 – 12, 2008. Special Issue on Security and Trust.

[3] J. P. Anderson. Computer security technology planning study. Technical report, Deputy for Command and Management System, USA, 1972.

[4] F. Y. Bannwart and P. Müller. A logic for bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141-1 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[5] G. Barthe, P. Crégut, B. Grégoire, T. P. Jensen, and D. Pichardie. The MOBIUS proof carrying code infrastructure. In *FMCO*, pages 1–24, 2007.

[6] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security—Theories and Systems. Mext-NSF-JSPS International Symposium*, volume 2609 of *Lecture Notes in Computer Science*, pages 154–171. Springer, 2003.

[7] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. *SIGPLAN Not.*, 40(6):305–314, June 2005.

[8] D. E. Bell. Secure computer systems: A refinement of the mathematical model. Technical report, MITRE Corp., 04 1974.

[9] D. E. Bell. Looking back at the Bell-LaPadula model. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 337–351, Washington, DC, USA, 2005. IEEE Computer Society.

[10] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[11] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997, The MITRE Corp., July 1975.

[12] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming*, 78(5):340 – 358, 2009.

[13] K. Bierhoff and J. Aldrich. PLURAL: Checking protocol compliance under aliasing. In *Companion of the intl. Conf. on Software engineering*, ICSE Companion '08, pages 971–972, New York, NY, USA, 2008. ACM.

[14] K. Bierhoff, N. Beckman, and J. Aldrich. Practical API protocol checking with access permissions. In *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *LNCS*, pages 195–219. Springer Berlin / Heidelberg, 2009.

[15] A. Bose and K. G. Shin. On mobile viruses exploiting messaging and bluetooth services. In *Securecomm and Workshops, 2006*, pages 1 –10, 28 2006-sept. 1 2006.

[16] R. Bubel, R. Carbon, N. Diakov, I. Schaefer, J. Schäfer, B. Weitzel, Y. Welsch, and P. Wong. Evaluation of the core framework, deliverable 5.2 of project fp7-231620 (HATS). `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable52.pdf`, August 2010.

[17] F. Chen. Java-MOP: A monitoring oriented programming environment for Java. In *In Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550. Springer, 2005.

[18] E. Chin and D. Wagner. Efficient character-level taint tracking for Java. In *Proc. work. Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[19] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *CSF*, pages 200–214, 2010.

[20] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[21] F. J. Corbató. On building systems that will fail. *Commun. ACM*, 34(9):72–81, Sept. 1991.

[22] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining and certification for multithreaded Java. To appear in Mathematical Structures in Computer Science.

[23] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings*, pages 546–569. Springer-Verlag, 2009.

[24] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Security monitor inlining for multithreaded Java. In *European Conf. of Object-Oriented Computing*, pages 546–569. Springer-Verlag, July 2009.

[25] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 18(1):37–59, 2010.

[26] M. Dam, B. Jacobs, A. Lundblad, and F. Piessens. Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 18:37 – 59, 2010.

[27] M. Dam, G. Le Guernic, and A. Lundblad. TreeDroid: A tree automaton based approach to enforcing data processing policies. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 894–905, New York, NY, USA, 2012. ACM.

[28] M. Dam and A. Lundblad. A proof carrying code framework for inlined reference monitors in Java bytecode. *CoRR*, abs/1012.2995, 2010.

[29] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM.

[30] R. Deline and M. Fahndrich. Typestates for objects. In *European Conf. of Object-Oriented Computing*, volume 3086 of *LNCS*, 2004.

[31] Department of Defense. *Department of Defense Trusted Computer System Evaluation Criteria*, dod 5200.28-std (the orange book) edition, December 1985.

[32] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens, and D. Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *CSAW*, pages 19–28, 2007.

[33] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, 2008.

[34] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .NET platform. *Journal Information Security Tech. Report*, 13(1):25–32, 2008.

[35] P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. *Information Processing 71, Proc. IFIP Congress*, 1:320–326, 1971.

[36] N. Dragoni and F. Massacci. Security-by-contract for web services. In *SWS*, pages 90–98, 2007.

[37] N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *EuroPKI*, pages 297–312, 2007.

[38] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. conf. Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[39] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, School of Computer Science, Reykjavík University, Ithaca, NY, USA, 2004. AAI3114521.

[40] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Dep. of Computer Science, Cornell University, 2004.

[41] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, Washington, DC, USA, 2000. IEEE Computer Society.

[42] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, page 0246. IEEE Computer Society, 2000.

[43] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. Workshop on New Security Paradigms (NSPW '99)*, pages 87–95. ACM Press, 2000.

[44] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[45] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *In proceedings of the national computer conference*, pages 329–334. AFIPS Press, 1979.

[46] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. work. Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[47] Ford Aerospace. Secure minicomputer operating system (KSOS) executive summary: Design of the Department of Defense Kernelized Secure Operating System. Technical report, Ford Aerospace and Communications Corporation, 1978.

[48] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, 2002.

[49] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *Transactions on Programming Languages and Systems*, pages 307–318. ACM Press, 2001.

[50] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617 (Draft Standard), June 1999.

[51] S. Furnell. Handheld hazards: The rise of malware on mobile devices. *Computer Fraud & Security*, 2005(5):4 – 8, 2005.

[52] M. Gandhe, G. Venkatesh, and A. Sanyal. Labeled lambda-calculus and a generalized notion of strictness (an extended abstract). In *Proc., Concurrency and Knowledge*, ACSC '95, pages 103–110, London, UK, 1995. Springer-Verlag.

[53] Gartner, Inc. Gartner says sales of mobile devices in second quarter of 2011 grew 16.5 percent year-on-year; smartphone sales grew 74 percent. `http://www.gartner.com/it/page.jsp?id=1764714`. Accessed February 17, 2012.

[54] S. Ghoshal, S. Manimaran, G. Roşu, T. F. Şerbănuţă, and G. Ştefănescu. Monitoring IVHM systems using a monitor-oriented programming framework. In *The Sixth NASA Langley Formal Methods Workshop (LFM 2008)*, 2008.

[55] D. Goldberg and L. Larsson. *Svenska hackare: En berättelse från nätets skuggsida.* Norstedt, 2011.

[56] D. Gollmann. *Computer Security.* Wiley, 2011.

[57] Google, Inc. Google Play: Auto Birthday SMS. `https://play.google.com/store/apps/details?id=es4b.apps.birthday`. Accessed April 23, 2012.

[58] Google, Inc. Google Play: Bankdroid. `https://play.google.com/store/apps/details?id=com.liato.bankdroid`. Accessed April 23, 2012.

[59] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification, 3rd Edition.* Addison-Wesley Professional, 2005.

[60] P. Gray. The hack of the year. `http://www.smh.com.au/news/security/the-hack-of-the-year/2007/11/12/1194766589522.html`, November 2007.

[61] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proc. Annual Computer Security Applications Conf.*, pages 303–311, 2005.

[62] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2008. ACM.

[63] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proc. work. Programming Languages and Analysis for Security*, pages 11–20, Tucson, Arizona, June 2008.

[64] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. work. Programming languages and analysis for security*, PLAS '06, pages 7–16, New York, NY, USA, 2006. ACM.

[65] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS*, pages 7–16, 2006.

[66] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[67] K. Havelund and G. Rosu. Monitoring programs using rewriting. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, Washington, DC, USA, 2001. IEEE Computer Society.

[68] S. Hudson. Java CUP. `http://www2.cs.tum.edu/projects/cup/`, March 2003.

[69] M. Hypponen. Malware goes mobile. *SCIENTIFIC AMERICAN*, Nov. 2006.

[70] Infoniac.com. List of computer viruses developed in 1980s. `http://www.infoniac.com/hi-tech/list-of-computer-viruses-developed-in-1980s.html`, November 2012.

[71] T. Jensen, D. Le Metayer, and T. Thorn. Verification of control flow based security properties. In *Security and privacy, 1999. Proceedings of the 1999 IEEE symposium on*, pages 89 –103, 1999.

[72] M. Jones and K. W. Hamlen. Disambiguating aspect-oriented security policies. In *AOSD*, pages 193–204, 2010.

[73] P. A. Karger, U. Roger, and R. Schell. Multics security evaluation: Vulnerability analysis. Technical report, HQ Electronic Systems Division: Hanscom AFB, MA. URL: http://csrc.nist.gov/publications/history/karg74.pdf, 1974.

[74] P. A. Karger and R. R. Schell. *Multics Security Evaluation: Vulnerability Analysis*. AD-A001. Deputy for Command and Management Systems (MCI), Electronic Systems Division, 1974.

[75] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: A run-time assurance tool for Java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218 – 235, 2001. RV'2001, Runtime Verification (in connection with CAV '01).

[76] G. Klein. JFLEX. `http://jflex.de/`, October 2007.

[77] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[78] G. T. Leavens and Y. Cheon. Design by contract with JML. `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`, 2006.

[79] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML reference manual, 2008.

[80] X. Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reasoning*, 30(3-4):235–269, 2003.

[81] J.-J. Lévy. *Réductions correctes et optimales dans le lambda calcul*. PhD thesis, Paris 7, 1978.

[82] J. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.

[83] J. Ligatti, L. Bauer, and D. Walker. Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec.*, 4(1-2):2–16, 2005.

[84] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[85] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proc. of the European Symposium on Research in Computer Security*, Sept. 2010.

[86] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[87] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18:717–721, December 1975.

[88] A. Lundblad. Inlined reference monitors for multithreaded java: Case-studies. `http://www.csc.kth.se/~landreas/mt_inlining/`, 2010.

[89] A. Lundblad and T. Andréasson. TreeDroid: Tree automaton based policy inlining. `https://sites.google.com/site/treedroidcasestudies`. Accessed May 4, 2012.

[90] F. Massacci and I. Siahaan. Simulating midlet's security claims with automata modulo theory. In *PLAS*, pages 1–9, 2008.

[91] E. J. McCauley and P. J. Brongowski. KSOS - the design of a secure operating system. *Managing Requirements Knowledge, International Workshop on*, 0:345, 1979.

[92] P. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient monitoring of parametric context-free patterns. *Journal of Automated Software Engineering*, 17(2):149–180, June 2010.

[93] S. M. Montazeri, N. Roy, and G. Schneider. From contracts in structured english to CL specifications. In *Proc. Work. Formal Languages and Analysis of Contract-Oriented Software*, volume 68 of *EPTCS*, pages 55–69, Málaga, Spain, Sept 2011.

[94] K. N. N. Dragoni, F. Massacci and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. 4th European PKI Workshop*, volume 4582 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2007.

[95] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.

[96] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. *Sigops Oper. Syst. Rev.*, 30(si):229–243, Oct. 1996.

[97] P. Neumann, R. Boyer, R. Feiertag, K. Levitt, and L. Robinson. *A provably secure operating system: The system, its applications, and proofs. Final report.* SRI project. Stanford Research Institute, 1977.

[98] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. A provably secure operating system. Technical report, Stanford Research Institute Menlo Park, June 1975.

[99] ObjectWeb. ASM web page. `http://asm.objectweb.org/`, February 2008.

[100] OWASP. OWASP top 10 - 2010. `http://owasptop10.googlecode.com/files/OWASPTop10-2010.pdf`, 2010.

[101] B. Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian\_signed\_spyware`, May 2007.

[102] S. Reynal. JPicEdt website. `http://jpicedt.sourceforge.net/`, 2010.

[103] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[104] G. Roşu and S. Bensalem. Allen linear (interval) temporal logic –translation to LTL and monitor synthesis–. In *Proc. intl. conf. Computer Aided Verification*, volume 4144 of *LNCS*, pages 263–277. Springer, 2006.

[105] S^{3}MS. Security of software and services for mobile systems. `http://www.s3ms.org/`, 2007.

[106] Project web page. `http://www.s3ms.org`, 2008.

[107] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17(7):388–402, 1974.

[108] J. H. Saltzer. Protection and the control of information sharing in Multics. *Commun. ACM*, 17(7):388–402, July 1974.

[109] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[110] M. Schaefer, B. Gold, R. Linde, and J. Scheid. Program confinement in KVM/370. In *Proceedings of the 1977 annual conference*, ACM '77, pages 404–410, New York, NY, USA, 1977. ACM.

[111] R. R. Schell, P. J. Downey, and G. J. Popek. *Preliminary Notes on the Design of Secure Military Computer Systems.* Defense Technical Information Center, 1973.

[112] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[113] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, Feb. 2000.

[114] Securelist.com. Macro virus. `http://www.securelist.com/en/glossary?glossid=189267795`, November 2012.

[115] K. Sen, G. Rosu, and G. Agha. Generating optimal linear temporal logic monitors by coinduction. In *Proc. Asian Computing Science Conference*, pages 260–275. Springer-Verlag, 2004.

[116] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 418 – 427, may 2004.

[117] C. Skalka and S. F. Smith. History effects and verification. In *APLAS*, pages 107–128, 2004.

[118] M. Sridhar and K. W. Hamlen. Actionscript in-lined reference monitoring in Prolog. In *PADL*, pages 149–151, 2010.

[119] M. Sridhar and K. W. Hamlen. Model checking in-lined reference monitors. In *Verification, Model Checking, and Abstract Interpretation*, pages 312–327, 2010.

[120] M. Sridhar and K. W. Hamlen. Flexible in-lined reference monitor certification: Challenges and future directions. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 55–60, New York, NY, USA, 2011. ACM.

[121] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, Jan. 1986.

[122] H. Sutter. A fundamental turn toward concurrency in software. `http://www.drdobbs.com/web-development/a-fundamental-turn-toward-concurrency-in/184405990?pgno=1`, March 2005.

[123] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, Sept. 2005.

[124] Symantec Corporation. Android.Lovetrap. `http://www.symantec.com/security_response/writeup.jsp?docid=2011-072806-2905-99`. Accessed May 3, 2012.

[125] The Tor Project. Anonymity online. `https://www.torproject.org/`, December 2012.

[126] R. Thion and D. Le Métayer. FLAVOR: A formal language for a posteriori verification of legal rules. In *Proc. Symp. Policies for Distributed Systems and Networks*, pages 1–8. IEEE Computer Society, June 2011.

[127] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *Proc. Programming language design and implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[128] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *FMOODS*, pages 240–258, 2008.

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

[130] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *WOOT*, pages 81–90, 2011.

[131] M. Viswanathan. *Foundations for the run-time analysis of software systems.* PhD thesis, University of Pennsylvania, 2000.

[132] T. V. Vleck. How the Air Force cracked Multics security. `http://www.multicians.org/security.html`, November 2012.

[133] D. Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM.

[134] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Trans. Softw. Eng. Methodol.*, 9(4):341–378, Oct. 2000.

[135] Ware, W.H. and United States. Defense Science Board. Task Force on Computer Security and Rand Corporation and United States. Dept. of Defense. *Security Controls for Computer Systems: Report of Defense Science Board, Task Force on Computer Security.* R (Rand Corporation). Rand, 1979.

[136] J. Whitmore. Design for Multics security enhancements. Technical report, Honeywell Information Systems, Inc., 1973.

[137] J. Whitmore, A. Bensoussan, P. Green, D. Hunt, A. Kobziar, and H. I. S. I. C. MA. *Design for Multics Security Enhancements.* Defense Technical Information Center, 1973.

[138] M. Yamamura. Palm.Liberty.A technical details, Symantec. `http://www.symantec.com/security_response/writeup.jsp?docid=2000-121918-3730-99&tabid=2`, November 2012.

[139] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. *Security and Privacy, IEEE Symposium on*, 0:79–93, 2009.

[140] X. Zhang, M. Leucker, and W. Dong. Runtime verification with predictive semantics. In A. Goodloe and S. Person, editors, *NASA Formal Methods*, volume 7226 of *Lecture Notes in Computer Science*, pages 418–432. Springer Berlin Heidelberg, 2012.

[141] C. Zhao, W. Dong, M. Leucker, and Z. Qi. Security goals assurance based on software active monitoring. In *Secure Software Integration and Reliability Improvement (SSIRI), 2011 Fifth International Conference on*, pages 70 –79, June 2011.