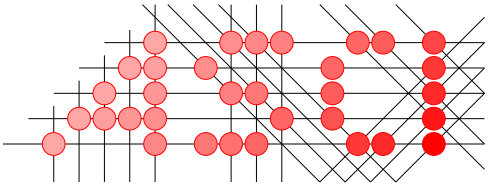
REMOTE POLICY ENFORCEMENT USING JAVA VIRTUAL MACHINE

SRIJITH KRISHNAN NAIR



Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 189.

VRIJE UNIVERSITEIT

Remote Policy Enforcement Using Java Virtual Machine

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. L.M. Bouter,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de faculteit der Exacte Wetenschappen
op dinsdag 19 januari 2010 om 10.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Srijith Krishnan Nair

geboren te Trivandrum, India

promotor: prof.dr. A.S. Tanenbaum
copromotor: dr. B. Crispo

to Surya

ACKNOWLEDGEMENTS

A Ph.D. degree can be a journey of delight, discovery, joy, loneliness, frustration and desperation, maybe not in that order. I have been lucky to have colleagues, friends and family to support and help me through this amazing voyage and see me through to its end.

First and foremost, I would like to thank my advisors Andrew Tanenbaum and Bruno Crispo. Andy and Bruno have guided me over the years, taking in their stride my crazy ideas and letting me explore topics that lay at the fringes of my research area, while at the same time always steering me towards a cohesive structure for my dissertation work. Andy has always been an inspiring advisor, providing constant support and motivation. His ability to identify the crux of the problem, provide sharp, incisive and critical comments has helped me throughout the work and I hope I have imbibed some of those strengths over the years. Bruno has been a dedicated advisor, teaching me a lot about security, the need for scientific rigour and the need to consider the whole breadth of the research area. I owe a lot to them and I thank them both.

Let me also take this opportunity to thank my doctoral committee: Prof. Frank Piessens, Prof. Sandro Etalle, Prof. Luigi V. Mancini, Dr. Herbert Bos and Prof. Frances Brazier for their valuable comments and suggestions on the dissertation work.

My colleagues have been a constant source of support and help through the years. A big thanks to Patrick for doing the heavy lifting of implementing a lot of the ideas presented in this dissertation. The conversations with Chandana, Mohammad, Hugo, Jorrit, Melanie, Gabriela, Ben, Thomas, Martijn and Philip on security issues and beyond have enriched my work several times over. A big thanks to my room mates Michel, Reza, Maik and Elzbieta for putting up with me! I would also like to thank Swami, Spyros, Michal, Bogdan, Daniela, Albana, Guido, Konrad, Vivek, Asia, Sander, Guillaume, Wilfred, Jan-Mark, Arno, Berry and Elth for making

the workplace fun and enjoyable.

I was fortunate to do a couple of internships during the course of these years and would like to thank my former colleagues at BT: Theo, Ivan, Leonid, David, Dinesh and at VMWare: Andy, Tal, Jim, Min, Mark, Swathi, who hosted me and provided great work environments and welcome break from Ph.D. work.

I owe this dissertation work to my family: my dad (who did not live long enough to see it in this form), my mom, my wife and my brother. My parents have been a constant source of comfort and support over these years. My brother has been an unquestioning yet constant support. My wife Surya has been an unflinching support through thick and thin. I know it has not been an easy few years and I thank you for being there, edging me along towards the goal. To you all I say–"Thank You!"

Srijith Krishnan Nair
Colchester, UK, November 2009

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

CHAPTER 1

Introduction

As the reach and power of the Internet and networked systems widen, and thanks to the emergence of paradigm shifting technologies and delivery models like Web Services (WS) and Software as a Service (SaaS), ever larger numbers of users are sending huge amounts of private data to remote systems that they do not have any control over. On the other side of the same technology-coin, commercial digital content distributors are using the wide reach of the Internet to help disseminate digital content like music, videos and software to individual client machines, be it generic desktop machines or consumer appliances like multimedia players.

In general, these data and content providers have a strong interest in protecting their data from being misused. They would like their data to be used as specified by them, accessible only to explicitly allowed external parties and even after said access has been granted, allowing only specific actions to be performed on the data.

These access and usage specifications are usually expressed in the form of *policies* which can then be bundled with the data that they govern and sent over to the remote machines.

Several previous works have focused on how to express the restrictions that the policies define at the level of specification languages, while others have considered the problem from a more theoretical angle by formally defining models and classes of the policies that can be enforced based on various assumptions and capabilities of the system. Fewer works have, however, investigated the actual system level requirements involved in enforcing these policies on the remote machines. This is the angle from which we approach the problem in this dissertation.

PROBLEM STATEMENT

The broad problem statement that the work reported in this dissertation addresses can be expressed as follows:

Given a data object that the user wishes to submit to an remote host and a policy that defines access and usage restrictions on the data object, design and implement an architecture that enables the enforcement of these policies at the remote host.

While the development of a fully functional policy enforcement framework would involve several complementary areas of research including the policy expression language, policy modelling, formal analysis and the system architecture development, it is only the last of these research areas that form the subject of this dissertation. Where possible, existing works in the other areas are leveraged to fill in the gaps in the rest of the framework.

1.1. TRUST MODEL

In order to clarify and understand the requirements of the system that needs to be implemented at the remote host and the measure of trust that the data provider must place on the remote party, a judicious threat model that captures the interest of the various parties involved is necessary.

While the specifics of the threat model would vary with the different application scenarios' environment, like the open or closed nature of the remote host environment, certain characteristics of the threat model can still be generalised for most policy enforcement scenarios.

The data that are submitted to the remote host is assumed to be of high value to the data provider, like financial, personal or similar. Hence the data provider is trusted to compose the right access and usage policies. At the same time, the data provider is assumed not to have any direct control over the working of the remote host.

The entity in charge of the remote host (user or administrator) is assumed to have its own interest at heart when dealing with the data provider and it is assumed to be untrusted from the data provider's point of view. It has varying degree of control over the run-time environment of the remote host. On open systems like a typical desktop or a server, the entity has almost full control over the hardware and the software that power the host. However, in the case of closed application devices like mobile phones or

multimedia players like iPod, the entity has limited control over the machine's hardware and the software. In general, the remote host is assumed to be capable of running any application locally and these applications are not trusted to adhere to the policies defined by the data provider.

The remote host is assumed to run a middleware, developed as part of this dissertation work, that enable the enforcement of the policies. This middleware, the operating system and all the hardware below it is assumed to be trusted to behave as expected. In order for the remote host to obtain the policy-attached data, it needs to prove to the provider that it is running this stack of trusted hardware and software layers. However, as mentioned before, the application is assumed to be untrusted in nature.

Thus a "Trust but verify" mantra sums up the trust relationship of the provider with the remote host. This ability of the data providers to verify the environment of the remote system, termed *attestation* in literature, is a strong assumption and requirement, which though discussed in detail later on, is not the primary focus of this work.

1.2. OUR APPROACH

The policy enforcement problem, as defined here, can be approached from various angles and using various levels of abstraction. Some of the earlier works and systems consider enforcing policies for specific applications or classes of applications [Jobs, 2007; Microsoft Corporation, 2009]. In these works the logic required to make enforcement decisions are built into the application code itself. Others, on the other hand, take a much lower level approach and consider the problem at the level of the operating system, exploring intricacies at the level of the operating system processes, describing which process can communicate with which other processes or access specific input or output channels [Zeldovich, 2007].

Based on the level at which policy enforcement is performed, the classes of policies that can be interpreted (and hence enforced) also varies. Operating system level enforcement limits the enforcement classes to those that are readily describable at the process and system call level, while those at the level of specific applications confine themselves to those applications and their semantics alone.

In this dissertation, we consider the policy enforcement problem from a data-centric view point, assuming that the policy is attached to the data

that are operated on by the applications. Our work approaches the problem at the middleware level, with the intention of exploiting the features of the higher and lower level solutions. This approach allows the architecture to enforce data-specific, and not application specific, policies across multiple applications while at the same time not running the risk of losing application-semantic level information that would be valuable in enforcing a wider variety of policy classes. In particular we consider the enforcement of policies for applications run in the Java Virtual Machine (JVM) [Gosling et al., 1996] environment. The rationale for this choice and details of the design of such an architecture are discussed in detail further on in this dissertation.

One of the key concepts that we leverage in our architecture is that of Information Flow Control (IFC), which deals with restrictions placed on how information can be transferred from one entity to another. While IFC as a research topic can be investigated from various angles, our work considers it from the perspective of the application's programming language semantics. Works in the area of IFC can be divided into two broad approaches–compile time and run time. In compile time systems, the information flow constraints are checked and verified at the time of compilation. Run-time systems, on the other hand, perform these checks dynamically during the execution of the application. While each approach has its pros and cons, our architecture uses a hybrid approach, using the run-time mechanism enhanced with static control flow analysis, due to two main considerations: the ability of the enhanced run-time system to work without having access to the actual source code of the application and the larger classes of policies that can be enforced using this hybrid run-time approach.

1.3. THESIS CONTRIBUTION

In this dissertation we present the design, implementation and application of a Java Virtual Machine based policy enforcement architecture. The contributions of this work are as follows:

- We examine in detail the previous work done in the problem space of policy enforcement and highlight the gaps our work aims to fill.
- We present the design and implementation of a JVM based informa-

tion flow control based middleware architecture aimed at enforcing policies associated with data objects.

- The middleware developed is used to implement an application independent Digital Rights Management (DRM) system using a widely studied usage control model as its basis.

- The JVM framework is also used to design a Web Service architecture capable of enforcing usage policies associated with the submitted data, as specified by the data provider.

The results obtained during the course of this work has been published in several peer-reviewed international journals, conferences and workshop proceedings.

The initial thoughts on the design of the policy enforcement architecture was presented in a paper in 2006 [Nair et al., 2009] and the preliminary results of the implementation were discussed in a subsequent international workshop the same year [Nair, 2006]. This was followed up by a detailed paper on the implementation of Trishul in REM 07 [Nair et al., 2008b] which explained in detail the approach we took, as detailed in Chapter 3 of this dissertation, as well the initial set of performance results obtained. The work done on using Trishul to enforce DRM policies, as presented in Chapter 4 of the dissertation, was published in ACM DRM 08 [Nair et al., 2008a]. This approach can be used to implement the Nuovo DRM Paradiso system aimed at enabling DRM-preserving digital content redistribution, proposed initially in the CEC 05 paper [Nair et al., 2005] and later expanded in the Fundamentae Informatica journal paper [Dashti et al., 2009].

1.4. STRUCTURE OF DISSERTATION

The rest of the dissertation is organised as follows.

Chapter 2 introduces the concept of access and usage control and information flows and outlines the existing research work in these areas within the problem space of this dissertation. We highlight the limitations of the current state of the art and explain the need for the work that this dissertation undertakes.

In Chapter 3 we present the design and implementation of the JVM based information flow control system, named Trishul, that forms the core of the dissertation. We explain in details the various design and implementation choices made during the course of the work and how they influence the performance of the achieved system, which is investigated in detail using microbenchmark performance measurements reported in the chapter.

In chapters 4 and 5, we present the applications of the Trishul framework in building policy enforcement systems in various application scenarios:

- In Chapter 4 we present the implementation of a Digital Rights Management (DRM) system using the theoretical concepts proposed in a widely researched usage control model, UCON_{ABC}, and the Trishul framework.
- In Chapter 5 we show how Trishul can also be used to implement a policy enforcement architecture for Web Services, the core components of the Service Oriented Architecture paradigm.

Finally, we conclude the dissertation in Chapter 6. There we review the contributions of the dissertation work, the lessons learnt and point out directions for future work.

CHAPTER 2

Background and related work

In this chapter we discuss the background work related to the area of security policy enforcement. We start with an introduction of basic access control and usage control policies as they form the core of any policy enforcement system. We then introduce the concept of information flow control. Information flow control plays an integral part in the systems we consider in this dissertation as a form of very fine grained abstract access control policies. We look at various types of systems that enforce these flow-based policies and discuss their key differences. A discussion on the security of Java architecture and that of trusted computing systems is also presented in this chapter.

2.1. SECURITY POLICIES

Broadly defined, security policy is a *"statement or set of statements that partitions the states of the system into a set of authorised, or secure, states and a set of unauthorised, or insecure, states"* [Bishop, 2002] or *"a formal specification of the restrictions to be enforced"* [Sterne, 1991].

This definition allows for categorisation of security policies into three: *confidentiality* policies which deal with (read) access to restricted information, *integrity* policies which deal with who can alter restricted information and how it can be transformed and *availability* policies which specify which entities can have access to specific system resources [Bishop, 2002].

In this work, we concentrate on the enforcement of confidentiality and availability policies and integrity policies are not discussed further on in

this dissertation. Interested readers are referred to [Bishop, 2002] for a systematic analysis of this category of policies.

2.1.1. Access Control

Access control policies, a form of availability policies, define which subjects can gain access to a restricted object, usually files, processes or machine resources like disk, network and CPU.

An *access control matrix model*, proposed by Lampson [Lampson, 1971] and later refined by Denning and Graham [Graham and Denning, 1971], provides the simplest framework for such a policy. The framework describes the rights R of domains over resources using a matrix representation A, an example of which is shown in Table 2.1. The protected entities (files in this case) are termed *objects* O while *subjects* S denote the set of active players in the system like users and processes. Each element $a[s, o] \in A$ denotes the rights exerted by the subject on the object.

object→ subject ↓	file 1	file 2	process 1
user 1	read, write	read, execute	execute
user 2	read	write	-

Table 2.1: An access control matrix with two subjects (users) and three objects (files).

Access control policies can be divided into two broad categories. In *discretionary access control* (DAC) [US DoD, 1985], individual users are given control over specifying who can access a particular resource they own. On the other hand, in *mandatory access control* (MAC) [US DoD, 1985], the system specifies access control restrictions associated with an object and the individual user is not allowed to change the settings. Modern operating systems in general enforce DAC, using a set of rules which describe conditions under which the access is allowed. When the access constraints are specified in terms of the identity of the subject, these kind of policies are know as *identity based access control* (IBAC) policies.

Originator controlled access control (ORCON) [Bishop, 2002] is a hybrid form of the DAC and MAC systems proposed to model scenarios where the originator of the objects retain control over them even after they

have left the originator. In ORCON, implemented as a decentralised access control system, the originator[1] of an object controls who can access it. In such a model, the owner of the object cannot override the originator's settings on the access control restrictions on the object. Thus if the originator has specified that subject $\mathbb{S}_1$ cannot read a particular file, the owner cannot overrule this and provide $\mathbb{S}_1$ with the read rights.

Often, the role played by the subject in an organisation–more than the identity of the subject–defines the access control rights he or she enjoys. For example, a company policy could be defined that the CEO of the company should have access rights to reports marked 'top-secret', while the manager of a group should have access to only reports marked 'secret.' Instead of associating the read right to the top-secret to a specific subject whose identity might change, the right is associated with the role of the CEO and the subject is then assigned this role. In this way, a change in the role can be reflected by simply changing the access matrix instead of having to re-assign the role to all the subjects. Such a model in which the role (a set of job functions) of the subject defines the access rights to restricted objects is termed *role-based access control* (RBAC). RBAC can be considered as a form of MAC model with DAC providing further restrictions on the allowed actions.

Bell-LaPadula [Bell and LaPadula, 1975] is a hybrid MAC-DAC access control policy model aimed at enforcing a confidentiality policy in a military-like environment for subjects, which constitute active components in the system, and objects, which form the passive entities. In this model, a set of security clearances is associated with the subjects and security classifications with the objects. The "need to know" principle is captured using the concept of *categories* for each security classification. Objects can be placed in several of these categories and subjects are given access to power set of the category set. Each security clearance/security classification and security category form a *security level*.

The Bell-LaPadula security model is described using three properties, the first two specifying MAC policies, while the last being DAC:

- The Simple Security Property states that a subject at a given security level may read an object at the same or lower security level and if it has discretionary read access to the object. This property is often described as the "no read-up."

[1] Note that originator need not be the owner of an object.

- The *-Property states that a subject at a given security level cannot write to any object at a lower security level. This is often described as the “no write-down" property.
- The Discretionary Security Property states that access must be permitted by the access control matrix.

“Unclassified,” “Confidential,” “Secret" and “Top Secret” is one example of one such classification. Consider the example of subject $\mathbb{S}_1$ with a clearance of “Top Secret” and $\mathbb{S}_2$ with clearance “Secret” and objects $\mathbb{O}_1$ with security classification “Top Secret” and $\mathbb{O}_2$ with classification “Confidential.” As per the simple security principle, $\mathbb{S}_1$ can read both $\mathbb{O}_1$ and $\mathbb{O}_2$ while $\mathbb{S}_2$ can read only $\mathbb{O}_2$. *-property places the restriction that $\mathbb{S}_1$ can write into $\mathbb{O}_1$ but not $\mathbb{O}_2$ while $\mathbb{S}_2$ cannot write into $\mathbb{O}_1$ or $\mathbb{O}_2$.

However, there are occasions where a subject at a high clearance has to communicate with a subject with a lower clearance. The Bell-LaPadula restrictions prevent this from happening. Instead, in order to allow such a communication, the concept of *maximum security level* and *current security level* are introduced in the model. A subject is given a maximum security level clearance to which it can raise itself but at the same time is able to lower his current security level to a lower value than this maximum in order to communicate with another subject at that lower security level. Of course, with the current security level, the associated access privilege will be reduced.

Over the years, several other access control models have been proposed and studied, including Biba [Biba, 1977], LOMAC [Fraser, 2000], System Z [Mclean, 1987] etc. These are not discussed here for reasons of brevity.

2.1.2. Usage Control

Since its introduction in early military systems, the use of authorisation has been a means to limit access to resources. However, researchers have been coming to the realisation that the traditional classic models are falling short of serving the needs of the modern distributed computing environment and associated work-flow management.

Once a subject has been given access to the object, one may still wish to control the way in which the object is used by the subject. Classically, this had to do with the time allocated to the access of the object. For example, a subject which has been granted access to use the network should not

be allowed to hog the resource in such a way that no other subjects can gain access to it. In recent years, the use of usage control has expanded to encompass issues like privacy and intellectual property protection. For example, a subject who has been granted access to a piece of data should still be restricted on what operations can be performed on the data. For example, a policy could state that playing an MP3 file could be allowed but not more than three times.

One of the more recent and coherent theoretical modelling of the various aspects of usage control is that of UCON_{ABC} model.

UCON_{ABC} Model

The core aspect of the UCON_{ABC} model proposed by Park and Sandhu [Park and Sandhu, 2004] deals with the decision-making aspects of the subjects' usage of the objects. Subjects and objects are endowed with *attributes* that capture the properties and/or capabilities of these components. One of the main innovative aspects of the UCON_{ABC} model is its concept of mutable attributes of the subjects and objects. While earlier models considered only immutable attributes that can only be modified by the manual intervention of an administrator, like the clearance of the subject or the classification of the object, in this model subject and object attributes can be modified as a result of the exercise of a right by the subject. This allows the framework to model modern applications like DRM, where the subject attributes capture variables like credit balance and object attributes the variables like cost per-use.

Subjects can be either *provider subjects*, *consumer subjects* or *identifiee* subjects. Provider subjects are the provider of the objects and hold some limited rights on it while the consumer subjects are the ones that exercise most usage rights on the objects. The identifiee subjects are those entities that are identified by the object, for example patients in medical records.

The subjects exercise privileges termed *rights*. However, unlike the traditional predefined access matrix based rights, the rights in UCON_{ABC} model are approved/provided in real-time as and when the access is attempted after the evaluation of various constraint conditions. This is similar in nature to the task-based authorisation [Thomas and Sandhu, 1998] model, which was proposed as an extension to the traditional model in which authorisation decision is made during the completion of the task. In it, the rights are created and consumed just-in-time and the exercise of a

consumable right can in turn enable other rights for different subjects and objects.

In UCON_{ABC}, the decision to allow the exertion of the rights by the subject on the object are based on—*Authorisation* (A), *oBligation* (B) and *Conditions* (C).

Authorisation evaluates the rights requested, the subject and the object attributes and provides decision on whether to allow it or not. These authorisation checks could be performed before the rights are granted (pre-authorisation or *preA*) or while the rights are being exercised, at periodic intervals (ongoing-authorisation or *onA*). Obligations [Park and Sandhu, 2002], an aspect not considered in most traditional access control models, specify the mandatory requirements that the subjects have to perform before access is provided (*preB*) or during the exercise of the rights (*onB*). For example, accepting the End User License Agreement is an obligation, specifically a *preB*. Conditions capture the system and environment states that influence the decision to allow a right. Unlike authorisation and obligation evaluations, condition evaluations do not lead to the update of the subject or object attributes. The associated condition variables are also not mutable.

The primary components of the framework and their relationships with each other are illustrated in Figure 2.1.

Denoting the mutability of the attributes by 0 for immutable, 1 for pre-updates, 2 for ongoing and 3 for post-updates, the core UCON_{ABC} models can be represented as preA_0, preA_1 and preA_3, onA_0, onA_1, onA_2, onA_3, preB_0, preB_1, preB_2, preB_3, preC_0 and onC_0.

UCON_{ABC} model has been used in modelling various scenarios like resource sharing in collaborative computing systems [Zhang et al., 2008a] and data control in remote platforms [Berthold et al., 2007a]. Work has also been done in expanding the scope of the model. Katt et al. [Katt et al., 2008] has extended the original UCON_{ABC} model by adding the notion of *post-obligations* using a continuity-enhanced usage control enforcement model and adding continuous usage sessions.

2.2. INFORMATION FLOW CONTROL

When a security policy is associated with a data object, it becomes imperative to trace the flow of the data within the system as it is used by

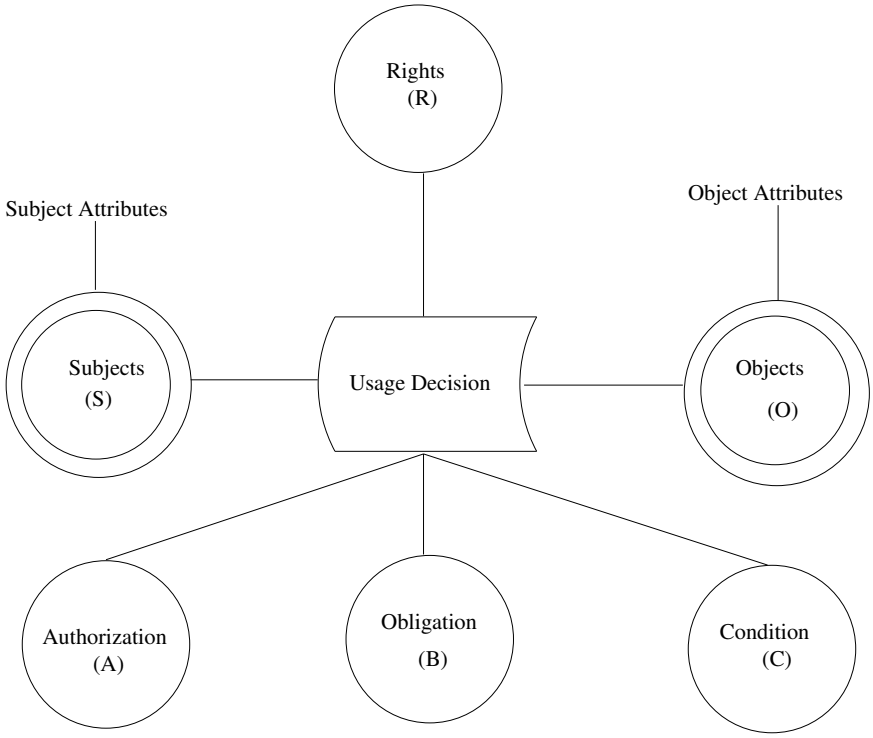


Figure 2.1: Components of the UCON_{ABC} model.

various applications and that the access to this data object by these applications be mediated as per the policy restrictions. The field of work related to *information flow control* studies this requirement and a brief discussion on it is presented here.

2.2.1. Information Flow Problem

Information flows from object[2] x to object y, denoted $x \Rightarrow y$, whenever information stored in x is transferred to, or used to derive information transferred to, object y. Denning formally defined a control flow model FM [Denning, 1976] as

$$FM = \langle N, P, SC, \oplus, \rightarrow \rangle$$

[2]used here in a broader sense than that of Java objects

N denotes a set of storage objects that receive and store data. P is a set of processes that move information around in the system. SC is defined as a set of security classes that each object in N is bound to. This security class also includes L, the lower bound of the security classes which is attached to objects in N by default. $\oplus$ is the binary operator that defines the security class of the result of a binary operation performed on any pair of operand classes. In most cases it is equivalent to an OR operation. $\rightarrow$ denotes the legal *can-flow* of information from one security class of object to another. Without losing generality and specifically in the context of this dissertation, the objects that form N can be considered as program variables, and in certain instances, as seen later on, blocks of program code.

Many programs perform their computation using one or more variables as operands and store the resulting values into another variable. For example, in the pseudo-code $y = x$, when the value of x is transferred to y, information is said to *flow* from object (variable) x to object (variable) y and the flow can be denoted as $x \Rightarrow y$ [Denning, 1976].

Flows due to codes like $y = x$ are termed *explicit flows*, more accurately *explicit direct flows* after the convention in [Guernic et al., 2006], because the flow takes place due to the explicit transfer of a value from x to y. On the other hand, consider the code shown in Listing 2.1. Even though there is no direct transfer of value from x to y, once the code is executed, y would have obtained the value of x.

Listing 2.2 shows another example of a convoluted information flow causing code fragment. At the end of the execution of this code fragment, boolean variable b ends up having the same value as boolean a. In traditional literature these two examples are grouped together as examples of implicit flows but to differentiate the level of complexity needed to trace information flow in them, we follow the convention in [Guernic et al., 2006] and terming the information flow exhibited in Listing 2.1 as *explicit indirect flow* and that in Listing 2.2 as *implicit indirect flow*. Thus $x \Rightarrow y$ is an explicit indirect flow and $a \Rightarrow b$ is an implicit indirect flow.

```
1 boolean x
2 boolean y
3 if (x == true)
4     y = true
5 else
6     y = false
```

Listing 2.1: Explicit indirect flow.

```
1 boolean b = false
2 boolean c = false
3 if (!a)
4         c = true
5 if (!c)
6         b = true
```

Listing 2.2: Implicit indirect flow.

Fenton [Fenton, 1974a] proposed a mechanism to handle implicit information flow by adding a new security class for the program counter $\underline{pc}$. Whenever a control branch occurs, $\underline{pc}$ is set to the $\oplus$ of the class of objects that form the arguments of the branch decision. Within the branch block, $\underline{pc}$ is added to every control flow. Thus in the example illustrated in Listing 2.1, when the if statement is executed, $\underline{pc}$ is set to $\underline{x}$ and $\underline{y}$ is set to $L \oplus \underline{pc} = \underline{x}$. Thus the implicit information flow from x to y is captured by the security label $\underline{y}$ and the process $\underline{x} \Rightarrow \underline{pc} \Rightarrow \underline{y}$.

However, the proposed solution does not capture the trickier *implicit indirect flow* shown in Listing 2.2. When a is $true$, the first if fails so $\underline{c}$ remains L. The next if succeeds and $\underline{b} = \underline{pc} = \underline{c} = L$. Thus, at the end of the run, b attains the value of a, but $\underline{b} \neq \underline{a}$. The same is true when a is $false$. The underlying problem is that even though the first branch is not taken, the very fact that it is not followed contains information, which is then leaked using the next if.

A trivial (and ineffective) approach to this problem is to ignore it, as done by Beres and Dalton [Beres and Dalton, 2003]. Fenton [Fenton, 1973] and Gat and Saal [Gat and Saal, 1976] proposed a solution which works by restoring the value and class of objects changed within the branch structure, back to the value and security class it had before entering the branch. This however would not work in practice since existing application codes routinely use similar control structures without paying any consideration to information flow leaks.

Aries [Brown and King, 2004] takes a more drastic approach wherein a *write* to an object within a branch structure is disallowed if its security class is less than or equal to the security class of the program counter, $\underline{pc}$. Thus, in the previous example if a is $false$, when the program tries to write to c, the compile time system prevents it from doing so, since c's security class $L <= \underline{pc}$ ($= \underline{a}$). This approach works only if the security classes have an explicit notion of high and low.

Denning [Denning, 1975] proposes a more secure approach whereby

the compiler inserts an extra instruction at the end of the $if(!a)\{c = true\}$ code block to update $\underline{c}$ to $\underline{pc}$ (= $\underline{a}$). Thus, irrespective of whether the branch was followed or not, the class of object acted upon within the branch is updated to reflect the information flow.

2.2.2. Managing Information Flow

In general, two different approaches have been explored with the aim of providing information flow control—static and dynamic, each associated with compile-time and run-time systems.

In the *compile-time* approach, applications are written in specially designed programming languages in which special annotations are used to attach security labels and constraints to the objects in the program. At compile time, the compiler uses these extra labels to ensure the security of the flow control model. These compile-time checks can thus be viewed as an augmentation of type checking.

We say that $\underline{x}$ *can flow* to $\underline{y}$, denoted by $\underline{x} \rightarrow \underline{y}$, iff information in x is allowed to flow into y [Denning, 1976]. In the context of information flow, the necessary and sufficient condition for a system to be considered secure is that, for all (x, y), $x \Rightarrow y$ is allowed iff $\underline{x} \rightarrow \underline{y}$ [Denning and Denning, 1977]. When information flow occurs between more than two objects, the compiler has to verify that each of the flows is allowed. For example, in the code segment $z = x + y$, it is clear that information flows from both x and y to z. A compiler would, in theory, need to verify $\underline{x} \rightarrow \underline{z}$ and $\underline{y} \rightarrow \underline{z}$. In general, if $b = f(a_1, a_2 \ldots a_n)$, each $\underline{a_i} \rightarrow \underline{b}$ has to be verified. However for the sake of simplicity, the compiler computes $\underline{A} = \underline{a_1} \oplus \underline{a_2} \ldots \underline{a_n}$ and then verifies $\underline{A} \rightarrow \underline{b}$.

Compile-time information flow analysis was used by Denning [Denning, 1976; Denning and Denning, 1977] as a mechanism to add a certification mechanism into the compiler analysis phase in order to prove the security of the system. In JFlow [Myers, 1999], an example of a modern compile-time system, the Java [Gosling et al., 1996] programming language is extended in order to let the programmer specify security labels to the objects. At compile time, a special compiler uses the labels to verify the information security model of the system. Once this has been verified, the code is translated to normal Java code and a normal Java compiler transforms it into bytecode.

Run-time solutions take a different approach by using the labels as an

extra property of the object and tracking their propagation as the objects are involved in computation. Instead of verifying $\underline{x} \oplus \underline{y} \rightarrow \underline{z}$ at compile time, the system propagates the security class of the information source into the information receiving object. Thus, the assignment $\underline{z} = \underline{x} \oplus \underline{y}$ occurs. These assignments however only track the flow of information as it moves through the system. The actual enforcement of security policies is carried out by another part of the system, termed the "*Policy Engine*." It intercepts all information flows from program objects (such as variables) to output channels, and allows the flow to proceed only if they are not disallowed by the relevant policies. Examples of such output channels are files, shared memories, network writes, etc. Whenever an object x tries to write information into an output channel O, the policy engine checks whether $\underline{x} \rightarrow \underline{O}$ is allowed by the specified policy and if not, the flow is disallowed, by aborting the program or silently returning the failure return value for that execution.

Pure run-time enforcement systems are, however, unable to distinguish implicit information flows and hence control them since by definition these flows exploit execution branches that have not been executed in a particular run, which hence have not been under the scrutiny of the pure run-time system. In order to enforce information flow policies on systems that have implicit flows in it, the policy engine also has to consider the program as a whole and perform a non-realtime analysis of all execution branches in the program.

Fenton's Data Mark Machine [Fenton, 1974a] was one of the earliest systems that proposed the use of run-time information flow control to enforce policies. However the machine was an abstract concept and no implementation was attempted. The security mechanism proposed by Gat and Saal [Gat and Saal, 1976] works in a similar fashion. The system however relies heavily on specialised hardware architecture to trace information flow. The RIFLE architecture [Vachharajani et al., 2004] is a more recent system that implements run-time information flow security with the aim of providing policy decision choice to the end user. They use a combination of program binary translation and a hardware architecture modified specifically to aid information flow tracking. Again, the use of the modified hardware architecture prevents it from being used on a normal machine.

Beres and Dalton [Beres and Dalton, 2003] use the DynamoRIO [MIT, 2003] dynamic instruction stream modification framework to dynamically rewrite machine code in order to support dynamic label binding. Taint-

Bochs [Chow et al., 2004] uses a similar idea to track flow of information within a system but with the aim of tracking how 'tainted' data flows in the system. With a similar objective in mind Haldar et al. [Haldar et al., 2005a] use bytecode instrumentation to track tainted data received from the network. They also attempt to extend this idea by using bytecode instrumentation to perform mandatory access control on Java objects, in order to enforce security policies [Haldar et al., 2005b]. However, the level of granularity that is considered (objects) is too coarse-grained to be useful in many applications. For instance, they provide as an example a class method that tries to leak a secret file into a public file [Haldar et al., 2005b]. This is prevented by tagging the whole class instance as 'secret' as soon as the secret file is read and denying access to public channels once this tag has been set. The coarse nature of this tagging however prevents the class method from accessing any public channels even if the operation it wishes to perform is not on the data read from the secret file.

Recent years have seen considerable interest in research of the compile-time approach towards information flow [Myers, 1999; Sabelfeld and Myers, 2003]. One of the reasons for favouring the compile-time approach is that all such systems are deemed to be secure even before execution of the program. The belief was that these systems leak only at most one bit of information per program execution and hence are inherently more secure than run time systems [Myers, 1999]. However, it has been shown by Vachharajani et al. [Vachharajani et al., 2004] that termination channel attacks, usually considered the Achilles' heel of run-time systems, can be engineered to leak arbitrary number of bits in both compile-time as well as run-time systems.

One of the most important distinguishing points of the two approaches is the kinds of policies that can be enforced by them. Compile-time systems suffer from the important limitation that the policies are bound to the code in a static manner. There is no easy way to handle scenarios where the policies are not purely information flow based and where different policies need to be attached to different runs of the application using different input data. These system perform policy-code binding early in the lifecycle, preventing their use in application scenarios where the policy is bound, not to the application but instead, to the data. An example application where such limitations occur is that of an email system in which each incoming email has its own specific policy, none of which are constant within the same or across different application runs or known at compile-time. Compile-

time systems are also limited by the fact that these systems cannot enforce policies that depend on the dynamic run-time properties of the system and the user. For example, a policy that states "This application should not be allowed to send more than 1 MB of data across the network in one day" cannot be verified at compile time, since the enforcement requires the maintenance of a state that tracks the network usage of the application at run-time. While some sophisticated compile-time systems address this indirectly by checking that the application itself contains the logic to perform this enforcement check, such a mechanism cannot be formalised for a generic class of policies.

Compile-time systems are in general more efficient than run-time systems in that the verification is done only once, at compile time. At run-time, these systems can thus confine themselves to checking the proof of the verification. However, run-time systems perform flow control on each run of the code, slowing the system. The gain in speed enjoyed by compile-time systems however is in exchange for the limitation on the kind of policies that can be enforced. These include policies that depend on the dynamic run-time properties of the system and the user. Similarly, compile-time systems cannot ensure the enforcement of system-wide obligations [Park and Sandhu, 2002] that may be stated in the usage policy, unless they can be expressed at compile-time in a static, immutable manner.

Compile-time systems are written in special languages; hence most existing applications, written in C, C++, or Java, will have to be rewritten in these languages before they can be verified. Yet another shortcoming is that the verification process is performed by the programmer and the user has to trust the programmer. Although proof carrying codes [Necula, 1997] can be used to enhance the trust, practical use of the concept has not reached a critical mass.

Inline reference monitors (IRMs) [Erlingsson, 2004] use a hybrid reference monitor with post-compile-time (but not strictly run-time) code rewriting approach to the problem of high-level policy enforcement. However, McLean [McLean, 1990] proved that information flow policies equivalent to noninterference are not *trace properties* and Schneider [Schneider, 2000] has shown that execution monitors (IRMs being a form of execution monitors) are only capable of enforcing *properties*. Information flow, not being a *safety property* is thus not enforceable by the use of reference monitors [Schneider, 2000]. This limitation can be intuitively understood

as follows—monitors see executions as a series of executed actions, however in order to enforce strong information flow constraints as discussed earlier, the enforcement system must also be aware of actions which were not evaluated by a given execution, something that monitors are not capable of.

What this means at the enforcement level is that because they are unable to trace information flow within the system, in order to enforce a fine grained policy like 'do not allow data accessed from /secret to be sent over the network,' IRMs have to resort to enforcing a coarser policy like 'do not allow data accessed from anywhere within the local file system to be sent over the network.' Hence, while IRMs are able to enforce a policy like 'do not allow transmitting on network once data has been read from /secret,' because they are unable to trace information flow within the system, they cannot enforce a finer policy like 'do not allow data accessed from local file system to be sent over the network.' In the former, all access to network resources will have to be denied, irrespective of the origin of the data that are being attempted to be sent, as long as the local file system has been accessed before the network usage. In the latter, network access will be denied only if the data that are attempted to be transferred is actually read from the local file system.

This conservative approach to dealing with the enforcement of information control policies is by itself a property of compile-time enforcement system, due to its inability to use the information available at runtime.

2.2.3. Covert Channel and Noninterference

Covert channels are a form of hidden communication channels that use the bandwidth of a legitimate channel to leak information about various aspects of the system. A simple example of such a channel is that of a process that opens and closes a file, leading to a lock / unlock being placed on the file, in a timed pattern. This lock on the file can be observed by another process in the system, which may not, in the first place, be allowed to communicate with the first process. However, the former process could represent the information it wants to leak (basically a string of bits) as the timed pattern, allowing the latter to gain knowledge about the information. The Trusted Computer Security Evaluation Criteria [US DoD, 1985] defines two general forms of covert channels—storage channels where communication is established by the act of storing or modifying objects, and timing channels

in which relative timing of events is used to convey information.

Covert channel attacks are known to be hard to eliminate by the very nature of the attacks. Since the channel from which the covert channel 'steals' the bandwidth to leak the information is almost always a legitimate channel and since the subjects have legitimate need and proper authorisation to access this channel, the use of this channel is hard to control, especially given that the timing pattern could be adhoc and dynamic in nature.

The use of covert channels points to an alternative way of looking at information flow security—in the form of interference at the system level. Viewed this way all channels, and not just those which are explicitly designed to conduct information between subjects, need to be controlled for controlling information flow within a secure system.

Goguen and Meseguer define their notion of *noninterference* [Goguen and Meseguer, 1982] as follows:

"commands in A, issued by users in G, are noninterfering with users in G' provided that any sequence of commands to the system, given by any users, produces the same effect for users in G' as the corresponding sequence with all commands in A by users in G deleted"

Though modelling security policies as noninterference assertions and system security as a set of state transitions fulfils strong security requirements, the sweeping nature of the model makes it difficult to design and implement actual generic systems that satisfy the model at the system level. Hence, in this dissertation, noninterference aspects of security are not considered at the overall system level. Similarly, covert channels are also not considered in this dissertation due to this inherent difficulty in building covert-channel-free as a practical system and its limited usage in the broad scope of policy enforcement. Furthermore there is a large body of existing work that investigates various aspects of this problem [Shaffer et al., 2008; Cabuk et al., 2004].

However, information flow control at the code semantic level as discussed before can itself be modelled as a noninterference system. A narrower definition of noninterference stated as follows is however used in these contexts. A process, P is said to be noninterfering if the values of its public (or low) outputs do not depend on the values of its secret (or high) inputs. The system presented in this dissertation can be used to enforce/implement such a noninterference policy.

2.3. JAVA SECURITY

Since its early days of the 1990s, Java technology has seen wide acceptance in the whole spectrum of computer systems, from backend servers to embedded devices. This was due mainly to the “write once, run anywhere" cross-platform nature of Java applications as well as to the rich programming language available to application developers. Built-in mechanisms like type-safe reference casting, automatic garbage collection and structured memory access make the language inherently more secure than other commonly used languages. Java Virtual Machine's (JVM) features like the class loader architecture and class file verifier further enhance the security of the execution environment.

The Java security manager is assigned the responsibility of managing the access control restrictions of the code running inside the JVM to resources external to the JVM. The Java API asks the security manager for permission to perform potentially unsafe actions by invoking the checkPermission method. Only if allowed by the manager will the API go ahead with the execution. If the permission is denied, a security exception is thrown.

The early implementation used the concept of *sandboxing* to create two levels of security environment. This was refined later on (JDK 1.2 and above) to provide more levels of security environments whose security permission could be specified with a finer granularity [Gong et al., 2003]. Cryptographic signatures are used to bind the application code to the origin of the code and policies are defined based on the principals (origin) of the code.

For example, consider an application trying to read the */etc/passwd* file. The Java API would create a java.io.FilePermission object and pass the strings ‘/etc/passwd’ and ‘read’ to the object's constructor. It then passes this Permission object to the checkPermission() method of the java.security.AccessController object. The AccessController uses the information contained in the protection domains objects (which encapsulates the permissions granted to the code source in the policy file) of classes whose methods are present in the call stack (using stack inspection [Wallach and Felten, 1998]) to determine whether the action is to be allowed or not.

In Java 2 the use of -Djava.security.manager initiates a concrete SecurityManager class and allows the system administrator to specify the access policy (used by the security manager to make it decisions) via a *policy file*.

Policy File

The Java policy file is used to grant permission(s) to class files loaded into the JVM. Each class file is associated with a *code source* which indicates where the code came from. This allows application developers to vouch for codes they develop using digital certificates and code-signing methods and users to grant permissions based on their trust on these developers.

```
1 grant signedBy ``VU-CA" {
2        permission java.io.FilePermission
3                        ``/etc/passwd", ``read";
4 }
```

Listing 2.3: Policy object example.

Listing 2.3 shows a sample policy file which grants specific permission to codes signed by 'VU-CA.' A permission object has three parts—type, name and optional action. The permission class's name indicates the *type*, for example java.io.FilePermission. The *name* is obtained from the Permission object, */etc/passwd* being the example used above. The *action* property of the Permission object specifies the action requested, for example read. One or more such Permission objects is associated with a CodeSource and forms a Policy object. The policy file consists of several such objects.

The current security manager design, even with all the above-mentioned features, still has limitations. Consider an application that wants read access to the /etc/passwd file of an UNIX/Linux system. Such an access is normal, since information present in the file is used to perform routine housekeeping operations. However, there is no reason to send the information obtained from the file outside the system via the network connection. An application that tries to do so would, for example, be trying to harvest system user information in order to perform an efficient brute force password attack. What is needed is a policy that allows an application to read the content of the password file but prevents it from sending that information out on the network. Current policy architecture does not support this level of control. In order for it to enforce a similar functionality, the Security Manager would have to prevent all writes to the network, thus preventing all network communication capability of the code.

Furthermore a more fundamental issue with the stack based approach to JVM security has recently been identified. When an application attempts

to access a restricted resource, the JVM performs a walk over the execution stack to verify that all callers currently in the stack have been granted permission to access that resource [Wallach and Felten, 1998] using the checkPermission primitive, in order to prevent the Confused Deputy Attacks [Hardy, 1988]. However it has been shown [Pistoia et al., 2007] that the stack-based access control approach is not secure as it allows untrusted code to influence the execution of trusted code that accesses restricted resources. Consider as example the code fragment in Listing 2.4.

```
1 public class Trusted {
2   public static main void(String[] args) throws Exception{
3     Helper h = new Helper();
4     String fname = h.name();
5     FileOutputStream f = new FileOutputStream(fname);
6   }}
7 public class Helper {
8  public String name(){
9    return ``/home/user/secret.txt";
10 }}
```

Listing 2.4: Class Trusted code fragment.

Assume that the class Trusted is provided by a trusted party and is allowed to perform a security sensitive operation like creating a FileOutputStream. However, unknown to the user, the class is using an untrusted Helper class to supply the name of the FileOutputStream, fname.

When the JVM performs a stack walk, it sees the following callers on the stack—security.checkPermission, FileOutputStream.<init>(File, bool), FileOutputStream.<init>(String) and Trusted.main. Since all these callers have permission as strong as FilePermission “/home/user/secret.txt", “write", checkPermission will pass. However this allows the class Helper, an untrusted code, to influence the execution. Since h.name was not in the stack when the stack walk was performed, this influence was not captured.

In addition to all this, the current Security Manager architecture is only capable of enforcing basic credential based access control policies and not usage control policies. In addition the notion of *obligations* as a set of directives that has to be carried out before or after the access/usage is allowed is neither conceptually expressible nor enforceable by the current Java Security Manager implementation.

2.4. TRUSTED COMPUTING

Trusted computing aims to provide open commodity systems with certain desirable properties usually associated with high-assurance closed systems. Cryptographic co-processors [Smith et al., 1998] work as secure tamper resistant processing units used to perform processing of sensitive operations, including tamper-proof execution of programs and protection of secrets. However, due to the high degree of secure functionalities it implements, these processors are very expensive and cannot be cost-effective for use in large-scale deployment of cheap commodity systems.

One of the fundamental problems that these trusted platforms try to address is allowing external parties to measure and evaluate the security of the platform. Software based solutions that aim at providing such functionalities can be easily circumvented by basing the attack at a lower level of the computer architecture than which the solutions work at.

2.4.1. Trusted Platform Module

The Trusted Platform Module (TPM) specifications [Trusted Computing Group, 2006], defined by the Trusted Computing Group (TCG) [Trusted Computing Group, 2009], provide a mechanism to implement a cheap trusted computing architecture. The TPM, implemented as a chip that is attached to the motherboard of the machine, is aimed at providing a hardware root of trust for, among other functionalities, implementing a foundation for verifying the software processes running on the system.

The chip implements several cryptographic operations, such as random number generation, asymmetric and symmetric key encryption and decryption, signing, secure hashing, etc. The architecture uses a combination of hardware and software features to provide a high-assurance environment. For this, each TPM has several cryptographic keys either built in or generated within the chip.

The Storage Root Key (SRK) always resides in the nonvolatile memory of the TPM and its asymmetric private part never leaves the TPM. When the TPM generates a new key, it is encrypted by its parent key and SRK forms the root of this tree, forming the Root of Trust for Storage. Endorsement Key (EK) is used to uniquely identify the TPM. Each TPM manufacturer provides a certificate to the EK attesting the compliance of the TPM to the specifications. The TPM produces Attestation Identity Keys (AIKs)

that are linked to the platform using certificates from the EK. The private AIK never leaves the TPM unless it has been encrypted by the SRK. A Privacy Certificate Authority uses the certificate issued by the EK and the manufacturer's certificate of EK to attest the authenticity of the AIKs.

Remote Attestation

A crucial functionality provided by the TPM is that of *remote attestation* that allows for the platform to attest its state in response to a challenge from an external party. The state of the system is captured in the form of a log of events, maintained by an *integrity measurement architecture* (IMA) like that of IBM IMA [Sailer et al., 2004]. It is the responsibility of the IMA to produce measurements as requested by the external parties, as follows.

The TPM contains a number of Platform Configuration Registers (PCRs) that hold the SHA-1 cryptographic hash of the event. Each measurement is extended into one of the PCRs by hashing the result of the concatenation of the PCR's current value and the new measurement value: PCR_i = SHA-1(PCR_i + m). Thus the PCR value reflects the digest of all the measurements taken so far as well as the order in which they are taken. The latest version of the TPMs have at least 24 PCRs. In order to capture the complete boot sequence of a platform, each step of the boot process is captured as a hash in the PCR. This starts when the system is booted. At this point the TPM take control, hashes the BIOS and stores the value in the PCR@. It then hands control to the BIOS, which in turn computes the hash of the operating system and extends the PCR with the measured value and transfers control to the OS. Taking this process further, the IMA captures details of the various binaries that are being executed by the operating system and stores the hash in the PCR. This process is shown in Figure 2.2, where CRTM stands for Core Root of Trust for Measurement, which forms the part of the platform whose integrity is trusted.

When an external party contacts the platform with an attestation request, the TPM uses its AIK's private key to sign the content of the requested PCR and sends it to the challenging party. The verifier authenticates the public AIK by validating the AIK's certificate chain provided by the Privacy Certification Authority. It then reads the value of the PCR and decides to trust the integrity of the platform by comparing it against a list of know 'safe' values. In turn, the challenging party must have in place a policy on how to classify the reported fingerprint values if they turn out to

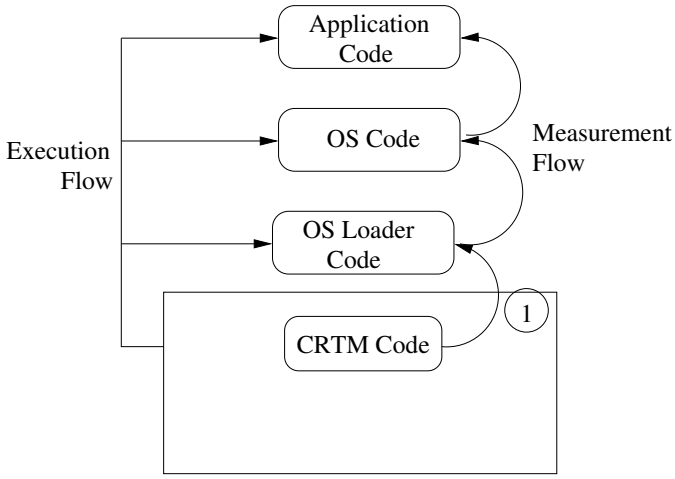


Figure 2.2: Extending trust from trusted (root of trust) hardware to the higher level of application code using induction of digest measurement.

be unknown or untrusted fingerprints.

Sealed Storage

The TPM also provides the functionality of *sealed storage* by which data can be encrypted using a key whose private part never leaves the TPM. In addition, the sealing process can be bound to a particular state of the platform as specified by the value contained in a specific PCR(s) of the TPM. Later the TPM performs the decryption operation only if the PCR(s) contain the same value as it did when the sealing was performed, thus ensuring that the decryption happens only if the system is in the same state as when it was performing the decryption.

Dynamic PCRs

TPM specification v1.2 [Trusted Computing Group, 2006] extends the nature of PCRs by introducing dynamic PCRs. In this version, unlike PCRs 1-16 which are reset only when the system reboots, PCRs 17-23 can be reset to zero dynamically on the receipt of a hardware command from the CPU. On system reboot these PCRs are reset to the value of -1 to distinguish static and dynamic resets of these PCRs.

The dynamic sets of the PCR work as follows: AMD's Secure Virtual Machine (SVM) extensions allow for the *late launch* of a Virtual Machine Monitor (VMM) with built-in protection against software-based attacks. In order to launch the VMM, the kernel code residing in protection ring 0 of the CPU invokes the *SKINIT* instruction[3] with a physical memory address as the only argument. The first two words in the memory at this location, termed the Secure Loader Block (SLB), are the SLB's length and entry point (max of 64KB). In order to protect the SLB launch from software attacks, the CPU disables direct memory access to memory pages composing the SLB and also disables interrupts to prevent codes that were executing earlier from regaining control. The processor then enters the 32-bit protected mode and jumps to the entry point specified.

In order to support attestation of the proper invocation of the SLB, as a part of the SKINIT instruction the processor resets the dynamic PCRs values (PCR 17-23) to zero and then sends the content of the (max 64KB) SLB at the entry point to the TPM. The TPM in turn computes the hash of the content and extends the value into PCR 17. Future TPM attestation can then include the value stored in PCR 17, attesting to the invocation of the SKINIT instruction and the (hash based) identity of the SLB code.

Though the SVM technology's use of SLBs and dynamic PCRs was meant for use in the secure late launch of VMMs, the Flicker project by McCune et al. [McCune et al., 2008] has extended the SKINIT technology for the execution of other security sensitive application codes in a secure and isolated environment with support for multiple session runs using the TPM sealing functionality. They have demonstrated its use in applications like SSH password authentication, distributed computing applications and certificate authority application.

We make use of these TPM functionalities to provide attestation assurance for our system.

2.5. SUMMARY

In this chapter, we looked at background work related to various aspects of security policies, information flows and Java security. However,

[3] Intel has a similar GETSEC[SENTER] instruction for its Trusted eXecution Technology extension

as evident from the discussion in this chapter, current systems have various failings and are not engineered towards enforcing comprehensive sets of policies and there is a need for a JVM based dynamic run-time time policy enforcement architecture with information flow tracing capability.

In the next chapter we present the design and implementation of such a system, which forms the core of this dissertation.

It is to be noted that this chapter only provides a basic overview of the various aspects of security we are interested in. We go into more detailed analysis of the various related works and associated issues in the chapters that follow this.

CHAPTER 3

Trishul

For a system to be able to enforce policies attached to data, it needs to be able to trace the data as they are used within the system and then verify that such usages are allowed by the policy. Therefore one of the important aspects of a data-oriented policy enforcement system is the information flow tracing system.

In this section we present the design and implementation of Trishul, a Java Virtual Machine (JVM) based IFC system. Java was chosen to implement Trishul on because of its wide use as a mature platform-independent technology. Furthermore, the interpreted nature of the code execution within the JVM allows Trishul to interpose itself between the Java application and the lower level system on which it is being executed, the details of which are discussed in this chapter.

The source code of Trishul has been released under the GPL license and can be obtained from the project's homepage [Nair, 2009].

3.1. ARCHITECTURE

In this section we provide an introduction to Trishul's architecture and discuss various associated design issues. Discussion of implementation specific issues are left for later parts of the chapter.

3.1.1. Design

Trishul is designed to be an information flow based policy enforcement architecture aimed at providing an application independent platform for enforcing policies. Hence, instead of handcrafting it as a specialised system to solve a specific application's requirements, Trishul is designed in such a way as to be generic and extensible to suit various applications' needs. With this in mind it is engineered to implement the following features:

- dynamic runtime information flow tracing mechanism that is capable of introducing and propagating taint labels
- Java method call interpositioning mechanism that allows for examination of all aspects of the method call
- pluggable modular policy engine that controls various aspects of the information flow tracing mechanism and makes decision on whether to allow the method call to proceed or not

In addition to this we also provide a Java-like language to allow the policy engine developers to write the engines modules for the various application scenarios that Trishul can be used in.

Figure 3.1 illustrates Trishul's architecture and its basic working. It consists of two parts: the core Trishul JVM system and the pluggable policy engine module. The core JVM implements information flow tracing as explained later in this chapter and provides the policy engine the hooks needed to specify the method calls made by the untrusted application that it is interested in examining. These hooks allow the policy engine to load appropriate policy enforcement logic into the Trishul system based on the policy associated with the data being used by the application and later, based on the operation being performed on the data by the application, to decide whether to allow the application's function call.

The system works as follows. When a Java application is started, the application is loaded and executed in the JVM as usual. A policy engine is also loaded into the JVM, based either on the application being run or via a command-line argument to the JVM. When the application performs certain actions which introduces data into the JVM that the policy engine is interested in, for example reading from a local file (step 1 in Figure 3.1) or receiving data over a network socket, the JVM intercepts the call and attaches a *taint label* to the data (step 2). The information flow tracing

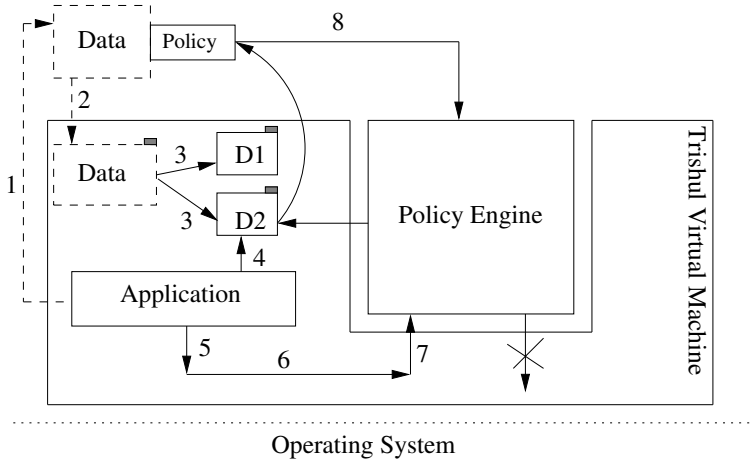


Figure 3.1: Trishul architecture.

functionality of Trishul ensures that, irrespective of what the application does or where the data are moved in the JVM (step 3), the label remains associated with the data. When the application tries to act on the data (step 4), for example send it over a socket connection (step 5), Trishul interposes (step 6) and transfers the control to the policy enforcement engine (step 7). The engine checks with the respective data's usage policy (step 8) and decides whether or not to allow the application to proceed. For example, if the application tries to write the data chunk D2, which originated from 'Data' which has an associated policy 'Do not send over the network,' to a network socket, the call will be blocked.

As seen in the figure, the policy enforcement engine is a pluggable module separate from the core of the Trishul VM. By allowing the engines to be loaded as pluggable modules, the same framework can be used to enforce policies based on the logic provided by various trusted third parties. For example consider a policy 'play 3 times' associated with a media file. A vendor V_1 would consider a 'play' as having played more than half the file's content, while another vendor V_2 would consider it a play only if the whole file is played fully. Thus V_1 and V_2 would be able to specify different interpretations of the application semantic of 'play' and provide different engine codes to enforce the policy.

In addition, by design Trishul's architecture is not bound to the use of any specific policy specification language for expressing the data's usage

policy. It is up to the policy engine writer to decide which sets of policy specification language he wishes to write the language parser in and provide the interpretation logic of policy expressions for.

3.1.2. Handling Indirect Flows

In order to capture the explicit and implicit indirect flow of information discussed earlier in Chapter 2, Trishul introduces the concept of a *context taint*, which extends the concept of associating a security class with the program counter *pc*, as proposed by Fenton [Fenton, 1974a].

The context taint is used to capture the indirect taint flow associated with a code branch, for example the case blocks in a switch statement or an if/else statement, by examining the variables that influence the conditional branch's control flow instruction (CFI) and then passing the taint of these variables into the branch blocks and augmenting the already existing direct taint flows with this additional direct taints.

```
1 boolean a
2 boolean b = false
3 if (a)
4    b = true
```

Listing 3.1: Explicit indirect flow code.

Consider the code fragment in Listing 3.1 that causes a simple explicit indirect information flow. The context taint $\underline{ct}$ is initialised to null at the beginning of the application run. The CFI is in line 3 and is influenced by the value of variable *a*. Trishul captures this influence in the context taint by adding to it the security/taint label of the variable *a*. For a CFI like *if (a == 5) && (b == 6)*, where the value of *a* and *b* influences the CFI, $\underline{ct}$ is computed as $\underline{ct} = \underline{ct} \cup (\underline{a} \cup \underline{b})$, where as before $\underline{a}$ denotes the security label associated with the object *a* and $\cup$ denotes a union/combination operator that combines taint labels.

It is to be noted that Trishul taint labels are implemented as binary bitmaps (more on that later) and hence $\cup$ operator is equivalent to the binary $\vee$ (OR) operator and is used henceforth in the explanations below.

Once the current context taint for a CFI is calculated, Trishul then identifies all the objects (variables, actual Java objects, its members etc.) whose values are modified within the taken and *non-taken* branch blocks. This is done at the loadtime of the application bytecode into the JVM and exact mechanism is described further in Section 3.3.3. At run-time, when the

Line	Branch	a=true, Taint computation	a=false, Taint Computation
01	no	-	-
02	no	$\underline{b} = \underline{L}$	$\underline{b} = \underline{L}$
03	yes	$\underline{ct\_03} = \underline{a}$	$\underline{ct\_03} = \underline{a}$
04	no	$\underline{b} = \underline{ct} \vee \underline{L} = \underline{ct\_03} \vee \underline{L} = \underline{a}$	$\underline{b} = \underline{ct} \vee \underline{b} = \underline{ct\_03} \vee \underline{L} = \underline{a}$

Table 3.1: Reasoning of how the concept of branch context taint is used to capture the indirect flow present in Listing 3.1.

conditional CFI is actually executed, the objects that are modified in any of the possible (taken and non-taken) paths are tainted with the context taint $\underline{ct}$ using the following rule:

- If the branch is taken: $\underline{object} = \underline{ct} \vee explicit\_flow\_in\_statement$
- If the branch is not taken: $\underline{object} = \underline{ct} \vee \underline{object}$

This captures the fact that irrespective of whether the branch is taken or not, the CFI objects influence the value of the objects in these branches and this influence of the CFI is captured in the taint propagation through the use of the context taint.

Consider again the code in Listing 3.1. The analysis at load time computes that the $\underline{ct}$ at line 3 ($\underline{ct\_03}$) is $\underline{a}$. It also computes that the block of code (just one line in this simple case) 4-4 is modified based on the CFI branch in line 3. Based on the value of a, this block (line) can either be executed or skipped. Table 3.1 looks at these cases.

```
1 boolean b = false
2 boolean c = false
3 if (!a)
4         c = true
5 if (!c)
6         b = true
```

Listing 3.2: Implicit indirect flow code.

Consider again the pseudo-code introduced in Section 2.2 as implicit indirect flow, reproduced here for convenience as Listing 3.2. The analysis at load time computes the $\underline{ct}$ at line 03 ($\underline{ct\_03}$) as $\underline{a}$ and $\underline{ct\_05} = \underline{c}$. Consider a scenario when $a = false$ at run time. Table 3.2 shows how the context taint approach described above correctly identifies implicit flow of information from a to b by successfully computing $\underline{b} = \underline{a}$. Assume that lines 1 and 2 when executed set the taint label of the variables b and c to L. Line 3

Line No.	Branch?	Taken?	Taint computation
01	no	-	$\underline{b} = \underline{L}$
02	no	-	$\underline{c} = \underline{L}$
03	yes	yes	$\underline{ct\_03} = \underline{a}$
04	no	-	$\underline{c} = \underline{ct} \vee \underline{L} = \underline{ct\_03} \vee \underline{L} = \underline{a}$
05	yes	yes	$\underline{ct\_05} = \underline{c}$
06	no	no	$\underline{b} = \underline{c} \vee \underline{b} = \underline{ct\_05} \vee \underline{b} = \underline{b} \vee \underline{c} = \underline{a}$

Table 3.2: Reasoning of how the concept of branch context taint is used to capture the implicit flow present in Listing 3.2.

is a CFI whose associated context taint was already calculated as $\underline{a}$ at load-time. Line 4 is executed since a was $false$. As per the rule stated earlier, taint on the variable c is calculated as $\underline{ct} \vee explicit\_flow\_in\_statement$. This calculation leads to $\underline{c} = \underline{a}$. Line 5 is again a CFI whose associated context taint was calculated as $\underline{ct\_05} = \underline{c}$. Since c is $true$, line 6 is not executed and hence the taint label of the implied flow into b is calculated as $\underline{ct\_05} \vee \underline{b}$, which is equal to $\underline{a}$. A similar result is computed when $a = true$, the details of which are left as a thought exercise.

In a way, the use of context taint can be thought of as performing a translation that converts all indirect flows into direct flows using augment security flow instructions, without actually effecting the execution logic of the actual instructions.

Simplifying the complexities associated with the objects and arrays and their use of the heap (we get into them later in section), the interpreted mode of the JVM can be considered as using a stack oriented approach wherein which the VM executes the instructions by moving data from local variable arrays to the operand stack, or vice versa, and performing computation on these values in the operand stack using it to also store intermediate values.

In effect Trishul virtual machine extends every slot on the variable array as well as the operand stack to store a bitmap based labels. In addition to performing the traditional bytecode instructions, the Trishul JVM uses the augmented stack to track the direct information flows by executing instructions that store into the stack slot's taint value the bit-wise $\vee$ of the taint values of the stacks involved in the traditional instruction. In order to track indirect flows, the JVM uses additional security registers to hold the context taints and instructions to manipulate these security registers and labels associated with the stacks and arrays. For every instruction i_b influ-

encing the control flow of the program, the label of the taint label of the stack slot conditioning the behaviour of i_b is stored in security register r_b as the context taint. This label is added to the label of every instruction which is control dependent on i_b. This handles the explicit indirect flows. In order to take into account implicit indirect flows, the label of the security register r_b is added to the taint value of every stack slot in which a value is assigned to the variable array. An important thing to note is that these additional instructions are performed dynamically at run-time (using information gained by performing static analysis performed at load-time) and that the actual bytecode of the compiled Java classes are not modified at all.

3.2. THE POLICY ENFORCEMENT ENGINE

The policy enforcement engine module of Trishul is responsible for providing two main kinds of functionality:

- tainting the data as it is introduced into the JVM by the application. The data of interest are usually those which have an access or usage policy or similar restrictions associated with them.
- deciding on how the tainted data can be used at a later stage by the application, in accordance with the policy associated with them.

In order to ease the development of the policy engines, a Java-like language named *Trishul-P* was developed as part of this dissertation work. The language is used for two main purposes:

- as a mechanism to specify the Java method calls that the policy engine is interested in, so that at run-time, Trishul can transfer the control to the policy engine when they are invoked
- provide the logic to be used to decide on how to enforce the policies when these methods are invoked.

Trishul-P has three key abstractions: actions, orders and policies. *Actions* allow the engine writer to specify the method calls that are of interest to the policy engine by abstractly specifying the method calls performed by

the Java application. Each time an action specified by the engine is about to be executed by an application, the JVM intercepts it and passes the control over to the policy engine and queries it for a decision. The decision is returned in the form of an *Order* indicating a specific action the JVM should take, such as disallowing the action, or attaching a taint label to the data.

Let us consider an example engine code fragment shown in Listing 3.3 for further explanation.

```
1  public class TestAbstractActionPolicy extends Engine {
2  public Order query (Action a) {
3    private enginetaint {secretTaint, topsecretTaint, pwdFile}
4    aswitch(a) {
5      case <* java.io.PrintStream.println(..)>:
6        return new OKOrder(this, a);
7      case <* java.io.PrintStream#<secretTaint>.println (String s#<topsecretTaint>)
           >:
8        return new HaltOrder(this, a);
9      case <* java.io.FileInputStream.<init>(..,File f)>:
10       if(f.getName().indexOf ("/etc/passwd") >= 0) {
11         return new ObjectTaintOrder(a.getThisPointer(),#object:pwdFile);
12     case <abstract * trishul.test.trishul_p.TestAbstractAction(int p1, String p2)
           >:
13       System.out.print (p1 + ":" + p2);
14       return new OKOrder(this, a);
15       }
16     break;
17   }
18   return null;
19 }
20 public void handleResult (Action action, Order s, Object result, boolean
       isException)
21 {
22   System.out.print ((isException ? "Exception":"Normal") + result);
23 }
24 }
```

Listing 3.3: Example of Trishul enforcement engine code expressed using Trishul-P.

Every policy engine class is defined as an extension of the parent Engine class (line 1) and has to provide a definition of the query class method (line 2), in which the core code for the policy engine is defined. Since the policy engine provides the reply to the JVM in the form of an Order, the return type of the query method is specified as an Order (line 2).

Line 3 defines enginetaints that are used in this code segment. enginetaints are used to assign values to the taint labels and are discussed in detail in Section 3.2.3. Line 4 marks the beginning of an aswitch block. aswitch, like the traditional switch statement, is a control statement which allows the

value of the action *a* to control the flow of execution of the decision logic of the policy engine. The case statements that follow (lines 5, 7, 9 and 12) are presented as action patterns, which are discussed in detail below. Lines 6, 8, 11 and 14 specifies the return Order for each of the case blocks. Orders are discussed in detail in Section 3.2.4. As in the case of regular Java code, further code logic can be added to the case statements (line 10).

handleResult method (lines 20–23) is executed after the execution of the actual method the application was attempting to execute when it was intercepted, following a match among one of the case statements for the earlier aswitch block. Its functionality is also explained further on in this chapter.

3.2.1. Actions

The method invocations that the policy engine is interested in intercepting are specified as Action objects within a set of case statements (Listing 3.3) using the syntax

< modf retTp pkg.class#<thisTaint>.mthd(..#paramTaint)#<contextTaint> > (3.1)

The action objects are contained within the outer $<$ and $>$ delimiter characters. The constraints that can be specified for the Java method include the modifier modf (like public, private, final etc.), the return type of the method retTp, the calling object's identity pkg.class, the method's name mthd and its parameters. The policy engine distinguishes between different actions using the aswitch statement. It is similar to Java's switch statement; the switch expression being an action and the case labels the *action patterns*.

Action signatures can also use wildcard patterns: '*' matches any one constraint and '..' matches zero or more parameter types. For example, the first case statement in Listing 3.3 (line 5) matches the println method call defined in the PrintStream Java class file of the java.io package for zero or more parameters of any type, while the second case statement (line 7) matches a similar method call only if there is only one parameter of type String. The various Taint matching criteria specified in syntax (3.1), thisTaint, paramTaint, contextTaint, enclosed using delimiters $<$ $>$, are explained further in Section 3.2.3.

3.2.2. Abstract actions

Consider a policy engine which is interested in any application's write access to a specific output channel. Java provides several different library methods that can be used to perform this operation. In order to capture these different methods, a single Order would have several actions associated with it. It becomes cumbersome to list each of these actions separately. Abstract action makes writing policies in these circumstances easier by providing a syntax to group several related actions into a single abstract action and referencing this abstract action in the policy engine code. In other words, abstract actions summarise a set of application method calls into a single action statement.

Listing 3.4 shows an example of how an abstract action TestAbstractAction is defined. Line 5 declares a matches method that returns true or false based on whether the Action under consideration matches one of the action patterns specified in the method. As before an aswitch statement is used to check the various action patterns that constitute the abstract action, specified in the case statements of lines 7 and 12. Lines 8, 9, 13 and 14 are used to provide a uniform parameter list of the TestAbstractAction action. Thus in Listing 3.4, TestAbstractAction is defined to be consisting of two actions trishul.test.action1(int a1, String a2) and trishul.test.action2(String a2, int a1), and line 12 of Listing 3.3 shows how the abstract action is used within a policy engine code.

3.2.3. Taint Labels & Patterns

Taint labels are considered within Trishul as bitmaps whose bits can be set or unset as a part of the tainting process. The enginetaint keyword is used to assign values to taints labels, as used in line 3 of Listing 3.3. Taint labels declared in this way can then also be specified as constraints in the action pattern. It need to be kept in mind that enginetaint does not introduce any implicit ordering to the lattice.

In syntax (3.1) thisTaint, enclosed with delimiters $< >$, specifies the taint of this pointer of the Java class. paramTaint can be used to match tainted parameters. It can be specified for individual parameters or '..', in which case it matches if any of the parameters is tainted. contextTaint, again enclosed within $< >$, can be used to match a tainted context. Thus the case statement in line 7 of Listing 3.3 matches println method only

```
1  public class TestAbstractAction extends AbstractAction {
2    private int param1;
3    private String param2;
4
5    public boolean matches (Action a) {
6      aswitch(a) {
7        case <* trishul.test.action1(int a1, String a2)>:
8          param1 = a1;
9          param2 = a2;
10         _this = a.getThisPointer ();
11         return true;
12       case <* trishul.test.action2(String a2, int a1)>:
13         param1 = a1;
14         param2 = a2;
15         _this = a.getThisPointer ();
16         return true;
17     }
18     return false;
19   }
20 }
```

Listing 3.4: Example of abstract action definition.

when the string parameter is tainted with value topsecretTaint and when the object instance is tainted with secretTaint taint label.

Several additional options are available for matching taints labels in action patterns. If multiple taint patterns need to be specified, it is possible to match when any taint match occurs, or only when all the taints matches. Furthermore, when matching against an object's taint, either the reference taint or the object taint can be matched. The following Backus-Naur Form-like syntax is used for defining these taint patterns

$$\#<[\text{type:}]\{\text{taint1,taint2,..}\}[\text{how}]> \tag{3.2}$$

The '#' delimiter is used to separate the pattern from the rest of the statement, while $< >$ delimiters are used to enclose the pattern. The optional parameters are denoted within the [] characters.

The type is either object, primitive or auto, to match either an object's taint, a primitive value's taint, or an object taint in case of an object and a primitive taint in case of a value. The main purpose of this flexible syntax is to allow matching against the reference's taint label when matching an object, by specifying the primitive keyword. For example, if a String parameter matching pattern uses:

$$\#<\text{object:secretTaint}> \tag{3.3}$$

the match will happen only if the String object is tainted. The pattern below

```
#<primitive:secretTaint>                    (3.4)
```

on the other hand provides a match when the reference to the string is tainted. This provides greater flexibility for an engine writer to identify taints at greater granularity.

The taint keyword in syntax (3.2) is either an asterisk (*) to specify any taint value except 0, or a set of comma-separated taint labels, declared previously using the enginetaint keyword and enclosed in curly brackets '{}', as shown in (3.5).

```
#<{secretTaint, cryptoTaint} &>             (3.5)
```

how keyword in (3.2) is either an ampersand (&) or pipe symbol (|), to match either all or any specified taints. For example the pattern in (3.5) matches if both secretTaint and cryptoTaint taint values are set. On the other hand (3.6) matches if either or both of the taint values are set.

```
#<{secretTaint, cryptoTaint} |>             (3.6)
```

This flexibility allows the policy engine writer to enforce various logic based on circumstances. For example, he could decide to halt the application exception if the input channel being read from is labelled both secretTaint and cryptoTaint or just throw an exception if it is tainted only secretTaint or cryptoTaint.

Just as object and primitive can be specified as pattern matching syntax for case statements of the aswitch block, they can also be specified when (un)tainting an object as a result of the Order returned by the policy engine. In these cases, the taints are specified as named literals of the format

```
#[type:]taint                               (3.7)
```

The type (object or primitive) is optional and specifies whether to (un)taint the object or the reference. If not specified, object taint is assumed if the taint applies to an object, and primitive is assumed otherwise. As in taint patterns, taint specifies the actual taint label. Line 11 of Listing 3.3, extracted here in (3.8), provides an example of such a tainting, in which the this object is tainted with the pwdFile label.

```
#object:pwdFile                             (3.8)
```

3.2.4. Orders

Once the policy enforcement engine intercepts the application's method call specified by the actions, it ascertains the consequence of the action, decides on the way to handle the action and returns the decision back to the JVM in the form of an *Order* object. In order to capture the various flow control requirements, Trishul-P implements the following subclasses of this Order object:

- OKOrder: the matched method is allowed to be executed
- InsertOrder: decision is deferred until after some specified code is executed and evaluated
- ReplaceOrder: instead of executing the method, this Order returns the value specified in the Order as the return value of the method execution
- SuppressOrder: suppresses the method execution and throws a Runtime exception
- HaltOrder: method call is not allowed and the application is terminated
- Param(Un)TaintOrder: (un)taints the specified parameter and then invokes the method
- RetVal(Un)TaintOrder: the return value of method call invocation is (un)tainted
- Object(Un)TaintOrder: calling object is (un)tainted
- ExceptionOrder: same as SuppressOrder, except that the class of exception thrown is specified by the policy engine
- CompoundOrder: allows for multiple orders to be combined, as explained further below

Consider again the example code in Listing 3.3. If the second case statement (line 7) is matched, line 8 instructs the JVM to terminate the execution of the application and exit the JVM by returning a HaltOrder while the third case statement (line 9) causes the policy enforcement engine to instruct the JVM to taint the FileInputStream object associated with the file */etc/passwd* with the pwdFile taint label (line 11).

Listing 3.5 shows an example of how InsertOrder is used. The first time the println action pattern is matched, InsertOrder specified in line 6 is executed (since boolean variable first is true) which in turn invokes the test method call. test sets first to false, preventing further execution of InsertOrder (line 6) when the next instance of the println pattern matches. Once test is executed, the control is again passed back to line 6. ConcreteAction, line 6, is used to create actions that can be inserted into the system. It takes three parameters, the this object, the method name and the parameters for the method.

```
1  boolean first = true;
2  int counter = 1;
3  public Order query (Action a) {
4    aswitch(a) {
5      case <* *.*.println(..)>:
6        if (first) return new InsertOrder(new ConcreteAction (this, "test(int)", new
             Object[] {counter}), this);
7        break;
8    }
9    return null;
10
11   private void test(int i) {
12     System.out.print (i);
13     first = false;
14     counter ++;
15   }
16 }
```

Listing 3.5: Example of InsertOrder usage.

Note that Trishul-P also provides for a way to analyse the state of the system after the method call has been executed. This is done by providing an implementation of the handleResult interface in the policy engine code, as shown in lines 20-23 of Listing 3.3 and extracted here in Listing 3.6 for easy reference.

```
1  public void handleResult (Action action, Order s, Object result, boolean
2  isException)
3  {
4     System.out.print ((isException ? "Exception":"Normal") + result);
5  }
```

Listing 3.6: Example of handleResult usage.

If the order returned by the query method of the policy engine allows for the application to execute the matched method call, once the method is invoked, the handleResult method is executed. This allows the engine to check whether the system is still in a specific (secure) state after the

method has been invoked. In line 1 of Listing 3.6, action is the action originally passed to the query method, order is the order returned by query method, result is the action's result as an object. If the execution of the action caused an exception, captured with the boolean isException, the exception is passed in the variable result.

3.2.5. Policy Engine Tree

Trishul allows policy engines to be loaded and unloaded at runtime as well as to be combined with the existing engines. These new engines can then load other policy engines, thereby creating a tree of policy engines. This allows for the creation of a flexible hierarchy of policy decision engines in a scenario that involves multiple interested parties. For example, a mobile phone could be shipped with the basic policy engine of the phone manufacturer, which could then be supplemented by the policy engine of the carrier. Later, when the mobile phone is used to buy and play a multimedia content, the manufacturer and carrier policy engines can be supplemented by the content provider's policy engine to enforce policies specific to the use of the multimedia content.

The loading is performed by using the addEngine call while the unloading is done using the removeEngine function. addEngine takes two parameters, the class file of the policy engine and the traditional Java policy file that specifies what access rights are allowed for this newly added policy engine. The getDisallowEnginePolicy() method loads an engine that has no access rights. Listing 3.7 is an example of a code fragment which loads a new policy engine contained in the Java classfile 'Local.class' and sandboxes its privileges with no access rights. Security aspects of the policy engine is discussed below.

```
1  private EngineHandle localEngineHandle;
2  public Order query (Action a) {
3    aswitch (a) {
4      case <* *.*.testLoadUnload (boolean b)>:
5        if (b)
6          localEngineHandle = addEngine (Local.class, getDisallowEnginePolicy());
7        else {
8          if (localEngineHandle != null)
9            removeEngine (localEngineHandle);
10       }
11       break;
12   }
13   return null;
```

```
14 }
```

Listing 3.7: Example of loading an engine with no access rights.

With such a tree in place, a very interesting situation rises with regards to action pattern matching and the execution of the orders as returned by these engines. In order to preserve the hierarchy of the engines inherent in the tree, the following logic is used–each engine in the tree is allowed to match against any action and return any order. However, a child engine's order must be at least as restrictive as its parent. Since the default action as per Trishul-P's syntax is to allow an action, this restriction means that a child policy engine cannot allow anything that its parent explicitly forbids.

This is implemented internally using CompoundOrder by placing all the orders returned by the different policies in the CompoundOrder, which is then evaluated by the policy engine. In some cases, evaluating the combined orders is straightforward. For example when multiple (un)tainting orders are combined, they are executed in order. However, when a Halt and OKOrder are combined the results must be specified explicitly. The following rules are used to decide on the outcome of multiple orders.

- If any order is a HaltOrder, the program is halted.
- If any of the order is a Suppress or ExceptionOrder, the first one encountered (in breadth-first search order through the tree of policies) is executed.
- If any order is an InsertOrder, it is executed.
- The last ReplaceOrder encountered is executed.
- All taint orders are executed as and when it is encountered.

In general, when there is a conflict, the most restrictive order is executed.

3.2.6. Policy Engine Security

Trishul policy engines are executable code in their own right and following the principle of least privileges, should be run with minimal permissions.

The standard Java security model [Gong et al., 2003] is used for this purpose. The policy engine is by default run in a sandbox without any permission. Additional permissions have to explicitly assigned using the Java policy files by the administrator of the machine. This ensures that the execution of the policy engine code is as secure as the standard Java security model.

A separate Java security manager is used to enforce the engine's permissions in Trishul. Whenever the JVM is about to make a call into the Trishul-P policy, the security manager is installed. When the call returns the original security manager is restored. The advantage of using a separate security manager for the engine code is that it can be stricter than the application's security manager, often granting no permissions at all to the engine.

Permissions are granted to the engine only when strictly required, for example when it needs to load a secondary engine. In this case, it can selectively be granted the permission to load the new engine class, not a blanket permission to load any file or access any network resource. When a secondary engine is loaded, it will again be initialised with no permissions. Its set of permissions, specified when it is loaded, must not exceed the permissions that are granted to the parent engine, to avoid permission escalation loophole. Likewise, the default base policy engine receives no permissions, unless a set of permissions for it is specified explicitly when Trishul starts.

For example, if the default policy engine is granted the permission to read files in the '*/engine*' directory, and it loads 'engineA' from this directory, it may assign 'engineA' the permission to read '*/engine*', or a subdirectory of that, but not the permission to read '*/secret*'. If 'engineA' is granted permission to read '*/engine/A*' and it in turn loads 'engineB' , it may grant 'engineB' the permission to read the contents of '*/engine/A*', but not of '*/engine/misc*', as it does not itself have permissions to that directory.

Now that we have described how the Trishul JVM and the policy engine works and have introduced Trishul-P, the language for writing policy engines, we go into the actual implementation details of this information flow tracing system in the next part of this chapter.

3.3. IMPLEMENTATION

In this section we discuss the details of Trishul's implementation, with specific reference to its information flow tracing capability as well as the policy engine module. Since Trishul is implemented as a Java virtual machine, we start with with an overview of some salient aspects of Java architecture and the JVM's working.

3.3.1. Java Architecture

The Java architecture comprises of two distinct environments: compile-time and run-time. In the compile-time environment, programs written in Java programming language are compiled into machine architecture independent *bytecodes* using the Java compiler and stored in what are called *class files*. At runtime an abstract computer called Java Virtual Machine (JVM) loads these class files and executes the bytecode instructions in them in a platform-dependent manner. By keeping the bytecode platform independent and the JVM implementation platform specific, the Java architecture is able to support platform-independent applications that can be compiled once and run everywhere.

The JVM specifications [Lindholm and Yellin, 1999] define the functionality that every virtual machine implementation should support, while leaving design choices to the individual implementations. This open nature of the specifications has led to the development of several proprietary as well as open source JVM implementations. Trishul was implemented as a modification to the existing codebase of the open source Kaffe JVM version 1.1.7 [Kaffe, 2009]. In the rest of this subsection we describe the internal design of the JVM that is relevant to the implementation of Trishul. A detailed treatment of the full design aspects of a JVM is beyond the scope of this dissertation and interested readers are referred to other resources [Venners, 2000].

The simplest implementation of the JVM is an *interpreter*. In the interpreted mode the JVM executes each bytecode instruction one at a time. The drawback of a pure interpreted mode is that the execution is time consuming. Instead, a *just-in-time* (JIT) compilation mode is often used wherein which the execution starts off in the interpreted mode but as the lifetime of the application progresses, the JVM performs profiling of the application execution and dynamically converts the frequently run methods from

bytecode into native machine code which have much faster execution time. A JIT-based information flow tracing system is however much more difficult to implement and debug. Hence the initial development of Trishul was done for the interpreted mode of the JVM and once the core code was stable, the JIT mode was also implemented.

A JVM interpreter has three distinct parts (1) the class loader, responsible for loading Java classes and interfaces and performing associated security checks; (2) the execution engine which executes each bytecode instruction; and (3) the runtime data area, consisting of a method area, heap, Java stacks, native method stacks and a program counter (pc) register. Each Java application is run inside a separate virtual machine. The method area and the heap are shared across all threads running in a JVM. The method area holds per-class structures including method data, method code and runtime constant pool[1], while the heap holds all the objects dynamically instantiated by the VM.

The Java architecture consists of two kinds of methods: Java and native. Java methods are written in the Java programming language, compiled into bytecode, stored in classes, and interpreted by the JVM. Native methods are typically written in C or C++ and compiled into machine code and stored as machine architecture specific system libraries. They usually provide direct access to host resources. Java code can call these native methods directly from the JVM using the Java Native Interface (JNI). Direct access to these native methods however renders the Java code platform specific, so their use is discouraged. Instead, Java distributions are packaged with a set of Java classes that abstract away the native method calls and Java applications are encouraged to call the method in these classes instead.

Every thread started in the JVM is given a separate Java stack that is used to maintain the state of all Java methods called by the thread, like the local variables, intermediate calculations and parameters used for its invocation. The state of the native methods invoked by a thread is saved using a separate native method stack, registers and platform specific memory areas. This internal layout of the JVM is represented in Figure 3.2. If the thread is executing a Java method, the program counter (pc) register indicates the next instruction to be executed.

Each Java stack is made up of frames, with each frame containing the

[1] A runtime constant pool is a per-class or per-interface runtime representation of the constant_pool table in a class file.

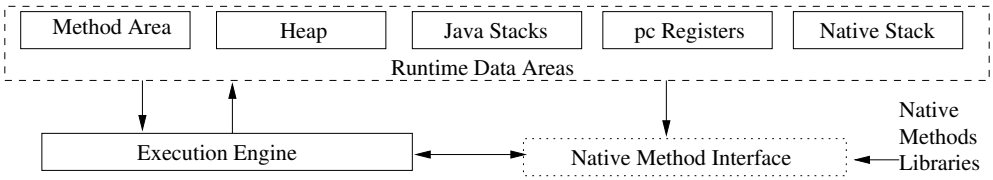


Figure 3.2: Internal layout of a JVM.

state of a separate Java method invocation. In interpreter mode, Kaffe JVM uses the variables array to hold the local variable values and the operand stack to hold intermediate operation results. The VM executes the instructions by moving data from the local variable array to the operand stack or vice versa and performing computation on these values in the operand stack using it also to store intermediate values. In order for the virtual machine to track the flow of information as the instructions are executed, every slot on the variable array as well as the operand stack has to be extended to store the label of the information that is stored in the slot.

In the next section we describe the actual implementation details of Trishul, starting with how taints for various Java system pieces are stored.

3.3.2. Taint Propagation

Trishul enforces information flow control for access to three types of application data: locals, which reside on the stack and in registers; objects, which reside in the heap; and statics, which reside in a global table.

Taints are implemented as bitmaps in Trishul and the associated join operator $\vee$ is implemented as the bit-wise OR of the values. Each local variable, parameter, return value and all values that are present in the JVM's operand stack have individual taints. Taints are stored in two places: stacks for local and temporary variables and, the heap for object members and array elements.

In Trishul, the taints for variables in the stack are stored within the same stack, implemented by extending the stack-entry structure with a taint entry. Since the memory allocation for the stack–and hence the stack related taints–are automatically handled by Kaffe's stack management subroutines, our extension did not have to do it. Object taints are stored directly in the memory allocated by the JVM for the object while the mem-

ber taints are stored in specially allocated shadow memory to optimise their allocation only when the member is assigned an initial value.

Each traditional Java object has a taint, termed the *object taint*, associated with it, while each of the object's member variables have their own individual *member taints*. Whenever an object member is assigned a value, the value's taint is included in the *object taint*. Thus the object taint is the combination of all the associated member taints.

In order to be efficient, Trishul calculates the object taint in a lazy manner. The taint is not reset automatically when member taints are reset. Since multiple member taints may have the same taint value, simply subtracting the reset member's taint value from the object taint bitmap would not capture the taint update correctly. A full scan and combination operation of each of the member taints on the object is needed to capture the object taint and performing this frequently would result in large overheads. Instead, this scan is performed only when the policy engine explicitly asks for the object taint's value.

In addition to the object and member taints, the reference used to access the object is also tainted. This taint is included whenever a member is read or written into, along with the context taint.

Since static variables defined in the class are not object members and there exists only one instance of these variables in a process, only one taint value is associated with these variables, stored in the variable descriptor used by the JVM. Array taint and associated element taints work in a similar way to the object and member taints.

The process of taint propagation is implemented in Trishul in a straightforward manner–by extending the macro code implementing the Java bytecode instructions to combine the taint labels when the values are computed. For example, consider the iadd instruction, which removes two integers from the top of the stack, adds them and places the result on the top of the stack. This is a simple case of explicit flow of information from the variable containing the two integers to the variable containing the result. Internally the iadd instruction was realised by Kaffe using the C macro shown in Listing 3.8 (tint is the internal datatype representing integers and v is the C struct for the slots). In Trishul this macro was extended to propagate the taint label's value of the operands, as shown in Listing 3.9.

```
1 #define add_int(t, f1, f2)  (t)[0].v.tint = ((f1)[0].v.tint) + ((f2)[0].v.tint)
```

Listing 3.8: C macro that implements iadd.

```
1 #define taint2(t,f1,f2)  (t)[0].taint=taintMerge2 ((f1)[0].taint, (f2)[0].taint)
2 #define taintMerge2(t1,t2)  ((taint_t) ((t1) | (t2) ))
3
4 #define add_int(t, f1, f2)  (taint2(t,f1,f2), (t)[0].v.tint = ((f1)[0].v.tint) +
      ((f2)[0].v.tint)
```

Listing 3.9: Modified C macro that implements iadd and propagates the taint.

3.3.3. Indirect Flows

As described earlier in this chapter Trishul uses the concept of context taint to capture the indirect flow introduced by control flow branches. In order to capture the context taint Trishul performs a postdominator data flow analysis [Aho et al., 2006] using a two-stage process. It is implemented as a hybrid of the static and dynamic information flow control systems by using a combination of static analysis and run-time enforcement.

The static load-time analysis captures the indirect flow contained in the Java bytecode instructions while the runtime enforcement part allows for the late binding of policies to the system at runtime instead of compile time.

Load-time Static Analysis

The static analysis is performed during the initialisation/loading phase of the application. However, in order to be efficient, instead of performing the whole analysis at the load time of the class, it is deferred until the first time each method is invoked by the application. This allows the process to skip the analysis of those methods that are defined in the class but not used by the application. In the first stage of the analysis, when a method is invoked for the first time, its control-flow graphs (CFGs) [Aho et al., 2006] with *branch bitmaps* are computed to detect context blocks. In the second stage, these CFGs and branch bitmaps are summarised into *context bitmaps*. These processes are explained in details below.

Creating the CFGs The CFGs are used to determine the conditional flow instructions (CFI) that control the execution of a statement. Even though the bytecode verifier of the Kaffe JVM already creates a CFG for its internal use, the produced graph is not in a readily useful format for analysing context taints. Hence in this implementation of Trishul, a separate CFG is calculated.

The CFG is created using a single forward pass over the method's code with a node for each basic block. A basic block is a sequence of instructions with a single point of entry (the first instruction) and a single point of exit (the last instruction). A CFI always forms the last instruction of a basic block. Directed edges represent transitions between basic blocks, either caused by the normal flow of instructions or by a CFI.

A basic block with a goto instruction or one without any CFI has an outward edge leading to another block, while that with if-statements have edges to two other blocks and those with switch instructions lead to any number of other blocks. The basic block containing the last instruction in a method has an outward edge leading to a special *exit* block. CFIs that exit the current method (return and throw instructions) are linked to the exit block ensuring that all blocks (other than the exit block) will have at least one outward edge.

The CFI's targets are checked to ensure that each basic block has a single point of entry. If the target is before the current program counter (i.e. a backward branch) and it branches into the middle of a basic block, the target basic block is split so that the target instruction is the starting point of its block. In the case of a forward branch, a new basic block is created starting at the target instruction, which is initially empty. This block is stored in a forward list, which is checked when a new basic block is created. Later, when the basic block that includes the target instruction (identified earlier in the forward branch) needs to be created, the basic block from the forward list is used. If the target instruction is not the first instruction of the new basic block, this block is split as required.

Consider the Java code in Listing 3.10 and the corresponding Java bytecode in Listing 3.11. Figure 3.3 shows the CFG generated for this opcode using the process explained above. The '?' symbol in front of the bytecode in Figure 3.3 denotes a CFI.

```
1  public static void
2          main(String args[]) {
3     boolean a = true;
4     boolean b;
5     if (a)
6     {
7        b = true;
8     }
9     else
10    {
11       b = false;
12    }
13 }
```

Listing 3.10: Code for CFG example.

```
00: iconst_1
01: istore_1
02: iload_1
03: ifeq 11
06: iconst_1
07: istore_2
08: goto 13
11: iconst_0
12: istore_2
13: return
```

Listing 3.11: Bytecode of Listing 3.10.

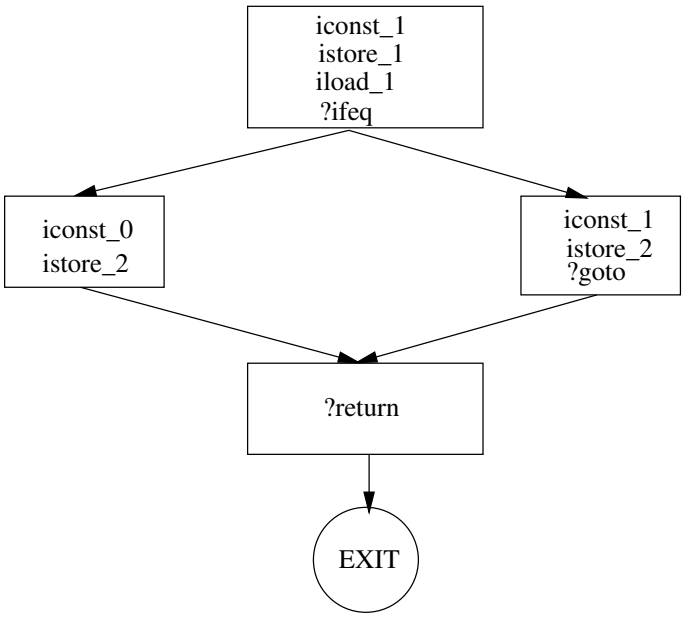


Figure 3.3: Control-flow graph created from Listing 3.11.

Branch bitmaps Once the basic block of the CFG is calculated, a branch bitmap is associated with each of the blocks. It contains a number of bits for each conditional CFI, one bit representing each possible target of the CFI. In the case of an if-statement there are two bits: one representing

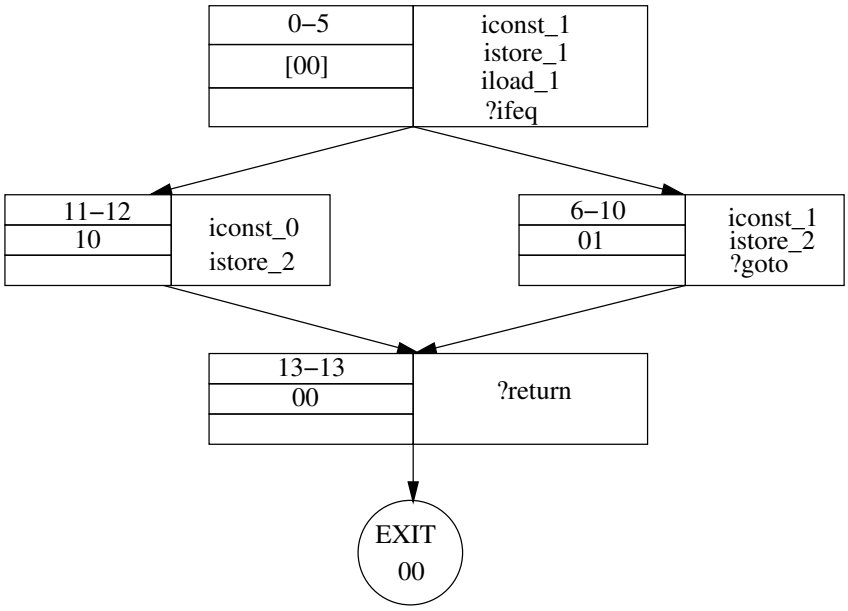


Figure 3.4: CFG showing initial branch bitmaps.

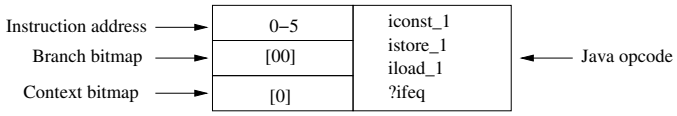


Figure 3.5: Details of the variable fields calculated in the CFG.

the case when the branch is taken, and one representing the case when the branch is not taken. A switch instruction has one bit per case, and possibly one bit for the default case.

The branch bitmaps are shown in the centre-left field in each node of Figure 3.4, which is the same CFG as Figure 3.3. The rest of the numbers shown in the left hand column of the block are explained in Figure 3.5. The bitmap consists of two bits, both referring to the if-statement in the top-most basic block. The fact that these bits represent the if-statement at the end of the block is indicated by the '[]' that enclose these bits.

At the start, the branch bitmaps are initialised to zero. Bits that repre-

sent a branch target are initialised to 1 in the basic block containing that instruction. Thus the bitmap in basic block <11 12> (indicating the program counters in the top-left field) is initialised to 10, because block <0 5> branches into this block. Likewise, block <6 10> is initialised to 01. Block <13 13> is initialised to 00, as the earlier branch instruction does not branch directly into it. These are recorded in Figure 3.4.

Once all bitmaps have been thus initialised, they are then recursively updated until each bitmap satisfies the condition that each bit that is set in any block that precedes the block in question is also set in the current bitmap. In other words, each bit that is set in a block, flows into every block following it. Thus the bitmaps in <13 13> are updated from 00 to 11, as shown in Figure 3.6. Once this is done, the bits controlled by a specific CFI can be in one of two states: all bits have the same value (00,11), or they have different values (01,10). In the first case, each possible path starting at the CFI includes the basic block (11), or no path includes the basic block (00). Either way, the execution of the basic block is not influenced by the CFI anymore. When the bits have different values, only some of the paths starting at the CFI reach the basic block, therefore the execution of the block is influenced by the CFI. This is captured by a context bitmap.

Context bitmaps Context bitmaps summarise the information stored in branch bitmaps. The bitmap contains a single bit per CFI. The bit is set to 0 if all the bits in the branch bitmap are the same, else it is set to 1. Thus the bit is set if the basic block is controlled by the CFI represented by that bit. Context bitmaps are shown in Figure 3.6 in the bottom-left fields. Again, rectangular brackets are used to show which bit represents the CFI in a basic block. The basic blocks <6 10> and <11 12> are controlled by the ifeq instruction in block <0 5> and ends up being calculated as 1, while block <13 13> is not and its context bitmap is set to 0.

The context bitmaps are stored in a list, sorted on the program counter of the first instruction in the basic block and passed to the run-time system for use in updating the context taint accordingly.

Run-time analysis

Context taint At runtime the stack frame of each method contains an array of partial context taints. This array contains an entry per conditional CFI, and thus has as many entries as the context bitmap has bits. When a

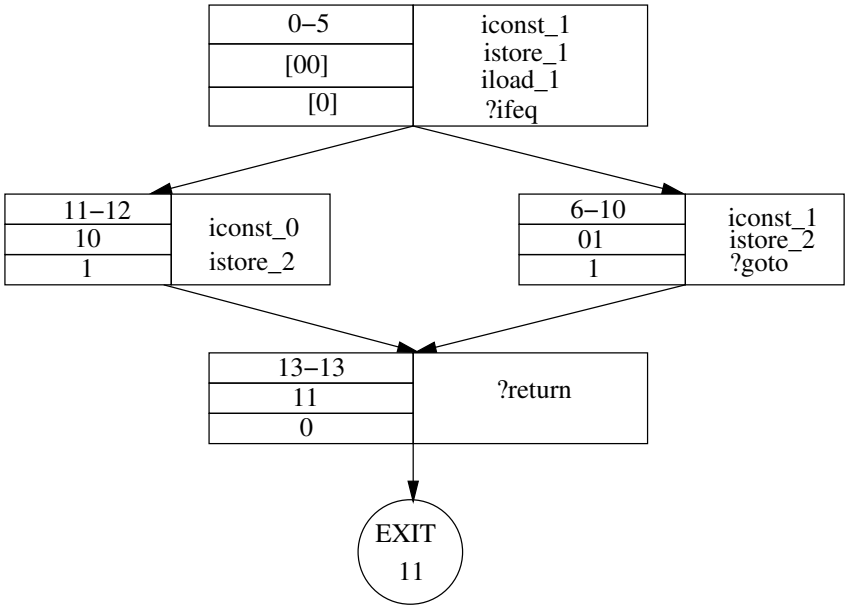


Figure 3.6: CFG showing final branch bitmaps and context bitmaps.

conditional CFI is executed, the condition's taints are stored in the appropriate array entry. When a new basic block is entered, either through the execution of a CFI or when the program counter advance beyond the current basic block, the context taint is updated by combining the partial context taints of all conditional CFIs whose bits are set in the context bitmap. The code in Listing 3.9 which showed how Kaffe's C macro was extended to capture the taint propagation is further extended as shown in Listing 3.12 to capture the effect of the context taint. The way in which partial context taints and context bitmaps are stored differ between the interpreted and JIT modes of Trishul and is explained in detail later in the chapter.

```
1 #define taint2(t,f1,f2)  (t)[0].taint=taintMerge3 ((f1)[0].taint, (f2)[0].taint,
      current_context_taint)
2 #define taintMerge3(t1,t2,t3)  ((taint_t) ((t1) | (t2) | (t3)))
3
4 #define add_int(t, f1, f2)  (taint2(t,f1,f2), (t)[0].v.tint = ((f1)[0].v.tint) +
      ((f2)[0].v.tint)
```

Listing 3.12: C macro that implements iadd modified to propagate the taints including the context taint.

Method call Java methods by themselves do not have taints but as they can be invoked at any point during the execution of the Java code, especially within a context that is tainted, it is instrumented to inherit an *initial context taint* that captures the context taint (0 for the main method) at the call point, and that of the this-pointer when an object method is invoked. This initial context taint is then included in the method's own context taint, ensuring that invoking a new method call does not let the execution escape the context taint in the caller's execution context. When the method call returns, the context taint of the caller code is updated to the value it was before the method call.

The parameters passed to the method call preserve their existing taints and their use within the method results in the propagation of these taints.

Non-taken branches As explained earlier, in order to capture indirect flows, it is necessary that even those variables that are present in the non-taken branches of a conditional CFI be tainted with the context taint. For this, a list of all variables that are modified in each basic block of the CFG is maintained. This list is then extended by including lists of any basic blocks that are accessed through method calls. At runtime, even when a branch basic block is not executed, the taints of the variables in the list

are extended with the current context taint, since the decision to not execute the block is (potentially) influenced by the variables of the CFI whose taints make up the context taint.

However, in some cases it may not be possible to create a complete list of variables modified in the block until the branch is actually executed, because of dependence on information available only at runtime or because the analysis is not rigorous enough. In order to ensure that this limitation does not lead to the leakage of information, Trishul provides an optional fallback taint termed the *global context taint* that can be disabled or enabled as per the system administrator's requirement. Once it is enabled, whenever the list of modified variables cannot be accurately determined, the current branch context taint is added to the global context taint. Then, by ensuring that this global context taint is always included in the currently active context taint all through the execution of the application, Trishul ensures that any later use of the undetermined variables in the non-taken branch block will be tainted indirectly with the earlier branch's context taint.

However, since this global taint cannot be reset or untainted automatically, it is up to the policy engine writer to decide on how to control it. It is possible to disable it when the JVM is first invoked or it could be untainted manually using the policy engine syntax made available by Trishul-P or using the concept of *annotations* discussed below. The use of the global context taint allows the system administrator to be very conservative in approaching the tainting problem at the expense of *taint creep* (the phenomenon of taints spreading uncontrollably throughout the system), rendering the application unusable if the policy enforcement is strictly adhered to. In our non-exhaustive analysis, Trishul was able to identify the complete list of modified variables in a basic block 96% of the time across various applications and did not have to report to using the global context taint. However, in the 4% of the cases where the global context taint was used, it did lead to extensive taint creep when manual untainting was not performed.

3.3.4. Manual Taint Propagation

While most taints are propagated automatically as described in earlier sections, native method invocations necessitate manual taint propagation. Since these methods also create and move values, they would also be creating

and propagating taints. However, as these methods are executed outside the JVM's control (see Section 3.3.1), the tainting has to be implemented either as modification to the native code by adding taint propagation code (see Listing 3.13 for an example of System.arraycopy native method that has been modified to propagate the taint this way) or by using the *annotation* propagation method provided by Trishul. While the modification of the native code provides better performance compared to the annotation method, the permanent hardcoded nature makes it harder to maintain the native method codebase.

```
1  in = $src[srcpos];
2  inTaint=src->member_taint;     /* added */
3  out = &dst[dstpos];
4  outTaint = dst->member_taint; /* added */
5
6  for (; len > 0; len--)
7  {
8    *out++ = *in++;
9    *outTaint++ = *inTaint++; /* added */
10 }
```

Listing 3.13: Native method System.arraycopy modified to propagate taints.

Annotations are Java classes that contain policy writer specified hook methods for existing classes' methods. When the original method is invoked by a Java application, control is transferred to the hook method instead, which then adjusts taints before and after invoking the original method.

Annotations can also be used to let the policy engine writer manipulate the taints at a level higher than the implementation unit of objects and variables that Trishul supports. For example, consider the String class. Since a String is conceptually made up of a sequence of characters, it would be natural to assume that any tainted character would taint the string as a whole. However, in practice, the String class is implemented as an array of characters. If one of the characters is tainted, it causes the array to be tainted but not the String object. Annotations can be used to carry this taint from the array to the object. Trishul's code base is already supplied with several of such annotations that ensure parity between the logical and practical way of tainting the Java units.

Listing 3.14 shows how the String object annotation (trishultaint String) is implemented in Trishul. Two hook methods are specified in the listing, one for the String Java method and the other for the hashCode method. In the former, the array's taint is applied to the String object while in the latter,

the integer returned by the invocation of the hashCode method is tainted with the taint of the String object.

```
1  package java.lang;
2
3  trishultaint String  {
4    notrishultaint int hash;
5
6    public String (byte[] b, int offset, int len)
7    {
8      setObjectTaint (this, getArrayTaint (b));
9      super (b, offset, len);
10   }
11
12   public int hashCode ()
13   {
14     int h = super ();
15     taint (h, getObjectTaint (this));
16     return h;
17   }
18 }
```

Listing 3.14: Annotation applied to the String object.

Annotations can also be defined to specify that some variable do not propagate taint values using the notrishultaint label as used in line 4 of Listing 3.14. This is a clear security violation and a potential security hole, but can be necessary in some cases to avoid taint creep. It can be used securely if it can be guaranteed that the further use of the variable does not transfer information, as can be the case if the variable is used for, say, caching. A similar method is available to allow methods to be invoked without a context taint, again to handle problems in certain application scenarios.

In order to ensure security of the implemented annotations, they have to be delivered alongside the Java-library and included in the library's signature, making it impossible to load the core Java library without the associated annotations or being able to add new unapproved annotations to the system. For this, all annotations are stored in a single signed jar file, which is loaded at start up and if its signature is invalid, the system refuses to start, thus preventing non-administrative users and potential unsafe code from adding security compromising annotations into the system. Since Kaffe does not support verification of JAR signatures, this feature was added.

Furthermore, in order to ensure that all native functions are properly annotated, a list of the allowed native libraries that are approved for use is stored in Trishul along with the digest of these libraries. Thus any unauthorised library is prevented from being loaded and native methods defined

in it are prevented from being invoked by Java programs.

3.3.5. Exception Handling

There are two kinds of exceptions found in Java: checked and unchecked runtime exceptions. A checked exception must be handled explicitly when the code is being written, either by containing the offending throw instruction in an appropriate catch block, or by declaring it as part of the method's signature. If declared as part of the method's signature, the exception will cause the current method call to be terminated and transfer the control to the caller. If the caller contains an appropriate catch block, that will be invoked. Otherwise, the process is repeated until a method with an appropriate catch block is reached, which must be present in the main function, since Java main methods do not allow exceptions to be declared. Run-time exceptions, on the other hand, are not declared. They are used by the JVM to signal internal errors that may not be recoverable, such as dereferencing a null pointer or division by zero.

As exceptions cause changes in the flow of control, they require special handling to avoid information leaks. Exception handling in the JVM is identical for normal and run-time exceptions. However the fact that runtime exceptions are not declared makes their analysis harder, as does the fact that most Java instructions can cause some form of run-time exception. Because these runtime exception causing instructions are so common and the likelihood of them happening is low, tainted run-time exceptions may be treated as rare abnormal events that causes the program to terminate. Hence, they are disregarded during the exception analysis of the taint propagation system in Trishul. Doing so does allow an application to leak information. However, since runtime unchecked exceptions invariably cause the application to terminate, the amount of information leaked is very limited, mostly as little as 1 bit per exception.

In a checked Java exception, the throw statement transfers control to the appropriate catch block, much like a goto statement, with the difference that in the case of exceptions, the target address may be in a different method if the throw statement is not inside an appropriate try/catch block. Also, unlike a goto statement which always has a fixed target address, the target of a throw statement may not be known before runtime. This is due to the fact that an exception that is thrown is just a normal object that resides on the heap, the parameter to the throw instruction being a reference

to that object. Before run-time only the static type of the reference can be determined. The actual type of the exception object may be a subclass of that type. As the catch block that is invoked depends on the actual type of the exception object, the catch block may not be known before the throw statement is actually executed. Also note that if the exception is thrown to a different method, it is generally not possible to determine the catch block before runtime, since that would require knowledge of each possible call site.

In Trishul we aim to determine the run-time type of the exception object by finding the instruction that places the reference to the exception on the stack. If this is a new instruction (which it frequently is), the run-time type is known. With the run-time type being known and there being an appropriate catch block, an edge is added in the CFG from the throw statement to the catch block. In other cases, an edge is added to the method's exit block. This errs on the side of caution by assuming that no catch block will be executed and hence all the variables written in any of the catch blocks need to be tainted, possibly triggering the global context taint fallback mechanism. This is mostly not necessary as almost all non-malicious (more that 97%) occurrences of exceptions are analysed judiciously by Trishul's algorithm.

Method invocations also require special care in the light of exceptions. If a method can throw an exception, the flow of control will not necessarily pass to the instruction following the method invocation, but may instead pass to a catch block or the caller of the method. This turns a method into a conditional CFI. If run-time exceptions are treated as normal exceptions, each instruction that can cause a run-time exception also becomes a conditional CFI. In Trishul, when the CFG is being calculated, a method that can throw an exception is treated as a CFI with an edge to the next basic block, as well as an edge to each catch block that may be invoked, or the exit block if a catch block cannot be determined. There can be multiple such edges, as a method may declare different distinct exception types.

If an exception is thrown, the current context taint and the exceptions taint are stored. At the catch block, this taint is included in the context taint; each catch block has a bit in the context bitmap and thus an entry in the context taint for this purpose. If the catch block is not in the same method as the throw instruction, the call stack will be unwound. Each method invocation on the stack is treated as a conditional CFI and requires tainting of the variables that are written to in the current stack frame, since the instructions following the method invocation are analogous to a control

path that is not executed. As the stack will be unwound anyway, tainting local variables and stack elements is not required. The entire unwinding of the stack can be skipped if neither the exception nor the context were tainted.

Information can be leaked if a method that declares an exception does not end up throwing such an exception, much like an if statement can leak information by not executing a certain control path. Consider the following example in Listing 3.15

```
1  boolean b = true;
2  try
3  {
4    leak (secret);
5    b = false;
6  }
7  catch (Exception e) {}
8
9  void leak (boolean secret) throws Exception
10 {
11   if (secret) throw new Exception ();
12 }
```

Listing 3.15: Leaking information through an exception that is not thrown.

In this case, if an exception is not thrown, it conveys the fact that secret is false, which is captured in the variable b. In order to capture this information flow, the assignment to *b* must also be tainted. This is handled by maintaining in the leak function a taint of exceptions that are not thrown. When the method executes, each conditional CFI that skips executing of a throw statement causes the context taint to be included in this taint. As the invocation of leak is considered to be a conditional CFI, it has a bit in the context bitmap and an entry in the context taint. This entry is set to leak's non-thrown taint, which will ensure the assignment to *b* is tainted. Note that after the catch block, the control paths merge, so the context is untainted.

Finally blocks, which are executed when leaving a try/catch block regardless of whether an exception is thrown, are implemented in Java as catch blocks for any type of exception. The case when no exception is thrown is handled by an explicit jump into the finally block. Therefore these blocks are handled automatically in Trishul.

3.3.6. Just-in-Time Mode

In order to improve the performance of the bytecode execution, the Just-In-Time (JIT) compiler of a JVM compiles sections of the often used platform independent Java bytecode to machine specific lower level code. Implementing Trishul's taint propagation and policy engine hooks in such a system presented various challenges. The corresponding design decisions made for such an implementation are discussed here.

In Kaffe, the JIT compiler mode is setup as follows. The Kaffe compiler generates and places short segments of code called trampolines [Inaba, 1998] at the address of the actual method. When an attempt is made to invoke the method, the trampoline is invoked instead, as it occupies the method's address. The trampoline in turn invokes the JIT compiler to generate the method's actual machine code and in the process replaces the trampoline. Finally, the generated code is invoked. The next time around when the method is invoked, the method's native code in invoked directly as the trampoline is no longer present.

In the JIT mode, Kaffe generates functions with this layout:

1. Load locals and parameters into registers
2. Perform calculations on registers
3. Store registers into locals and return value

Thus, there is no direct connection between the locals in steps 1 and 3, except through the registers used in step 2. In order to ensure that the taints on the locals and parameters are propagated properly, the JIT compiler was modified so that each instruction generated in step 2 that operates on registers also generates instructions to modify the associated taint registers in order to track the taint flow.

In Trishul's current implementation, each normal register used by Kaffe has an associated (part of an) Streaming SIMD[2] Extensions (SSE) register that holds the corresponding taint value for that register. So the layout is modified to:

1. Load locals and parameters into registers and corresponding taint values into SSE register

[2]SIMD stands for Single Instruction, Multiple Data, colloquially, "vector instructions".

2. Perform calculations on registers and associated taint registers
3. Store registers into locals and return value and SSE registers into taint values

This process is explained in detail below.

Register Taints

The IA-32 architecture provides three sets of registers: eight 32-bit general registers, eight 80-bit floating point registers and eight 128-bit Streaming SIMD Extensions (SSE) registers. The program counter is a 32- bit register known as the Extended Instruction Pointer (EIP).

The general registers (EAX, EBX, ECX, EDX, EDI, ESI, EBP and ESP) are used for most integer calculations and control flow instructions. Operations are generally performed on two operands: one or two input operands and one output operand. Both operands may be registers, or one may be a memory location. The floating point registers (FP0-FP7) are organised into a stack, with FP0 being the initial top element. Floating point operations are performed between two stack elements, or between the top of the stack and a memory location. The result is always left on top of the stack.

The SSE registers are not used by Kaffe and hence can be used by Trishul to hold the taints of values stored in the registers. Each of the eight 128-bit SSE registers (XMM0-XMM7) are made up of four 32-bit parts. However in Trishul, each SSE register is used to hold only three 32-bit taints, leaving the last 32-bit part free for computations. This is to overcome the limitation of the SSE instruction set that it does not have an operation to move a part of the register into another part of another register, only supporting shuffle operations that combine different parts of a single register into another register.

Hence, in order to move, for example, the taint of register ECX to the taint of ESI, three operations are needed: the destination part is cleared, the source part is moved into the correct position in a temporary register, and this temporary register is OR-ed into the destination register. As only the destination part must be affected, the remainder of the temporary register must be zeroed. Since the SSE instruction set does not provide an operation to clear a specific part of a register, the highest 32-bit part is always kept to 0, allowing a shuffle operation to copy that 0 into one or more specific parts

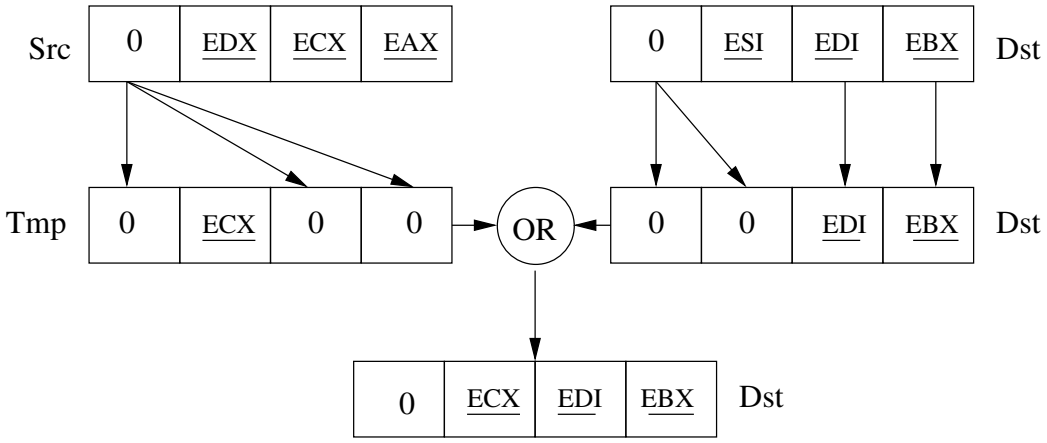


Figure 3.7: Moving ECX to ESI using SSE registers in 3 operations.

in a single operation and also move the source part to the correct position. Figure 3.7 illustrates these steps.

Variable and Argument Taints

Trishul stores variable and argument taints on the stack. The layout of a typical Kaffe stack is shown in Figure 3.8. The call instruction pushes the arguments to the method on to the stack before storing the return address[3].

In the prologue of the newly invoked method, the previous frame pointer is then stored and the base pointer register EBP is made to point to the current top of the stack. The local and temporary variables are stored below that. Since it is known at compile time how many temporary variables are required, Kaffe is able to reference them using addresses relative to EBP, just like local variables. The stack pointer register ESP is used only when a new stack frame must be created.

Figure 3.9 shows the layout of the modified stack frame in Trishul, holding the taint values represented by the underlined names. All the taint values for the method's arguments are pushed onto the stack before the arguments. This requires that the list of arguments be iterated twice. Had the arguments and taint values been pushed as ⟨argument, taint⟩ tuples, the

[3]Remember that the stack grows downwards.

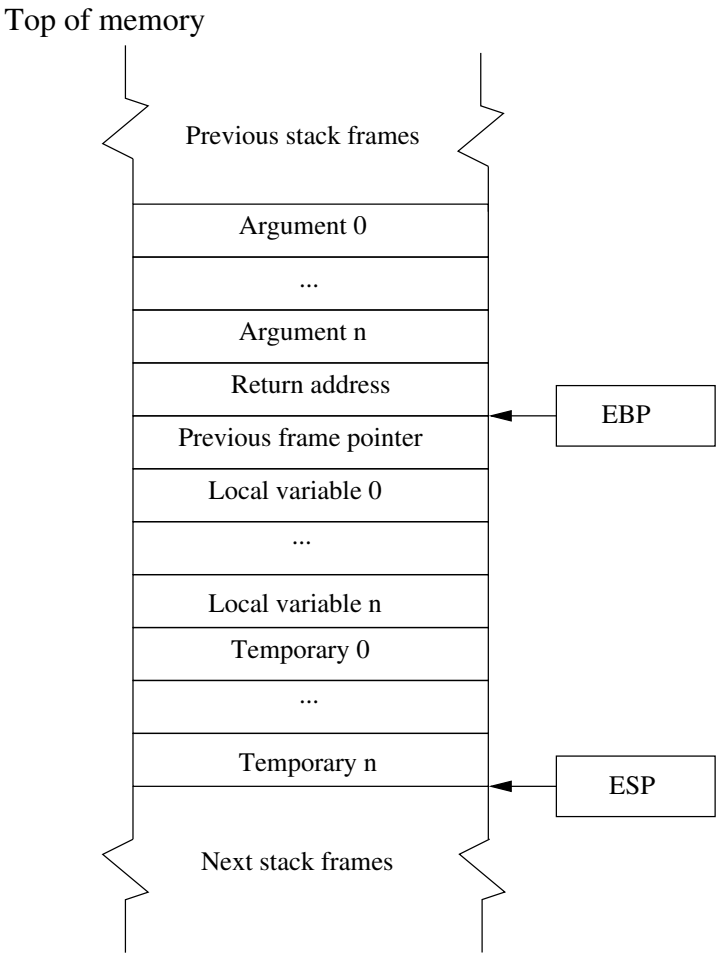


Figure 3.8: Kaffe stack frame.

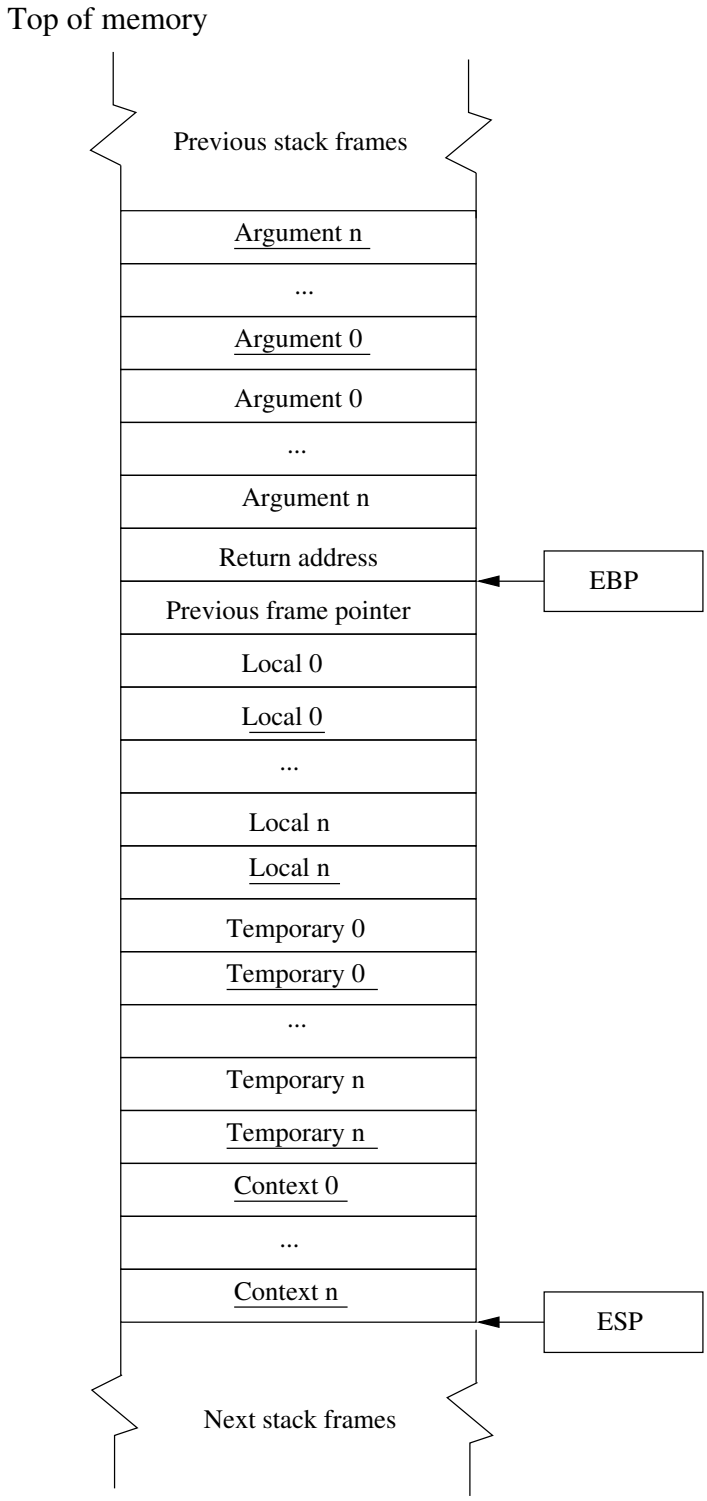


Figure 3.9: Trishul's stack frame holding taints, denoted by underlined names.

list would only have to be iterated once, but that would cause problems for native methods, which expect arguments to be pushed using the C's calling convention (which expects them to be pushed one after the other in reverse order). It would also pose problems for double-sized values, which occupy two consecutive fixed-sized argument slots.

The argument stack pointer, stored in a global variable, points to the taint values. This allows native methods to access the taint values. The taint values are pushed onto the stack in the opposite order from the argument values, allowing the taint values to be accessed using the index of the formal parameter as it appears in the C code, whereas the argument values are pushed in the opposite order, as required by C.

Taint values for local and temporary variables are stored as tuples of the form ⟨argument,taint⟩. Because it is not known how many temporary variables will be required before the full method's code is generated, it is not possible to determine the offset of a taint storage location if the taint values are stored following the variables. This would require two passes of the JIT compiler: one to generate the code and one to fix the offsets of taint values. Using the tuple approach, only one pass is needed, but, as with arguments, double-sized variables present a problem, as they require two consecutive slots. To handle this, the order of variable and taint values are reversed, so that two consecutive slots are used for the value and two for the taint values. Two slots are used for the taint value for a double-sized variable to make accessing slots easier. If a single slot was used instead, finding the proper slot for a variable would require scanning the list of variables to see if any double-sized variables precede the variable.

Context Taints

The Kaffe JIT compiler uses only six of the general registers present in IA-32 architecture to hold program values. As all floating point operations are implemented as register-to-memory operations, only FP0 floating point register is used by Kaffe. Both EBP and ESP registers are used only for bookkeeping and not for actual operations. Thus Kaffe needs to store only 7 register taint values (for the six general registers and FP0), each of which can be stored in a full SSE register. The one remaining SSE register is thus free to be used for storing the context taint, which can be thought of as the taint for EIP register.

The different parts that make up the context taint are stored in the stack

frame, as shown in Figure 3.9. Because the number of local and temporary variables is known at compile time, the offset of the context taint parts (relative to EBP) is known at compile time and can be fixed in the generated code. Entry and exit of basic blocks can be detected when code is generated by comparing the address of the instruction being generated to the addresses in the current basic block. If it is detected that a different basic block is entered, code to update the context taint is emitted.

Exception handling

The JIT compiler taints the required objects when an exception is thrown at the time the stack is unwound to locate the exception handler. Tainting these values is complicated by the fact that the code to handle nontaken branches is invoked at JIT-compilation time which generates machine code to handle tainting. The exception code is invoked at run-time, and must taint values directly. To this end, it uses information in the exception's stack trace to locate the run-time information generated by the load-time analysis and the locations of values that must be tainted. When the exception handler is invoked, the runtime simply jumps to the correct address, without an opportunity to initialise the partial context taint array with the exception's taint correctly. Therefore, this is handled when the last stack frame is unwound.

To handle non-thrown exception taint, an unused SSE register is used as a special taint register, allowing easy update of the taint value. When a method that may throw an exception is invoked, the current value of the taint register is stored on the stack and the register is cleared. When the method returns, the current taint value is stored in the partial context taint array and the original value is restored from the stack. As the instruction could throw an exception, it is treated as a conditional control flow instruction and the context taint is rebuilt before the next instruction is executed.

3.3.7. Trishul-P

The Trishul-P code compiler was implemented using a modified Java Compiler Compiler [JavaCC, 2009] and the policy engine was implemented to run within the JVM, allowing it to compare and match static properties of the method call, like the signature, and the dynamic taints of objects, parameters and context taints.

To match against run-time information as efficiently as possible, a two-stage matching strategy was used. During the first stage, which is invoked only when matching against a specific action for the first time or when the policy changes, the static information is matched. The result of this match is stored and reused whenever the action is executed again. The second stage matching, that compares the dynamic information of taints, is performed only when the first stage match is successful. This stage needs to be executed every time the action is executed. For example, when matching:

void java.io.PrintStream#<secretTaint>.println(String s)

the package, class, method names and parameter and return types are checked in stage one, since they are the same for every invocation of the method. However, as the taint values may be different for each invocation, it must be rechecked every time the method is invoked.

To handle loading and unloading of policy engines, a global iteration counter is maintained. This is initially set to 0 and increased every time an engine is loaded or unloaded. When a first-stage match is executed, the current iteration counter is stored with the match result. When an action is executed, the stored value is compared against the global value. A mismatch indicates that a new policy engine has been loaded or unloaded and the first-stage match must be executed again.

When a method is invoked, Trishul-P passes information on this action, including the values of actual parameters passed to the method, to the engine. This information is copied into the Action object passed to the engine, possibly converting primitive values to objects to allow them all to be stored in a single array. As an optimisation, the parameters are now only retrieved when they are accessed. To this end, a handle is passed in the Action object. The first time any parameter's value is accessed, the handle is used to create the array containing the value for actual parameters. How this is done differs between the interpreter and JIT compiler.

In the interpreted mode, as the interpreter has direct access to the objects that represent methods and variables at runtime, the matching is performed directly on the method object passed to the method call function. This object always represents the actual method that is invoked, which may not be the declared method in the case of polymorphism, making the matching process a simple case of comparing values. The actual parameters and the location used to store the return value that are passed when a

method is invoked, exist as simple objects on the interpreter's stack, as do the associated taint values. Therefore, they can be inspected and modified as required, to handle inspection of parameter values and replacement of the return value, as indicated by Trishul-P's Orders.

In order to perform matching in the JIT implementation, the object representing that method is required, as it contains the values that must be compared in the Trishul-P matching process. When a polymorphic method is invoked, this is not known, as only the address of the method's code retrieved from an object's dispatch table is known. The code layout was modified slightly in order to retrieve the method object correctly. When Kaffe generates code for a method, it generates a method header (which includes a pointer to the method object), followed by a variable length constant pool, followed by the actual code. This has been rearranged in Trishul so that the constant pool now follows the code. Thus a pointer to the method object is always available at a fixed offset before the code address.

A potential problem to this approach is the use of trampolines mentioned earlier–when a code address which is the target of a method invocation might actually contain a trampoline. Luckily, the trampoline already contains a pointer to the method's object, so this can still be retrieved. However, an extra check must be made to see if an address points to a code segment or a trampoline. This can be accomplished since a trampoline always starts with a jump instruction and a code segment always starts with a push instruction.

The actual parameters and taint values can be accessed using the argument stack pointer, as described earlier. The return value and taint value can be accessed since the register in which they are stored is known. To allow compatibility with the interpreter, the parameters and return values are copied into objects used by the interpreter if they are accessed.

3.3.8. Platform Integrity

The enforcement of policies using an architecture like that of Trishul involves the assumption first and foremost that Trishul is installed on the designated machine and that the install's integrity is protected.

The problem of ensuring that the software is installed on the machine and that the data to which the policy is attached is accessible only within the Trishul-based JVM is easier to solve in an application specific manner

than as a generic solution. Such a solution would be able to exploit the nuances of the application: closed or open systems, end-user or server-backend applications, etc. Examples of such solutions will be described in the next chapters. In general, these solutions would involve the use of trusted computing technology presented earlier in Section 2.4, exploiting the functionality provided by features like sealing and attestation.

Trishul has also been designed to make it hard for the attackers to compromise the integrity of the installation. As explained earlier, the effectiveness of the system depends on the application's inability to invoke native methods that have not been approved and packaged with the installation of Trishul. For this, a list of all allowed native libraries and its SHA-1 digests are stored securely in Trishul. Any attempt to load any native libraries not present in this list is denied. Furthermore, if the attacker tries to substitute one of the approved libraries with an unsafe one, the digest of this fake library would not match the digest stored in Trishul and any attempt to load it will also be denied.

Trishul is also designed to load a default policy enforcement engine that can be chosen by the system administrator. This default engine is then allowed to load others as and when required. In order to ensure that this engine, endowed with large privileges, is not subverted by an attacker, during the compile process of the JVM (when the engine's identity is specified by the administrator), the SHA-1 hash of the engine is also securely stored inside Trishul. At the start of every JVM instance, this hash is integrity checked to ensure that the default policy engine has not been removed or tampered with.

Now that we have introduced Trishul-P and explained in detail how Trishul is implemented in the interpreted and JIT modes of the JVM, let us look at some simple applications that provide examples of how Trishul-P is used and at the same time showcase the power of Trishul's functionalities discussed earlier. Larger application scenarios are discussed in later chapters of the dissertation.

3.4. EXAMPLE APPLICATIONS

Now that Trishul's architecture and the policy engine language has been covered, in this section we consider some examples of how Trishul can be used to solve policy enforcement problems that the current JVMs cannot.

3.4.1. Protecting system password file

Consider the scenario where an application wants read access to the */etc/-passwd* file of a UNIX/Linux system. Such an access request is normal and mostly legitimate, since information present in the file is used to perform several routine housekeeping operations in these systems. Since the actual passwords are not stored in plaintext in the file, the read operation by itself is not dangerous. However, consider an application trying to send out the information read from this file via the network to a remote party. There is almost always no reason why the information obtained from the file needs to be sent out into the network. An application which tries to do so could, for example, be trying to harvest user information in order to perform an efficient brute force password attack using known user names.

```
1 grant signedBy ``VU -CA" {
2   permission java .io. FilePermission ``/etc/ passwd", ``read";
3 }
4 grant codeBase ``file :/usr/share/java/repository /-" {
5   permission java . security . AllPermission ;
6 }
```

Listing 3.16: Java policy aimed at disabling leak of password file content into the network.

What is required is a policy setting that allows an application to read the content of the password file but prevents it from sending that information out into the network. Policies expressed in the form of currently supported Java Policy objects do not support this level of control. For example, if the policy in Listing 3.16 is used, it will allow read access to the password file but prevent the application from creating a socket connection to a host. But this is too broad a denial as it will also prevent the application from ever sending anything over the network, irrespective of the actual content that the application is trying to send. This is due primarily to the JVM's inability to trace the flow of information in the system and take access control decision at the flow level.

In order to prevent this, the enforcement system needs to ensure that the data read from the */etc/passwd* file is (1) tainted with a label (2) the label is propagated within the system alongside the data and (3) when attempt is made to send the data via the network, it is prevented. This can be done using Trishul with relative ease. Listing 3.17 shows the fragment of a Trishul-P code used to write such an access control engine.

```
1  private policytaint {
2    pwdF , netC
3  }
4  aswitch (a) {
5    case <* java .io. FileInputStream .<init >(.. , File f)>:
6      if(f. getName (). indexOf ("/etc/ passwd ") >= 0) {
7        return new ObjectTaintOrder (a. getThisPointer (),# object :{ pwdF });
8      }
9      return null ;
10   case <* *. Socket . getOutputStream (..) >:
11     return new RetValTaintOrder (# auto :{ netC });
12   case <* *. PrintStream #<{ netC }>. write (..# <{ pwdF }>)>:
13     return new ExceptionOrder (new java .lang . RuntimeException ("Leak !"));
14 }
15 return null ;
```

Listing 3.17: Trishul-P policy engine to prevent leak of password file information into the network.

It works as follows. The first action and associated Order (lines 5-7) taints the FileInputStream object with a label pwdF if the file being used for initialisation is */etc/passwd* while the second case statement (line 10-11) intercepts and returns a network socket with taint label netC. Trishul's underlying taint propagation mechanism would then ensure that any object that uses this socket would be tainted with the netC label while any data read from the FileInputStream object would be tainted with pwdF label. The third action in the policy engine file (line 12) checks for a call to the write method of a PrintStream object tainted with the netC label, which uses pwdF tainted data as argument. If the application invokes the method within these taint constraints, it is trying to send the data obtained from the '*/etc/passwd*' file via a socket to an external host. As a response Trishul returns an ExceptionOrder which in turn causes the JVM to throw a java.lang.RuntimeException exception.

A couple of things have to be noted with regards to the example mentioned above. In Listing 3.17, the name of the password file is fixed to make the code easier to read. In a real system, a SELinux [National Security Agency, 2009]-like policy structure would exist for every file that has a usage policy associated with it. The Trishul engine would query the related policy file first and then, based on the policy specified in that file, would perform the engine logic. Attacks like copying the password file to a new file and then reading this new file to perform network actions could be stopped in two ways (1) disabling the writing of the file's content into a new file or (2) carrying the policy of the original file to the new file. These are not shown in the listing above in order to keep the example code

simple. Canonical attacks targeted at the path filename of the file are not considered in this code fragment but a full policy engine would have to defend against them using well-know counter-measures.

It should also be noted that PrintStream.write() is only one of the possible methods that an application can invoke to write data to network. A comprehensive policy engine would use an abstract action that encompasses all possible methods that perform similar writes.

3.4.2. Multi-Level Security Systems

Multi-Level security (MLS) systems take inspiration from the defense community's security classification system. Most MLS computer systems use the Bell-LaPadula model [Bell and LaPadula, 1975] introduced earlier in Chapter 2, that proposes two main mandatory access control security properties. The *no read-up* property states that a subject at a given security level may not read an object at a higher security level, while the *no write-down* property states that a subject at a given security level must not write to any object at a lower security level.

Consider a CEO who has 'Top-Secret' security clearance. He has two files, one with classification of Top-Secret and another with classification Public, both of which he wants to write into. Current MLS system would require that the CEO open the Top-Secret file, edit it, close the application and change his current security level to Public by logging out of the system and logging in again with the lower clearance. Only then would he be able to open and edit the Public file. This is needed to prevent the CEO from copying content from the Top-Secret file and writing it into the Public file, which could then be read by anyone.

An MLS system implemented using Trishul JVM can avoid the need for the manual change of the current security level without compromising the security of the system. Trishul achieves this by preventing writes to an object only if its classification (Public) is lower than that of the content that is being written (Top-Secret). Thus the CEO is able to open and edit the Top-Secret file and the Public file simultaneously and even copy content from the Public file into the Top-Secret file but will be prevented from copying the data from the Top-Secret file to the Public file.

Listing 3.18 shows a portion of the Trishul-P engine code that was used to prototype such an enhanced MLS system in Trishul JVM. When an application tries to access a protected file on behalf of a subject, the invoked

method call (say java.io.FileInputStream) is intercepted and the control is transferred to the policy enforcement engine. The engine checks the clearance of the subject to access the file and if cleared, it labels the input stream with the security classification of the object (specified in a global system configuration file). Trishul then taints any data originating from this input stream with the stream's label and propagates the taint as the data gets used in the system. Later when the application tries to write data into an output channel (OutputStreamWriter), the engine throws a RuntimeException using ExceptionOrder if the output channel's security level label is lower than that of the data being written (confidential > public). Note that in the process above, only the specific instance of FileInputStream is tainted and a new FileInputStream created later will remain untainted, preventing taint spread. While our prototype system uses a custom configuration file to specify the clearance of the subjects and objects, a production system could use the information provided by a SELinux-like system file.

```
1 case <* java.io.FileInputStream.<init> String path,..)>:
2   oLabel = objectLevel (path);
3   switch (oLabel) { // confidential =5, public =1...
4   case 5:
5     return new ObjectTaintOrder(a.getThisPointer(),#object:{confidential},this,a);
6   ....
7   }
8 case <* *. OutputStreamWriter #<{publiclabel}>.write(..# <{confidential}>)>:
9   return new ExceptionOrder (new java.lang.RuntimeException (``Disallowed''),this
        ,a);
```

Listing 3.18: Trishul-P code fragment that implements the enhanced MLS system.

Note in this example that since Trishul's taint label system supports arbitrary lattice structure, it becomes necessary to explicitly code the logic of the structure within the policy engine. This is abstracted away here as a function in line 2. While this may seem cumbersome, the flexibility of an unstructured lattice allows the policy engine writer to support arbitrary decision logic, even those that do not use a lattice structure.

3.5. PERFORMANCE

As evident from the discussion on the implementation of Trishul-P, a lot of work goes into the creation of the CFGs, calculation of the context taint as well as the actual propagation of the taint labels. All these create

overheads when using the Trishul system. We investigate this additional overhead added by various parts of the Trishul system in this section.

Since Trishul itself is application independent, instead of comparing the performance of Trishul and Kaffe when running a specific application, in this section we concentrate on using microbenchmarks to compare the two. All tests were performed on single node of a four-node AMD Opteron system (model 852, 1Mb cache, 2593 MHz), with 1.5 GB of RAM. All performance measurements were taken using the JIT version in a release configuration and were compared against a standard Kaffe-1.1.7 release built using the same compiler options.

The overhead introduced by Trishul architecture can be categorised into three main components: (1) that due to the actual taint propagation mechanism as well as the dynamic calculation of context taints etc. (2) that incurred during the analysis of the bytecode to obtains CFGs, context bitmaps etc. and (3) that introduced by the hooks needed to examine the JVM's method invocations to intercept method of interest to the policy engine. The performance measurements were performed in such a way as to isolate these overheads.

3.5.1. Taint Propagation Overhead

The run-time overhead due to taint propagation was measured by observing the execution times of the inner loops of a prime number sieve and a file reader program. In order to measure only the runtime of the taint propagation mechanism and not the load-time analysis, the inner loop was executed twice and measured only the second time. The first execution ensures that all the required classes have already been verified and analysed. No policy engine is used for these benchmark applications to avoid the overhead introduced by the policy enforcement engine module.

Prime Number Generator

A prime number generator Java was used to test the performance of a CPU-bound application. It loops over the first 16384 integers and determines whether they are prime or not. As Table 3.3 shows, an overhead of 167% was observed when Trishul's performance was compared to that of the unmodified Kaffe. Most of the overhead can be attributed to the repeated recalculation of the context taint due to the tight for loops in the algorithm. In

Kaffe	Trishul	Increase
685ms	1828ms	167%

Table 3.3: Performance of prime number generator when run in Kaffe and Trishul JVMs.

Kaffe	Trishul	Increase
7.7ms	7.8ms	1%

Table 3.4: Time taken to read and print a 10Mb file when run in Kaffe and Trishul JVMs.

this example, for each outer loop (n = 2 to n = 16,384), an inner loop from i = 2 to i=sqrt(n) is calculated, leading to around 230000 (re)calculations of the context taints at the CFIs. We have identified ways to decrease this overhead, as discussed later in this section; but the implementation has been left as future work.

File Reader

This benchmark application measured the performance of I/O-bound applications. The application read a 10Mb file with randomly generated content, into a 64Kb buffer. The data is then printed to standard output, which is redirected to /dev/null. As Table 3.4 shows, a very low overhead of 1% was measured for this benchmark application. In this I/O application, major part of the run-time is spent on the actual reading of the content from the file as well as the writing onto the standard output. The time taken to do this dwarfs the extra overhead introduced by the taint propagation mechanism of the Trishul JVM, leading to an overall low overhead when the application is run inside Trishul.

Since typical real-world applications are likely to be neither fully CPU-bound nor fully I/O-bound, it is expected that the taint-propagation overhead for these applications will be somewhere in between these measures.

3.5.2. Load-time Overhead

In order to measure the overhead due to the load-time analysis, an application that prints a fixed date (1/1/1970) was executed in Trishul JVM. This application was chosen because it was noticed that its invocation forced a

Kaffe	Trishul	Increase
1052ms	1188ms	12.9%

Table 3.5: Runtime overhead due to load-time analysis of an application printing a specific date.

Context	254800 bytes
Non-taken branches	1668928 bytes
Total	1923728 bytes
No. of methods	986
Bytes per method	1951 bytes

Table 3.6: Memory overhead due to load-time analysis.

large part of the Java library to be loaded and therefore a large number of analysis to be performed and this is a good candidate for measuring the load-time overhead. For example, this specific application run caused 986 methods to be analysed.

Table 3.5 shows that Trishul's load-time analysis incurred a 12.9% overhead compared to Kaffe. Table 3.6 shows the memory that was required to transfer information from the load-time analysis to the run-time system. It was measured by recording all allocations of the objects that are used to pass this information; these objects are used exclusively for this purpose alone. On an average, 1951 bytes are required to hold all required information for a single method, the main part being the information on non-taken branches, i.e. the lists of variables that are modified in a CFI branch. Some optimisations that may reduce the size of these lists are discussed further on.

3.5.3. Policy Engine Overhead

A microbenchmark application that invoked a specific method 200,000 times repeatedly was used to measure the overhead introduced by Trishul-P's hooks. The run-time taken to execute actions specified in various policy engines were recorded. Table 3.7 summarises these measurements.

The first case statement in the policy engine code 'never matched, static' specified a method that was not invoked by the application at all. It also did not contain any taint comparison requirements and was discarded purely based on static properties of the method's signature. The second

Policy	Run-time (ms)
None	4.5
Never matched, static	4.6
Never matched, dynamic	31933
Matched, static	212842
Matched, dynamic	212245
Matched, dynamic, order	216000

Table 3.7: Runtime overhead due to Policy engine.

policy 'never matched, dynamic' specified a method which though was invoked by the application, was not matched due to the specified object taint being different at run-time. While in the first case the overhead was just 2%, the second matching process performed by the engine increased the runtime by 7100 times. This big increase is due to the fact that the dynamic properties are checked during the second phase of the two-stage matching process described in Section 3.3.7. In other words, the taint value needs to be rechecked every time the method is invoked, 200,000 times in this case. Matching on parameter taints and context taints show similar performance results.

The next two policies 'matched, static' and 'matched, dynamic' matches the method, either the static properties or the dynamic taint values. The larger overhead observed is caused by the work needed to hook into the policy engine: creating objects and arrays expected by the policy engine, installing the security engine for the policy engine, etc.

The last policy 'matched, dynamic, order' also returns a taint order and is used to capture the overhead of handling an order. When compared to the case where no order is returned (matched, dynamic), this increases the runtime by less than 1%. This shows that hooking into the policy incurs a lot of overhead, regardless of the amount of work done inside the policy engine.

The performance measurement suggests that the most efficient policies are the ones that hook into the policy engine as little as possible, and perform as much work as possible whenever such a hook is eventually made. Note however, that the performance reported here records a worst-case scenario. Such high overhead is not expected of normal applications, since, unlike the microbenchmark application which performs a very tight loop for 200,000 times with only one method being called in the body of the CFI block, they would spend more time calling other methods that may

not be of interest to the policy engine, and performing IO processes in its lifetime, decreasing the overall impact of Trishul-P's hooks.

3.5.4. Optimisations

The prototype implementation of Trishul, though as stable as Kaffe, has not been optimised due to lack of time and resources. Several possible optimisation and fine tuning of the prototype implementation has however been identified. Some of these are discussed here.

Currently the CFGs and context bitmaps of each method used by the application are generated at normal verification time of the bytecode. This overhead can be reduced by storing the calculated bitmaps and related information of the Java system libraries in a secure, integrity protected manner and reusing it the next time. Along similar lines, as of now Trishul creates its own CFG separate from the CFG used by the JVM's bytecode verifier due to earlier developmental constraints. However it is theoretically feasible to reuse the JVM's own internally calculated CFG, thereby decreasing runtime as well as memory overheads.

Some optimisation is also possible in the process of creating the lists of variables and objects that are modified in nontaken branches needed to handle indirect flows. For example, locations that are modified in each branch do not need to be tainted explicitly, nor do locations that are modified in the branch that is actually executed. Additionally, locations may appear in the lists multiple times; they of course need to be tainted only once. These optimisations would not only benefit the run time of the taint propagator, but also reduce the amount of memory required to perform the propagation steps.

As seen in Table 3.3, Trishul suffers a large overhead in the presence of tight loops found in CPU-bound mathematically intense application codes. As mentioned earlier, this overhead is mainly due to the repeated (re)calculation of the context taint for each run of the conditional branching involved in the code. This overhead can be reduced by exploiting the observation that if the arguments involved in the calculation of the context taint (i.e., the CFI's argument) have not changed their taint value in either the taken or the non-taken conditional branches, the context taint would also not have changed and hence need not be re-calculated. For example in the calculation of the prime number generator, the variables used with the CFI (n and i) are modified only once in the block where their val-

ues are incremented by 1 for each loop (i.e., $n++$ and $i++$). Since these operations do not change the values of taint labels $\underline{n}$ and $\underline{i}$, there is no need to re-calculate the context taint introduced by the CFI and hence can be skipped, saving over 230,000 re-calculations, reducing the overhead.

A re-examination of the way registers are used for storing taints could provide further optimisation in the JIT mode implementation of Trishul. Due to the nature of the SSE instruction set, currently chosen to hold register taints, accessing of individual elements is a slow process. Storing a single taint per register could provide an improvement over this. In addition, unused MMX registers (Kaffe uses only one for floating point calculations) can also be used to store the taints. Additional low level instructions available for manipulating these registers may make them more suitable for storing the taints.

The global context taint used in Trishul is an imprecise part of Trishul's architecture. The use of a better reaching-definition analysis algorithm should be able to reduce the number of times the global context taint fallback is invoked. In addition, several ways to automatically reduce the scope of the taint can be studied as future work. For example, if it is determined that all the variables that are control-dependent on the branch are assigned new values before the method returns, the global context taint needs to be increased only for that methods and can be reset after that method has returned. Similarly, if the effected variables are all members of a single object, the fallback taint could be limited to methods in that object.

As of now, the code to handle Trishul-P matches during method invocations is generated for each method. Every time a method is invoked, it must be checked if the policy engine's list of actions have changed, in which case the match must be performed again. If the list of actions change infrequently (i.e. no dynamic addition or removal of policy enforcement engines), it might be more efficient to use a different logic to regenerate the method hooks whenever the policy engine tree changes, thus removing the need to check if the tree has changed whenever a method is invoked. This would lead to quicker method invocations. Methods that are not inspected by the policy enforcement engine would incur only the overhead of regenerating the code, which in turn can be reduced further by regenerating only when the method is used again, which might not happen at all.

3.6. RELATED WORK

In order to certify software as complying to a static security policy, Denning proposed a compile-time approach to solve the implicit information flow problem [Denning, 1975]. In her system the compiler added extra instructions to the existing instructions of the application such that, irrespective of whether the CFI branch is followed or not, the class of object acted upon within the branch is updated to reflect the information flow. The approach relies on the properties of a lattice structure among the security classes and assumes that this structure is known to the compiler at the compile time, like in a typical confinement problem [Lampson, 1973; Lipner, 1975]. This restricts the class of security properties that such a system can be used to enforce. The wide range of security policies that current application scenarios present go beyond these static lattice confinement and confidentiality policies.

Volpano et al. [Volpano et al., 1996] later formalised Denning's work as a type system for which well typed programs respect the non-interference property. A type system is a set of rules used to check if a typing environment is compatible with a given program. These works have however been proposed as models or as a purely theoretical system whose proposed implementation depended on the use of specialised 'tagging' supported hardware for supporting tracing. Trishul is a practical system that does not rely on uncommon hardware support for its information flow control.

The system proposed by Andrews and Reitman [Andrews and Reitman, 1980] uses correctness proofs to establish the correctness of information flow constraints. This allows security classes to change at run-time, but is not applicable to practical systems as it is required that a program can be analysed as a single entity, whereas most software is developed as a set of modules. This approach can be extended to modular systems, as demonstrated by Mizuno and Schmidt [Mizuno and Schmidt, 1992] where they overcome this limitation by extending the work to work on modular systems by using a link-time algorithm to combine multiple modules.

Jif [Jif, 2009], the successor to JFlow [Myers, 1999] implements a compile time system by extending the Java type system to include security information. Jif introduces two new concepts into Java for information flow security: the labelled type and the switch label statement. A labelled type is a Java type annotated with an extended security class. A custom compiler then ensures that no information flow violation occurs by validating the

labels when values are assigned to the types. Generic labels may be used to write code that works irrespective of the actual security class, much like generics or templates allow code to work on variables of different types. The switch label statement allows a program to inspect a variable's actual label, allowing security decisions to be made. JFlow includes some dynamic features through the use of the decentralised label model [Myers and Liskov, 1997]. In this model, variables can be of type label. Value of this type can be used as near first-class values or as a label for other values. However they are only partially dynamic since variables of type label are immutable after initialisation.

Sabelfeld and Myers [Sabelfeld and Myers, 2003] survey several compile time static analysis IFC system, most of which are based on non-standard type systems like Jif. Type-based analysis in general are not flow, context or object sensitive, leading to higher false alarm. For example, consider the Listing 3.19. It is deemed unsafe by type-based systems because of the potential flow of information from confidential to public in the if...else block. The system does not capture the fact that any potential information flow is killed by the last assignment.

```
1 if (confidential ==1)
2   public = 42;
3 else
4   public = 17;
5 public = 0;
```

Listing 3.19: Flow that raises false positive in type-based systems.

The approach of using program dependence graph (PDG) in combination with constraint solving proposed by Hammer et al. [Hammer et al., 2006] in order to perform static analysis of Java program codes produce a graph similar in functionality to the one produced by our CFG approach. Our system however does not have the 'high' and 'low' levels built into the analysis, being lattice structure independent, and is thus more generic than theirs. As is the case with other compile-time approach systems, Hammer et al.'s system accepts or rejects a program based on whether information is allowed to leak from high to low level security classes. It does not support any run-time analysis, which means that like other purely static analysis system, the system gives judgement for *all executions of a program as a whole* and not *for a single execution alone*.

Fenton's Data Mark Machine [Fenton, 1974a] was one of the earliest systems that used the concept of run-time information flow control to en-

force policies. It adds data marks (fixed except for the program counter's data mark) to the abstract computer model of Minsky [Minsky, 1967]. It introduced the concept of adding a security class to the program counter ($\underline{pc}$) to handle indirect flows. When storing a value v to a fixed data mark storage location l, the machine checks whether the data mark of l is higher or equal to the upper bound of the data mark of v and the one of the program counters. If it is not, the operation is considered as a NOP. When storing v to a dynamic data mark location dl, the machine updates the data mark of dl to the least upper bound of data mark of v and that of the program counter, allowing for the program counter's data mark to monotonically increase at each conditional jump.

However, it has been shown [Fenton, 1974b; Guernic, 2007] that the proposed system is not able to handle implicit indirect flows like in Listing 2.2 when destination of flow has a dynamic mark, as the machine is not able to see the operation causing the flow and make necessary updates to the data mark of the dl. Since Trishul uses a combination of static and run-time analysis for handling indirect flows, the JVM is able to analyse the operation causing the flow, even in the presence of implicit indirect flows. Furthermore, as in Denning's system, the machine was considered as a purely abstract concept and no implementation was ever attempted.

The security mechanism proposed by Gat and Saal [Gat and Saal, 1976] tries to handle the indirect implicit flow but ends up preventing reuse of procedures due to its inability to store output of procedures in dynamic storage. Brown and King [Brown and King, 2004] proposes a similar system but it too has been shown to be unsafe for handling implicit indirect flows, see Section 2.2.1 of [Guernic, 2007].

Beres and Dalton [Beres and Dalton, 2003] use the DynamoRIO [MIT, 2003] framework to dynamically rewrite machine code in order to support dynamic label binding. The underlying concept behind the architecture of our system *Trishul* resembles that of this system with an important practical difference: instead of using a separate code modification framework, we make use of the interpreted nature of Java's bytecode instructions to perform dynamic tracing at runtime. Their system has the limitation that since it works at the machine code level, there is limited support for implicit information flows and it also assumes the existences of certain enhanced hardware support in order to perform the label tracing. Trishul has the advantage that because it operates at the Java bytecode level, the control flow and hence the information flow can be modelled much more

precisely. In their approach to non-taken branches the system either aborts the program or uses an approach similar to Trishul's global context taint mechanism. Since it is implemented completely in the JVM, Trishul requires no changes to the operating system kernel. Working at the low level of machine code also means that the system is not suitable for enforcing security policies that are rich in application semantic level restrictions.

The RIFLE architecture [Vachharajani et al., 2004] was proposed as a system that implements run-time information flow security with the aim of providing policy decision choice to the end user. They use a combination of program binary translation and a hardware architecture modified specifically to aid information flow tracking. Their work uses security registers to address explicit indirect flows, by capturing the data mark of registers conditioning the behaviour of the CFI and using it in addition to existing data marks for every instructions which is control-dependent on the CFI. This is very similar in concept to the use of context taint within Trishul.

The difference between RIFLE and Trishul occurs in the way implicit indirect flows are handled by the systems. In Trishul, the context taint is added to taint of all the instructions along the non-taken path of a branch. However, in order to prevent memory redirection problems inherent in the way instrumentation is performed in RIFLE at the binary level, the security register value is appended to the label of all instructions that potentially use values defined by instructions control-dependent on the branch. While this means that the append action is performed when the label is used rather than when it is defined (as in the case of Trishul), the strategy is proven to be safe [Vachharajani et al., 2004]. However the inherent truncation automata-like [Ligatti, 2006] behavior of dealing with insecure outputs create a new information flow that is not considered in their framework [Guernic, 2007]. Trishul-P's use of an edit automata-like[Ligatti, 2006] policy engine framework prevents such leaks by allowing the engine writer to handle such occurrences using SuppresOrder and InsertOrder. RIFLE also requires enhanced hardware architecture support for the binary-level information flow security instructions that need to be performed and hence can only be run using a simulator environment.

Chandra [Chandra, 2006] proposed a hybrid taint propagation approach for Java, similar to Trishul's but by instrumenting the bytecode with taint propagation code. One interesting aspect of the system is that the non-taken branch is not tainted until the context taint is untainted. Until the untainting of context taint happens any reference to the variables will al-

ways include the correct taint due to the inclusion of the context taint in the taint propagation process. The assumption is that untainting of context taint will happen infrequently, thus providing an optimisation over the process of tainting the non-taken branch every time, like in Trishul.

However, the approach used by Chandra has several shortcomings. For example, when considering native methods the return value of the native method is tainted with the method parameters' taints. This works only if the native methods are referentially transparent–the return value depends only on the parameters. In reality this is most often not the case. Furthermore the native methods can modify any value in the system and hence a more radical approach like the manual annotation system used by Trishul is required to fully capture the information flow in them. Exceptions are also handled incorrectly in their system–though throw statements are correctly identified as a form of goto, they ignore the effect of stack unwinding caused due to exceptions and the fact that method invocations can turn into conditional statements, leading to control flow attacks as explained in Section 3.3.6. Their work also does not implement any policy engine expression framework like Trishul-P nor is the architecture flexible enough to implement the range of policies that Trishul can.

Newsome and Song [Newsome and Song, 2005], Argos [Portokalidis et al., 2006], TaintBoch [Chow et al., 2004] and Haldar et al. [Haldar et al., 2005a] use taint tracing to track the use of untrusted data from potentially unsafe input channels, like networks. Haldar et al. [Haldar et al., 2005b] also attempt to extend this idea by using bytecode instrumentation to perform mandatory access control on Java objects, in order to enforce security policies. The level of granularity that is considered [Haldar et al., 2005a, b]–objects–is however too coarse-grained to be useful in many applications. For instance, they provide as an example a class method that tries to leak a secret file into a public file [Haldar et al., 2005b]. This is prevented by tagging the whole class instance as 'secret' as soon as the secret file is read and denying access to public channels once this tag has been set. The coarse nature of this tagging however prevents the class method from accessing any public channels even if the operation it wishes to perform is not on the data read from the secret file. Furthermore these systems are designed in general to solve specific application problems and do not consider the enforcement of general access and usage policies like Trishul nor provide any mechanism like Trishul-P to write policy enforcement engines for using the system in other application scenarios.

Le Guernic et al. consider an automaton-based dynamic monitoring of information flow for a single execution of a sequential [Guernic et al., 2006] and concurrent [Guernic, 2007] program. Like in our system, the mechanism is proposed as a combination of dynamic and static analysis allowing or rejecting a single execution of the program without doing the same for all other executions, unlike pure static system. The automaton is used to guarantee confidentiality of secret data and takes into account explicit and implicit flows. However, the work is purely theoretical in nature and while interesting theoretical results have been derived, no implementation has been attempted. Our IFC system can be seen as a generic implementation of this work. Instead of confining to a binary high-low system, our system allows for propagation of arbitrary taint labels which the policy enforcement engine can then use to implement the required guarantee. It is not far-fetched to assert that the policy enforcement engine of our system can be programmed to enforce the confidentiality high-low policy.

Le Guernic's work can thus be thought of as a theoretical treatment of a simplified version of Trishul. While formalisation and soundness proof of Trishul was not part of the work covered in this dissertation, Le Guernic's work provides a good starting point for such an effort.

Information-Based Access Control (IBAC) [Pistoia et al., 2007] has been proposed as an alternative to the traditional stack-based and history-based access control for ensuring that all codes that influence a security sensitive action is sufficiently authorised. The work also presents a mechanism to convert an access-control policy into an implicit integrity policy in order for IBAC to enforce it. While the work proposes the use of static as well as dynamic enforcement of IBAC using PDGs, no implementation is reported. Unlike our system, the proposed IBAC system addresses the specific problem of code security in Java and .Net Common Language Runtime and does not provide a generic policy enforcement system.

Xu, Bhatkar and Sekar [Xu et al., 2006] use a notion of taints, similar to taint mode in Perl [Wall, 1987], to track dangerous data that originate from user in order to prevent execution of bad data and prevent attacks like SQL injection. Their source code analysis of C takes into consideration direct flows and some indirect explicit flows but do not consider indirect implicit flows as they assume that such an analysis is not necessary for the kind of attacks they aim to prevent. Lam and Chiueh [Lam and Chiueh, 2006] proposed another similar framework for dynamic taint analyses but again does not take any indirect flows into consideration.

Polymer [Bauer et al., 2005] is a general purpose policy engine for Java that rewrites the application bytecode as well as instruments the system libraries to enforce security policies. While Trishul-P's language syntax is inspired by Polymer, the implementation is a complete rewrite since Polymer worked as a Java-Java compiler system that rewrote the system libraries independently of the specification of the security policy and application bytecode as per the policy specification, both permanently. Policy enforcement engines written in Trishul-P on the other hand is compiled into Java code and then into Java bytecode, which in turn uses the hooks in place in the Trishul JVM system to interposition itself between the method calls at runtime. One consequence is that the system libraries need not be instrumented outside the run of the JVM against a static set of method calls specified in Polymer's *action declaration file*. Furthermore, Trishul-P also supports the ability to introduce taint labels into the system (the various taint Orders), something which was not considered at all in the Polymer system. Polymer, being an execution monitor, also does not address the information flow problem inherent in process of enforcing security policies.

The side-effect of using a Polymer-like structure for Trishul-P is that the policy engine can be informally thought of as an *edit automata* based monitor, which is proven to enforce a much wider class of properties [Ligatti, 2006].

Viega et al. [Viega et al., 2001] has proposed the use of aspect oriented programming to security by using an aspect language to specify security transformations on a program. At compile-time, their language takes any specified aspects along with regular C program and weaves them into a single C program which is then compiled. The aspect language is similar to Trishul in that it supports wildcards, allows for insertion of code before or after point of interest or replace the code at the given point of interest. The aspect language is more similar to Polymer than Trishul as it does not support for specifying taint labels or introduction of these taints into the system. In fact information flow is not at all considered in the system.

One aspect that has not been considered in the design and implementation of Trishul is that of multi-threading in programs. Enforcing information flow control when dealing with synchronisation and concurrency issues brought on by threading can lead to subtle information flow channels that are difficult to capture. Synchronisation commands may prevent some output sequence from occurring and when the execution of such a synchronisation command is conditioned by a tainted data, the value of

the data may be revealed whenever the program outputs a sequence which cannot occur if synchronisation command is executed. Compile-time solutions to the problem has been proposed [Barthe et al., 2007; Roy et al., 2009] but a dynamic runtime monitor-based solution has not been implemented as of yet. That being said, the theoretical treatment of the problem for runtime monitors presented in [Guernic, 2007], even though restricted to high-low lattice systems, offer a possible route towards implementing such a solution.

There have been several work done in designing system call interpositioning architectures. It has been used in recent years for addressing both the confinement problem [Acharya and Raje, 2000; Goldberg et al., 1996] and intrusion detection [Wespi et al., 2000]. Janus, for example, originally prototyped by Goldberg et al. [Goldberg et al., 1996] and later implemented as a loadable kernel module, provides one such mechanism to restrict the application's interaction with the underlying operation system at the level of the system method calls performed by the application. However, hardly any of these dynamic systems have been built with the intention of supporting information flow control. That said, the lessons learnt [Garfinkel, 2003] from building such interpositioning systems, like protecting against canonical attacks, can be equally applied in the implementation of the decision logic within Trishul's enforcement engine.

3.7. CONCLUSION

In this chapter we presented the design and implementation of Trishul, the information flow based policy enforcement architecture that forms the core of this dissertation work. In particular we discussed in detail the Trishul-P language syntax which can be used to write modular policy enforcement engines and the Trishul Java Virtual Machine capable of supporting tracing of information flow caused by both direct and indirect flows. Design decisions to handle complications arising from control flow instructions, exceptions and native methods are also discussed.

In order to have a better idea of how the system works, we then looked at how Trishul can be used in solving some small application scenarios like protecting the password file of a Unix platform and implementing an enhanced Multi-Level Security (MLS) system.

We then demonstrated through performance measurements that Trishul,

though burdened by the extra effort involved in the load-time analysis and the run-time taint propagation mechanism, is still a practically usable system. The measurements helped us identify that the major overhead is introduced by the taint pattern matching steps associated with the enforcement engine hooks, in particular those that deal with dynamic taint label patterns.

Finally, we also presented various optimisations that have been identified which could potentially improve the performance of the Trishul system by a large factor. These form part of future work along with considering other possible optimisation avenues.

In the next two chapters we consider the use of Trishul in implementing larger applications in order to prove its usability in more real life scenarios.

CHAPTER 4

Application: Digital Rights Management

Now that the design and implementation of Trishul has been presented, we consider an application scenario for the architecture.

In this chapter we present Trishul-UCON (T-UCON), an implementation of a Digital Rights Management (DRM) system based on the UCON_{ABC} usage control model, built using the Trishul system. T-UCON is designed to be capable of enforcing not only application-specific policies, as most existing software-based DRM solutions do, but also DRM policies across applications. This is achieved by binding the DRM policy only to the content it protects with no relation to the application(s) which will use this content. Since T-UCON is implemented as a JVM-based middleware that mediates the usage requests of any Java application to the protected content, it can be used to enforce the guarantee that the usage policy is continuously enforced. Each request is granted or denied as per the rules laid down by the usage policy of the content. We illustrate the unique features of T-UCON by using typical examples of DRM policies such as the *pay-per-use* and the *use only N times* scenarios. Preliminary results on the overhead of our solution are also provided.

4.1. INTRODUCTION

As more and more digital content is being distributed online, the owners of this content are increasingly relying on digital rights management

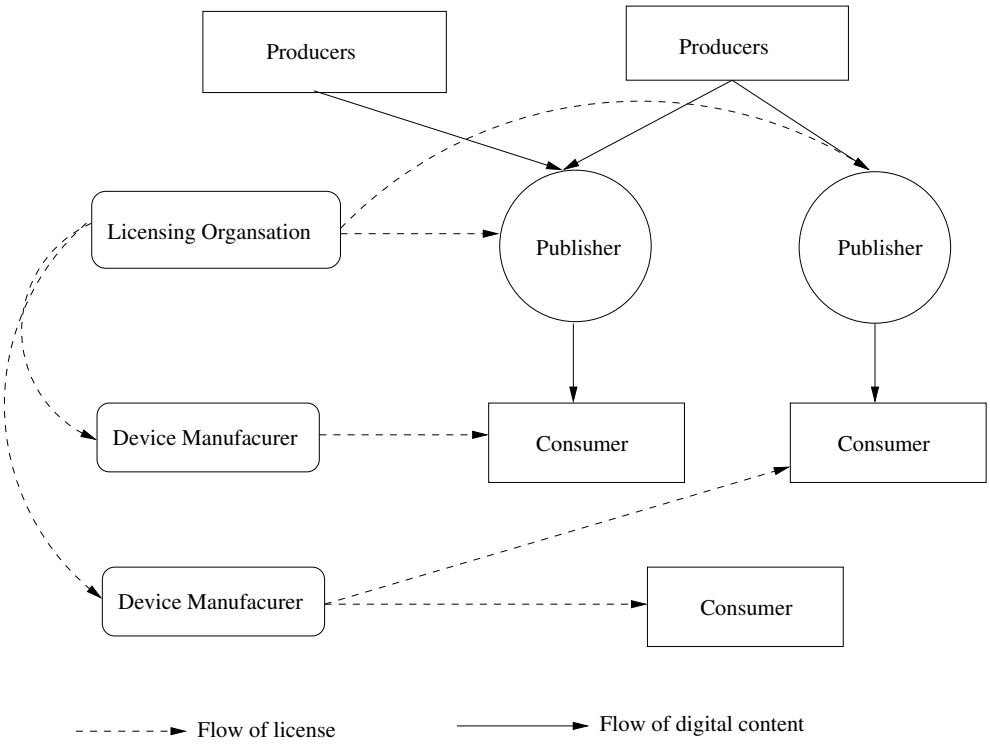


Figure 4.1: Stakeholders involved in a typical DRM setup.

systems to help in monetising the content. These systems manage the various usage aspects of the content. For example the content could be restricted as to how many times it can be played or in the case of digital redistribution, how many times the content can be resold.

After being in the news for some years now, DRM is currently approaching a more mature phase, gradually attracting a steadier research community. This trend is partially reflected in the industry too. Despite less emphasis compared to the early heady days, there are still many companies [Jobs, 2007] and industry alliances [OMA, 2009] highly interested in flexible, cheap and secure DRM technologies. This is motivated by the indisputable fact that an increasing amount of digital content is produced everyday and there is an overwhelming desire to protect both its distribution and consumption.

A generic DRM setup is composed of three prominent stakeholders and two less prominent ones as shown in Figure 4.1.

The three prominent stakeholders are:

- Producers: The producers are the entities that own the rights to the content. They form the starting point of the DRM chain and heavily influence the business model associated with the content. They could be individual artists, bands or record labels (like EMI, Sony, Time Warner etc.)

- Consumers: The consumers occupy the other end of the DRM chain and are made up of individual users who wish to obtain the digital content and consume them as per the restrictions laid down by the producers. It is assumed that the consumers can access protected digital content only by means of compliant devices. Compliant devices, by definition, enforce the policies set by the producers.

- Publishers: The publishers are responsible for managing and running the DRM network used to distribute the digital content to the consumers. They form the middleman between the producers and the consumers. iTunes is an example of a publisher.

The two less prominent stakeholders involved in the framework are:

- Device manufacturers: These are the manufacturers of certified compliant devices. It is assumed that there exists a industry-wide set of specifications that define the minimum capability of these devices, much like the specifications of the Open Mobile Alliance [OMA, 2009].

- Licensing organisation: This central entity is responsible for testing the devices made by the manufacturers and (digitally) certifying the manufacturers as complying to the industry-wide set of specifications. It is trusted by all parties involved in the system and its public key is embedded in all compliant devices and forms the root of trust for digital certificate chains. It is also responsible for certifying publishers as being part of the DRM system.

While a lot of work has been done at the cryptographic protocol level defining the interaction of the various players with respect to each other, less work has been done in examining the actual enforcement of the DRM policies at the consumer side, on the compliant devices. In this chapter

we try to address this shortcoming by considering the design of a policy enforcement architecture for the DRM system.

Broadly speaking, existing DRM solutions can be classified as hardware-based and software-based. Trying to determine which one is better in terms of security alone could be misleading since each of them serves different needs. In practice the main discriminant between the two is the business model of the distribution system rather than their actual security strength.

Hardware-based solutions (e.g. Zune, iPod, etc.) present closed systems consisting of *compliant* devices that are, by construction, made to conform to the DRM specifications. The security of these systems rely on the impossibility to *fake* a compliant device and on the admission control protocol that allows only compliant devices to interact with other compliant devices. An advantage of these solutions is the simplicity of the design, but the disadvantage is the cost of the device and the infrastructure. Building devices impossible or hard to fake or break is expensive. Thus in practise an acceptable compromise between manufacturing costs and estimated loss of revenue due to DRM failure is often considered in deciding which approach to implement.

On the other hand, software-based solutions, like iTunes Fairplay, are cheaper and more flexible since they do not require special hardware and they can share a computer with other non-DRM applications. These solutions build a software-protected environment (e.g. player, reader, etc.) within which (and only within which) the protected content can be consumed. Other applications cannot access the protected content since it is typically encrypted with a decryption key embedded in the protected environment. Software-based solutions are secure, assuming the operating system is trustworthy and that it is hard to extract the encryption key embedded in the software. These software-based solutions are the best choice in all those scenarios where the content provider does not have control over the hardware used by the consumers, but it is still in its own interest to make the DRM content available to as many users as possible.

Despite being more flexible than hardware-based systems, current software based solutions still suffer from many drawbacks that limit the type of DRM policies they enforce. Due to their design, existing solutions cannot, for example, implement cross-application DRM policies. Thus typical *use only N times* policies like '*play the song "Imagine" no more than 5 times*' cannot be enforced. Rather, what is now enforced are policies like '*play the song "Imagine" no more than 5 times using 'ThisPlayer'*,' thus bind-

ing the policy to a specific application. Similarly, *pay-per-use* policies like '*the cost of playing the song "Imagine" is 50 cents the first 10 times, then 5 cents for the next 10 times and 1 cent for the next 100 times*' are impossible to implement if one tries to enforce them at the song level rather than for a specific application.

While there has been previous work done on modelling DRM architectures and associated policy languages [OMA, 2009; Park and Sandhu, 2004], there are fewer implementations of such systems. In this chapter we present the design and implementation of T-UCON, an open and generic software-based architecture that enforces DRM policies. In particular, we will show how T-UCON can be used to enforce DRM policies both application specific and more importantly across applications. After providing solutions for the two examples mentioned above we show that T-UCON also enforces DRM policies that use *obligations*. Preliminary performance tests confirm the feasibility of our approach.

4.2. MODELLING DRM

A formal model for the DRM system goes a long way in ensuring that all the various usage restrictions that can be specified as part of DRM policies can be captured correctly. It has been argued that the notion of DRM is more than a set of enabling technologies and that it overlaps a lot with the notion of access control and usage decision models [LaMacchia, 2002]. At the same time, traditional access control models (MAC, RBAC etc.) do not capture the requirements of modern DRM application scenarios.

The UCON_{ABC} model [Park and Sandhu, 2004] provides such a model. In this section we briefly recap this model, first introduced in Chapter 2, and explain how it can be used to model various DRM scenarios.

4.2.1. The UCON_{ABC} Model

Historically the UCON_{ABC} model was introduced as an extension of the traditional access control model in order to take into consideration various missing requirements needed to model the wide range of usage restrictions seen in real-life scenarios.

The main components of the UCON_{ABC} model, as shown in Figure 4.2, are the *Subjects*, which wishes to assert various *Rights* over certain *Objects*.

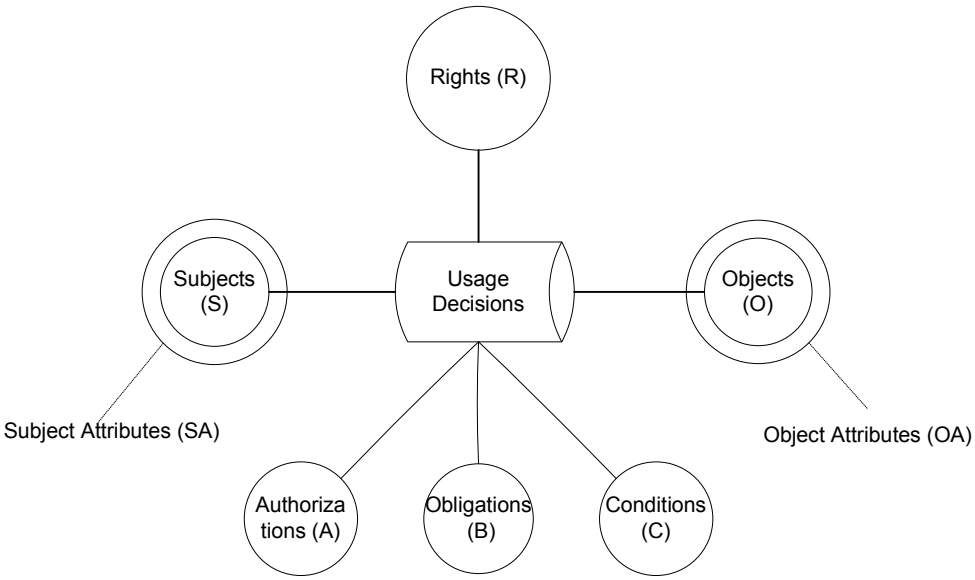


Figure 4.2: Components of the UCON_{ABC} model.

Subjects and objects are endowed with *Attributes* that capture the properties and/or capabilities of these components.

The usage decision on the requested rights by the subject on the object is made based on 3 factors: *Authorisation* (A) where attributes of subjects and objects are checked in order to make decisions on whether the subject is authorised to access the object, *oBligation* (B) where checks are performed to ensure that certain actions are performed by the subject and *Conditions* (C) where environmental (system) attributes are checked to see if they are in a predefined secure state.

The generality required by complex usage control scenarios is achieved by adding the notion of decision continuity and subject and object attribute mutability to the model. The continuity property is added to the decision phase by allowing the decision to be made before the usage is permitted (*pre*) or during the usage session (*on*), while the attributes mutability is supported by allowing them to be updated before (*pre*), during (*on*) or after (*post*) the usage has been granted. These notions of usage decision continuity and attribute mutability addressed by UCON_{ABC} enable it to meet the requirements of the generic DRM model laid down by Erickson [Erickson, 2003]:

– Use of information resource by user: this forms the very premise of usage control models

– Implementation of control: while [Erickson, 2003] considers authentication, metadata and proprietary infrastructures for data distribution, identification and cryptography, the $UCON_{ABC}$ model (being a generic system model) does not cover in specific detail how to address these issues. However, it does provide basic control mechanisms to achieve control based on the various forms of checks and attribute updates

– Set of policies for controlling use of resources: In both the DRM model and $UCON_{ABC}$ every action on a resource is governed by a corresponding set of checks which form one or more policies

– Fixed or built-in policies: refers to how policies are attached to the content they are protecting. Neither $UCON_{ABC}$ nor DRM models address this issue, but in both cases the aim is to cryptographically ensure that the policy is inseparable from the object being protected.

4.3. TRISHUL-UCON ARCHITECTURE

Trishul-UCON was designed and implemented on top of the Java Virtual Machine architecture provided by Trishul and hence can work for all Java applications. Unlike traditional DRM solutions that restrict policy enforcement to specific applications, T-UCON is therefore capable of enforcing policies independent of and across Java applications. This is done by associating the DRM policies to objects, mediating any access to these policy-restricted contents using the T-UCON system and by capturing the state of the system across application runs in the object and subject attributes.

Figure 4.3 provides a high level overview of the various components of T-UCON and their relationship with each other. The Policy Enforcement Point (PEP) intercepts Java method calls made by the applications that are of interest to the DRM system. Once the relevant calls are intercepted, the control is passed to the Policy Decision Point (PDP). It is the responsibility of the PDP to decide whether the application call can be allowed to proceed or not. To make this decision, the PDP consults the policy associated with

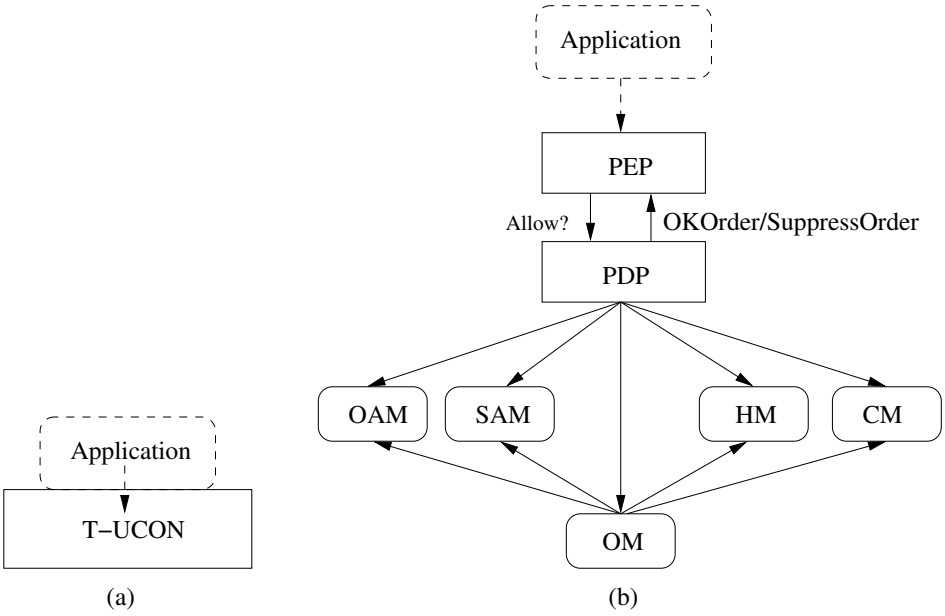


Figure 4.3: A schematic representation of (a) T-UCON intercepting application methods calls and (b) various components of the Trishul-UCON architecture.

the object through the Object Attribute Module (OAM). The OAM also provides an interface to query and update the object attributes. The Subject Attribute Module (SAM) provides an interface for querying and updating the subject attributes, while the Condition Module (CM) provides a similar functionality for the system attributes. Once a decision has been made by the PDP on whether to allow the action, it is communicated to the PEP. The PEP forwards this decision to the JVM, which then halts the action by throwing an exception, if needed. The Obligation Module (OM) is tasked with enforcing obligations in the policies while the History Module (HM) provides logging and log querying capability to the system.

In the rest of this section, we look at each of these components in detail.

Policy Enforcement Point (PEP)

The Policy Enforcement Point is the central point where the application and the policy enforcement mechanism of T-UCON intersect.

When T-UCON is launched, the PEP registers all the Java method calls

(actions) that are of interest to the DRM system. For generic DRM scenarios, these include the file open and read method calls which need to be mediated and any network based calls which are denied by default.

When an application tries to execute any of the restricted methods, the PEP intercepts it and passes control to the PDP, sending along all the available information regarding the method call. Once the PDP makes a decision on whether to allow the call or not, the decision is passed to the PEP, which enforces the decision by letting the application call to proceed or by terminating the application run.

The PEP is implemented using a combination of the Trishul JVM hooks that intercept the method calls at runtime and Trishul-P language based enforcement engine that lists the signature of these method calls. Exploiting the flexible engine hierarchy inherently provided by Trishul, T-UCON starts off with a default policy enforcement engine which can be extended by loading additional engines as the need arises.

Policy Decision Point (PDP)

Once an action is intercepted by the PEP, it is passed on to the PDP. The PDP implements the core of the decision logic of the decision engine and is responsible for ensuring that the required authorisations, obligations and conditions are met for the method call to proceed and if they are not, to disallow the action.

The PDP is responsible for interpreting the object policy and enforcing the various constraints associated with the usage of the object. T-UCON by design does not support one specific Rights Expression Language (REL) for expressing the object policy, instead it allows the PDP to load individual expression language parsers to support any standard RELs like XACML [OASIS, 2008], and ODRL [ODRL, 2002] or even proprietary ones.

The PDP is written in Java and uses the Trishul-P Order object to pass back the decision to the PEP. Since the SuppressOrder and ExceptionOrder are implemented in Trishul as Java exceptions, a T-UCON aware application, knowing that a method call it is invoking is a restricted call, like opening an MP3 file, could be written in such a way as to catch and handle the exception. A T-UCON unaware application on the other hand is terminated when SuppressOrder and ExceptionOrder is received. Since HaltOrder is handled by Trishul as an exit call which halts the application and does not

allow the application to recover, it should be used only in extreme circumstances.

The rest of the helper modules of T-UCON: OAM, SAM, OM, HM and the CM, are implemented as normal Java classes and invoked by the PDP as required in the normal Java style, in order to handle the various DRM policy scenarios.

Object Attribute Module (OAM)

The OAM provides an interface to query and update object attributes. The OAM can be independently called by the PDP and the OM as both these modules need access to the object attributes. This modular design allows the OAM to rely on the PDP and the OM to initiate the pre/ongoing/post updates to the attributes while freeing them from having to interpret the syntax of the attribute specification.

As the policy associated with the object is considered as an object attribute, the OAM is also designed to query this information. It should be noted however that the OAM itself is not responsible for interpreting the policy nor the state of the object as stored in the attributes. These are still the responsibility of the PDP. For the current prototype implementation the object policy is assumed to be placed at a fixed location /etc/tucon/object_id.xml where object_id is a unique identifier of the DRM object. The policy is expressed in an ad-hoc XML format, an example of which is shown in Listing 4.1, though support for other formats can be easily added.

```
1 <objAttri>
2    <song>
3       <classification>level 2</classification>
4       <value>15</value>
5        <usageNum>3</usageNum>
6        <role>pRole</role>
7    </song>
8 </objAttri>
```

Listing 4.1: Example Object policy.

Subject Attribute Module (SAM)

The SAM provides a similar functionality to query and update the subject attributes. These are again invoked by the PDP and the OM when they need to perform pre/ongoing/post attribute updates. In the current implementa-

tion, the attributes implemented are those required by the pay per-use, use n-times and metered payment DRM scenarios, as explained in detail later.

Condition Module (CM)

System attributes are queried using the CM. These include the date, time and other system variables that can be considered as *conditions* in the UCON_{ABC} model. The CM contains platform dependent codes that provide the relevant system information to the query. Since system attributes should not in general be changed, the CM does not provide the functionality to update these attributes.

History Module (HM)

Many DRM policies require history based decisions, since they typically span across several usage sessions of the object. Such history based decisions are often associated with obligation policies that could potentially need to check if particular actions were performed by the subject before certain rights are allowed. We implement the history by associating with each object a state that is global with respect to the applications and time.

The History Module provides two distinct functionalities: a convenient mechanism to log events/actions that have occurred and an efficient mechanism to query these logged events. As seen from Figure 4.3, the HM is called from the PDP as well as the OM to log and query actions, associated decisions as well as any other relevant checks performed prior to making the decision.

In order to provide an efficient service, the current implementation of the HM logs only the essential details including a timestamp, the action identified by the intercepted method's name and parameters, the identity of the logging entity (PDP/OM), the decision returned (in case of PDP), the state of obligation requirement (in case of OM) and the identity of the object. The query functionality accepts queries based on the action name, while additional constraints can be specified on the parameters, logging entity as well as the time period. The query response contains the timestamp of the matched query, the decision returned as well as the identity of the object.

Obligation Module (OM)

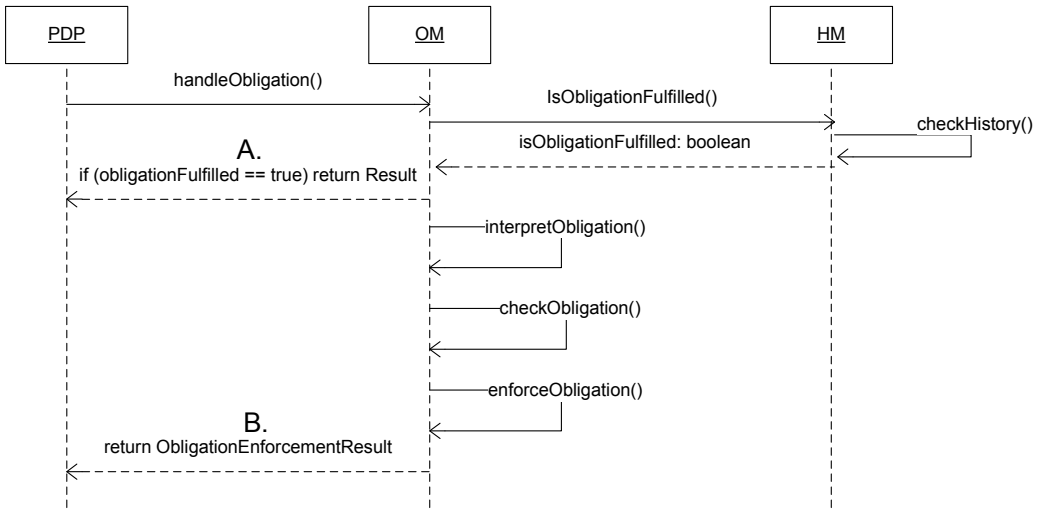
In the model, obligations are actions that are required to be performed by the subject before, during or after the usage of an object (e.g. subject must accept the license agreement before using the software application).

While most of the authorisation and condition requirements are sufficiently straightforward for the PDP to check and enforce by itself, the complicated nature of obligation requirements warrants a dedicated enforcement module: the Obligation Module (OM). When the PDP encounters an obligation requirement (e.g. user needs to accept the license agreement) in the object policy passed to it by the OAM, it passes the obligation part of the check to the OM for handling. The OM implements all the logic needed to enforce the obligations requirement. Such a modular architecture allows all the logic required to interpret, check and enforce the obligations to be completely contained within the OM.

Currently, we have only implemented a subset of possible obligation types. In particular, we do not deal with obligations that require calls external to the applications. For example, if a service requires that the subject has a digital certificate, the system does not proactively make the external call to obtain the certificate. Rather it checks if the certificate is present in the system and if not, disallows the access.

Figure 4.4 provides an overview of the obligation enforcement process. The PDP, on encountering an obligation in the usage policy invokes the OM to handle the obligation, passing it the associated policy fragment. The OM first checks whether the obligation has already been fulfilled by querying the HM. If it has, the OM returns a $true$ value to the PDP which then continues to process the rest of the usage policy restrictions. If the HM query turns up $false$, the OM interprets, checks and enforces the requirements of the obligation and sends back the result of the enforcement action back to the PDP. Based on whether the enforcement by the OM was successful or not, the PDP continues with the rest of the restrictions or terminates the resource usage.

If the policy specifies an ongoing obligation, the OM registers a timer with the PDP, associating it with a unique identifier to identify the specific obligation. When the timer fires, the PDP passes the control back to the OM to check for the obligation compliance.



A. prior-to-usage obligation checking
B. prior-to-usage attribute update added to case A., by calling attribute modules

Figure 4.4: Working of the Obligation Module of T-UCON.

4.4. ENFORCING DRM POLICIES

In this section we look at some typical DRM scenarios and explain how the architecture has been used to develop prototypes that implement such scenarios.

As with the generic UCON_{ABC} model, a DRM system consists of two main entities: subjects and objects. Subjects are users or applications (e.g. a multimedia player) being executed on behalf of the user. Objects are content files, such as music or video files, whose access and use are subject to various restrictions (when, where or how they can be used).

Subject attributes are properties and capabilities associated with the user that allow him/her to exercise rights over objects. Attributes relevant to DRM systems include credit card details, prepaid credit balance and similar financial details of the user. Object attributes are properties associated with object's usage, like the cost of the media file, meta-data, like artist name, bitrate of the MP3 file, etc. Properties like the remaining play count and age of the file are also considered as object attributes in our system. Conditions are used to express environment variables (e.g., date, time) that could be used to evaluate DRM policies. As in the original model obliga-

tions express actions that must be executed to use the object (e.g., accept the license first).

4.4.1. Pay-per-use

Conceptually a pay-per-use service is one of the simplest DRM scenarios. An object (o) has a value associated with it, which forms its attribute $ATT(o)$. The subject's (s) credits form his attribute $ATT(s)$. The policy associated with the object states that every use right (R) (say view) of the object requires the value of the object to be decremented from the credit balance of the user. In order to implement this scenario the authorisation for use of the object is checked before the usage is allowed and the mutable subject attribute is updated as a *pre*-update process. Using UCON_{ABC}, a generic pay-per-use policy can be modelled, using the notation in [Park and Sandhu, 2004], as:

M is a set of monetary amounts
$credit : S \rightarrow M$
$value : O \times R \rightarrow M$
$ATT(S) : credit$
$ATT(O,R) : value$
$allowed(s,o,r) \Rightarrow credit(s) \geq value(o,r)$
$disallowed(s,o,r) \Rightarrow credit(s) < value(o,r)$
$preUpdate(credit(s)) : credit(s) - value(o,r)$

The $value(o,r)$, specified as its object attribute, could be a static value like \$0.50 per play or could change over usage, being 50 cents for the first 10 times, 10 cents for the next 10 and 1 cent from then on.

When the application, on behalf of the user, tries to perform a file-open operation on the object, the PEP–in the form of Trishul–hooks intercepts the action for mediation in order to enforce the policy. The intercepted action is then forwarded to the PDP, which then queries the OAM to check the exact policy associated with the object. After interpreting the policy, the PDP queries the OAM again to read object attribute *value* and the SAM for the subject attribute *credit*. It then decides on the authorisation based on the the value of *credit* and *value*. Figure 4.5 shows the details of the steps involved in the process.

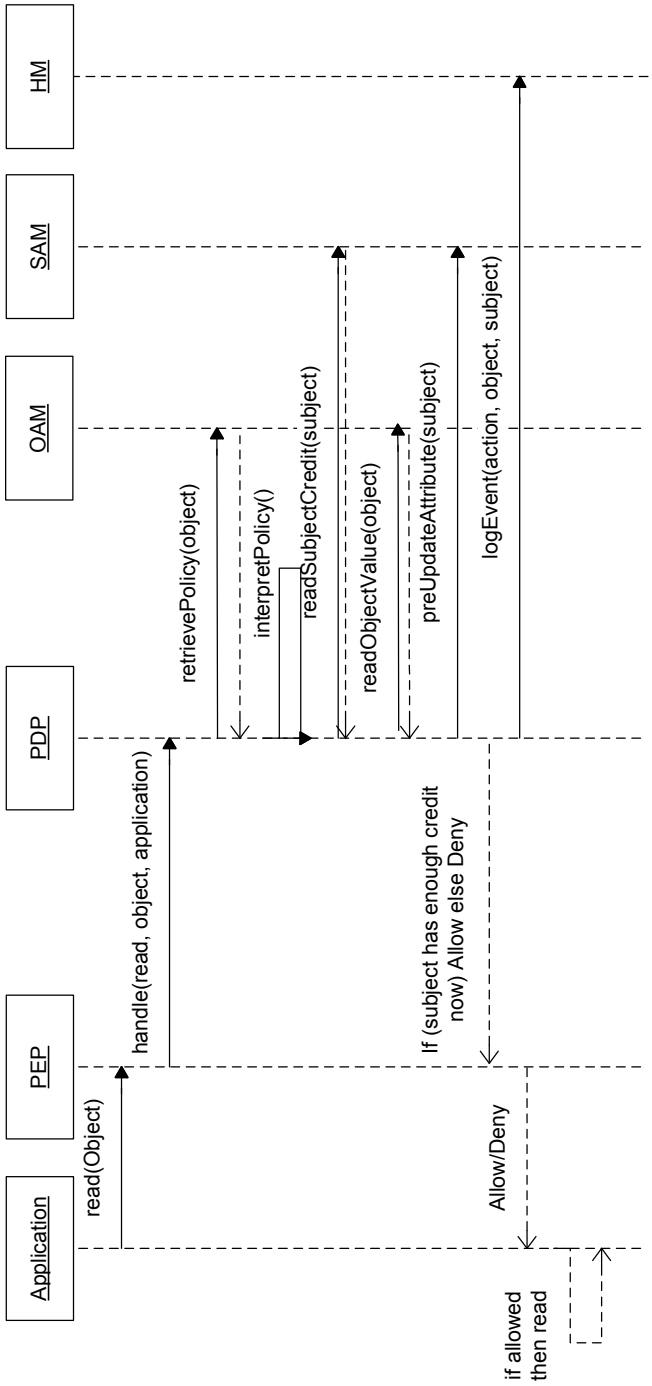


Figure 4.5: Implementing pay-per-use policy using T-UCON.

4.4.2. Use N times

Next, let us consider the often-discussed DRM policy of 'use only N times'. Of course, the semantic of 'use' is application dependent. In our example it is "play." This scenario can be complicated by the practice followed by many content providers of letting anybody play a certain percentage (say 50%) of the song as a means of advertising the song for free. We support this feature too in our model. So a "play" action counts as such only if at least 50% of the song has been played (read from the file). In UCON_{ABC} such a policy can be expressed as:

B is number of bytes
N is an integer
$size : S \rightarrow B$
$played : S \rightarrow B$
$plays\_left : O \rightarrow N$
$allowed(s,o,r_1) \Rightarrow plays\_left > 0$
$disallowed(s,o,r_1) \Rightarrow plays\_left \leq 0$
$allowed(s,o,r_2) \Rightarrow true$
$postUpdate(played(s),r_2) : played(s) + read$
$postUpdate(plays\_left(o),r_2) : plays\_left(o) - 1$; if $played(s) > size(o)/2$

Where r_1 is the *open* right and r_2 the *read* right and in this example O is an audio file. The last line of the listing is the one which decreases the count of plays left if the amount of file read is more than half the size of the file.

On interpreting the policy, the PDP queries the OAM for the objects' use_left attribute. The action is allowed if this value if greater than 0. This implementation has the limitation however that once the use_left has reached 0, the object can never be opened, even for providing a preview of the content. If such an access is to be allowed, the following logic is used instead:

$allow(s,o,r_2) \Rightarrow use\_left > 0 \vee read(s) < size(o)/2$
$disallow(s,o,r_2) \Rightarrow use\_left \leq 0 \wedge read(s) > size(o)/2$

The process-flow involved in implementing this modified policy using T-UCON is shown in Figure 4.6. As these steps show, T-UCON performs an update of $plays\_left$ object attribute irrespective of which application

is playing the file, ensuring that the the object policy is enforced across different applications and application runs.

4.4.3. Metered payment

A membership-based metered payment DRM system presents a slightly different scenario. In such a system, the subject needs to be a valid member possessing an expense account to access the object and the expense associated with the usage is dependent on the usage duration. Thus the membership is the object attribute while the subject attributes is the cost of usage per unit time. The DRM scenario usage control can be expressed as:

M is monetary amount
ID_{mem} is a set of membership IDs
$Time$ is a current usage unit of time
$expense : S \rightarrow M$
$usage : S \rightarrow Time$
$member : S \rightarrow ID_{mem}$
$value_t(o,r) : O \times R \rightarrow M$, cost of usage right per unit time
$ATT(S) : member, expense, usage$
$ATT(O,R) : value_t$
$allowed(s,o,r) \Rightarrow member(s) \neq \phi$
$disallowed(s,o,r) \Rightarrow member(s) = \phi$
$postUpdate(expense(s)) : expense(s) + value_t(o,r) \times usage(s)$

As the formalisation above shows, the key subject attributes are its membership ID, the total expense and the current usage time while the object attribute is its usage value in unit time. What exactly constitutes the action/rights (R) is scenario dependent. For example, in our implementation, a simple read of the object is used as R.

On parsing the policy and noting the metered payment restriction, the PDP asks the OAM to look up the object attribute. At the same time, the subject attributes are queried using the SAM. The PDP then performs the necessary authorisation checks, the results of which are used to decide whether to allow the rights or not. Once the method call has been executed, the PDP then performs the post update on the subject attribute *expense* by calling the SAM, as shown in Figure 4.7.

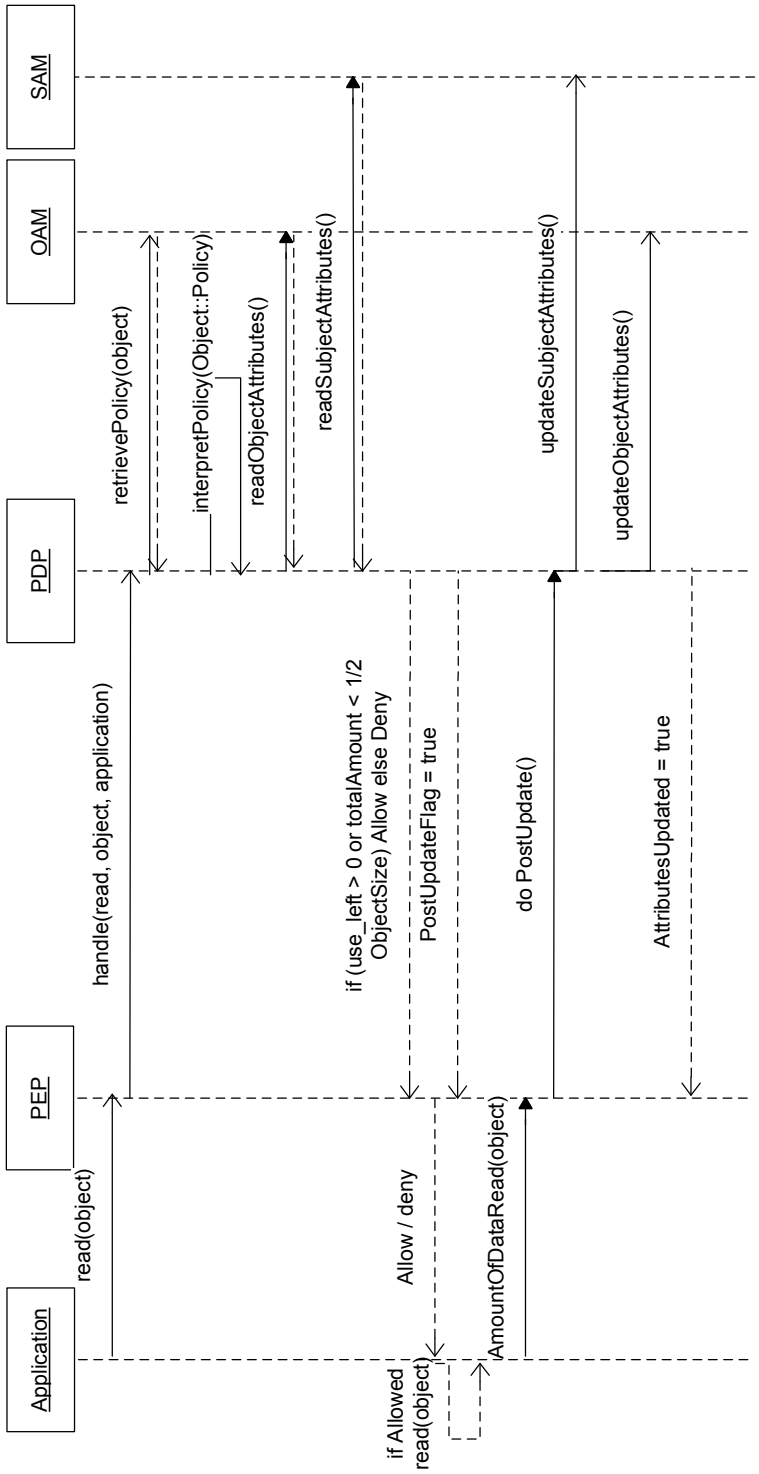


Figure 4.6: Implementing the 'play *N* times' policy using T-UCON.

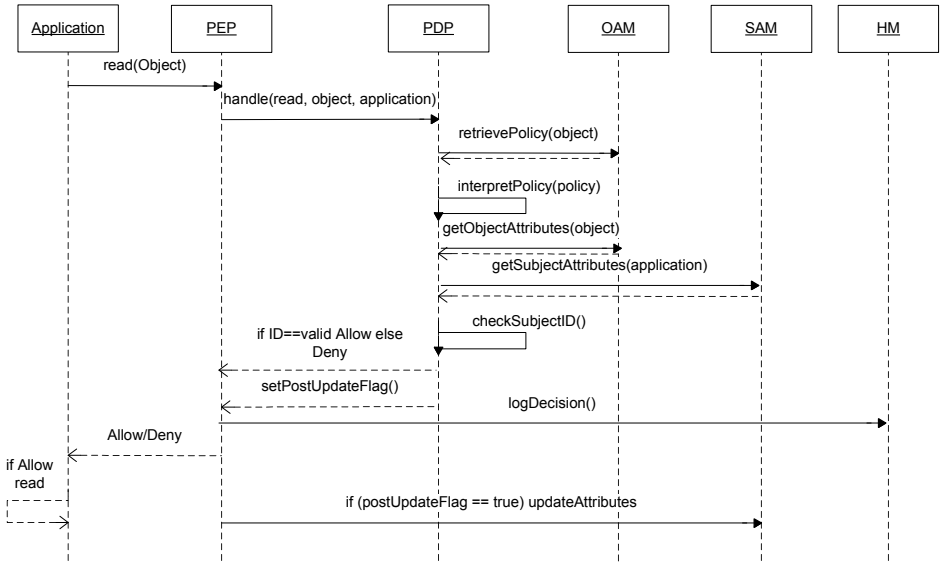


Figure 4.7: Implementing metered payment policy using T-UCON.

4.5. PERFORMANCE

The T-UCON architecture is designed with the conscious aim of being modular and generic enough to implement a wide range of DRM policies in an application-independent manner. Such a design however has the downside of introducing performance overhead. In this section we perform an empirical study of these overheads by examining the results of performance measurements conducted using our prototype implementation of T-UCON. The measurements were performed on an Intel Core 2 Duo 2 GHz machine with 2 GB RAM running Ubuntu 7.10 with a 2.6.22 Linux kernel.

We consider the specific example of pay-per-use policy discussed earlier, specifically to a music player application playing an MP3 file. The open access to an MP3 file is allowed only if the credit remaining for the user is more than the per-use value of the file and the appropriate object attribute is updated after the decision to allow the read is made.

In the first set of measurements, the time taken between the application invoking the file open command and the actual creation of the file object is measured. When an unmodified JVM is used, with no policy enforced, the application was able to start reading from the MP3 file in just over 1 ms.

Environment	time (ms)
Unmodified JVM, no policy	1
T-UCON, no policy	10
T-UCON, pay-per-use (XML)	1783
T-UCON, pay-per-use (txt)	241

Table 4.1: Performance comparison of T-UCON prototype in pay-per-view microbenchmark.

Environment	time (s)
Unmodified JVM, no policy	303
Unmodified JVM, SM policy	303
T-UCON, no policy	304
T-UCON, pay-per-use (XML)	311
T-UCON, pay-per-use (txt)	308

Table 4.2: Performance comparison of T-UCON prototype in pay-per-view application run.

In the second case the application was inside the T-UCON enabled JVM but the setup was done in such a way that the PDP invokes the OAM, which instead of returning the object policy, returns a null string and the PDP in turn returns an *OKOrder*. This set of experiments was intended to measure the overhead due to interception process and the loading of the basic T-UCON modules. It was observed that a high extra overhead of 9 ms, 9 times the original value, was introduced.

In the next experiment, the pay-per-use policy was enforced using T-UCON. With the subject and object attributes being represented using XML, to simulate the use of an XACML [OASIS, 2008]-like REL language, resulting in an observable overhead of 1780 ms. This includes the time taken to read the policy, the object and subject attributes, the execution of the decision logic and return of the decision to the JVM from the PEP. Closer analysis revealed that more than 75% of this overhead is introduced by the process of reading, parsing and updating the XML files. In order to estimate a less biased overhead, simple text files were used next to represent the attributes of the subject and the object. The overhead reduced to a lower value of 240 ms. Table 4.1 summarises the results of the measurements. The lower overhead observed in the second and last set of experiments when compared to the overhead observed when using

XML shows that the system overhead is dependent on the complexity of the policy checks involved and on the way the policy and attributes are represented and as such cannot be attributed solely to the architecture.

Furthermore once the action is allowed and the necessary object updates have been carried out, T-UCON does not have to perform any further mediation and hence the overall overhead should be small when the application run is considered in its entirety. To verify this, we next considered the case of playing a 5.6 MB, 4.56 minute long MP3 file, again subject to the pay-per-use policy. Table 4.2 shows the time required to play the file for various test cases, averaged over 5 runs. The base measurements was performed on an unmodified Kaffe JVM as noted in row one of the table. The second row denotes the time taken for the unmodified JVM to play the file when a simple grant read permission policy is specified by the Java Security Manager, while the rest of the rows of test cases are similar to those in Table 4.1. It is worth noting that the current Security Manager supports only simple access control policies and not usage control restrictions. The low overhead figures observed when using T-UCON supports our claim regarding the practicality of using our architecture.

4.6. TRUSTED SYSTEM CONSIDERATIONS

The T-UCON architecture proposed here, being a pure software based solution, does not rely on any hardware functionality to perform policy enforcement. However, a software-only solution to DRM policy enforcement on open platforms is hard, since it is susceptible to attacks from the owner of the platform who might attempt to circumvent even sophisticated software protection by trying to replace the middleware, or even the underlying operating system. Thus, in order to ensure the integrity of the architecture, it is imperative to leverage on security provided by trusted hardware technologies.

While integrity measurement architectures [Sailer et al., 2004] provide a mechanism by which TPM-based hardware can endorse the configuration of the system's boot process and the libraries loaded in the system to a third party, this by itself does not prevent a malicious replacement of the DRM architecture by the user or the attempt to play the DRM enabled content by an application that does not reply on the T-UCON middleware, basically non-Java based applications.

The basic functionality required to prevent such exploitation by completely replacing the middleware or circumventing its invocation is to ensure that applications do not get access to the DRM content if it is not accessed within the T-UCON middleware setup. The 'seal' functionality of the TPM [Trusted Computing Group, 2006] is the key to implementing this restriction. McCune et al. took the first step in this direction in the Flicker [McCune et al., 2008] project whereby the dynamic root of trust of the TPM and the secure kernel hardware support (SKINIT instruction) of modern CPU architectures is used to provide an isolated execution environment for a *Piece of Application Logic* (PAL). This combined with the TPM-based sealed storage functionality allows for maintaining state across multiple Flicker sessions. They have demonstrated the use of this architecture in protecting SSH password authentication mechanism on the server side by implementing part of it as a PAL and also in protecting the private key of a certificate authority server. While such work shows the potential of using the TPM's dynamic root of trust in association with the CPU features to provide a sealed environment for running T-UCON, the size of T-UCON, a whole JVM implementation, would be a big hurdle in protecting it using a similar approach.

One way of assuring integrity of the T-UCON architecture is through the deployment of a trusted subsystem similar to the one proposed by Zhang et al. in [Zhang et al., 2008b]. Using such a system, a TPM's sealing and trust chain based attestation functionality are used to ensure that the objects can be accessed only by applications running inside T-UCON.

The concept of *trusted channel* introduced by Sadeghi et al. [Sadeghi et al., 2007] provides the most feasible mechanism to provide the TPM-sealed environment required for protecting the T-UCON setup. A trusted channel is defined as a secure channel that can validate the configuration of the other endpoint of the compartment and bind the data to this configuration such that only the compartment with the specified configuration can access the data. The compartment configuration in their architecture maps to a hash value of the software binary (T-UCON in our case) and all the initialisation information including the default policy enforcement engine. The trusted channel is powered by the *trust manager* that abstracts the trusted computing services and the *storage manager* that provides persistent storage while preserving integrity, confidentiality and authenticity.

In the proposed architecture, very similar to the one in [Sadeghi et al., 2007], the publishers verify the bootstrapping of the trusted computing

base and once accepted, asks the trust manager on the DRM device to calculate the configuration of the compartment running T-UCON. If this configuration matches the globally known and approved configuration value, the publisher, using a protocol similar to the one proposed in [Sadeghi et al., 2007] ships off the DRM content and the license (policy) encrypted in such a way as to be sealed against the verified configuration value. The DRM content, in the encrypted state, is not useful outside the compartment. An application that tries to access it outside the scope of a compartment running T-UCON will not be able to get to the unencrypted content as the storage manager part of the trusted computing base would refuse to decrypt the DRM content to any compartment that does not match up to the same configuration as what was used to seal the content. When the compartment with T-UCON running in it attempts to access the DRM content and the configuration integrity check has been passed, the storage manager goes ahead and decrypts the contents and passes it to the T-UCON compartment for its use within the compartment. The vitalisation layer of the architecture ensures strong isolation between the compartments.

4.7. RELATED WORK

The financial incentive involved in developing an architecture for enforcing the usage restriction of digital content has seen the development of several DRM systems.

Most of the work has however been in the form of proprietary systems. Windows Media DRM [Microsoft Corporation, 2009] (WMDRM) from Microsoft allows protected audio and video to be played on Windows PCs and portable devices. The WMDRM provides the full infrastructure of the DRM structure including the content packaging, distribution and licensing as well as restricted playback. The Open Mobile Alliance [OMA, 2009] provides one of the few open specifications for a DRM system, specifically for mobile service providers and device manufacturers. But even in these specifications, the actual mechanism for policy enforcement is left as an open problem for the device manufacturers. In that respect the work we have reported in this chapter compliments these specifications and DRM systems.

Jamkhedkar and Heileman [Jamkhedkar and Heileman, 2004], taking inspiration from the OSI layer framework, propose a generic layered ar-

chitecture for DRM systems. The upper layer made up of application and negotiation layer are the end-to-end layers that create services that are used by the applications that involve DRM. In the middle, the rights expression layer provides the minimal support for the management of the digital rights while the lower layer is concerned with the actual enforcement of rights restrictions. This layer is itself divided into upper category layer, responsible for handling content according to its type and then lower category layer which is responsible for ensuring that no low-level illegal access to the DRM program are allowed. T-UCON system sits somewhere in between the upper and lower category layers, providing a middleware based DRM enforcement system.

Michiels et al. [Michiels et al., 2005] extends the work of [Jamkhedkar and Heileman, 2004] by extracting the high level usage scenarios according to the functionality of the players, the content producers, publishers and consumers. Though they claim to present the "next step towards a software architecture that supports reuse and co-operation of multiple domain-specific DRM technologies and standards," the discussion is confined to the higher framework level and the actual enforcement mechanism is not considered.

Recent years have seen an increased interest in the area of enforcing usage control policies in distributed systems. Considering this area as a superset of the DRM enforcement studies, here we take a look at the existing proposals and highlight their differences compared to our approach.

Berthold et al. [Berthold et al., 2007a], tackle the enforcement of usage control requirements in Service Oriented Architectures. The paper suggests a client-side architecture which is able to support domain separation and policy enforcement for various Java services or objects. The granularity of usage decisions supported by their architecture is at the level of applications, while T-UCON provides a very fine-grained control at the level of Java method calls. Furthermore in T-UCON, applications are not assumed to be trusted.

The client-side enforcement approach proposed by Schaefer [Schaefer, 2007] considers the use of a reference monitor to enforce usage policies on the objects of interest. Although the architecture is similar to that of T-UCON, in their approach the monitor on the client side needs to be continuously updated with the information on a usage control server. Moreover, the work is purely theoretical in nature, while in this thesis work we present the design and implementation of a practical enforcement architecture.

A slightly different approach is taken by Zhang et al. [Zhang et al., 2008a] where they propose an authorisation enforcement architecture for collaborative systems. While their focus is on the general collaborative systems, ours concerns enforcement on the consumer-side. Although typical UCON_{ABC} mechanisms like attribute updates and obligation checks are dealt with in detail, the proposed solution commits to the study of authorisations rather than usage control and hence among others, ongoing obligations are not considered. Trusted computing in usage control is approached in [Zhang et al., 2008b], and while the suggested architecture is similar to [Zhang et al., 2008a], the focus is on the integrity of inner security modules and details of specific enforcement scenarios are not presented.

Katt et al. [Katt et al., 2008] extends the original UCON_{ABC} model by adding the notion of *post-obligations*. Focusing on obligations from the point of view of subject, object and fulfilment time, the paper stresses an enforcement framework incorporating these aspects. However, in their architecture the PEP is embedded with the target application. This prevents policies from being enforced across applications and assumes a trusted application or the existence of a mechanism to safely direct application actions, neither of which are explained in detail in the paper. T-UCON, on the other hand, is firmly based on the premises of controlling untrusted applications and allows the policies to be associated with objects and enforced across applications.

Jamkhedkar and Heileman draws inspiration from the UCON_{ABC} model to propose a formal conceptual model for rights statements that aims to reduce interoperability complexity between various RELs [Jamkhedkar and Heileman, 2008]. T-UCON's OAM would be able to handle such a formal expression model equally well, if a suitable parser is available.

4.8. CONCLUSION

In this chapter we have presented the design and implementation of T-UCON, a generic software-based Digital Rights Management architecture using Trishul framework. Being a middleware solution, unlike other DRM solutions T-UCON is able to enforce policies associated with DRM objects across multiple applications and application runs. Performance results show that while the checks associated with the DRM logic have the

potential to introduce larger overheads, the framework in itself adds a small amount of overhead.

CHAPTER 5

Application: Web Services

Over the years business operations of various organisations have used ad-hoc setups to interact with each other over the Internet. This has, however, lead to the emergence of a large number of incompatible frameworks. In order to tackle this problem the Web service (WS) technology [W3C, 2009] along with associated specifications have been developed to provide a standards-based open framework for application-to-application interaction.

In its current form, WS technologies have been widely used to implement the Service Oriented Architecture (SOA) paradigm by which monolithic standalone systems are decomposed into smaller loosely coupled modular systems that can then be used in an on-demand fashion to compose larger service offerings in a heterogeneous environment by various organisations. This allows for business processes to be composed of these smaller *services* that could even span across organisational boundaries. This also allows a service provider to offer the user, be it an individual or an organisation, a standard public endpoint for accessing a particular service while at the same time allowing it to compose the service internally using various sub-component services, while hiding the complexity from the user. All aspects of the service discovery as well as inter-service interactions and communications are structured using open standards allowing for maximum interoperability.

With more and more users relying on the WS platform for their needs, the question of data integrity, confidentiality and equally importantly the user's ability to impose specific usage restrictions on the data, are becoming big issues. Recent work like WS-Policy [W3C, 2006c] and WS-

ABAC [Shen and Hong, 2006] have allowed users to specify basic authorisation rules for accessing a service and/or the data used by the service. Most of the existing work has, however, concentrated on protecting the web service, specifying and enforcing policies that define who can access the service and how it can be used. However, much less has been done on specifying and enforcing access and usage policies as defined by the data provider. In general the provider would like to ensure that the data it had provided to a remote WS is being used only by services it trusts and has explicitly allowed and in ways specifically allowed by policies it has defined.

In this chapter we describe an architectural framework that exploits the capabilities of Trishul to implement a policy enforcement architecture for user-defined policies in a Web Service environment.

5.1. WEB SERVICES

Service Oriented Architecture (SOA) paradigm promises simple, fast, secure and interoperable integration of services, enabling the creation of large business processes and service using a collection of smaller self-contained components. By allowing the overall service to be accessed at a publicly addressable *service endpoint*, the individual component services that make up the service are hidden from the view of the service user.

At present, Web Services (WS) are the only concrete technology that is used in implementing an SOA framework. These technologies in turn use various XML-based open standards to implement such services. In this section we provide a brief introduction to the main standards used in most WS systems. While the actual process of forming and using Web Services use a top-down approach, expressing the business process first and then implementing and expressing various aspects of the process using different services, in this discussion we use a bottom-up approach in order to describe the various technologies involved.

Simple Object Access Protocol (SOAP) [W3C, 2007] is an XML-based protocol that aims to standardise the communication aspect of the WS technology. It is used for messaging and remote procedure calls between service components as well as the user and the service endpoint, as shown in Figure 5.1. At the transport level, SOAP reuses the existing HTTP protocol to carry the message, which, when stripped to its essentials, is an XML

document containing a header and a body. The header section contains metadata associated with the message while the body contains the actual payload of the SOAP message. WS-Security [OASIS, 2006] specification provides protocol level security by specifying means of supporting encryption and digital signing of SOAP messages.

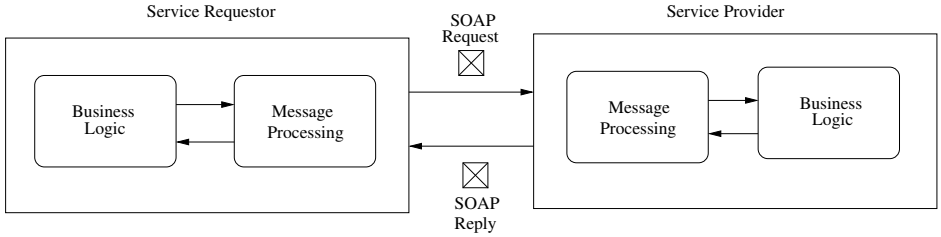


Figure 5.1: The use of SOAP messages between web service entities.

The Web Service Description Language (WSDL) [W3C, 2006a] provides a way to formally describe the service being offered at the WS end point as well as the structure of the expected client-service interactions. It allows for specification of the vocabulary of the message, the interaction of the application-level abstract interface as well as the protocol-dependent details that the user must follow to access the service. A WSDL document consists of two parts: logical and physical. The logical part defines *types* of data being carried, *message* representing the input and output parameters associated with an operation, the actual *operation*, which defines an actual action performed by the service and *portType*, which defines an abstract set of operations supported by the service. The physical part of WSDL describes more concrete aspects of the service including *binding* associating a concrete protocol and message format to operations and message defined within a particular portType, the *port* that associates the endpoint with a physical network address and *service* that contains one or more port elements representing related endpoints.

Universal Description, Discovery and Integration (UDDI) specification [OASIS, 2004] provides potential users a unified and centralised way to find service providers. It specifies the definition of how to define information about the service as well as the query and update APIs for accessing and updating this information from/at the centralised listing.

Once all the services required to implement an SOA has been implemented, it needs to be brought together. The most common way to do that is using Web Service Business Process Execution Language (WS-BPEL) [OASIS, 2007], a language used to create *orchestrations*, which are composite, controller services defining how the services being consumed will interoperate to provide the SOA functionality. At the core, it is an XML-based programming specification that is used to describe high level business processes as interactions between different businesses fashioned as Web Services. An alternative to Web Services Orchestration is Web Services Choreography and its associated specification Web Services Choreography Description Language (WS-CDL) [W3C, 2005] which defines a more descriptive process-less service-service relationship between various WS endpoints.

Yet another XML-based specification, WS-Policy [W3C, 2006c], is used to express the capabilities, requirements and characteristics of Web Services involved in a SOA framework. A policy is composed of policy alternatives, each of which is a collection of *policy assertions*. An assertion is defined as “an individual preference, requirement, capability or other property." Examples of policy assertions include authentication schemes, privacy policy, QoS guarantees etc. WS-Policy provides a common fine-grained syntax for specifying these different kinds of assertions in a consistent manner. An example of a WS-Policy document is shown in Listing 5.1. It describes a web service instance invocation which uses a ‘SecurityToken’ assertion of the type ‘Kerberos’ [Kerberos Consortium, 2009].

```
1 <wsp:Policy>
2   <wsp:ExactlyOne>
3     <wsse:SecurityToken>
4       <wsse:TokenType>wsse:Kerberosv5GTGTS</wsse:TokenType>
5     </wsse:SecurityToken>
6   </wsp:ExactlyOne>
7 </wsp:Policy>
```

Listing 5.1: Example of WS-Policy specifying that a WS instance uses a Kerberos token.

As there are several entities involved in the life-cycle of a Web Service, it follows that these actors would be interested in specifying their own policies at various phases in the service life-cycle. Figure 5.2 shows these actors and the various policies they define.

Service policies are defined by the developers of the web services as

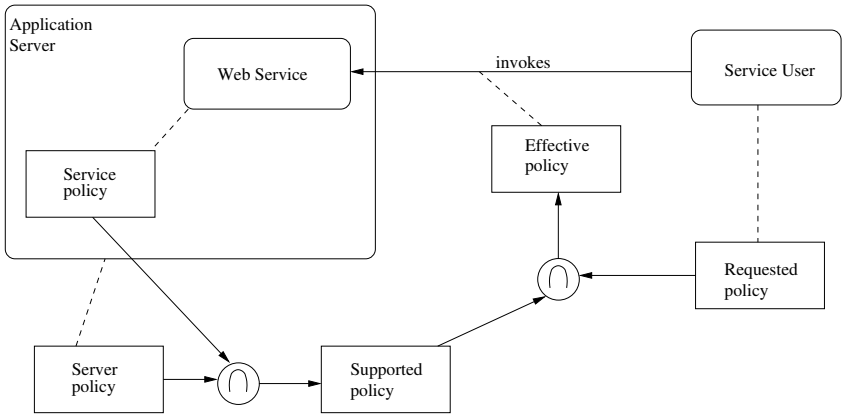


Figure 5.2: WS-Policy definitions.

properties that must hold true for all instances of the web service irrespective of the hosting environment. *Server policies* on the other hand are defined by the hosting providers and specify the features that are supported by a specific application server on which the service is instantiated. *Supported policies* are formed by the intersection of these two policies and represent the effective policy of a WS running on a specific hosting infrastructure. The WS user is also able to state the features of the services it would like to invoke, to support. These policies are called the *requested policies*. The intersection of the supported policies and requested policies form the *effective policies* of the Web Service instance. Approaches to policy intersections are discussed in [Mukhi and Plebani, 2004; Verma et al., 2005].

WS-Policy by itself does not provide all the functionalities required for using policies in web services. For example, while it can be used to specify service policies, it does not concern itself with how the policies are attached to a web service. Yet another specification, WS-PolicyAttachment [W3C, 2006b], defines the required mechanism. Specifically, it defines a mechanism to reference policies from a WSDL document, associate them with a specific instance of a WSDL service as well as with UDDI entities.

While the specifications discussed above provide a standards based framework for enforcing policies, the actual design and implementation of a policy enforcement architecture has been less common. In this chapter we consider the use of Trishul in designing such an architecture.

5.2. SCENARIO OVERVIEW

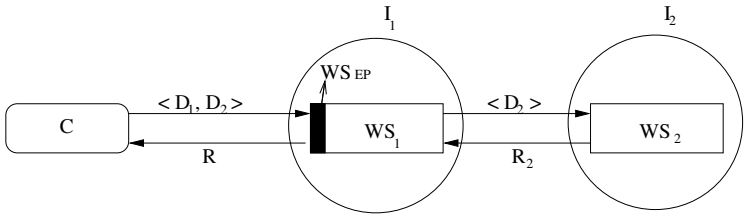


Figure 5.3: Example scenario providing motivation for the policy enforcement architecture.

Consider the example scenario shown in Figure 5.3 in which an entity I_1 provides a web service WS_1 at the publicly addressable endpoint WS_{EP}. The service offered at WS_{EP} is internally fulfilled by using the capability of the local WS_1 and another Web Service WS_2 provided by an external entity I_2. A client C wishing to use the service provided by I_1 sends the required *confidential* input data D to WS_{EP} for processing. Assume that the data consists of two independent segments: D_1 and D_2, each to be used by I_1 and I_2 respectively. On receiving the input data, WS_1 sends D_2 to WS_2 for processing and uses the response R_2 obtained from WS_2 along with the user-submitted data piece D_1 to compute R, and sends the response to the user.

The client C wishes to associate policies (these policies are discussed below) P_1 and P_2 to the data segments D_1 and D_2 respectively. It would like to ensure that D_2 is sent to I_2 alone and that the policies specified by the client and attached to the data are enforced at WS_1 and WS_2.

5.2.1. Policy Classes

The classes of policies that a WS client would like to see enforced remotely on the submitted data can be broadly divided into two – access control and usage control.

Previous studies on access control policies in the context of web services [van Bemmel et al., 2005; Shen and Hong, 2006] have concentrated on issues related to allowing the services to define and control who invokes

it or on allowing the user to specify the *requested policies* to form the *effective policy* of the web service instance. In this work, we extend this by allowing the clients to specify the identity of the web services that can access the confidential client data as well as specify the conditions under which such access can be provided.

Usage control policies on the other hand specify how input data can be used by the web service components once access rights have been granted to it, either unconditionally or after a specified policy requirement has been fulfilled. Such policies could span a wide range of functional restrictions. For example, *C* could specify that the data D_2 should not be stored in the remote WS's permanent storage device but rather be used for processing entirely in the memory. Another possible policy could be that the data should not be sent outside the WS host unless encrypted beforehand or that all network communication using the data should use secure end-to-end encryption protocol TLS 1.2 or higher.

5.2.2. Threat Model

In a traditional web service setup the resource provided by the service components are considered to be the asset to be protected from attackers who might be interested in exploiting the setup to gain unauthorised or unaccounted access to them. However, in our scenario, the user data is the protected asset while the web service components are assumed to be untrusted. They cannot be trusted to enforce the policies on their own, that is, *the remote service component may be able to violate the usage control requirements that data provider has imposed on the data*.

We also assume that since the remote platform on which the service is running is not directly under the control of the data provider, it cannot be blindly trusted to enforce the policies specified by the provider. Further steps need to be taken in order to ascertain the integrity and functional state of the software stack that is running on the remote side before such trust can be placed.

In the next section we introduce Trishul-WS, an architecture proposed to implement a policy enforcement framework in a Web Service framework using Trishul.

5.3. TRISHUL-WS

As discussed before, the WS architecture provides a framework for user-submitted data to be processed by the various component services that make up the WS. However, the problem of ensuring that the policies associated with the submitted data is enforced at these services has largely remained unresolved. Trishul-WS architecture aims to fill this gap in research by implementing a middleware-based WS policy enforcement architecture.

5.3.1. System Architecture

The proposed architecture of Trishul-WS is shown in Figure 5.4.

Let us consider the various components involved in the system by following the data as it traverses the system. Before the service user sends any data to the web service instance, it invokes the Attestation Service Module (ASM) of the WS to check for its compliance with the policy enforcement architectural requirements (1). The detailed working of ASM is discussed in Section 5.3.4.

Once the compliance check is successfully completed, the user sends the data to the WS at the endpoint advertised in the UDDI document (2). When this ⟨data, policy⟩ packet is received at the remote endpoint, the Access Policy Enforcement Point (PEP_A) enforces any access control restrictions specified in the policy of the user of the WS. For this, it invokes the Policy Decision Point (PDP) (3) to check whether the conditions for access have been met.

The PDP is internally assisted by the various helper modules in its decision making process, making the architecture extensible. The detailed working of these modules are not explicitly defined in the architecture itself since they depend on the specific policies that would be enforced at the WS, allowing for the architecture to support various classes of access control and usage policies. For example, all system specific attribute checks ("Java version should be Sun Java $\geq$ 1.5") could be provided by a 'System Attribute Module' which could then be queried by the PDP during the decision process to check whether Sun Java is installed and if so, whether the version is $\geq$ 1.5.

The decision is sent back to the PEP_A (4), which forwards the data to the service component if access restrictions are met (5). Once the data has

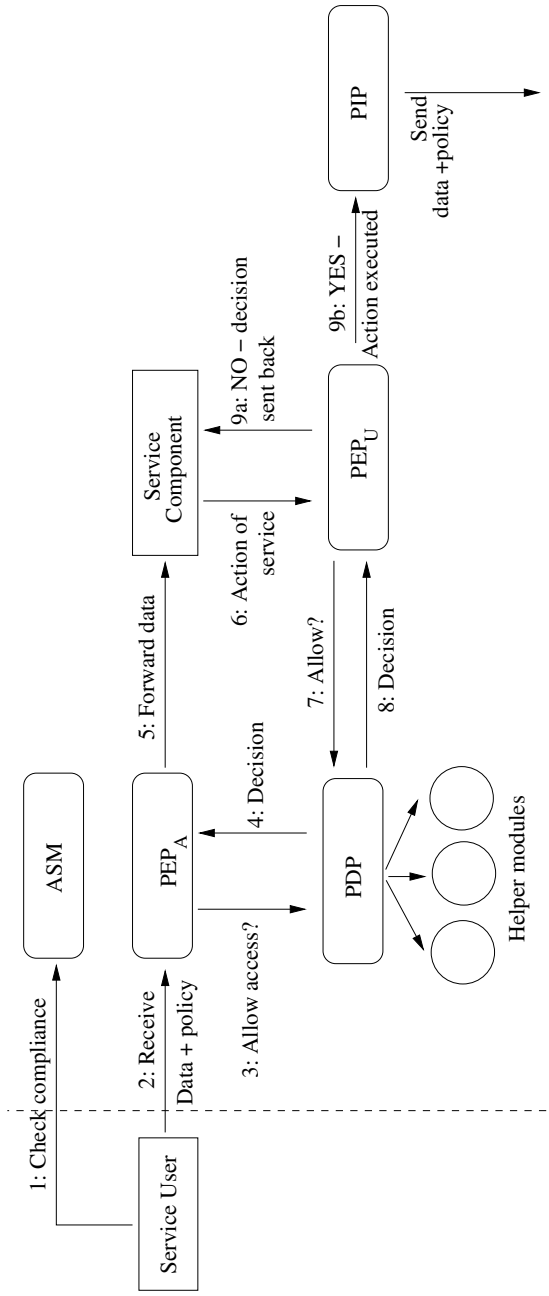


Figure 5.4: The architecture of Trishul-WS designed to develop policy enforcement in web services framework.

been passed on to the service, every action performed on it is subject to usage restrictions specified by the user requested policy. To enforce this, each action that involves the use of the policy-tagged data is *intercepted* by the Usage Policy Enforcement Point (PEP_U) of the middleware (6) and passed on to the PDP (7) to make a decision on whether it should be allowed to be executed as per the (usage) policy attached to the data. The decision made by the PDP is conveyed to the PEP_U (8) which then informs the application of the negative decision (9a) or lets the application's action through (9b).

One special action that needs to be handled differently is that of writing the data to a network socket. When a data segment leaves the system for an entity other than the service user, it has to be re-tagged with the effective policy that is currently associated with the data. This data segment under consideration need not be the user provided data as such but rather any data that has been tainted explicitly or implicitly by information flow from the user provided data. This process of data re-tagging is handled by the Policy Insertion Point (PIP).

5.3.2. Properties

The architecture described above is designed with the following properties in mind:

- Modular: the functionality provided by the various components are compartmentalised in such a way as to make the system very modular.

- Policy language agnostic: the architecture does not specify or depend on a specific policy specification language that *C* needs to use. WS-Policy [W3C, 2006c] and its extensions like WS-CoL [Baresi et al., 2006] and other similar expressive languages can be used to specify the policy as long as the PDP has the equivalent interpreter engine to parse the policy and understand the specification syntax.

- Message passing independence: the architecture is not tied down to any specific message passing specification, allowing for the use of SOAP-like XML based message passing protocol which is normally used in web services framework or any other markup language that might be used in other SOAs.

– Extendable: extra helper modules can be added to the architecture and invoked by the PDP to help enforce a wider range of policies.

– Technology independence: the architecture does not explicitly depend on or use specific technology properties in its design. This means that in principle, a implementation of the architecture could be realised using Java, .Net or any other technologies. However, the system-base used for implementation does need to provide certain functionalities, as described below, for the actualisation of the architecture.

5.3.3. Functional Requirements

While the proposed architecture is independent of any implementation technology, the platform choice for implementation of such a system should be made with the following requirements in mind:

– Interception capability: when a service uses a policy-tagged data in an operation, the PEP has to intercept the action and pass the control to the PDP. Whether this interception is done at application level (harder to implement but easier to express and interpret the policy) or at a lower level, for example at the Java method calls level, (easier to implement but harder to translate high level semantics to low level calls) is implementation dependent.

– Information flow tracing: the ability to associate a policy with the data and robustly trace the flow of the data within the system is a crucial requirement of the architecture, without which the (untrusted) application could try and disassociate the policy from the data, in an effort to circumvent the policy enforcement.

  Furthermore, as the web service components work on the data, the policies associated with the data can change. For example, consider the following operation performed by the service on two data pieces it received:

  $D_{temp} = D_1 + D_2$

  As per the information flow principles [Denning, 1975], the policy associated with D_{temp} should effectively be $P_1 \cup P_2$, that is the policies P_1 and P_2. This means that when the service use D_{temp} in later

stages, the PDP needs to ensure that both P_1 and P_2 are adhered to. Similarly if the service is trying to send the data to an external party, the PIP should tag D_{temp} with these two policies and the tuple sent into the network should be $\langle D_{temp}, P_1, P_2 \rangle$.

In short, the implementation should be able to handle both normal and malicious explicit and implicit flow tracing problems described in earlier chapters of the dissertation.

5.3.4. Platform Security

For the proposed architecture to be able to enforce client-specified policies, every WS component involved in providing the service has to implement the middleware platform and run its component service on top of this middleware.

Since the assurance of policy enforcement depends on the existence of the middleware at the remote web service, C needs some form of check to ensure that the policy enforcing middleware is indeed running on the remote system before it can safely send the data to the WS_{EP} of WS_1.

Furthermore, access to the data should be prevented if the middleware is found not to be running, in order to protect it against the untrusted platform owner. In a similar manner, WS_1's middleware needs to ensure that the WS_2 is also running the expected middleware before sending the data across.

In our architecture, this is implemented by the use of the Trusted Platform Module (TPM) [Trusted Computing Group, 2006] hardware. TPM provides the technology to confirm that a remote machine is in a specific (trusted) state using the core root of trust for measurement and the process of remote attestation [Trusted Computing Group, 2006]. It also provides a functionality called secure sealing that enables data to be encrypted to a specific state of the machine in such a way that decryption can be performed successfully only if the machine is in the same state for which it was encrypted for. These functionalities are used in our architecture to provide platform security.

At startup, the state of each of the components of the layers below the middleware – the BIOS, the boot record and the operating system – is captured by binary hashing and stored in a Platform Configuration Register (PCR) of the TPM. The OS, or a dedicated kernel module [Sailer et al., 2004], can in turn capture the state of all the code running in the system,

including the Trishul-WS middleware, and again binary hash it and store the new value by extending the PCR value. Whenever a remote party, like the service users, wishes to validate the state of the software stack running on the system, it can initiate an attestation challenge to the TPM which will report the value of the stored PCR value, signed with a hardware-protected Attestation Identity Key (AIK). A certificate from a mutually-trusted Certification Authority can in turn bind the AIK to a specific legitimate TPM.

Semantic Remote Attestation [Haldar, 2006] has been proposed to help the attestation process capture the dynamic state of the application running on top of the middleware. All these functionalities are provided by the Attestation and Security Module (ASM) of our middleware shown in Figure 5.4.

Once the remote system is verified to be in a state that is known to be trusted, data can be sent to it encrypted such that it can be decrypted only if the machine is in the same trusted state.

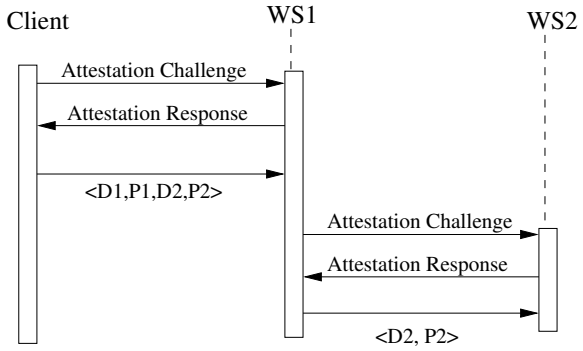


Figure 5.5: Steps involved in the platform attestation process.

Figure 5.5 provides an overview of the attestation process. The client C kick-starts the process by issuing an attestation challenge to WS_1 and only if the attested value matches a known trusted value would it submit the data to the WS_{EP}. Similarly, the middleware on WS_1 would issue an attestation challenge to a remote component service WS_2 before a policy-tagged data is sent to it. This is a typical implementation of the attestation procedure proposed by the TCG. In fact, the attestation process described here is similar in nature to the WS-Attestation [Yoshihama et al., 2005] specification.

Recent advances in technology in the form of support for Secure Virtual Machine (SVM) [Strongin, 2005] extensions in modern CPUs have provided a more dynamic way to ensure the integrity of the platform and confidentiality of the submitted data. McCune et al. [McCune et al., 2008] have developed *Flicker* architecture that allows a piece of code to execute in isolation by disabling direct memory access (DMA) and interrupts. In our architecture the Flicker code can be fashioned to run the ASM.

On its first run, the ASM creates a private-key public-key pair pair and seals the private key using the TPM such that no other code can gain access to the key. The attestation of this step can assure the remote client that a trusted ASM was run to generate the key. The public key is then made widely available. When the client needs to send any data over to the remote host, it can be encrypted with the public key of the ASM.

When the ASM receives the encrypted data, it uses TPM unsealing functionality to retrieve the private key and use it to decrypt the data. This decrypted data is used by the rest of the middleware for processing. As long as the PDP prevents the data from being stored locally, any untrusted application will be unable to circumvent the policy enforcement specification associated with the data.

5.3.5. Implementation

The Trishul framework is well suited for implementing the WS policy enforcement architecture proposed earlier in this chapter due to its support for information flow tracing and the ability to perform method call interception needed to monitor the web service actions.

The PEP_A is implemented by extending the Axis2 [Apache Software Foundation, 2009] SOAP engine as it anyway needs to interact with the SOAP engine of the already existing WS framework. The PEP_U interface is implemented using the Trishul JVM hooks that intercept the method calls invoked by the web service components. In the current prototype implementation, it is assumed that the identity of these method calls that need to be trapped are known in advance.

In the simplified prototype developed to demonstrate the feasibility of using Trishul to implement the Web Service policy enforcement framework, we consider a simple WS system which accepts two strings as input data, concatenate them and return the resulting string back to the user. An extension of the SAML [OASIS, 2005] *condition* element in an SAML as-

sertion was used to specify the user restriction attached to the use of the data supplied to the web service. WS-PolicyAttachment is used to attach the policy to the user supplied data.

In the example policy in Listing 5.2, the input data is associated with the policy that the data can be used only if the JVM version is at least 1.5 and that the data cannot be sent to any other third party over the network.

```
<Assertion AssertionID="3425">
  <Conditions NotOnOrAfter="2009-01-12T09:03:187">
    <policyCondition>
      <policy xmlns="http://www.srijith.net/t-ws">
        <appliesTo serviceId="T-WS_Example">
          <URL>http://srijith.net/t-ws1</URL>
        </appliesTo>
        <partyId>TWS1</partyId>
        <JVMVersionMin>1.5</JVMVersionMin>
        <thirdparty>
          <allow>No</allow>
        </thirdparty>
      </policy>
    </policyCondition>
  </Conditions>
</Assertion>
```

Listing 5.2: SAML based policy attached to user submitted data for prototype implementation.

The enforcement of the policies using the Trishul-WS architecture obviously impacts the performance of the Web Service infrastructure. In order to gauge this performance overhead of the prototype implementation of Trishul-WS, a number of measurements were performed, comparing the enforcement architecture prototype performance with that of the normal setup. All tests were performed with the Web Service component implemented on a single node of a four-node AMD Opteron system (model 852, 1Mb cache, 2593 MHz) with 1.5 GB of RAM. Table 5.1 lists the measurement results of the processing time between the instance the input data is received at the WS and the instance the result data is sent to the user.

In the case where Trishul-WS is used, the web service is executed in a policy complying environment, i.e. JVM version 1.5 is used and the component does not attempt network access. As the measurements in Table 5.1 show, the use of the policy enforcement T-WS architecture introduces a processing overhead of 97%. Given the overhead introduced by the Trishul framework as seen in the microbenchmark reported in Chapter 3, this finding is not surprising.

Setup	Processing time (ms)	Overhead
Normal WS	39	-
T-WS	75	97.4%

Table 5.1: Performance comparison of normal web service processing time and that using the T-WS policy enforcement architecture for policy in Listing 5.2.

5.3.6. Advertising Enforceable Policies

The set of policies that a Web Service can enforce at its end can be considered as description of services that is offered to the client. Hence, just like in the case of typical Web Services properties, it should be able to advertise the types of policies that are available for the user to attach to the submitted data. Just as WSDL is used to describe the services offered, a similar extension of the XML-based specification language may be used to describe the set of policies the WS can enforce at its end. This is not considered as within the scope of this work.

In order for the client to specify which parts of the data can be accessed by which specific internal components of a WS setup, the identity of all components that make up the service must be advertised to the user. For example, C should know that the service at WS_{EP} is composed of two individual services WS_1 and WS_2. With this knowledge, C could then specify the access control policy that WS_2 can have access to say, only D_2. Again, support for this is not considered as essential part of current work and is deferred for further research.

5.4. RELATED WORK

Some of the works done in the area of Web Services security has been in providing protocol level integrity and confidentiality assurance, for example WS-Security [OASIS, 2006]. They mainly deal with specifications on how to use cryptographic primitives to protect the SOAP messages sent between the different components of the Web Service. Our proposal on the other hand is an architectural solution to the problem of enforcing user-specified policies for the data provided by them. In fact solutions like WS-Security can be used within our architecture, just like in the normal WS setup.

WS-Policy [W3C, 2006c] and WS-PolicyAttachment [W3C, 2006b] specifications provide a framework to define and attach policies (capabilities, requirements etc.) to various entities associated with a WS based system. Unlike our proposed architecture however, they do not specify the actual system design needed to ensure the enforcement of the associated policies. Our architecture uses these specifications to define the policies that the data provider would like to attach to the data submitted to the Web Services.

van Bemmel, Wegdam and Lagerberg have proposed 3PAC, an enforcement architecture for credential-based access policies for Web Services [van Bemmel et al., 2005]. Similar in concept to the working of Kerberos [Kerberos Consortium, 2009], but for Web Services, the 3PAC architecture provides a signed token-based system for controlling access to Web Service resources. Other than access control decisions based on tokens, 3PAC does not support any other policy restrictions, specifically usage control restrictions. In fact the actual implementation of the 3PAC access control engine at the Web Service can be implemented using the Trishul-WS architecture proposed in this chapter as a modular helper module to the PDP.

Attribute-based access control (WS-ABAC) [Shen and Hong, 2006] was proposed as an alternative to the identity-based access control model for Web Services in order to address the administrative scalability and control granularity of the identity-based approach. XACML [OASIS, 2008] is used as the policy specification language. The implementation architecture is similar in design to that of Trishul-WS, with a SOAP handler, PEP, PDP among others to implement the access control model.

Baresi, Guinea and Plebani have proposed a monitoring framework and a language named WS-CoL for letting the user specify requirements on the execution of Web Service composition in WS-BPEL processes in [Baresi et al., 2006]. However, their work in itself is mainly concerned with restrictions to be placed on execution of composed Web Services, i.e. the WS-BPEL process. This is done by instrumenting the original WS-BPEL specification such that at all locations specified in the policy, the invocation of the WS-BPEL activity is substituted by a call to the monitor manager. The policy restrictions specified in WS-CoL can, in theory, be supported by Trishul-WS by writing a suitable parser and decision engine logic, while the monitor manager is similar in design to the Trishul-WS architecture. The policies supported in their work can be considered as a subset of the

policies that can be implemented using Trishul-WS. One main difference between their work and Trishul-WS is that in our work, the policies can be attached to the data in a very secure manner using the information flow tracing functionality while in their work the policies are specified for business processes between Web Service compositions.

Berthold et al. have proposed an approach to model usage control requirements on remote clients in Service Oriented Architectures [Berthold et al., 2007b]. While in some way this is similar to our approach, their work is aimed at providing, for example, a secure client environment in which confidential data can be used. More importantly, their approach assumes that the applications at the client side are trusted to behave according to the policy specification. Our proposed solution explicitly assumes that the service running on the remote WS host is untrusted and proposes a middleware based solution that works at the granularity of the Java method calls to enforce the policies associated with the data these applications/services use.

In Entropia [Chien et al., 2003], the authors propose an architecture similar to [Berthold et al., 2007b] but in context of desktop grid systems where the data to be processed is sent to the desktop of the client in a secure way and the client is trusted to use the data only in the correct way. Their architecture does not consider generic usage policies and the only overlap with our work is its ability to provide an encryption mechanism protocol to ensure the integrity of the data on the desktop grid.

5.5. CONCLUSION

In this chapter of the dissertation we have presented Trishul-WS, a generic middleware based architecture for the enforcement of user specified data-centric policies in WS and SOA frameworks, making use of the information flow tracing and method call interpositioning capability of Trishul.

Through the use of TPM technologies, the user (and the middleware) ensures that the remote machine is indeed running the trusted middleware and only then is the data sent to the endpoint. Once the data is received, the information flow tracing property of the middleware ensures not only that the data and policy cannot be separated but also that the application's operation on the data will preserve the correct policy requirements as per

the information flow principles. The middleware also enables the interception of application action performed on the data and ensures that these are allowed as per the policies specified by the data provider.

Performance measurements performed on an unoptimised prototype implementation of the architecture showed a high overhead. While this is not surprising given the performance overhead measured for Trishul framework in the earlier chapter, this means that Trishul-WS would need a lot more work before it can be considered for use within production environments of Web Services.

CHAPTER 6

Summary and Conclusions

This last chapter concludes the dissertation and is organised as follows: Section 6.1 summarises the previous chapters, in Section 6.2 we present our conclusions and lessons learnt from this research work and in Section 6.3 we discuss future directions for research.

6.1. SUMMARY

In Chapter 1, we introduced the research issues motivating this dissertation, namely the need for an architecture that enables the enforcement of various classes of policies that are attached to sensitive data submitted by external parties and processed by the end system. Our proposal was to design and implement a Java middleware based security architecture in order to realise this requirement.

Chapter 2 presented some of the background work related to this dissertation. We discussed some basic access control and usage control models that are usually used to specify restrictions on when access can be allowed to a protected resource and once this access has been given, what restrictions can be placed on the actual usage of that resource. The need for information flow tracing and its control was also discussed in this chapter with references to previous works done in this area. The stack based Java security model was shown to be inadequate in helping us implement the envisioned policy enforcement architecture. In the last part of this chapter we discuss the trusted computing technology, in particular the functionality provided by the TPM in the form of remote attestation and sealed storage.

In Chapter 3 we introduced Trishul, the Java based policy enforcement architecture that forms a major part of this dissertation work. Explaining the design of the system, we looked at the two main design techniques used by Trishul: information flow tracing and Java method call interception. Trishul-P was also introduced as a Java-like language that helps policy engine writers specify the method calls that Trishul should intercept and also as a way to introduce taints into the system that helps in information flow tracing. In the section on implementation of Trishul we explained in detail how Trishul performs information flow tracing accurately using a hybrid load-time and run-time analysis process. The implementation description covered both interpreted as well as just-in-time modes of the Trishul Java Virtual Machine. To get a better understanding of how Trishul works we then looked at couple of example applications where Trishul's generic policy enforcement framework has been used to enforce basic access control policies.

Microbenchmark performance measurements conducted to infer the overhead introduced by interception and information flow tracing functionality of Trishul were reported. The measurements showed that the method call pattern matching and interception module incurred heavy overheads while the hybrid load-time and run-time process used to perform correct taint propagation introduced moderate overhead into the Trishul system. Possible design and implementation optimisations that have been identified were also mentioned in this chapter. A comprehensive review of the related works in the area of information flow control, which forms the major part of Trishul, and policy enforcement, is presented at the end of the chapter.

In chapters 4 and 5 we presented the application scenarios where the Trishul policy enforcement architecture is used in solving the problem of digital rights management and the enforcement of policy attached to user-supplied data in Web Services systems.

In Chapter 4 we looked at how Trishul can be used to implement a DRM solution based on the UCON_{ABC} usage control model. We described the various functional components of the resulting T-UCON system and how Trishul's functionalities can be exploited to implement them. Then we showed how T-UCON could be used to enforce three kinds of DRM scenarios. Performance measurements, reported in the chapter, showed that T-UCON incurs only marginal overhead in exchange for being able to enforce common DRM policies.

In Chapter 5 we considered the problem of policy enforcement in a Web Services framework. Departing from the usual model of considering the offered service as the restricted resource and the need to control who can access to it, the application scenario considered in the chapter assumes that the *data* submitted by the service user to the web service is the valued commodity. The policies, specified by the data provider, that govern this data's access and usage need to be enforced at the Web Service component levels. We then described the design of a system architecture that can be used to enforce such policies in a WS setup. After noting that Trishul's framework would suit the implementation of such a system, we went on to describe the prototype implementation of Trishul-WS for a simple Web Service application scenario and examine its overhead.

The purpose of this last chapter is to summarise the work reported in this dissertation, the conclusions of this research and to point out directions for future research work.

6.2. CONCLUSIONS

This dissertation presented, discussed and evaluated the main ideas behind the design and implementation of a Java Virtual Machine based policy enforcement framework named Trishul. The goal of the research presented in this dissertation was to examine whether it was possible to develop a generic policy enforcement architecture, in which the policy was attached to the data that were being operated on and which defined the access and usage restrictions on that data. To answer this question we analysed the shortcomings of the policy enforcement mechanisms available currently, especially with respect to the Java architecture. We then went on to propose a new policy enforcement architecture as an extension of the normal Java Virtual Machine that was more powerful and flexible than the current available solutions.

One of the key functionalities introduced into this new JVM framework was that of information flow control, which allowed for very precise tracking of the data as they are used within the system as well as very flexible control over how the data can be used within the system. We also proposed a Java-like language for developing the core decision engine of Trishul.

The microbenchmark performance analysis performed on the just-in-time mode implementation of Trishul revealed areas of high overhead in

the system. Subsequent analysis revealed design and implementation choices that could potentially decrease these overheads, which can be considered for potential future work. The inevitable overhead present in pure run-time policy enforcement systems like Trishul (remember that Trishul performs the traditional static analysis steps at load-time of the application) suggests that any analysis that can be done offline in a secure manner, say of the system libraries, should be done so. The analysis also showed that Trishul is not the best system to enforce policies for computation heavy applications.

In order to highlight the power and flexibility of the developed Trishul system, and to demonstrate that the system does indeed help in enforcing data-attached policies, we then used it as the building block of a policy enforcement architecture for two different application scenarios. In the first scenario, we designed and implemented a Digital Rights Management (DRM) system that was capable of enforcing several typical DRM policies attached to the multimedia content rendered by DRM applications running on top of the enforcement architecture. In the second application scenario, the Trishul framework was used to build a policy enforcement architecture for Web Services (WS), in which policies attached to the user data submitted to the WS were enforced by all the component services that make up the WS.

By verifying that the various component services run on top of the proposed enforcement architecture and in turn by designing and implementing the policy enforcement architecture using Trishul in a secure and extendable (modular) manner, we achieved the set objective of demonstrating the power and flexibility of the Trishul framework.

6.3. FUTURE WORK

There are a number of possible directions for future work, among others, open issues that have been identified.

The prototype implementation of Trishul, while ideal for showcasing the power and flexibility of the system to be used in various application scenarios, is far from an efficient implementation. As discussed in Section 3.5.4, several points of improvements have been identified, the implementation of which could constitute a direction for future work. Reusing JVM's internally calculated control flow graph, omitting tainting of variables and objects that are modified in each branch of a CFI, completely

skipping recalculation of context taint if the variables that influence its value has not been modified in the branches of the CFGs and the offline computation and storage of CFGs for system libraries are some of them.

We have developed the Java-like Trishul-P language to allow policy engine writers to express the logic of the decision engines and hook it to the Trishul framework in an efficient manner. While the Java-like nature of the language lowers the barriers to adaptation of the system as well as makes it a powerful development tool, this same property makes it harder to theoretically analyse the expressibility of the language and the overall power of the enforcement system in terms of the classes of policies that can be enforced by it. As referred earlier in Section 3.6, several works have analysed the power of dynamic monitoring systems whose internal decision engine logic are expressed as automaton-based process steps. The structured and logical expression of these systems make them ideal for structured analysis in comparison to the Turing-complete nature of Java-like languages. In fact the work done by Le Guernic et al. [Guernic et al., 2006; Guernic, 2007] in analysing information flow based dynamic systems is very similar to the internal working of Trishul, except for the fact that in their work, the decision engine logic is expressed as automaton transitions.

Hence, a different direction of work would be to investigate either the implementation of Trishul-P with an automaton-based internal engine or the development of a front-end to the Trishul-P interface in the form of an automaton-based system. In the latter architecture, the policy engine writer would define the engine logic in the form of automaton transitions of allowed and disallowed states, which would then be transformed into Trishul-P code that, as before, can be compiled into loadable Java classes for use within Trishul. This allows for a more structured analysis of the power of Trishul architecture while at the same time making the development easier by reusing the existing solutions.

Trishul does not consider or address the impact of multi-threading in information flow or in application execution. As discussed earlier, multi-threading introduces a set of new challenges [Guernic, 2007] and handling them could form a direction for future work.

In our work we have leveraged on trusted computing technologies to attest the integrity of the platform in order to ensure that the remote computing environment is what it is supposed to be. However, more works needs to be done in ironing out the specific details of how this will be implemented in practice, in particular to ensure that the policies associated

with the data are neither separated from the data nor tampered with. While this dissertation proposed some basic mechanisms to enforce such a 'sticky policy' functionality [Karjoth et al., 2002], further work needs to be done and latest developments [Tang, 2008] need to be investigated to make it more secure. This could also thus form part of future work.

SAMENVATTING

Nu het bereik en mogelijkheden van Internet en genetwerkte systemen steeds groter worden en dankzij de opkomst van paradigma verschuivende technologieën als *Web Services (WS)* en *Software as a Service (SaaS)* sturen steeds grotere aantallen gebruikers enorme hoeveelheden privé gegevens naar externe systemen waar zij geen controle over hebben. Aan de andere kant van dezelfde technologie-munt gebruiken commerciële aanbieders van digitale informatie het grote bereik van het Internet om producten zoals muziek, video's en software te verspreiden naar individuele computers van cliënten, zijnde generieke *desktop* machines of consumentenelectronica zoals multimediaspelers.

Over het algemeen hechten deze gegevensaanbieders groot belang aan het beschermen van hun gegevens tegen misbruik. Zij willen graag dat hun gegevens alleen gebruikt worden zoals zij dat gespecificeerd hebben, en alleen toegankelijk zijn voor expliciet toegestane externe partijen. Sterker nog, zelfs nadat toegang verleend is mogen slechts bepaalde specifieke acties op de gegevens uitgevoerd worden.

Deze toegangs- en gebruiksregels worden gewoonlijk uitgedrukt in de vorm van voorwaarden, welke dan gebundeld kunnen worden met de gegevens wier toegang zij reguleren, en vervolgens samen verstuurd naar de externe machines.

Een aantal eerdere onderzoeken heeft zich gericht op hoe de beperkingen in de voorwaarden uitgedrukt kunnen worden op het niveau van specificatietalen, terwijl andere onderzoeken het probleem vanuit een meer theoretische hoek benaderden door het formeel definiëren van modellen en klassen van voorwaarden die gehandhaafd kunnen worden gebaseerd op verscheidene aannamen en mogelijkheden van het systeem. Veel minder onderzoeken hebben echter de vereisten op systeemniveau onderzocht die daadwerkelijk nodig zijn voor het handhaven van de voorwaarden op de externe machines. Dit is de invalshoek van waaruit wij dit probleem benaderen in deze dissertatie.

PROBLEEMBESCHRIJVING

Het globale probleem dat aangepakt wordt in deze dissertatie kan als volgt worden uitgedrukt:

Gegeven een dataobject dat een gebruiker wenst op te sturen naar een externe computer en voorwaarden die toegangs- en gebruiksregels specificeren op het dataobject, ontwerp en implementeer een architectuur die het handhaven van deze voorwaarden op de externe computer mogelijk maakt.

Bij het ontwikkelen van een volledig functioneel raamwerk voor het handhaven van voorwaarden zijn meerdere complementaire onderzoeksgebieden betrokken zijn, waaronder talen voor voorwaarden, modellering van voorwaarden, formele analyse en systeemarchitectuur ontwikkeling. Deze dissertatie gaat alleen in op het laatste onderzoeksgebied. Waar mogelijk wordt bestaand werk in de andere gebieden benut om de ontbrekende delen in het raamwerk op te vullen.

ONZE AANPAK

Het probleem van het handhaven van voorwaarden, zoals hier gedefiniëerd, kan vanuit verschillende invalshoeken benaderd worden en op verschillende niveaus van abstractie. Een aantal eerdere werken en systemen richten zich op handhaving van voorwaarden voor specifieke applicaties of klassen van applicaties. In deze werken is de logica die vereist is om de handhavings-besluiten te maken ingebouwd in de applicatiecode zelf. Anderen pakken het probleem aan op het lage niveau van het besturingssysteem en verkennen de complexiteit op het niveau van processen in het besturingssysteem en bepalen welk proces met welk ander proces mag communiceren of toegang heeft tot specifieke invoer- en uitvoerkanalen.

De klassen van voorwaarden die geïnterpreteerd (en dus gehandhaafd) kunnen worden hangen af van het niveau waarop voorwaarden worden gehandhaafd. Handhaving op het niveau van het besturingssysteem beperkt de klassen tot die welke direct te beschrijven zijn op proces- en *system call* niveau, terwijl handhaving op het niveau van specifieke applicaties de klassen beperkt tot die van applicaties en hun semantiek.

In deze dissertatie beschouwen we het probleem van handhaving van voorwaarden vanuit een gegevens-centrisch oogpunt, waarbij aangenomen wordt dat de voorwaarden gebundeld zijn met de gegevens waarop de ap-

plicaties werken. Ons werk benadert het probleem op *middleware* niveau, met de intentie om de mogelijkheden van de oplossingen op hoger en lager niveau te exploiteren. Deze aanpak stelt de architectuur in staat om gegevens-specifieke en niet applicatie-specifieke gebruiksvoorwaarden te handhaven over meerdere applicaties. Dit echter zonder het risico dat informatie op het niveau van de applicatiesemantiek verloren gaat, welke waardevol is bij handhaving van een grotere verscheidenheid aan klassen van voorwaarden. In het bijzonder beschouwen wij de handhaving van voorwaarden voor applicaties die in de *Java Virtual Machine (JVM)* omgeving draaien. De reden voor deze keuze en de details van het ontwerp van zo'n architectuur worden in deze dissertatie in detail beschreven.

Eén van de sleutelconcepten waarop wij bouwen in onze architectuur is *Information Flow Control (IFC)*, wat zich bezig houdt met beperkingen op hoe informatie overgedragen kan worden van één entiteit naar een andere. Hoewel IFC als onderzoeksonderwerp vanuit verscheidene invalshoeken onderzocht kan worden, bekijken wij het vanuit het perspectief van de semantiek van de programmeertaal van de applicatie. Onderzoek op het gebied van IFC kan grofweg onderverdeeld worden in twee verschillende categorieën: *compile time* en *run time*. In *compile-time* systemen worden de restricties op informatiestromen gecontroleerd en geverifiëerd ten tijde van het compileren. *Run-time* systemen, aan de andere kant, voeren deze controles dynamisch uit tijdens de uitvoering van de applicatie. Hoewel beide aanpakken hun voor- en nadelen hebben gebruikt onze architectuur een hybride aanpak, waarbij het *run-time* mechanisme wordt versterkt met statische *control flow* analyse. Hiervoor zijn twee hoofdafwegingen: het versterkte *run-time* systeem kan werken zonder toegang te hebben tot de daadwerkelijke broncode van de applicatie en deze hybride *run-time* aanpak kan een groter aantal klassen van voorwaarden handhaven.

BIJDRAGEN

In deze dissertatie presenteren wij het ontwerp, de implementatie en de toepassing van een architectuur voor het handhaven van voorwaarden gebaseerd op een *Java Virtual Machine*. De bijdragen van dit werk zijn als volgt:

– Wij onderzoeken in detail het eerdere werk gedaan op het gebied

van het handhaven van voorwaarden en identificeren de gaten die ons werk tot doel heeft te vullen.

– We presenteren het ontwerp en de implementatie van een op een JVM-gebaseerde *middleware* architectuur, genaamd Trishul, gericht op het handhaven van voorwaarden dat geassociëerd is met dataobjecten.

– De ontwikkelde *middleware* wordt gebruikt om een applicatie onafhankelijk *Digital Rights Management (DRM)* systeem te implementeren met als basis een veel gebruikt model voor de beheersing van gebruik.

– Het JVM-raamwerk wordt ook gebruikt om een *Web Service* architectuur te ontwerpen die in staat is om gebruiksvoorwaarden te handhaven dat geassociëerd is met aangeleverde gegevens, zoals gespecificeerd door de gegevens-aanbieder.

Eén van de sleutelfunctionaliteiten geïntroduceerd in dit nieuwe JVM-raamwerk was dat van *Information Flow Control*, welke het mogelijk maakte om zeer precies de gegevens te volgen terwijl deze in het systeem gebruikt werden, en welke ook zeer flexibele controle gaf over hoe de gegevens gebruikt konden worden in het systeem. We hebben ook een Java-achtige taal voorgesteld voor het ontwikkelen van de *core decision engine* van Trishul.

De prestatieanalyse d.m.v. *microbenchmarks* van de implementatie van Trishul in *just-in-time* modus bracht enkele delen van het systeem met hoge overhead aan het licht. Verdere analyse leverde ontwerp- en implementatiekeuzen op die deze overhead mogelijkerwijs kunnen verminderen. Deze keuzen kunnen beschouwd worden als toekomstig werk.

Om de kracht en flexibiliteit van het ontwikkelde Trishul systeem te laten zien en om aan te tonen dat het systeem inderdaad helpt bij het handhaven van aan gegevens verbonden voorwaarden, hebben we het als bouwsteen gebruikt voor een architectuur voor het handhaven van voorwaarden voor twee verschillende applicatiescenario's. In het eerste scenario ontworpen en implementeerden we een *Digital Rights Management* systeem dat in staat was meerdere typische voorbeelden van DRM-voorwaarden af te dwingen, geassociëerd met de multimediabestanden die weergegeven werden door de DRM applicaties die draaiden bovenop deze architectuur. In

het tweede scenario werd het Trishul raamwerk gebruikt om een architectuur voor het handhaven van voorwaarden voor *Web Services* te bouwen. Hierin werden de voorwaarden die door de gebruiker verbonden was met de gegevens en vervolgens opgestuurd naar de *Web Service* gehandhaafd in alle componentdiensten waaruit de *Web Service* opgebouwd was.

We hebben geverifiëerd dat de verschillende componentdiensten bovenop de voorgestelde uitvoeringsarchitectuur draaien en, vervolgens, hebben we de architectuur voor het handhaven van voorwaarden ontworpen m.b.v. Trishul op een veilige en uitbreidbare (modulaire) manier. Hiermee hebben we onze doelstelling van het aantonen van de kracht en flexibiliteit van het Trishul raamwerk bereikt.

TOEKOMSTIG WERK

Er zijn een aantal verschillende richtingen voor toekomstig werk, onder meer, open vragen die zijn geïdentificeerd.

De prototype-implementatie van Trishul, die ideaal is voor het tonen van de kracht en flexibiliteit van het systeem voor verschillende applicatiescenario's, is verre van efficiënt. Zoals besproken in Sectie 3.5.4 hebben we verschillende verbeterpunten geïdentificeerd. Het implementeren van die verbeterpunten zou als een richting voor toekomstig werk kunnen worden beschouwd.

We hebben een Java-achtige Trishul-P taal ontwikkeld om de ontwikkelaars van de voorwaarden *engine* in staat te stellen de logica van de *decision engines* uit te drukken, en deze op een efficiënte manier in het Trishul raamwerk te hangen. Hoewel de gelijkenis van de taal met Java de adoptie van het systeem makkelijker maakt, maakt dit het tegelijkertijd ook moeilijker om de uitdrukkingskracht van de taal en de algehele kracht van het handhavingssysteem met betrekking tot de klassen van de gehandhaafde gebruiksvoorwaarden te analyseren. De gestructureerde en logische beschrijving van systemen wiens interne *decision engines* beschreven worden als automaat-gebaseerde processtappen zijn ideaal voor gestructureerde analyse in vergelijking met de Turing-complete aard van Java-achtige talen.

Een andere onderzoeksrichting zou daarom ook zijn het onderzoeken van de implementatie van Trishul-P met een interne automaat-gebaseerde *engine* of de ontwikkeling van een *front-end* voor de Trishul-P interface in

de vorm van een automaat-gebaseerd systeem. In de laatste architectuur zou de schrijver van een voorwaarden *engine* de logica van de *engine* definiëren in de vorm van automaat transities van toegestane en niet-toegestane toestanden, die dan vervolgens, zoals eerder, gecompileerd zouden worden naar laadbare Java klassen voor gebruik in Trishul. Dit maakt een meer gestructureerde analyse van de kracht van de Trishul architectuur mogelijk en maakt tevens het ontwikkelen ervan makkelijker door bestaande oplossingen te hergebruiken.

Trishul laat zich niet uit over de invloed van *multi-threading* op informatiestromen of op de uitvoering van de applicatie. *Multi-threading* introduceert een nieuwe verzameling uitdagingen die in de toekomst aangepakt zouden kunnen worden.

In ons werk hebben we *trusted computing* technologieën benut om de integriteit van het platform te garanderen zodat zeker is dat de externe rekenomgeving is wat deze zou moeten zijn. Echter, het uitwerken van de details hoe dit in de praktijk te brengen is, vergt nog meer werk, in het bijzonder om te zorgen dat de gebruiksvoorwaarden onlosmakelijk verbonden blijven met de gegevens en niet ongewild veranderd kunnen worden. Hoewel deze dissertatie een aantal basale mechanismen voorstelt om 'klevende voorwaarden' te dwingen blijft toekomstig werk noodzakelijk om dit verder te beveiligen.

BIBLIOGRAPHY

Acharya, A. and Raje, M. (2000). MAPbox: using parameterized behavior classes to confine untrusted applications. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 1–17, Berkeley, CA, USA. USENIX Association.

Aho, A. V., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Massachusetts, USA, 2nd edition.

Andrews, G. R. and Reitman, R. P. (1980). An Axiomatic Approach to Information Flow in Programs. *ACM Trans. Program. Lang. Syst.*, 2(1):56–76.

Apache Software Foundation (2009). Apache Axis2. http://ws.apache.org/axis2/.

Baresi, L., Guinea, S., and Plebani, P. (2006). WS-Policy for service monitoring. In *Proceedings of International Workshop on Technologies for E-Services*, pages 72–83.

Barthe, G., Rezk, T., Russo, A., and Sabelfeld, A. (2007). Security of multithreaded programs by compilation. In *ESORICS*, pages 2–18.

Bauer, L., Ligatti, J. A., and Walker, D. (2005). Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA. ACM.

Bell, D. E. and LaPadula, L. J. (1975). Computer Security Model: Unified Exposition and Multics Interpretation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, USA.

Beres, Y. and Dalton, C. I. (2003). Dynamic Label Binding at Run-time. In *NSPW '03: Proceedings of the 2003 Workshop on New Security Paradigms*, pages 39–46, New York, NY, USA. ACM.

Berthold, A., Alam, M., Breu, R., Hafner, M., Pretschner, A., Seifert, J.-P., and Zhang, X. (2007a). A technical architecture for enforcing usage control requirements in service-oriented architectures. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services*, pages 18–25, New York, NY, USA. ACM.

Berthold, A., Alam, M., Breu, R., Hafner, M., Pretschner, A., Seifert, J.-P., and Zhang, X. (2007b). A technical architecture for enforcing usage control requirements in service-oriented architectures. In *SWS '07: Proceedings of the 2007 ACM workshop on Secure web services*, pages 18–25, New York, NY, USA. ACM.

Biba, K. J. (1977). Integrity considerations for secure computer systems. Technical Report ESD-TR 76-372, The MITRE Corporation, Bedford, MA, USA.

Bishop, M. (2002). *Computer Security: Art and Science*. Addison-Wesley Professional, Massachusetts, USA, 1st edition.

Brown, J. and King, T. F. (2004). A Minimal Trusted Computing Base for Dynamically Ensuring Secure Information Flow. Technical Report ARIES-TM-015, Massachusetts Institute of Technology, Cambridge, MA, USA.

Cabuk, S., Brodley, C. E., and Shields, C. (2004). IP Covert Timing Channels: Design and Detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 178–187, New York, NY, USA. ACM.

Chandra, D. (2006). *Information flow analysis and enforcement in Java bytecode*. PhD thesis, University of California, Irvine.

Chien, A. A., Calder, B., Elbert, S., and Bhatia, K. (2003). Entropia: architecture and performance of an enterprise desktop grid system. *J. Parallel Distrib. Comput.*, 63(5):597–610.

Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. (2004). Understanding data lifetime via whole system simulation. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 22–22, Berkeley, CA, USA. USENIX Association.

Dashti, M. T., Nair, S. K., and Jonker, H. (2009). Nuovo DRM Paradiso: Designing a Secure, Verified, Fair Exchange DRM Scheme. *Fundam. Inf.*, 89(4):393–417.

Denning, D. E. (1975). *Secure Information Flow in Computer Systems*. PhD thesis, Purdue Uinversity.

Denning, D. E. (1976). A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513.

Erickson, J. S. (2003). Fair use, DRM, and trusted computing. *Commun. ACM*, 46(4):34–39.

Erlingsson, U. (2004). *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University.

Fenton, J. S. (1973). *Information Protection Systems*. PhD thesis, University of Cambridge.

Fenton, J. S. (1974a). An abstract computer model demonstrating directional information flow. University of Cambridge.

Fenton, J. S. (1974b). Memoryless subsystem. *Computer Journal*, 17(2):143–147.

Fraser, T. (2000). LOMAC: Low Water-Mark Integrity Protection for COTS Environments. *Security and Privacy, IEEE Symposium on*, 0:230–245.

Garfinkel, T. (2003). Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS'03: Proceedings of Network and Distributed Systems Security Symposium*, pages 163–176.

Gat, I. and Saal, H. J. (1976). Memoryless Execution: A Programmer's Viewpoint. *Software: Practice and Experience*, 6(4):463–471.

Goguen, J. A. and Meseguer, J. (1982). Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. (1996). A secure environment for untrusted helper applications: Confining the wily hacker. In *SSYM'96: Proceedings of the 6th conference on USENIX Security Symposium*, Berkeley, CA, USA. USENIX Association.

Gong, L., Ellison, G., and Dageforde, M. (2003). *Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison Wesley, Massachusetts, USA, 2nd edition.

Gosling, J., Joy, B., and Steele, G. L. (1996). *The Java Language Specification*. Addison Wesley Publishing Company, Massachusetts, USA, 3rd edition.

Graham, G. S. and Denning, P. J. (1971). Protection: Principles and Practice. In *AFIPS '71 (Fall): Proceedings of the November 16-18,1971, Fall Joint Computer Conference*, pages 417–429, New York, NY, USA. ACM.

Guernic, G. L. (2007). *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University.

Guernic, G. L., Banerjee, A., Jensen, T., and Schmidt, D. A. (2006). Automata-Based Confidentiality Monitoring . In *Annual Asian Computing Science Conference ASIAN 2006*, pages 75–89. Springer Berlin / Heidelberg.

Haldar, V. (2006). *Semantic Remote Attestation*. PhD thesis, University of California, Irvine, California, USA.

Haldar, V., Chandra, D., and Franz, M. (2005a). Dynamic Taint Propagation for Java. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Washington, DC, USA. IEEE Computer Society.

Haldar, V., Chandra, D., and Franz, M. (2005b). Practical, Dynamic Information Flow for Virtual Machines. Technical Report UCI-ICS-TR-05-02, Irvine, California, USA.

Hammer, C., Krinke, J., and Snelting, G. (2006). Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *ISSSE '06: Proceedings of IEEE International Symposium on Secure Software Engineering*, pages 87–96.

Hardy, N. (1988). The Confused Deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38.

Inaba, K. (1998). What is trampoline code in Kaffe? http://www2.biglobe.ne.jp/~inaba/trampolines.html.

Jamkhedkar, P. A. and Heileman, G. L. (2004). DRM as a layered system. In *DRM '04: Proceedings of the 4th ACM workshop on Digital rights management*, pages 11–21, New York, NY, USA. ACM.

Jamkhedkar, P. A. and Heileman, G. L. (2008). A formal conceptual model for rights. In *DRM '08: Proceedings of the 8th ACM workshop on Digital rights management*, pages 29–38, New York, NY, USA. ACM.

JavaCC (2009). JavaCC. https://javacc.dev.java.net/.

Jif (2009). Jif: Java + information flow. http://www.cs.cornell.edu/jif/.

Jobs, S. (2007). Thoughts on Music. http://apple.com/hotnews/thoughtsonmusic.

Kaffe (2009). Kaffe.Org. http://www.kaffe.org.

Karjoth, G., Schunter, M., and Waidner, M. (2002). Platform for enterprise privacy practices: Privacy-enabled management of customer data. In *Privacy Enhancing Technologies*, pages 69–84.

Katt, B., Zhang, X., Breu, R., Hafner, M., and Seifert, J.-P. (2008). A general obligation model and continuity: enhanced policy enforcement engine for usage control. In *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 123–132, New York, NY, USA. ACM.

Kerberos Consortium (2009). Kerberos consortium. http://www.kerberos.org/.

Lam, L. C. and Chiueh, T. (2006). A General Dynamic Information Flow Tracking Framework for Security Applications. pages 463–472.

LaMacchia, B. A. (2002). Key challenges in DRM: An industry perspective. In *DRM'02: Proceedings of Digital Rights Management Workshop*, pages 51–60. Springer.

Lampson, B. W. (1971). Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton, New Jersey, USA.

Lampson, B. W. (1973). A note on the confinement problem. *Commun. ACM*, 16(10):613–615.

Ligatti, J. A. (2006). *Policy enforcement via program monitoring*. PhD thesis, Princeton University.

Lindholm, T. and Yellin, F. (1999). *The Java Virtual Machine Specification*. Prentice Hall, New Jersey, USA, 2nd edition.

Lipner, S. B. (1975). A comment on the confinement problem. In *SOSP '75: Proceedings of the fifth ACM symposium on Operating systems principles*, pages 192–196, New York, NY, USA. ACM.

McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. (2008). Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pages 315–328, New York, NY, USA. ACM.

Mclean, J. (1987). Reasoning About Security Models. In *IEEE Symposium on Security and Privacy*, pages 123–131, Los Alamitos, CA, USA. IEEE Computer Society.

McLean, J. (1990). Security models and information flow. pages 180–187.

Michiels, S., Verslype, K., Joosen, W., and Decker, B. D. (2005). Towards a software architecture for DRM. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 65–74, New York, NY, USA. ACM.

Microsoft Corporation (2009). Digital Rights Management. http://www.microsoft.com/windows/windowsmedia/forpros/drm/default.mspx.

Minsky, M. L. (1967). *Computation: Finite and Infinite Machines*. Prentice Hall, New Jersey, USA, 1st edition.

MIT (2003). The DynamoRIO Collaboration. http://www.cag.lcs.mit.edu/dynamorio/.

Mizuno, M. and Schmidt, D. A. (1992). A Security Flow Control Algorithm and Its Denotational Semantics Correctness Proof. *Formal Asp. Comput.*, 4(6A):727–754.

Mukhi, N. K. and Plebani, P. (2004). Supporting policy-driven behaviors in web services: experiences and issues. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 322–328, New York, NY, USA. ACM.

Myers, A. C. (1999). JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA. ACM.

Myers, A. C. and Liskov, B. (1997). A decentralized model for information flow control. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142, New York, NY, USA. ACM.

Nair, S. K. (2006). Policy Binding and Enforcement in Java. In *Proceedings of Workshop on Run-time Software Integrity and Authenticity*.

Nair, S. K. (2009). Trishul. http://purl.org/skn/trishul/.

Nair, S. K., Crispo, B., and Tanenbaum, A. S. (2009). Towards a Secure Application-Semantic Aware Policy Enforcement Architecture. In *Security Protocols: 14th International Workshop, 2006*, volume 5087 of *Lecture Notes in Computer Science*, pages 26–31. Springer.

Nair, S. K., Gheorghe, G., Crispo, B., and Tanenbaum, A. S. (2008a). Enforcing DRM Policies Across Applications. In *DRM '08: Proceedings of the 8th ACM workshop on Digital Rights Management*, pages 87–94, New York, NY, USA. ACM.

Nair, S. K., Popescu, B. C., Gamage, C., Crispo, B., and Tanenbaum, A. S. (2005). Enabling DRM-Preserving Digital Content Redistribution. In *Proceedings of Seventh IEEE International Conference on E-Commerce Technology*, pages 151–158, Los Alamitos, CA, USA. IEEE Computer Society.

Nair, S. K., Simpson, P. N. D., Crispo, B., and Tanenbaum, A. S. (2008b). A Virtual Machine Based Information Flow Control System for Policy Enforcement. *Electron. Notes Theor. Comput. Sci.*, 197(1):3–16.

National Security Agency (2009). Security-Enhanced Linux. http://www.nsa.gov/research/selinux/index.shtml.

Necula, G. C. (1997). Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA. ACM.

Newsome, J. and Song, D. X. (2005). Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS '05: Proceedings of 12th Network and Distributed System Security Symposium*.

OASIS (2004). *UDDI Version 3.0.2*. Organization for the Advancement of Structured Information Standards. http://uddi.org/pubs/uddi_v3.htm.

OASIS (2005). *Assertions and Protocols for the OASIS Security Assertion Markup Language*. Organization for the Advancement of Structured Information Standards. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

OASIS (2006). *Web Services Security: SOAP Message Security 1.1*. Organization for the Advancement of Structured Information Standards. http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

OASIS (2007). *Web Services Business Process Execution Language Version 2.0*. Organization for the Advancement of Structured Information Standards. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

OASIS (2008). OASIS eXtensible Access Control Markup Language. http://www.oasis-open.org/committees/xacml/.

ODRL (2002). Open Digital Rights Language (ODRL) Version 1.1. http://www.w3.org/TR/odrl/.

OMA (2009). Open Mobile Alliance. http://www.openmobilealliance.org/.

Park, J. and Sandhu, R. (2002). Towards Usage Control Models: Beyond Traditional Access Control. In *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 57–64, New York, NY, USA. ACM.

Park, J. and Sandhu, R. S. (2004). The UCON_{ABC} Usage Control Model. *ACM Trans. Inf. Syst. Secur.*, 7(1):128–174.

Pistoia, M., Banerjee, A., and Naumann, D. A. (2007). Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 149–163, Washington, DC, USA. IEEE Computer Society.

Portokalidis, G., Slowinska, A., and Bos, H. (2006). Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27.

Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. (2009). Laminar: practical fine-grained decentralized information flow control. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 63–74, New York, NY, USA. ACM.

Sabelfeld, A. and Myers, A. C. (2003). Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19.

Sadeghi, A.-R., Wolf, M., Stüble, C., Asokan, N., and Ekberg, J.-E. (2007). Enabling fairer digital rights management with trusted computing. In *ISC'07: Proceedings of 10th Information Security Conference*, pages 53–70. Springer.

Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. (2004). Design and Implementation of a TCG-based Integrity Measurement Architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA. USENIX Association.

Schaefer, C. (2007). Usage control reference monitor architecture. pages 13–18.

Schneider, F. B. (2000). Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50.

Shaffer, A. B., Auguston, M., Irvine, C. E., and Levin, T. E. (2008). A Security Domain Model to Assess Software for Exploitable Covert Channels. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 45–56, New York, NY, USA. ACM.

Shen, H. and Hong, F. (2006). An attribute-based access control model for web services. In *PDCAT '06: Proceedings of the Seventh International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 74–79, Washington, DC, USA. IEEE Computer Society.

Smith, S. W., Palmer, E. R., and Weingart, S. (1998). Using a High-Performance, Programmable Secure Coprocessor. In *FC '98: Proceedings of the Second International Conference on Financial Cryptography*, pages 73–89, London, UK. Springer-Verlag.

Sterne, D. F. (1991). On the buzzword 'security policy'. In *Proceedings of 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 219–230.

Strongin, G. (2005). Trusted computing using AMD "Pacifica" and "Presidio" secure virtual machine technology. *Information Security Technical Report*, 10(2):120–132.

Tang, Q. (2008). On using encryption techniques to enhance sticky policies enforcement. Technical Report TR-CTIT-08-64, University of Twente, Enschede.

Thomas, R. K. and Sandhu, R. S. (1998). Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented autorization management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI*, pages 166–181, London, UK, UK. Chapman & Hall, Ltd.

Trusted Computing Group (2006). TPM Specification Version 1.2 Revision 103: Part 1 - 3. http://www.trustedcomputinggroup.org/resources/tpm_specification_version_12_revision_103_part_1__3.

Trusted Computing Group (2009). Trusted Computing Group. http://www.trustedcomputinggroup.org.

US DoD (1985). Dep. Defense Trusted Computer System Evaluation Criteria. STD Document 5200.28-STD, U.S. Dep. of Defense.

Vachharajani, N., Bridges, M. J., Chang, J., Rangan, R., Ottoni, G., Blome, J. A., Reis, G. A., Vachharajani, M., and August, D. I. (2004). RIFLE: An

Architectural Framework for User-Centric Information-Flow Security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA. IEEE Computer Society.

van Bemmel, J., Wegdam, M., and Lagerberg, K. (2005). 3PAC: Enforcing access policies for web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 589–596, Washington, DC, USA. IEEE Computer Society.

Venners, B. (2000). *Inside The Java Virtual Machine*. McGraw-Hill Companies, Ohio, USA, 2nd edition.

Verma, K., Akkiraju, R., and Goodwin, R. (2005). Semantic matching of web service policies. http://lsdis.cs.uga.edu/~kunal/publications/SemanticPolicy-SWDP-final.pdf.

Viega, J., Bloch, J. T., and Chandra, P. (2001). Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39.

Volpano, D., Irvine, C., and Smith, G. (1996). A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187.

W3C (2005). *Web Services Choreography Description Language Version 1.0*. World Wide Web Consortium. http://www.w3.org/TR/ws-cdl-10/.

W3C (2006a). *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium. http://www.w3.org/TR/wsdl.

W3C (2006b). *Web Services Policy 1.2 - Attachment (WS-PolicyAttachment)*. World Wide Web Consortium. http://www.w3.org/Submission/WS-PolicyAttachment/.

W3C (2006c). *Web Services Policy 1.2 - Framework (WS-Policy)*. World Wide Web Consortium. http://www.w3.org/Submission/WS-Policy/.

W3C (2007). *SOAP Version 1.2*. World Wide Web Consortium. http://www.w3.org/TR/soap12/.

W3C (2009). Web Services @ W3C. http://www.w3.org/2002/ws/.

Wall, L. (1987). Perl security. http://perldoc.perl.org/perlsec.html.

Wallach, D. S. and Felten, E. W. (1998). Understanding Java Stack Inspection. pages 52–63.

Wespi, A., Dacier, M., and Debar, H. (2000). Intrusion detection using variable-length audit trail patterns. In *RAID '00: Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection*, pages 110–129, London, UK. Springer-Verlag.

Xu, W., Bhatkar, S., and Sekar, R. (2006). Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA. USENIX Association.

Yoshihama, S., Ebringer, T., Nakamura, M., Munetoh, S., and Maruyama, H. (2005). WS-Attestation: Efficient and fine-grained remote attestation on web services. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 743–750, Washington, DC, USA. IEEE Computer Society.

Zeldovich, N. (2007). *Securing Untrustworthy Software Using Information Flow Control*. PhD thesis, Stanford University, California, USA.

Zhang, X., Nakae, M., Covington, M. J., and Sandhu, R. (2008a). Toward a usage-based security framework for collaborative computing systems. *ACM Trans. Inf. Syst. Secur.*, 11(1):1–36.

Zhang, X., Seifert, J.-P., and Sandhu, R. (2008b). Security enforcement model for distributed usage control. In *SUTC '08: Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 10–18, Washington, DC, USA. IEEE Computer Society.

List of Citations

Acharya and Raje [2000], 92
Aho et al. [2006], 52
Andrews and Reitman [1980], 85
Apache Software Foundation [2009], 134
Baresi et al. [2006], 130, 137
Barthe et al. [2007], 92
Bauer et al. [2005], 91
Bell and LaPadula [1975], 9, 77
Beres and Dalton [2003], 15, 17, 87
Berthold et al. [2007a], 12, 118
Berthold et al. [2007b], 138
Biba [1977], 10
Bishop [2002], 7, 8
Brown and King [2004], 15, 87
Cabuk et al. [2004], 21
Chandra [2006], 88
Chien et al. [2003], 138
Chow et al. [2004], 18, 89
Dashti et al. [2009], 5
Denning and Denning [1977], 16
Denning [1975], 15, 85, 131
Denning [1976], 13, 14, 16
Erickson [2003], 100, 101
Erlingsson [2004], 19
Fenton [1973], 15
Fenton [1974a], 15, 17, 34, 86
Fenton [1974b], 87
Fraser [2000], 10
Garfinkel [2003], 92
Gat and Saal [1976], 15, 17, 87
Goguen and Meseguer [1982], 21
Goldberg et al. [1996], 92
Gong et al. [2003], 22, 47
Gosling et al. [1996], 4, 16
Graham and Denning [1971], 8
Guernic et al. [2006], 14, 90, 145
Guernic [2007], 87, 88, 90, 92, 145
Haldar et al. [2005a], 18, 89
Haldar et al. [2005b], 18, 89
Haldar [2006], 133
Hammer et al. [2006], 86
Hardy [1988], 24
Inaba [1998], 65
Jamkhedkar and Heileman [2004], 117, 118
Jamkhedkar and Heileman [2008], 119
JavaCC [2009], 71
Jif [2009], 85
Jobs [2007], 3, 96
Kaffe [2009], 48
Karjoth et al. [2002], 146
Katt et al. [2008], 12, 119
Kerberos Consortium [2009], 124, 137
LaMacchia [2002], 99
Lam and Chiueh [2006], 90

Lampson [1971], 8
Lampson [1973], 85
Ligatti [2006], 88, 91
Lindholm and Yellin [1999], 48
Lipner [1975], 85
MIT [2003], 17, 87
McLean [1990], 19
Mclean [1987], 10
McCune et al. [2008], 28, 116, 134
Michiels et al. [2005], 118
Microsoft Corporation [2009], 3, 117
Minsky [1967], 87
Mizuno and Schmidt [1992], 85
Mukhi and Plebani [2004], 125
Myers and Liskov [1997], 86
Myers [1999], 16, 18, 85
Nair et al. [2005], 5
Nair et al. [2008a], 5
Nair et al. [2008b], 5
Nair et al. [2009], 5
Nair [2006], 5
Nair [2009], 31
National Security Agency [2009], 76
Necula [1997], 19
Newsome and Song [2005], 89
OASIS [2004], 123
OASIS [2005], 134
OASIS [2006], 123, 136
OASIS [2007], 124
OASIS [2008], 103, 114, 137
ODRL [2002], 103
OMA [2009], 96, 97, 99, 117
Park and Sandhu [2002], 12, 19
Park and Sandhu [2004], 11, 99, 108
Pistoia et al. [2007], 24, 90
Portokalidis et al. [2006], 89
Roy et al. [2009], 92
Sabelfeld and Myers [2003], 18, 86
Sadeghi et al. [2007], 116, 117
Sailer et al. [2004], 26, 115, 132
Schaefer [2007], 118
Schneider [2000], 19
Shaffer et al. [2008], 21
Shen and Hong [2006], 122, 126, 137
Smith et al. [1998], 25
Sterne [1991], 7
Strongin [2005], 134
Tang [2008], 146
Thomas and Sandhu [1998], 11
Trusted Computing Group [2006], 25, 27, 116, 132
Trusted Computing Group [2009], 25
US DoD [1985], 8, 20
Vachharajani et al. [2004], 17, 18, 88
Venners [2000], 48
Verma et al. [2005], 125
Viega et al. [2001], 91
Volpano et al. [1996], 85
W3C [2005], 124
W3C [2006a], 123
W3C [2006b], 125, 137
W3C [2006c], 121, 124, 130, 137
W3C [2007], 122
W3C [2009], 121
Wallach and Felten [1998], 22, 24
Wall [1987], 90
Wespi et al. [2000], 92
Xu et al. [2006], 90
Yoshihama et al. [2005], 133

Zeldovich [2007], 3
Zhang et al. [2008a], 12, 119
Zhang et al. [2008b], 116, 119
van Bemmel et al. [2005], 126, 137