Thesis for the Degree of Doctor of Philosophy

Lightweight Enforcement of Fine-Grained Security Policies for Untrusted Software

Phu H. Phung

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2011

Lightweight Enforcement of Fine-Grained Security Policies for Untrusted Software
Phu H. Phung

ISBN 978-91-7385-578-5
© Phu H. Phung, 2011

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3259
ISSN 0346-718X

Technical Report 79D
Department of Computer Science and Engineering
Research group: ProSec Security

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers, Gothenburg, Sweden 2011

Lightweight Enforcement of Fine-Grained Security Policies for Untrusted Software

PHU H. PHUNG
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis presents an innovative approach to implementing a security enforcement mechanism in the contexts of untrusted software systems, where a piece of code in a base system may come from an untrusted third party. The key point of the approach is that it is *lightweight* in the sense that it does not need an additional policy language or extra tool. Instead, the approach uses the aspect-oriented programming paradigm – a programmatic means to modify the behaviour of an application based on *aspects* – to specify security policies and *embed* the policies into untrusted software. As a result, the policies can be fine-grained and application-specific, and can be *inlined* into the untrusted software without modifying the base system, in order to detect and prevent unintended behaviour of the software at runtime. The approach has been elaborated in two particular untrusted software contexts in this thesis.

Firstly, we have developed the approach in the context of a vehicle software architecture, where a third-party application can be installed and executed in a vehicle system. We have shown that various classes of fine-grained security policies can be specified and enforced in such a system by the approach. The security assurance provided by the enforcement mechanism is promising for deployment in an existing vehicle software system. Furthermore, we have identified a number of potential threats in the vehicle software architecture and developed countermeasures in terms of security policies. We have demonstrated the deployment of countermeasures to prevent possible attacks.

Secondly, we have studied web application security. We propose a novel enforcement method called *lightweight self-protecting JavaScript* by applying the lightweight approach in the context of web security. The method prevents or modifies inappropriate behaviour of JavaScript execution in web pages by intercepting security relevant API calls. Unlike other approaches to enforcing policies for JavaScript, the enforcement and policy code are provided as a library and therefore do not require a modified browser. Furthermore, the approach does not employ runtime parsing or transformation of code, and thus has low runtime overhead. We also present an application of the method in the context of untrusted JavaScript such as *mashups* by proposing a two-tier sandbox architecture in which untrusted JavaScript code can be loaded and executed dynamically. The execution of untrusted code is monitored by modular and fine-grained security policies defined via an adaptation of self-protecting JavaScript to ensure security for the hosting page.

Keywords: *security policy enforcement, vehicle software security, web-application security, JavaScript security, untrusted software*

Luận án này con xin dâng tặng Ba Mạ,
cho Thụy Uyên, con gái yêu của Ba,
và cho vợ Quỳnh Anh, đã chia sẻ cùng anh
trong suốt hành trình 5 năm qua.

This thesis is dedicated to my Parents,
to my angel daughter Thụy-Uyên,
and to my beloved wife Quỳnh-Anh.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my supervisor Dave Sands for his constant support and guidance. I am greatly indebted to him for his advice, encouragement, and every support during the last five years, especially during the last two months for his time for this thesis. Not only an excellent supervisor, Dave has also been a great friend in life with generous support, understanding, and sharing life experience.

I would also like to thank Andrei Sabelfeld and Marina Papatriantafilou for being in my PhD advisory committee with helpful comments and suggestions on my studies and this thesis.

Many thanks to V.N. Venkatakrishnan from University of Illinois at Chicago for being my faculty opponent and for providing useful feedback on my thesis draft, and to Frank Piessens from KUL for insightful discussions on my work, great comments on my thesis draft, and for being a member of my grading committee.

I acknowledge Aslan Askarov, Alejandro Russo, Daniel Hedin, Josef Svenningsson, Filippo Del Tedesco, Arnar Birgisson, Willard Rafnsson, current and former members in the ProSec Security research group of which I feel honoured and privileged to be a member. Special thanks go to Jonas Magazinius for in deep discussions on my research and for collaboration on one of the papers, to Niklas Broberg for being a good badminton partner.

I would like to take this opportunity to thank the faculty, the administrative staff and other current and former PhD students at the department for providing such a nice, friendly, and productive working environment. Special thanks to Dennis K. Nilsson and Ulf E. Larson for being the co-authors, to Vilhelm Verendel for sharing life experience, to Per Waborg for being a good ping-pong partner, and to Phuong Hoai Ha, Minh Quang Do, Philippas Tsigas, and Elad Schiller for the support from the beginning of my PhD studies until now.

The presentation of material in this thesis has been much improved by the comments of Hong-Linh Truong and some other friends. Thank you all!

I greatly appreciate John Mitchell at Stanford who hosted me for a research visit during summer 2010 from which I started the work in the last paper in this thesis. Thanks Ankur Taly for insightful discussions on the work.

Many thanks to my Vietnamese colleagues at Chalmers, especially to Tuan A. Le, Tung Hoang, Nhan Nguyen, Khoa Huynh and friends in Göteborg for their friendship, and for creating such a pleasant "Vietnamese life" in Sweden.

Last and most significantly, I wish to express my love and appreciation to my Parents for their love and support throughout my life, and for always encouraging my studies, to my beloved wife, Quỳnh Anh, for her continuous patience, understanding, and support during the last five years, and to my 24 day-old daughter Thụy Uyên for her wonderful arrival to our life. Special thanks go to my Mother-in-law, who has been with us in Göteborg for the last two months. Her time here is not only for her daughter and little granddaughter, but also help me to complete this thesis writing on time.

Phùng Hữu Phú

Göteborg, September 12, 2011

This research was partly funded by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under the SESAME project (in collaboration with Volvo Technology), the Swedish Research Council (VR), the Swedish Foundation for Strategic Research (SSF), and the European Commission under the WebSand project.

Preface

This thesis comprises a collection of papers contributing to the topic of security of untrusted software. The thesis is divided into two parts. Part I gives an overview of the thesis, summarises the included papers, discusses some practical issues, more recent related work, and some shortcomings which have not been discussed in the included papers. Part II reprints four published papers and one technical report, listed in order of publication as follows:

1. Phu H. Phung, and David Sands. Security Policy Enforcement for the OSGi Framework using Aspect-Oriented Programming. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC'2008)*, July 28- August 01 2008, Turku, Finland, pp. 1076-1082, IEEE Computer Society. ISSN: 0730-3157. ISBN: 978-0-7695-3262-2.

2. Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS' 2009)*, 10 - 12 March 2009, Sydney, Australia, ACM Press.

3. Phu H. Phung, and Dennis K. Nilsson. A Model for Safe and Secure Execution of Downloaded Vehicle Applications. In *Proceedings of Road Transport Information and Control - RTIC 2010 and ITS United Kingdom Members' Conference, IET*.

4. Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. Accepted at OWASP AppSec Research 2010, and published in *Proceedings of The 15th Nordic Conference in Secure IT Systems (Nordsec'10)*, LNCS, to appear.

5. Phu H. Phung. A Two-Tier Sandbox Architecture to Enforce Modular Fine-Grained Security Policies for Untrusted JavaScript. *Technical report No.2011:11-* Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, ISSN 1652-926X.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
CAN	Controller Area Network
CLI	Common Language Infrastructure
DOM	Document Object Model
ECU	Electronic Control Unit
ECUs	Electronic Control Units
FOTA	Firmware Updates Over The Air
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
GPS	Global Positioning System
GST	Global System for Telematics
IP	Internet Protocol
IRM	Inlined Reference Monitor
J2ME	Java 2 Micro Edition
JVM	Java Virtual Machine
JVML	Java Virtual Machine Language

LIN	Local Interconnect Network
OS	Operating System
OSGi	Open Services Gateway initiative
ROM	Random Access Memory
SASI	Security Automata SFI Implementation
SFI	Software-based Fault Isolation
SMS	Short Message Service
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language
XSS	Cross-site Scripting

Part I

Thesis Overview

Chapter 1

Introduction

Many modern software programs are application extensions, running within application platforms rather than running as separate processes in an operating system [Tan01, UE04]. Common examples of such programs include macros running on Microsoft Word or Excel, web browser plug-ins, JavaScript in web pages running on web browsers or email clients. Especially, with the advent of Internet, modern software systems may consist of several software components which are loaded from external or third-party sources. We refer such components as *untrusted software*, since the components loaded from external sources are possibly untrusted. Users of untrusted software, therefore, face security problems because untrusted software is potentially malicious and their execution may be harmful.

Conventional security mechanisms such as encryption, firewalls, anti-virus programs, and operating system-based access control cannot solve the problems of untrusted software because these mechanisms deal with the operating system-level and treat programs as a black-box and thus cannot *intercede inside* each application to enforce security. This is the source for some attacks in the past such as Melissa.V[1] or ILOVEYOU[2] worms which propagate themselves by a piece of script code hidden within an email message that is executed by the email-client when being opened. These worms were successful because the worm scripts are executed within the email-client application in which operating system level enforcement mechanisms cannot monitor. These limitations motivate the development of application-level security approaches generally known as *language-based security* where the security mechanisms use the developed techniques of programming lan-

[1]http://en.wikipedia.org/wiki/Melissa_(computer_virus)
[2]http://en.wikipedia.org/wiki/ILOVEYOU

guages [SMH01].

One of the approaches in language-based security to providing security for untrusted software is to monitor the execution of potential untrusted code so that any unintended behaviour can be detected and prevented according to some pre-defined security policies[3]. These *security policy enforcement* mechanisms consist of two parts: security policy definition and policy enforcement mechanism. A *security policy* defines which circumstance a behaviour is allowed to be executed, while a policy enforcement mechanism ensures that the behaviour of untrusted code complies with a security policy. A classical approach to implementing a security policy enforcement mechanism is the *security reference monitor* [And72] which imposes a security policy on an otherwise untrusted system. There are several ways to implement a security reference monitor. A traditional approach is to place it at the operating system level with hardware support to mediate system calls. This approach, however, cannot enforce security at the software level for untrusted software scenarios. Another implementation approach is to put applications running inside a virtual machine at the base system and implement the reference monitor as part of the base system so that the execution of application instructions is monitored by the reference monitor. This requires the base system running the untrusted software to be modified. A variant of the reference monitor concept is the automatic transformation of the application to check security-relevant events at application level at runtime to ensure its security.

Implementing a security policy enforcement mechanism for untrusted software using application level reference monitors is appealing because it can monitor the behaviour of the software at runtime with application-specific policies and it does not need to modify the base system running the untrusted software. Two key criteria for comparing security policy enforcement mechanisms are how expressive a security policy is specified and which mechanism a policy is enforced. The expressiveness of a security policy ranges from *coarse-grained* to *fine-grained*. *Coarse-grained* security policies can only define a limited set of conditions for program execution. *Whitelist* access control is a coarse-grained security policy example which only specifies a list of operations that are allowed to run; operations not in the list are denied. On the contrary, *fine-grained* security policy can define a richer set of conditions, not only allowing or denying an operation, but the decision on an operation depends on some conditions such as the his-

[3]Other related approaches for untrusted software security problem are reviewed and discussed in Chapter 2 (Section 2.1.

tory of the program execution. For example, stateful security policies are fine-grained policies which can specify in which conditions, based on some security states recording program execution history, an operation is allowed to execute. Security policies are defined in a specific language to be enforced by a policy enforcement mechanism.

Recently, there has been considerable interest in using reference monitors at a purely software level – for example by rewriting software components to "inline" security policies within them to provide expressive and efficient application-specific security policies for the software components, e.g. in [FBF99, UE04, Lig06, Ham06]. These approaches provide application level reference monitors for security policy enforcement. The implementation of these approaches consists of a trusted rewriter tool and a security policy language. Based on security policies defined in the policy language, the rewriter transforms a target application into a secured version that embeds security code which can monitor the application execution at runtime. A security policy defines which *security-relevant events* are monitored and which *security checks* are performed at a corresponding security-relevant event. A security check is a function that verifies whether an event is in a secure state. Examples of such approaches include: PoET-/PSLang [UES00b], Naccio [ET99], Polymer [BLW05], *Mobile* [Ham06], or ConSpec [AN08]. These are enforcement implementations consisting of a policy language and a software tool to transform an insecure program to a secured version of the program based on policies defined in the language.

The studies and realisations of the above mentioned enforcement approaches have shown that the approaches are quite promising, however, they face several drawbacks. Firstly, most existing enforcement tools in these approaches are research prototypes and lack the robustness and completeness of mature industrial tools. This makes them difficult to deploy in real contexts. Furthermore, these approaches need an additional language rather than the language of the software to specify security policies. This of course benefits in having policies constrained by e.g. a static type system [GSK10], however, the classes of policies are less expressive than those written in the language of the software. These points motivate an alternative approach to implementing security policy enforcement mechanism in which security policies can be expressively defined in the source language (the language of the untrusted software) and the policy enforcement can be implemented by a mature industrial tool. This thesis explores such an alternative approach by proposing a lightweight approach to security policy enforcement in order to enforce fine-grained security policies for untrusted software. The approach is lightweight in sense that it does not use an additional language other than the base language of the untrusted software to define security policies and uses an off-the-shelf implementation of the tool or plain library code to realise the enforcement. The overview of our approach is presented as follows.

1.1 Overview of the Approach

Independently of the security policy enforcement, aspect-oriented programming (AOP) [KLM+97] is a programming paradigm providing programmatic means to modularise the *cross-cutting* functionalities of complex software systems so that program concerns in a software system can be captured and encapsulated into so-called *aspects*. An *aspect* comprises a *pointcut*, which defines the point and the condition under which the aspect modifies the behaviour of an application, and an *advice*, which defines what modifications should be applied. At compile-time, a process called *weaving* is performed to analyse and modify the target program[4] by matching pointcuts and inserting advice. In principle, these features of AOP make its language and weaver tool suitable as an implementation of security policy enforcement mechanism where security policies could be defined by *aspects* and runtime security checks could be defined in *advice*.

This thesis studies an alternative realisation of a security policy enforcement mechanism by taking the advantages of the *aspect-oriented programming* [KLM+97] paradigm rather than using a security-specific policy language and program rewriting tool, or designing a new policy language and tool. We call this a "*lightweight approach*" to security policy enforcement for untrusted software. In the following subsections, we elaborate the key features of our realisation approach and relate to previous implementations of security policy enforcement mechanisms. We first characterise fine-grained security policies, and the class of security policies which our approach can describe and enforce. Second, we present the overview of lightweight features of the implementation of the enforcement mechanism.

1.1.1 Fine-grained security policies

Although the term "security policy" is fundamental to computer security [Ste91], a security policy has broad meaning. Conventional security policies are normally *coarse-grained* which is based on "all-or-nothing" approach,

[4]Note that the target program is in its source language or could be in object code, e.g. Java bytecode, not in an AOP language.

i.e. allow or deny an operation. These coarse-grained policies normally deny security-critical operations that may be harmful to the base system, for example do not allow the software to access system files. Smartphone operating systems such as Android currently implement these coarse-grained security policies for third-party applications installed on the phone [EGgC$^+$10]. For instance, when installing an third-party application, an Android-based phone user must allow the application to access a list of operations it requests but cannot specify only some separate operations as the user wishes. This class of policies is not adequate for untrusted software since it may need to use security-critical operations to function. As an example, suppose a smartphone user wishes to install a third-party software, and the software needs to be able to send SMS (text) messages in order to function properly. With *coarse-grained* security policies, the user can only allow or disallow the third-party software to send SMS. If the policy is to disallow to send SMS, the software of course cannot run properly. On the other hand, allowing the third-party software to send SMS, the mobile device still faces potential security problems: the software could be malicious and deliberately send too many messages, or the software may simply have bugs, causing it, under certain circumstances, to repeatedly send messages. This scenario motivates the need for more expressive security policies. Such an expressive policy example for the software might be something like the following [PS08]: "*allow a third-party software to use the SMS service, but:*

- *restricted to a specific recipient address,*
- *with a limit on the number of messages sent per day, and*
- *depending on the mobile's location."*

The above policy is one of the examples of "*fine-grained*" security policies which can be enforced for untrusted software in this thesis. Differing from previous security policy enforcement mechanisms, e.g. [ET99, UE04, BLW05, Ham06, AN08] which define a policy specification language to express security policies, our approach is to use a base language of the untrusted software in aspect-oriented programming style to define security policies, and thus we do not need an additional security policy language. As this thesis employs a reference monitor approach to security enforcement for untrusted software, our policies are reference monitor-style which define *unacceptable* behaviour. Since a security policy in our approach can be defined in a base language, it is *fine-grained* in the following sense:

application-specific: the policies can be defined specifically to each untrusted software.

stateful: security states can be defined and updated at runtime so that policy decisions can be based on execution history of the untrusted software.

expressive response actions: a classical reference monitor-style policy only stops an execution if it violates the policies. Our security policy specifications provide more powerful transformational abilities (inspired by the edit automata [LBW05]) such as the ability to *suppress* actions, *replace* actions, *insert* new actions, and to *truncate* an execution.

1.1.2 Lightweight enforcement of fine-grained security policies

The approach proposed in this thesis is to implement a security policy enforcement mechanism in a reference monitor style using an aspect-oriented style programming language. This has the benefits of providing a relatively complete and well-tested tool which can be applied at an appropriate level. The enforcement mechanism is *lightweight* in that:

i) it uses an off-the-shelf implementation of the tool,

ii) it does not need an additional security policy language, and

iii) it does not require modification of the base system.

However, an aspect-oriented programming language provides neither direct support for policy specification nor enforcement. Therefore, one of the goals of this thesis is to study whether an aspect-oriented programming language is suitable and adequate for a particular untrusted software system. In particular, we consider the following main questions in this study:

1. What classes of fine-grained security policies can be defined and enforced?

2. How can the approach be integrated with a base system without modifying the base system?

3. What security assurances can the approach provide to a particular untrusted software system?

4. What are the shortcomings of the approach?

To answer the above research questions, one needs to study the approach in particular untrusted software systems. In this thesis, we consider two areas of untrusted software to study the approach. The next section introduces the two untrusted software systems and the application of the approach in each system.

1.2 The Lightweight Enforcement Approach to Problem Areas

We study the lightweight enforcement of fine-grained security policies for two particular untrusted software systems. Firstly, we consider security aspects for a vehicle software architecture, analyse security threats and present the needs for fine-grained security policy enforcement in the architecture, and propose a lightweight enforcement approach to enforcing security policies for the vehicle software architecture. Secondly, we employ the lightweight enforcement approach to the context of web browser security. The enforcement method is called "*lightweight self-protecting JavaScript*" which controls and modifies the behaviour of JavaScript to make it self-protecting. In the following subsection, we describe each area, motivate the need for an alternative lightweight approach to implementing fine-grained security policy enforcement, and summarise our approaches in each area.

1.2.1 A lightweight enforcement approach to a vehicle software architecture

Vehicle systems today consist of many infotainment/telematics applications. The infotainment category consists of systems for information and entertainment for the driver and passengers in a vehicle, including digital broadcasting TV, audio streams, TFT displays as well as systems receiving data from external sources, e.g. traffic and weather information systems. The telematics category includes systems that integrate telecommunications and informatics. Such systems are used to provide networked software applications to the vehicle. Such infotainment/telematics applications not only run inside a vehicle but also communicate with other vehicles or servers in order to use external services such as road charges, traffic, weather, and travel support. A recent trend in telematics applications is to establish an open telematics market that allows the different players of the value chain to easily develop, implement and deploy new functionality or sub-systems [GST]. The open telematics market is dedicated to lightweight execution environments and enables third-parties to develop services that can be integrated into vehicle systems. The Open Services Gateway initiative (OSGi) [OSG07] is a framework which has been considered as one of the most prominent standards and an ideal solution for the open telematics market [GST]. OSGi is an open software architecture providing a *collaborative* software environment running on a Java virtual machine. An application in OSGi is composed of different components called *bundles* that can be downloaded and installed from external sources. Bundles can be developed by third parties and can be integrated with existing bundles to create new applications.

This trend also requires IT security considerations in order to guarantee system security. The OSGi specification Service Platform Release 4 (version 4.1, May 2007 [OSG]) has a security layer which is based on the Java 2 security model. In this security model, called a *sandbox*, all program code – regardless of whether it is installed locally or downloaded remotely– can be subjected to a security policy configured by a Java Virtual Machine (JVM) user. But this model unduly restricts the function of third-party services in the sandbox while these services might need to use system resources and sensitive information, such as sending SMS messages or getting GPS location from the vehicle system. In addition, although the security layer in OSGi also provides a mechanism to sign and validate code to ensure that the deployed code comes from a trusted source, this mechanism only certifies the origin and the integrity of the downloaded code, and is not able to address possible malicious actions such as deliberately sending too many SMS messages, for instance, to a high-cost service. Moreover, in an open environment it is hard to establish meaningful trust relationships, and even when one can, trust is not equated with quality. For example, a trusted third-party service may simply have bugs, causing it, under certain circumstances, to repeatedly send messages.

In this thesis, we have proposed a lightweight approach to enforcing fine-grained security policies for the vehicle application architecture. We study the implementation of security policy enforcement using AspectJ [Asp], an aspect-oriented programming language for Java, for the OSGi framework in the vehicle application architecture. We have identified and demonstrated the use of AspectJ to specify fine-grained security policies for untrusted software in the OSGi framework. The policies are enforced by being embedded in the untrusted software using the weaver tool of AspectJ. To the best of our knowledge, our work is the first study of security policy enforcement using an aspect-oriented language in an open system for untrusted software like the OSGi framework. We further study the application of our lightweight enforcement approach to the vehicle software architecture by developing a security model in order to prevent potential *cyber attacks* in the vehicle systems. We analyse possible threats and potential cyber attacks including add-on software and ECU firmware. For each threat, we introduce a corresponding countermeasure that can be specified as a security policy. We have proposed a model to deploy the countermeasures by using our security policy enforcement method in the vehicle context.

These contributions have been published in [PS08], and [PN10] which are included in this thesis as Paper A and Paper B, respectively. The contributions are summarised and reviewed in Chapter 3, where we also discuss some limitations of the approach and further directions to solve the issues.

1.2.2 A lightweight enforcement approach to web application security

The second domain we study and apply our lightweight enforcement approach to is the area of web application security. In this subsection, we review web application security at the client side, motivate a lightweight approach to fine-grained security policy enforcement, and summarise our contributions in this area.

Web application security becomes more and more important today as many financial transactions are performed over the web, and a lot of sensitive information such as credit card numbers are routinely handled by web applications. Although the sensitive information is encrypted to make it secret when being sent over the Internet, there are certain vulnerabilities in practice that allow attackers to steal the information. One way for the attacker to launch such an attack, e.g. steal user information from a web page, is to inject malicious code into the web page such that the malicious code is executed with the privileges of the web page. A web page normally contains script code written in JavaScript which is executed in the web browser to enhance the web page with dynamic contents. Running script code with the privileges of the web page means that the script can access any sensitive information of the web page, such as user information. The attacker can inject malicious JavaScript code into a web page by e.g. typing a piece of code in some unprotected user input fields in the web page, or in some fields that the web developers have failed to filter. This class of attacks is known as cross-site scripting (XSS) [XSS] where malicious scripts are injected into a web page by the attacker. According to the OWASP Top 10 Web Application Security Risks for 2010, XSS is the second largest security risk in web applications [OWA].

Preventing such XSS attacks is a difficult task in practice. Server-side defence mechanisms are a common approach in industry to deal with XSS. These mechanisms filter user input at the server-side to ensure that no script exists in user input. However, it has been shown in practice that these careful server-side filtering mechanisms can be evaded to inject malicious JavaScript code. The Samy [Sam] and Yamanner [Sec] worms are two popular real-world examples of XSS attacks where malicious JavaScript code is injected into web pages by disguising it so as to escape server filtering [Lev06, Rob].

An alternative to filtering mechanisms deployed on servers is to control the behaviour of the JavaScript code at runtime (in the client's browser) instead of statically validating the integrity of the code at the server. Suppose that the attacker can inject a piece of malicious JavaScript code in a given page. It may be enough to control the code execution in order to ensure that the code in the page does not behave in an unintended manner, such as sending sensitive information to the attacker. One way to implement this idea is to specify a policy which says under what conditions a page may perform a certain action, and enforce the policy at runtime to ensure that the code does not violate the desired policy. We refer this as the *client-side defence* approach, since security policies are normally injected into the web page somewhere between the server and the client, e.g. at server-side, proxy, or browser plug-in (client-side), and are enforced in the browser at runtime (c.f. [UELX07]). The attacks which evade filtering mechanisms rely on an inconsistency between the browser's view of the syntax of the script and the view of the filtering tool. The behavioural approach, however, has no such vulnerability.

Such *client-side defence* approaches have been widely studied in the literature[5]. Relying on a client-side defence approach, we propose a method called "*lightweight self-protecting JavaScript*" by employing our lightweight enforcement approach to enforcing security policies for JavaScript. One of the novel ideas of the method is that it does not require any aggressive code manipulation such as parsing or transforming JavaScript code in the body of the page at all. Instead, the policy code is assumed to be injected into the header of the web page at the server or by a trusted proxy. Injecting the policy code into the header ensures that the policy code is executed first, so the policy code gets to wrap the security critical methods before the attacker code can get a handle on them. A policy is a piece of JavaScript code specifying which method calls are to be intercepted, and what actions are to be taken. The implementation of policy enforcement is based on a

[5]These approaches are reviewed in detail in Section 4.1.1.

plain JavaScript library, without modifying the browser. As a result, unlike previous client-side approaches to instrumenting and monitoring JavaScript to enforce or adjust behaviour, our method can enforce fine-grained security policies for JavaScript code at runtime but (i) it does not require a modified browser, and (ii) it does not require any runtime parsing and transformation of code (including dynamically generated code) thus having low runtime overhead.

In this thesis, we present our *lightweight self-protecting JavaScript* method in Paper C. We address some security issues in the implementation of *lightweight self-protecting JavaScript* (which was born as an adaptation of a non security-oriented aspect-oriented programming library) and we provide a systematic way to avoid the identified vulnerabilities, and make it easier for the policy writer to construct declarative policies – i.e. policies upon which attacker code has no side effects. This work is presented in Paper D.

We have applied *lightweight self-protecting JavaScript* in the context of untrusted JavaScript by proposing a two-tier sandbox architecture, in which untrusted JavaScript code can be loaded and executed dynamically. A web mashup – a web application that integrates content such as data or code from multiple providers – is an example of untrusted JavaScript. As an example, a page containing a third-party advertisement is a mashup in which an ad, implemented as JavaScript code, is embedded into a hosting web page to display the ad content. This raises security problems for the hosting page since the ad code is potentially untrusted and malicious. In our proposed two-tier sandbox architecture, the execution of untrusted code, e.g. a component in a mashup, is constrained by modular and fine-grained policies. Policies are specified via an adaptation of the lightweight self-protecting JavaScript mechanism. By enforcing fine-grained policies in a sandbox, untrusted code is ensured not to perform harmful actions on the hosting page. This work is presented in Paper E in this thesis.

These contributions of the lightweight enforcement of fine-grained security policies for JavaScript are summarised and discussed in Chapter 4.

1.3 Thesis Contributions

Parts of the results in this thesis have been published in the following technical papers:

Paper A. Phu H. Phung, and David Sands. Security Policy Enforcement for the OSGi Framework using Aspect-Oriented Programming. In *Proceedings of the 32nd Annual International Computer Software and Ap-*

plications Conference (COMPSAC'2008), July 28- August 01 2008, Turku, Finland, pp. 1076-1082, IEEE Computer Society. ISSN: 0730-3157. ISBN: 978-0-7695-3262-2.

Paper B. Phu H. Phung, and Dennis K. Nilsson. A Model for Safe and Secure Execution of Downloaded Vehicle Applications. In *Proceedings of Road Transport Information and Control - RTIC 2010 and ITS United Kingdom Members' Conference, IET.*

Paper C. Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS' 2009)*, 10 - 12 March 2009, Sydney, Australia, ACM Press.

Paper D. Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. Accepted at OWASP AppSec Research 2010, and published in *Proceedings of The 15th Nordic Conference in Secure IT Systems (Nordsec'10)*, LNCS. To appear.

Paper E. Phu H. Phung. A Two-Tier Sandbox Architecture to Enforce Modular Fine-Grained Security Policies for Untrusted JavaScript. *Technical report No.2011:11*- Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, ISSN 1652-926X.

In summary, the contributions of this thesis fall into two folds:

Runtime Enforcement for Downloaded Vehicle Applications: Our lightweight enforcement approach has been deployed in a vehicle software architecture to provide security for untrusted applications downloaded from external sources. The untrusted applications are transformed, based on predefined policies, by the enforcement mechanism before deployment. Applying our mechanism, the execution of vehicle applications are monitored at run-time so that malicious behaviours violating the policies can be prevented and thus make the vehicle platform secure and safe. We have applied the enforcement mechanism to implement proposed countermeasures to prevent a number of concrete possible threats that we have identified in the vehicle software architecture. These contributions have been published in [PS08] (Paper A), and [PN10] (Paper B).

Lightweight Self-Protecting JavaScript: We have studied the lightweight enforcement approach in the context of web browser security and proposed a method called *lightweight self-protecting JavaScript*. The contribution of the method is that it can enforce fine-grained policies for JavaScript at run-time without browser modification. Using the self-protecting JavaScript method, a web developer can specify application-specific security policies for her web application and inject the policy and enforcement code into the web application to prevent e.g. cross-site scripting attacks. The method has been described and published in [PSC09] (Paper C), and a hardened implementation of the method has been published in [MPS10] (Paper D). We have also studied an application of self-protecting JavaScript by proposing a two-tier sandbox architecture in which untrusted JavaScript code can be loaded and executed dynamically within a sandbox environment. The execution of untrusted code in the environment is monitored and controlled by an adaptation of the self-protecting JavaScript method to ensure security for the hosting page running the untrusted JavaScript. This work has been reported in [Phu11] (Paper E).

1.3.1 Statement of personal contributions

I hereby state my personal contributions on joint-work publications.

Security Policy Enforcement for the OSGi Framework using Aspect-Oriented Programming: This paper has been written under the supervision of David Sands. I have written and developed most of the technical material.

A Model for Safe and Secure Execution of Downloaded Vehicle Applications.: Dennis Nilsson and I have made equal contribution in designing and implementing the system. I have written and developed most of the technical material.

Lightweight Self-Protecting JavaScript: I have contributed to the design, implementation and experimental evaluation of the methods and policies presented in the paper. I have written and developed most of the technical material.

Safe Wrappers and Sane Policies for Self Protecting JavaScript: I have partially contributed to the design, and implementation of the revised libraries fixing the identified vulnerabilities. I and the co-authors have made equal contribution in developing technical material.

Enforcing Modular Fine-Grained Security Policies for Untrusted JavaScript in EcmaScript 5: I have written and developed the paper by myself.

1.4 Thesis Organisation

This thesis contains two parts: an overview of the thesis (Part I) and included papers (Part II). **Part I** contains five chapters including this introduction chapter. The next chapter (**Chapter 2**) reviews in detail the background of the thesis including state-of-the-art literature and related work. First, approaches to untrusted software security are reviewed to provide a broader view of the area. Then, we discuss how reference monitors can be implemented by transforming programs so that security checks are inlined in the code – the so-called *inlined reference monitor* approach. Finally, we review the specific technology we apply in this thesis: aspect-oriented programming, and, together with a particular language such as the AspectJ language, and an AOP implementation for JavaScript.

In **Chapter 3**, we summarise our lightweight approach to enforcing fine-grained security policies for vehicle application architecture. We first review the vehicle software architecture and the OSGi framework used in vehicle systems. Then we introduce our approach to deploying the implementation of security policy enforcement using aspect-oriented programming for the OSGi framework.

We also present our further study on applying the lightweight enforcement approach to a vehicle application architecture to prevent potential cyber attacks in the system. In that work, we analyse the architecture to define the problems, and identify threats as well as propose countermeasures for the threat model. We present a model to deploy a reference monitor approach for vehicle applications for monitoring their execution to enable safety and security. Finally, we discuss some practical issues which have not been mentioned in the studies.

Chapter 4 presents our work on applying the lightweight enforcement approach of fine-grained security policies into the context of client-side JavaScript in web application scenarios. First the client-side JavaScript and the state-of-the-art of JavaScript security are reviewed. Then we briefly present our enforcement method called *lightweight self-protecting JavaScript* which implements the lightweight enforcement approach by *inlining* policy code into a web page to monitor and control the behaviour of the code so that the page becomes *self-protecting.* We discuss some implementation issues in the self-protecting JavaScript method and review a hardened version of the self-protecting JavaScript implementation that can fix some identified implementation issues. An application of self-protecting JavaScript in the context of untrusted JavaScript is also presented. We propose a two-tier sandbox architecture to allow untrusted JavaScript code to be loaded and executed dynamically in a sandbox environment enforced by a reference monitor adapted from self-protecting JavaScript. Finally, we discuss more recent related work, short-comings of the approach and suggestions for further work.

The last chapter (**Chapter 5**) presents the concluding remarks of the thesis.

Part II reprints our published papers and technical reports listed in the previous section (1.3). The papers are reprinted in their published form, modulo reformatting and typo correcting for the thesis. Further discussion of shortcomings beyond those provided in the papers is presented in **Chapter 3** for the vehicle software domain, and **Chapter 4** for web browser security domain.

Chapter 2

Background

In this chapter, two main strands of background material for this thesis are reviewed. First, we briefly overview the existing approaches to untrusted software security and review the execution monitoring approach and its related aspects to security policy enforcement that is employed in this thesis. Second, we review the Aspect-Oriented Programming (AOP) technique that we adopt in this thesis to implement the lightweight enforcement of security policies. More specifically, we review the AspectJ language and an AOP implementation for JavaScript.

2.1 Approaches to Untrusted Software Security

Conventional enforcement mechanisms implemented at the operating system (OS) level are not applicable to software security since they "*cannot enforce policies concerning application-implemented resources*" [SMH01]. As alternatives to OS level mechanisms, the problem of untrusted software security has been widely studied in a number of approaches in the literature, such as [Nec97, Eva00, CW02, VN04, UE04, Fon04, Ham06, Lig06, DJM$^+$07, Her07, DJM$^+$08] and so on. These approaches can be divided into two directions: static analysis and runtime execution monitoring. Static analysis is a technique that checks a program before executing it. By analysing programs, only those not violating the policy are allowed to be executed. The analysis is based on some safety properties e.g. [ECCH00, CW02] or type systems e.g. [MWCG99, STFW01]. The benefit of the program analysis approach is zero runtime overhead. However, static analysis techniques must inevitably reject some good programs since they use an abstracted model of program state that loses some information. In this thesis, our lightweight enforcement approach is based on the runtime execution monitoring approach, which is discussed in more details in the next section. Related work to our studies has been discussed in detail in the included papers. Alternative to static analysis and execution monitoring, some other approaches rely on a collaboration between the untrusted third-party producing the code and the host platform running the untrusted code. These approaches include Proof-Carrying Code [Nec97], Model-Carrying Code [SVB+03], security by contract [DJM+08].

Proof-Carrying Code (PCC) [Nec97] is an interesting static analysis technique to provide security for untrusted code security problem without establishing trust relationships. The idea of PCC is to replace a trusted compiler with an untrusted certifying compiler plus a trusted certificate checker. The basic idea is that it is much easier and simpler to check the validity of a given proof that a program is well-behaved than it is to construct a proof from scratch. In the PCC approach, a code producer (which could be untrusted) delivers code together with a *proof* that the code satisfies some property of interest (which could in principle be anything from basic memory safety to some complex security policy). The proof is generated by a so-called *certifying compiler* based on a previously defined policy. At the consumer side, a trusted certificate checker validates that the proof complies with the policy. If the proof is valid and the policy is appropriate, the program is guaranteed to be safe and allowed to be executed. The advantage of this approach is that validating such proofs is typically much easier and quicker than performing the analysis and transformations to generate certified code. Furthermore, PCC does not rely on a central trustworthy entity, which makes PCC eminently suitable for ensuring security for extensible and open systems. This approach is promising in principle, however, its implementation is quite challenging in practice. Despite more than 10 years of research, the PCC technology is still quite immature, and no real usable tools have emerged yet.

Model-Carrying Code (MCC) [SVB+03] combines a static analysis technique like PCC with an execution monitoring mechanism to provide safe execution of untrusted code. The idea of this approach is that the code producer generates a *model* capturing the security-relevant behaviour of the code, and supply this model together with the code to the base system. At the base system side (code consumer), the code consumer uses the model to check if the code violates predefined policies. As such, MCC is a collaborative framework between code producers and code consumers to provide safe execution of untrusted code. The implementation of MCC is based on system call interception in a Unix system. MCC is believed to be adaptable to other execution environments, however, it requires the base system to be modified. The policies in MCC are generic to the base system, not specific to an untrusted application.

Security by contract [DMM$^+$08, DJM$^+$08] is another approach, developed in the S3MS project [S3M], for providing security of untrusted mobile applications on mobile devices. A contract is a description of security-relevant behaviour of the application specified by the application developer before the development stage. Similar to MCC or PCC approaches, the contract is provided together with the application to the base system. When the application is deployed on the base system, the contract is checked: if the contract complies with the device policy, the application is allowed to run without any further enforcement; otherwise, the application will be monitored at runtime according to the base system policies.

In this section, some of the other approaches and more related work are briefly reviewed to provide a broader view of the area. The security problem of untrusted software is often referred to as *mobile code security* which has been widely studied in the literature. We refer the reader to [Fon98] for more details of the approaches in the area. In general, each approach has potential advantages as well as disadvantages. When employed in a particular system, each must overcome numerous technical problems to be practically applicable. We note, in particular, that with a dynamic language such as JavaScript, where code can be created or modified at runtime, there are many scalability challenges to deal with.

2.2 Security Policy Enforcement by Execution Monitoring

Security policy enforcement is a mechanism that enforces some desired policies in a computer system in order to prevent unacceptable behaviour [Sch00]. In contrast to static analysis techniques mentioned above, execution monitoring is a security policy enforcement approach that can enforce application-specific security policies to prevent bad behaviours at runtime. The execution monitoring approach is rooted in the classic *reference monitors* approach to intercepting security relevant resource requests and applying a security policy to decide whether to grant such requests [And72]. The lightweight enforcement approach presented in this thesis is based on the classic reference monitors. In this section, we review the reference monitors as well as some recent variant implementations of the reference monitors which can enforce application-level and more flexible security policies.

2.2.1 Classic Reference Monitors

The classic reference monitors were introduced in 1972 in a study plan of the U.S. Air Force [And72] which aims to propose a secure computing environment that can defend against malicious users in resource sharing systems with multiple categories of users. In such a system, all references to programs, data, and peripherals have limited control such that a malicious user (attacker) is able to access any references to any other data or programs in the system. Such limited control references let the attacker launch attacks such as getting other users' passwords, or denying legitimate use by others. Due to such threats, the goal of the reference monitor is to control the execution of users' programs in the system in order to ensure the security of the system. *"The function of the reference monitor is to validate all references (to programs, data, peripherals, etc.) made by programs in execution against those authorised for the subject (user, etc). The Reference Monitor not only is responsible to assure that the references are authorised to shared resource objects, but also to assure that the reference is the right kind (i.e. read, or read and write, etc.)"* [And72, p. 17]. In other words, the reference monitor observes the execution of a program in order to stop the program whenever the execution is about to violate some desired security policies.

Implementation of the reference monitor concept is based on the combination of hardware and software, which is called a reference validation mechanism. An operating system mediating access to files is an example of a reference monitor implementation. A reference monitor implementation must follow certain principles in order to provide high-assurance and complete mediation of relevant references. Those principles are described [And72, p. 17] as:

A) The reference validation mechanism must be tamper proof.

B) The reference validation mechanism must always be invoked.

C) The reference validation mechanism must be small enough to be subject to analysis and tests to assure that it is correct.

2.2.2 Implementing Reference Monitors

In practice, there are several approaches to implementing reference monitors described as follows.

Traditional approach The traditional approach is to place the reference monitor at the operating system (kernel) level with hardware support to mediate system calls. If a system call violates a security policy, the reference monitor halts the execution to protect the system. Since this implementation only concerns events at the system level, it cannot distinguish the origin of a system call, i.e. from which application the call originates, therefore, it cannot enforce application-level policies. In addition, this implementation uses a memory protection hardware mechanism which separates address spaces between the target application and the reference monitor, to ensure that the execution of one program cannot corrupt the instructions or data of another [SMH01]. Therefore, it creates performance costs for context switching of control between the reference monitor and the target application.

Interpreter approach Another approach is to run applications in an interpreter and implement the reference monitor as a part of the interpreter. Although this approach was inapplicable and was dismissed due to its high performance overhead [And72] at the time of the report (1972), there have been a number of recent implementations of this approach such as [JSH07, YCIS07, ML10] in the context of web-application security. These implementations modify the JavaScript interpreter in web browsers to implement a reference monitor within the browser. This mechanism provides precise and transparent security enforcement, however, the downside of this approach is that the modifications are certainly time consuming, error prone, and short lived in practice since the codebase of base systems, e.g. web browsers of a interpreter, is rapidly changing.

Program rewriting approach The third approach is to embed a reference monitor into a target application by modifying the application to include the functionality of a reference monitor. Recently, there have been such implementations in the literature such as *Inlined Reference Monitors* (IRM) [UES00a, UE04]. This implementation approach uses a trusted program rewriter which takes a target program (untrusted) and security policies as input and transforms the target program into a new version of the program that embeds the policy. Compared to the other approaches, the

IRM implementation has various advantages which are summarised in the following subsection.

Inlined Reference Monitors

An inlined reference monitor (IRM) [UES00a, UE04] is an approach to enforcing security policies for software by rewriting the software to "embed" (inline) security policies within it. More specifically, *security checks* will be added in security-relevant actions or events of the software.

The IRM implementation of reference monitors has various advantages compared to the other implementation approaches as follows:

- *It can enforce a richer set of security policies.* Since the code of the IRM is "inlined" to the target application, the IRM is a part of the application and can mediate any application instruction. This means that application-level policies can be enforced.

- *It can support more flexible policies.* The IRM code can be inserted before and/or after an application instruction execution or memory-relevant operation in the target application [UE04, WLAG93, PH98]. Moreover, application execution history can be tracked by storing corresponding events in *security states*. Therefore, the IRM can enforce history-dependent policies such as, for example, "do not allow network send after the file system is read by the application".

- *It is more efficient.* Since an IRM is located inside the target application, there is no overhead by context switching between the application and the operating system or the execution environment. Moreover, being located inside the target application provides the IRM with possibilities to optimise the program performance.

2.2.3 Fundamental aspects of Execution Monitoring

With execution monitoring, a program is monitored at runtime by some security checks which are defined in security policies. Specifying a security policy therefore involves three aspects [UES00a]:

- *security events*, the policy-relevant operations that must be mediated by the reference monitor;

- *security states*, information stored about earlier security events that is used to determine which security events can be allowed to proceed; and
- *security updates*, program fragments that are executed in response to security events and that update security states, signal security violations, and/or take other remedial actions such as suppressing or replacing an action.

For example, in the policy "*do not allow a network send after the filesystem has been read*", filesystem read and network send are two policy-relevant operations that must be mediated; the information whether the filesystem has been read is stored in a security state; for each policy-relevant operation, a policy will define a corresponding security update such as setting (updating) the security state, or stopping the network send if the security state of filesystem read is set. In the following subsections, the aspects of execution monitors including security policies and security automata are presented following [Sch00]. We also discuss the limitations of the security automata and review the edit automata which have more powerful remedial actions that we employ in this thesis.

Monitor-enforceable security policies

So far, we have mentioned *security policies*, but have not defined them. In the context of a reference monitor, Schneider [Sch00] introduced the term "*security policy*" as: "*A security policy defines execution that, for one reason or another, has been deemed unacceptable*". A security policy might include *access control policy*, which is to restrict access on objects or system operations; *information flow policy*, which is to restrict the leakage of confidential information by observing system behaviour; or *availability policy*, which specifies that the use of a resource must be released at some later point of the application execution. Formally, Schneider has provided the definition of a security policy as follows.

Definition of Security Policy: A *security policy* is specified by giving a predicate on a set of executions. A target system $\mathcal{S}$ *satisfies* security policy $\mathcal{P}$ if and only if $\mathcal{P}(\Sigma_S)$ is *true*, where Σ_S is the set of executions of a target system $\mathcal{S}$, which is represented by a finite or infinite sequence of states or actions.

Schneider [Sch00] defined a class of enforcement mechanisms which respond to policy violations by terminating the target application, and proved that security policies enforceable by such enforcement mechanisms are exactly the *safety properties*[1]. Liveness properties, which ensure "good events do occur" [AS87][2], and information flow policies are not in safety properties, and thus they cannot be enforced in this class of enforcement mechanisms.

Security automata

As identified in [Sch00], a reference monitor can be modeled by a *security automaton*, which can be used to specify monitor-enforceable security policies. By definition, a *security automaton* $\mathcal{A}$ is a deterministic finite or countably infinite state machine (Q, q_0, δ), where Q is the possible *automaton states*, $q_0 \in Q$ is the initial state, and $\delta \in Q \times I \rightarrow Q$ is the *transition function* where I is a countable set of *input symbols*. The transition function defines a next state for the automaton given its current state and an input symbol. For example, if the automaton is in state q and accepts an input symbol s and changes to a next state q' then $\delta(q, s) = q'$.

The set of input symbols I represents target execution events and is dictated by the security policy being enforced. The execution events might correspond to system states, atomic actions, e.g. machine instructions, or higher-level actions of the target system. Thus, a sequence $s_1 s_2 ...$ of input symbols in I represents a set of executions[3] in the target system. To process the sequence $s_1 s_2 ...$, the automaton starts with the starting state q_0 and reads one input symbol of the sequence at a time. Once an input symbol s_i is read, the automaton changes its current state q_i to $q_{i+1} = \delta(q_i, s_i)$.

If $\delta(q_i, s_i)$ is undefined, the input is rejected, *i.e.* stops the corresponding action violating the policy being enforced by the automaton; otherwise, the input is accepted, *i.e.* the action is allowed to execute normally.

Figure 2.1 illustrates a security automaton for a security policy that does not allow message sending (by operation *Send*) after files were read (by operation *FileRead*). There are two nodes, which represent the states of the automaton, in the figure labeled *start* (the initial state of the automaton) and *noSend*. The edges in the figure represent the transition function $\delta(q, s)$, where q is the node that an edge starts and s is an input symbol. Each edge is labeled by a so-called *transition predicate*: a Boolean-valued effectively computable total functions with domain I. Let p_{ij} denote the predicate for the transition from node q_i to node q_j, *i.e.* p_{ij} is labeled for the edge from

[1]Safety properties specify "nothing bad ever happens", i.e. preventing "bad events" from occurring in a valid run of the target [AS87].

[2]Availability policies are examples of liveness properties.

[3]Corresponding to security-relevant actions that a reference monitor mediates.

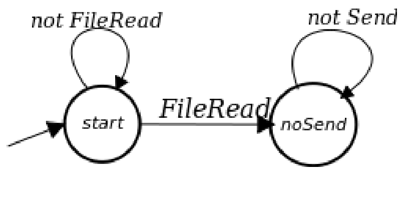


Figure 2.1: A security automaton: no *Send* after *FileRead* (taken from [SMH01])

node q_i to node q_j. The security automaton, after reading an input symbol s, changes its current state q_i to q_j if and only if $p_{ij}(s)$ is *true*, meaning that the input symbol s satisfies the predicate p_{ij}. For example, in Figure 2.1, the transition predicate *not FileRead* is satisfied by all input symbols except *FileRead* operations, or the transition predicate *FileRead* (from node *start* to node *noSend*) is satisfied only by input symbol *FileRead* operations.

If there is no transition defined for a input symbol s_r from a node q_r, it means that the security automaton rejects the input symbol s_r from the node q_r. For instance, in the security automaton depicted in Figure 2.1, no transition is defined from node *noSend* for input symbol corresponding to operation *Send* (message-send operation), therefore, the automaton rejects inputs *Send* from node *noSend* (*i.e.* after a file read by operation *FileRead*). Thus, the security automaton in Figure 2.1 describes a policy like "prohibit network send after file read".

Limitations As mentioned, the security automaton is only able to enforce safety properties, since it can only recognise acceptable actions and reject undefined actions, i.e. actions having bad behaviour. In [LBW05], the authors have identified several cases in which security automata fail to enforce a security policy. Firstly, security automata are unable to encode a policy that is not a predicate on execution sequences. For instance, the security automata cannot encode policies which require some external information that is hidden from the monitor. Thus, the monitor cannot enforce such a policy properly. Secondly, since security automata can simply accept or reject security-relevant actions, some effective manipulations such as insertion, suppression to certain security-relevant actions, cannot be performed in the monitor. Thirdly, the monitor is unable to observe certain security-relevant actions that has direct access to some hardware device. Lastly, if the monitor is compromised by the untrusted program, i.e. the program is able to corrupt the monitoring code, then the monitor cannot enforce any meaningful properties. In the next subsection, we describe *edit automata*, a more powerful automaton that acts as a program monitor with more response actions.

Edit automata

Examining the limitations of the security automata [Sch00] (introduced above), Ligatti, Bauer and Walker [LBW05] have introduced a more powerful automaton called *edit automaton* that acts as a program monitor. Differing from security automata, edit automata act as transformers that can modify security-relevant actions and behaviour of a target application. The authors have defined a hierarchy of enforcement mechanisms with different transformational capabilities:

- *Truncation automata:* Truncation automata can detect bad behaviour of a target program and terminate program execution if the security policy is violated[4].
- *Suppression automata:* Suppression automata are able to suppress policy-violated program actions without terminating the program.
- *Insertion automata:* Insertion automata can inject a sequence of actions into the target program[5].
- *Edit automata:* An edit automaton is the combination of a suppression automaton and an insertion automaton so as it has the ability to truncate or suppress security policy-violated actions as well as to insert action sequences into security-relevant events of the target program.

In summary, an edit automaton is able to enforce a wider range of properties rather than safety properties. In particular, an edit automaton, in addition to being able to terminate or truncate program execution, can suppress and insert actions. In [Lig06], Ligatti has demonstrated that, in practice, edit automata can enforce non-safety properties and even pure liveness properties.

Monitor-enforceable security policy languages

As we have mentioned, an execution monitoring responds to security-relevant actions at runtime based on security policies defined by a policy specifica-

[4]These automata are similar to security automata [Sch00].

[5]In fact, similar ideas have been implemented elsewhere e.g. [UES00a, UES00b].

tion language. In this section, we review such existing languages and their corresponding tools that enforce the policies on programs.

The *Ariel* project [PH98] introduced a declarative policy language to specify security policies and developed a program transformer tool to enforce such policies by injecting enforcement code for checking access constraints into the program (Java classes) and the target system's resource definitions. Ariel policies are described at the level of the Java APIs in terms of a set of constraints on accesses to local resources and the conditions under which they apply. Thus, the resources of the target system are protected since the executions of the transformed program are ensured to satisfy all access constraints. Since the language aims to specify access control policies, it is unable to describe policies that can modify the behaviour of a program.

The *Naccio* project [ET99] developed a general architecture for defining and enforcing code safety policies. The approach of Naccio is to modify method-call instructions, redirecting them to a wrapper method in order to enforce safety policies. Safety policies are defined by attaching checking code to resource operations. A policy consists of any number of safety properties that place constraints on resource manipulations. Naccio takes a program and a safety policy then produces a transformed program that behaves similarly to the original program except that it is guaranteed to satisfy the safety policy. Thus, Naccio has ability to define and enforce policies that place arbitrary constraints on resource manipulations as well as policies that alter how a program manipulates resources. However, Naccio cannot define or enforce liveness properties or policies that depend on the structural properties of the code.

Apart from Ariel and Naccio, Erlingsson and Schneider introduced Security Automata SFI Implementation (SASI) system [UES00b], an IRM system that implements the idea of security automata. SASI enforces safety properties by rewriting x86 machine code programs and Java bytecode (Java Virtual Machine Language (JVML)) programs based on security policies defined in *SAL* (Security Automaton Language). Later the authors introduced PSLang/PoET [UES00a, UE04], a language and system implementing IRMs. PoET, which stands for Policy Enforcement Toolkit, is a tool that rewrites the JVML class files of target programs and inserts IRM enforcement code implementing a reference monitor into programs. PSLang, the Policy Specification Language, is a policy language used to define security policies that identify security-relevant actions and the IRM enforcement code that the PoET tool should embed around each.

Mobile [HMS06, Ham06] is an extension of the .NET framework supporting certified Inlined Reference Monitoring. Mobile rewrites .NET CLI binaries based on a *declarative security policy language* such that the rewritten programs are guaranteed not to violate that security policy when executed. The policies denote types in the Mobile's type system. The rewritten programs also adhere a *policy proof* in the form of typing annotations which is verified by a trusted checker to guarantee the policy-adherence of rewritten code.

After developing a theoretical framework *edit automata* [LBW05], Bauer, Ligatti and Walker proposed a language and system called *Polymer* [BLW05, Lig06] that implements edit automata. The design of the Polymer system is similar to other policy enforcement languages/tools such as Ariel, Naccio, or PSLang/PoET in which a declarative language is used to define desired policies and policy enforcement is performed by a rewriter. Polymer is a policy specification language for specifying runtime policies on Java bytecode. It provides a methodology for conveniently specifying and generating complex monitors from simpler modules. The advantage of the language is the ability of composition of complex runtime security policies by making all policies first-class and composeable so that higher-order policies (superpolicies) can compose simpler policies (subpolicies). To use the system, a policy developer first identifies all program methods (on target system libraries) that might affect system security. Then, these methods are instrumented by a bytecode rewriter that inserts security checks into the methods in all the necessary places. Independently from this step, the developer specifies security policies in Polymer language. The security policies are then compiled by a policy compiler which translates the Polymer policy into ordinary Java and then invokes a Java compiler to translate it to bytecode. When loading the target application with the modified libraries, an implemented class loader rewrites the target code based on the compiled policy. By implementing the edit automata, the Polymer system has a wide range of remedial actions to security-violated program executions. Such remedial actions in the Polymer system include the ability to truncate, suppress, or replace violated actions as well as the ability to insert a sequence of actions into security-relevant events.

Strongly inspired by PSLang, ConSpec [AN08], a language for specifying policies and contracts, has been recently developed to exploit both for the specification of requirements and for the description of the security-relevant behaviour of actual systems. For providing the formal semantics of the language, ConSpec is more restrictive than PSLang. In particular, ConSpec does not allow arbitrary types in representing the security state and restricts the way the security states are updated. Interestingly, ConSpec has the ability to express policies on different levels such as on multiple executions of the same application, on executions of all applications of a system as well as on a single execution of the application and on lifetimes of objects of a certain class. This language is employed in the EU S3MS project [S3M] that aims to enforce security for untrusted applications in mobile devices.

Remark This section reviews the background of execution monitoring mechanism to enforce security policies which is the basic of our approach in this thesis. Our approach is a lightweight alternative implementation of a reference monitor. The key aspect to make the implementation lightweight is that it uses off-the-shelf tools and languages to define and enforce security. The tools and languages are in aspect-oriented programming paradigm which is presented as follows.

2.3 Aspect-Oriented Programming

Independently of security policy languages, a programming paradigm of *aspect-oriented programming* (AOP) [KLM+97] provides a means to modularise the cross-cutting functionalities of complex software systems so that the behaviour of an application can be modified by *aspects*. These features make AOP can be considered as a suitable implementation for security policy enforcement. In this section, we review the aspect-oriented programming paradigm and relate it to an alternative realisation of security policy enforcement which is investigated in this thesis.

Aspect-oriented programming (AOP) is a programming paradigm providing a programmatic means to modularise the *cross-cutting* functionalities of complex software systems by allowing the separation of cross-cutting concerns. The paradigm was originated by Kiczales and his research team at Xerox PARC [KLM+97] in 1997.

Separation of concerns [Dij76] is a general principle in software engineering to control the complexity of evergrowing programs by breaking down a program into distinct parts, i.e. *concerns*, the cohesive areas of functionality. In programming paradigms, these concerns are represented by new abstractions such as modules, and classes. These abstractions are separate and independent entities that support different levels of grouping and encapsulation of concerns. However, some concerns are called *crosscutting concerns* since their implementations do not provide the separation of concerns, *cutting across* multiple abstractions in a system. An example of such a crosscutting concern is the logging functionality in which the logging is normally invoked in every single logged class of the system, thereby crosscuts all logged classes or methods. As such, crosscutting concerns raise problems for programming standards, such as object-oriented programming, in which it is hard and error-prone to introduce such concerns in existing systems or to change afterwards.

AOP is an alternative solution[6] to deal with such problems of cross-cutting concerns by providing possibilities to cleanly separate concerns that would otherwise be cross-cutting using language mechanisms to capture explicitly cross-cutting structure. AOP implementations provide expressions for cross-cutting concerns and encapsulate each concern in one place. *Join point*, *pointcut, advice*, and *aspect* are terminologies used to express new concepts in AOP. *Joint points* are certain points in the dynamic execution of the program such as method calls, constructor calls, field access, exceptions, etc. A *pointcut* designates a set of join points for any program execution and the condition of execution under which crosscutting concern could be modified. An *advice* is the modification or additional code that would execute at each of the join points in a pointcut. *Aspects* are the modular units of cross-cutting implementation, comprising pointcuts, advice, that can modify the dynamic behaviour of program.

2.3.1 AOP implementation

An aspect language with different abstraction and composition mechanisms is needed in order to express appropriate AOP concepts, such as *pointcuts*, *advice*, and *aspects*. The main work of any AOP language implementation is to ensure that aspect and non-aspect (component) code run together in a properly coordinated fashion [KHH$^+$01]. The original method to ensure this coordination is to *weave* the advice code into the target program at desired jointcuts by a tool called *aspect weaver*, a special language processor to co-ordinate the co-composition of the aspects and components. The weaving process can be performed by different weaving techniques including source-level weaving and load time (or deploy-time) weaving. Source-level weaving is usually implemented using special preprocessors that require access to program source and performed during compilation. Load time weaving uses post-compiler processors [KLM$^+$97] as a part of virtual machine, using run-time instructions, or using the combinations of these approaches [KHH$^+$01]. In the case of Java bytecode, bytecode weavers can be deployed during the

[6]A possible treatment of cross-cutting concerns in software engineering is to *refactor* them away, i.e. to change the module hierarchy, by applying design patterns, so that the cross-cutting concerns become modular. This method, however, still faces limitations such as still leaving some cross-cutting boiler-plate, difficult to capture all aspects.

build process, or during class loading. In practice, various of such AOP implementations are available supporting most mainstream and other programming languages such as C/C++, Java, C#/VB.NET, JavaScript, PHP, and so on (see [AOPb] for more references). Here we briefly review two AOP implementations which have been studied in this research: AspectJ - an aspect-oriented language for Java, and an AOP implementation for JavaScript.

AspectJ

AspectJ [KHH$^+$01, Asp] is a seamless aspect-oriented extension to Java supporting two kinds of cross-cutting implementation, namely dynamic cross-cutting, and static cross-cutting. *Dynamic cross-cutting* is a mechanism that makes it possible to define additional implementation to run at certain well-defined points in the execution of the program. Static cross-cutting provides means to modify the static structure of a program such as adding new methods, implementing new interfaces, modifying the class hierarchy. "*In AspectJ's dynamic join point model, join points are well-defined points in the execution of the program; poincuts are collections of join points; advice are special method-like constructs that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprising pointcuts, advice, and ordinary Java member declarations. AspectJ code is compiled into standard Java bytecode*" [KHH$^+$01]. There are three types of advice in AspectJ, namely *before*, *after*, and *around.* For *before* and *after* advice types, aspect code is attached at before or after a defined pointcut, respectively. For advice type *around*, the execution behaviour of the concerned poincut can be controlled by allowing it to execute normally, suppressing the execution, or replacing the execution by aspect code. AspectJ has been first implemented with a source-level weaving technique, later a Java bytecode weaver has been developed to support load time weaving for Java bytecode. Detailed references for the language and tools can be found at [Asp].

An AOP implementation for JavaScript

We consider the client-side JavaScript context where JavaScript code is embedded into a web page to be interpreted and executed in a web browser. Differing from other languages, AOP implementation for JavaScript does not need a separate language to express cross-cutting concerns nor an independent aspect weaver. Instead, the reflective features of scripting language, and the prevalence of dynamic content can be exploited to add AOP features for JavaScript. Recently, various projects implementing AOP features for client-side JavaScript have been introduced (see [AOPb] for more references). For instance, jQuery AOP [AOPa] is a very small plugin that adds the features of AOP to jQuery[7]. jQuery AOP allows to add advice before, after, around and introduction to any global or instance object. In Paper C, we adapt this plugin to implement a lightweight IRM for JavaScript security. Our base library for AOP support, illustrated in Listing C.3, has only 454 bytes of code (compressed).

2.3.2 Security approaches using aspect-oriented programming

There have been recent interests in applying AOP techniques to security.

In [VBC01], an aspect-oriented extension to the C programming language was developed aiming to allow security policies to be separate from the code, enabling developers to write the main application and a security expert to specify security properties. The approach can be applied to security by different methods such as replacing insecure function calls by secure replacements, automatically performing error checking on security-critical calls, automatically logging data that may be relevant to security, etc.

Numerous research efforts e.g. [GRF02, SH03, YLH$^+$05, Set07] have shown some preliminary results in applying aspect orientation principle to secure software architecture design. More approaches to adopting AOP technique into security domain have been surveyed in [DS06]. Basically, these proposals only introduce the aspect-oriented approach as a part of the general design and development of security requirements for a software system.

In [Win04], the author has studied how aspect-oriented software development (AOSD) can be applied for the enforcement of application-specific requirements. The work elaborates on two specific AOSD techniques, namely interception-based and weaving-based. The interception-based AOSD technique is to operate on the execution of an application by capturing particular events to modify the execution whenever necessary. The weaving-based AOSD has been used for the modularisation of security based on the modification of development units to enforce the modularised security requirements within the boundaries of the application.

Hamlen and Jones [HJ08] have presented a language called SPoX (Security Policy XML), an aspect-oriented, declarative, security policy specification for enforcement by an IRM. The authors have introduced *aspect-*

[7]A JavaScript library that simplifies HTML document traversing, event handling, animating, and Ajax interactions to help JavaScript developers code their application faster.

oriented security automata, whose edge labels are encoded as pointcut expressions. The semantics of the language is defined to establish a formal connection between AOP and the IRM. This work is close to this thesis in reasoning about the connection between AOP and the IRM. They, however, provide a formal connection by designing a new language and providing the semantics for the language while the goal of this thesis is to study the application of exiting AOP languages for an IRM in particular untrusted systems.

Several studies such as [DWPJ06, SBM08] attempt to explore how secure an AOP approach to security can provide. These studies conclude that more formal security assurances should be provided to build a secure system with AOSD. However, according to a large number of AOP studies for software security, the AOP technique is a promising method to design and implement a secure software system. In particular, various authors, e.g. in [DW06, HJ08, BLW05, SH03, CR07], have observed that AOP can be considered as an implementation of security policy enforcement mechanism like IRMs. In [Yan10], Yang combines aspect-oriented programming techniques with static program analysis to provide a flexible and modular mechanism to enforce security policies on distributed systems. Alexandersson [Ale10] studies the use aspect-oriented programming to implement software-based fault tolerance and shows that the implementation results in clear advantages. Whether the AOP approach to implementing a security policy enforcement mechanism is adequate for particular systems needs to be investigated. Motivated by this, this thesis studies the application of aspect-oriented programming to the context of security policy enforcement. We call this *lightweight enforcement approach* to security policy enforcement. We investigate the approach for two specific untrusted systems: vehicle software architecture, and client-side JavaScript. In Chapter 3, we present our lightweight enforcement approach to vehicle software systems. We employ the lightweight approach to the context of JavaScript security which is summarised in Chapter 4.

Chapter 3

Runtime Enforcement for Downloaded Vehicle Applications

This chapter summaries our approach to enforcing fine-grained security policies for a vehicle application architecture. In the first section, the vehicle software architecture and the OSGi framework used in vehicle systems are reviewed. In Section 3.2, we introduce our approach to deploying the implementation of security policy enforcement using aspect-oriented programming for the OSGi framework. This work is included in this thesis in Paper A. Section 3.3 summaries our work to identify possible threats in the vehicle software architecture and deploy the security policy enforcement mechanism to design countermeasures against the threats. This work is included in this thesis in Paper B. Some practical issues are discussed in Section 3.4.

3.1 Vehicle Software Architecture

A modern in-vehicle system contains an in-vehicle network consisting of various *Electronic Control Units (ECUs)* and networks such as the *Controller Area Network (CAN)*, and the *Local Interconnect Network (LIN)*, among others. There is a wireless gateway connecting to the CAN bus that allows the in-vehicle network access to external networks such as fleet-management systems and the Internet. The in-vehicle system contains applications which can be divided into two categories: *vehicle software* and *ECU firmware*. The vehicle software requires an underlying platform to run the applications while ECU firmware is a self-contained application, flashed to a ROM, and run on a microprocessor responsible for a certain functionality in the vehicle [LPE06].

The functionality of ECU firmware ranges from small tasks, such as opening a window and unlocking a door, to more advanced functionality, such as automatic brake systems and collision warning systems. Recent administrative functions in vehicle systems, such as remote diagnostics and Firmware Updates Over The Air (FOTA), require the in-vehicle network to communicate with the Internet. However, allowing external communication with the previously isolated in-vehicle network introduces a number of security risks. As shown in a number of studies, e.g. [NLPJ08, NL08b, HD07, LNJ08, NL08a], ECU firmware is highly plausible targets for future attacks that could have serious consequences. In [NPL08] an ECU classification based on safety and security is presented in order to assist in designing security to know what to protect. Based on the classification it is suggested that automotive manufacturers should emphasise the security or restrict the remote diagnostics and FOTA procedures to certain ECUs.

For vehicle software, we consider the architecture described in the client part of the reference implementation of the open Global System for Telematics (GST) standard[GST] based on J2ME/OSGi. In this architecture (c.f. [GST], Open Systems Implementation Guide), software is installed and executed on an OSGi [OSG] software platform, that runs on a Java Virtual Machine (JVM) on an on-board vehicle computer with an underlying Operating System (OS). The on-board vehicle computer is connected to the CAN bus and provides interfaces for applications such that applications could access the underlying car infrastructure. The vehicle software applications are downloaded over the Internet from a control centre via the wireless gateway and then installed and run within the OSGi framework.

3.1.1 Open Services Gateway initiative(OSGi)

OSGi is a framework implementing a complete and dynamic component-model that is missing in stand-alone JVM environment. An application in OSGi consists of one or more components, called *bundles*. Bundles can be installed remotely and can be started, stopped, updated and uninstalled without restarting the JVM. The OSGi framework offers a co-operative model so that bundles can discover and use services provided by others in the same OSGi framework. The main advantage of using OSGi is to split systems into multiple components that can be dynamically loaded, unloaded or replaced by new implementations. Moreover, using OSGi makes it easy to develop new applications without recreating common services. Figure 3.1

depicts system layers in the software architecture using OSGi.

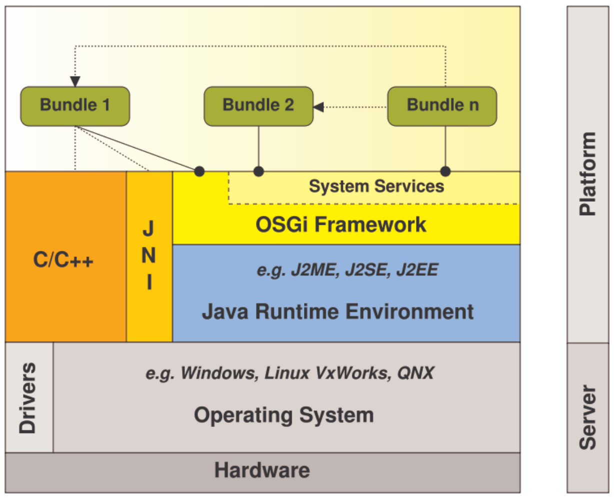


Figure 3.1: OSGi & System-Layering.

The OSGi technology based middleware has been deployed in many different industries such that it creates a large software market for OSGi software components [OSG07]. The OSGi framework has also been used in in-vehicle systems by several car manufactures. For example, the GST project defines an application runtime environment for a client system (vehicle) using the OSGi framework as mentioned; BMW used the OSGi specifications as the base technology for its high-end infotainment platform [OSG07].

OSGi Security The OSGi framework provides a secure environment that executes applications in a *sandbox* so that these applications cannot harm the environment, nor interfere with other resident applications. However, in an open system such as the open telematics market, one needs to allow potentially untrusted applications access to security sensitive resources in order to get the full benefits of extensibility. Such a simple sandboxing view which grants all-or-nothing access to a static set of resources, determined on the basis of trust, is too coarse-grained. The OSGi framework sits on top of a JVM and its security mechanism is based on the Java 2 security model [Jav]. In addition, OSGi Service Platform Release 4 [OSG] has provided an optional security layer that can help the infrastructure to deploy and manage applications that must run in controlled fine-grained environments. The main addition is simply the ability to authenticate bundles to be able to verify bundle integrity. Several security mechanisms have been proposed to secure the OSGi execution environment. For example, in [LKMB05] and [PF07], the authors proposed protocols for secure bundle deployment. These solutions, however, only help certify the origin and the integrity of code. Unfortunately, these security mechanisms are inadequate for an open telematics market that allows third-party providers to develop and provide services for in-vehicle systems. Since these security mechanisms cannot detect and prevent possible problems, e.g. downloaded applications could be malicious, or applications may simply have bugs, causing harmful to the system. To be flexible we need to be able to enforce application-specific policies at runtime. In the next section we present our work which identifies these limitations in details and studies the implementation of application-specific security policy enforcement for the OSGi framework.

3.2 Security Policy Enforcement in the OSGi Framework Using Aspect-Oriented Programming

This section briefly presents our work which has been published in COMPSAC'08 [PS08], and is included in this thesis in Paper A. In this work we have investigated on the implementation of security policy enforcement in the context vehicle telematics/infotainment systems – on-board vehicle computer and communications systems. The contribution of this work is a new combination of methods. We consider the OSGi (Open Services Gateway initiative) standard [OSG] as a representative open middleware platform for telematics systems. The main points of the study is to propose the architecture for enforcing security policies for third-party applications running on the OSGi framework and to implement the enforcement mechanism. We adopt a language-based approach using *aspect-oriented programming* (AOP) (c.f. Section 2.3) with the AspectJ compiler [Asp, KLM$^+$97], rather than a more security-specific program rewriting tools such as e.g. PoET/PSLang [UES00a] or Polymer [BLW05]. Security policies are specified as *aspects* in AspectJ language, an extension of Java programming language for AOP. Based on the policies, a third-party application distributed in Java source code or Java bytecode are rewritten the AspectJ weaver tool. After the rewriting process, the application is modified to a "secured" version that the policies are embedded into it. The secured application is deployed to the framework to run such that any behaviours of the application violating the policies at runtime are detected and prevented. The rewriting process is illustrated in Fig. 3.2.

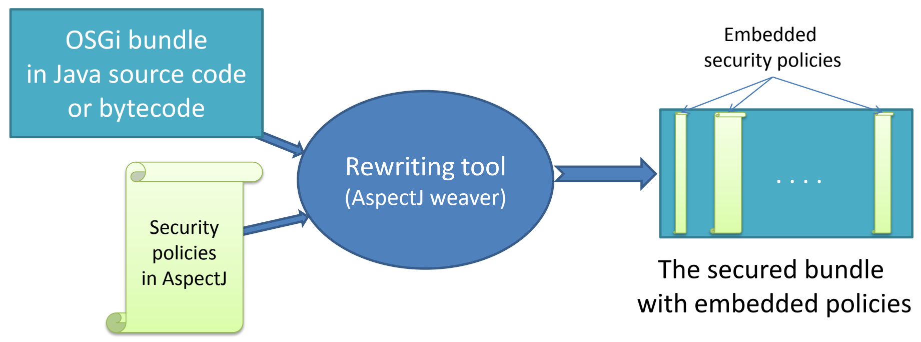

OSGi bundle in Java source code or bytecode
Security policies in AspectJ
Rewriting tool (AspectJ weaver)
Embedded security policies
. . . .
The secured bundle with embedded policies

Figure 3.2: The process to embed security policies for a vehicle application

A strength of this approach is that it uses a relatively complete and well-tested tool. A weakness, in principle, is that is does not provide direct support for policies (aspects). For example, in order to support different levels of security states such that history-dependent security policies, we design and implement a library for this purpose. We have illustrated how various sorts of security policies are categorised and described in AspectJ as advice. Our demonstration has resulted in the first study of security policy enforcement using an aspect-oriented language in an open system like the OSGi framework. The study differs from research of security policy enforcement in that it is based on the more industrially well-known language AspectJ and the main stream Java language without defining any new policy languages.

3.3 Preventing Potential Cyber Attacks for Downloaded Vehicle Applications

Based on the security policy enforcement method introduced in the above section, we have performed further studies in the vehicle software architecture to deploy the method in order to prevent potential *cyber attacks* [NL08a] in the vehicle systems. In this section, we summary our work performing this study which is included in this thesis in Paper B and published in RTIC'10 [PN10].

The work analyses possible threats and potential cyber attacks in the vehicle software architectures including add-on software and ECU firmware. For each threat, we introduce a corresponding countermeasure that can be specified as a security policy. We have proposed a model to deploy the countermeasures by using our security policy enforcement method in a vehicle context. In our proposed model, a transformation module is located at the wireless gateway as a proxy between the downloading module and the installation/updating module. After downloading, the code is rewritten to "embed" (inline) security policies within it. Thus, the execution of the modified code is enforced by application-specific security policies implementing countermeasures to prevent identified threats. We have illustrated the implementation of countermeasures in terms of policies in AspectJ, an aspect-oriented programming language. A similar solution for the implementation in the Electronic Control Unit (ECU) firmware architecture is also discussed in Paper B.

3.4 Discussion

Enforcing security for untrusted software in mobile systems like a vehicle software architecture has been an active research direction. There have been several research projects such as European S3MS [S3M, DJM^{+}07], GST [Sec06], VII [SWE08], EVITA [Evi] which investigate security mechanisms for such systems. In this chapter, we have summarised our contributions on providing security for the untrusted software architecture in vehicle systems. We have shown that our lightweight enforcement approach is promising for untrusted software scenarios in a vehicle software architecture. However, there are several issues which have not been discussed in detail in our studies i.e. [PS08, PN10]. In this section, we review the issues and discuss possible solutions.

We have demonstrated that various classes of security policies can be defined in AspectJ which can be embedded into untrusted programs to ensure the execution of the modified programs do not violate policies. However, several security aspects are not directly supported in an AOP language like AspectJ. For example, although we can encode security states in variables to define history-dependent security policies, we have to implement a library to deal with issues such as temporal policies and system level versus application level security states. Since we encode these temporal states via files, we assume that common access for multiple threads are handled by the platform, e.g. OSGi. We have not investigated the issues of concurrency for policy enforcement of multithreaded programs in detail. For instance, some of the concurrency issues such as race conditions when checking and updating security states have not been considered in [PS08]. The problem of time of check/time of use might happen when the reference monitors of two or more untrusted application concurrently access a global (system) security state. In a policy example such as “limit the number of SMS messages all applications in the system can send to 10 per day”, a global security state must be defined to store the number of SMS messages sent per day in the whole system. When an application tries to send a SMS, the reference monitor must first check the global state before granting sending operation to the application. If the operation is proceeded, the reference monitor then update the security state. The race problem happens when one application checks before another concurrent application updates the global state, the policy might be violated but the monitor could not detect to stop the operation. One possible solution to solve this problem could be to introduce a global lock for accessing global security states. However, it is needed to be further investigated to provide a complete solution as well as security assurance. While these problems have not been studied in this thesis, they have been handled by others in the literature such as [DJLP10, DJLP09]. In those studies, the authors deal with concurrency issues for multi-thread Java-like application in the context of inlined reference monitors by introduce race-free policies. The formal correctness of the approach has also been proved.

Although we have shown that the lightweight enforcement approach using aspect oriented programming is promising, there remain some theoretical and practical issues needed to be further investigated. For instance, we have demonstrated that fine-grained policies can be defined and enforced using an AOP language and tool, however the guarantees of the enforcement have not been shown yet. The design and implementation of AOP are not security-oriented, thus have not considered the case that applications may be malicious and can break the woven process. In the context of security enforcement, this issue should be studied to ensure that no policies can be bypassed. Moreover, the design and implementation of these tools do not consider the case that malicious code can be injected to an application which can break the security of the enforcement. In [DWPJ06], the authors have identified several security risks in using AspectJ to build secure applications. For example, a significant risk is *the level of control that aspects have over other modules in a system* [DWPJ06]. Moreover, it is possible that an aspect is malicious which can compromise the system. The work also suggests a number of steps should be investigated further that an AOP is mature for developing a secure system.

Chapter 4

Lightweight Self-Protecting JavaScript

In this chapter, we summarise our lightweight approach to enforcing fine-grained security policies for client-side JavaScript in order to prevent possible malicious actions. In the first section, we review client-side JavaScript and the state-of-the-art of JavaScript security. In Section 4.2, we briefly present our enforcement method called lightweight self-protecting JavaScript *and is presented in this thesis in Paper C. We discuss some implementation issues in self-protecting JavaScript method and review a revised implementation to provide a more tamper-proof library and better support for authoring sensible policies (Section 4.3.1). This is included in this thesis in Paper D. In Section 4.4, we present an application of the self-protecting JavaScript approach in the context of untrusted JavaScript. We have proposed a two-tier sandbox architecture so that untrusted code can be loaded and executed dynamically in a sandbox environment enforced by a reference monitor adapted from self-protecting JavaScript. This architecture is presented in Paper E in this thesis. Section 4.5 discusses more recent related work, short-comings of the approach and suggestions for further work.*

4.1 Client-side JavaScript and its security

JavaScript is a *dynamic scripting language* embedded in web (HTML) pages and executed in web browsers[1] (client machines). We call this client-side JavaScript because although JavaScript was originally created to run on

[1]In fact, JavaScript interpreters can also be found several tools such as Adobe Acrobat, Photoshop, Google Desktop Gadgets, email clients, and more [MDCa].

client-side, i.e. web browsers, the language is no longer limited to just client-side, e.g. server-side JavaScript is also available [MDCa, PS01]. JavaScript is tightly integrated with the browser Document Object Model (DOM) [W3C] to interact with the web page content. The DOM is a World Wide Web Consortium (W3C) standard, defining specifications for programs and scripts accessing document elements of a web page so that the programs or scripts can dynamically access and update the content, structure, and style of the document elements. JavaScript has dynamic features that allow web developers to build dynamic contents of a web page by generating the content based on user information when the JavaScript code in the web page is executed on a web browser. The key dynamic feature of JavaScript is that the code can be generated and executed at runtime by the `document.write`, and `eval` functions.

When a user visits a web page containing JavaScript code, the code is loaded to the web browser to be interpreted and executed within a JavaScript engine implemented in the browser. Each web browser has its own JavaScript engine implementation, and there are inconsistencies cross browsers. For example, some JavaScript code which runs in a browser properly might cause an error in another browser. ECMAScript is a standard specification, by Ecma International, which tries to solve the inconsistency problem by standardising the JavaScript language. ECMA-262 ECMAScript Language Specification 3rd edition (ECMAScript 3) is a standard version which is implemented by almost all current browsers. In December 2009, ECMAScript version 5 was released aiming to provide a huge improvement over ECMAScript 3. Latest version of major browsers, such as Mozilla Firefox 6.0, Google Chrome 15.0.865, Internet Explorer 10, support ECMAScript 5. Some new features of ECMAScript 5 are significant from a security perspective and are reviewed in this thesis in Paper E.

4.1.1 JavaScript security

Modern web applications generate web pages that contains both text (HTML) and JavaScript code. Since the JavaScript code can be arbitrary code from unknown sources, web browsers have security mechanisms to protect users and the system from such potential malicious code. One of the JavaScript security mechanisms is based on sandboxing, where JavaScript code is treated as an untrusted component and allowed to execute with only a restricted set of operations in order to isolate it from the rest of the operating system. For example, no access is granted to the local file system, the memory space of other running programs, or the operating system's network layer. This mechanism can prevent malfunctioning or malicious scripts from wreaking havoc in the user's environment [PS01]. The reality of the situation, however, is that scripts may conform to the sandbox policy, but still violate the security of the system, both by design and by accident. A simple example is to generate annoying pop-ups that difficult to terminate, i.e. by opening a pop-up window without control buttons or re-generating instantly when a pop-up window is closed. Phishing attacks [Phi] are a more serious example of the popup window problem where the location bar of origin web page is hidden and the attacker attempts to acquire sensitive information, such as credit card details, by redirecting a user into a malicious site.

The *same origin policy* [Rud] is another security mechanism provided by JavaScript and the web browser to prevent a document or script loaded from one origin from getting or setting the properties of a document from a different origin. For instance, JavaScript code from `othercompany.com` could not access the cookies' information of `company.com`. The origin, in this context, includes domain name, protocol, and port. In reality, however, the same origin policy has been exploited to launch attacks such as Cross-site Scripting (XSS) [XSS]. Cross-site scripting is a known attack where malicious scripts are injected into a web page at its origin, *i.e.* via wiki contents, blog comments, etc. and executed by the privileges of the web page which can bypass browser *same origin policy* mechanism to steal sensitive information of victims such as cookie, history.

As mentioned in Section 1.2.2, server-side and client-side defence are two approaches in the literature to preventing XSS attacks. Client-side defence mechanisms offer more advantages than server-side filtering mechanisms since its enforcement is performed in the browser at runtime, and it is easier for the enforcement mechanism to have a browser-consistent view of the source code. We divide client-side defence mechanisms into two directions based on whether they require browser modification or not. Modifying a browser has the advantage that it can access lower-level implementation details, so that the protection mechanism can modify or extend JavaScript in order to enforce richer precise security policies. However, this direction also has down side from an immediate practical perspective: it requires the browser users to be proactive to protect themselves. Moreover, modifying a browser requires much effort and it is difficult in practice to provide the same modification of all browsers for a protection mechanism. JCShadow [PDL$^+$11], and ConScript [ML10] are two state-of-the-art examples of this direction.

Avoiding browser modification, on the other hand, is an advantage in itself. For example it could allow a server to protect its own code from

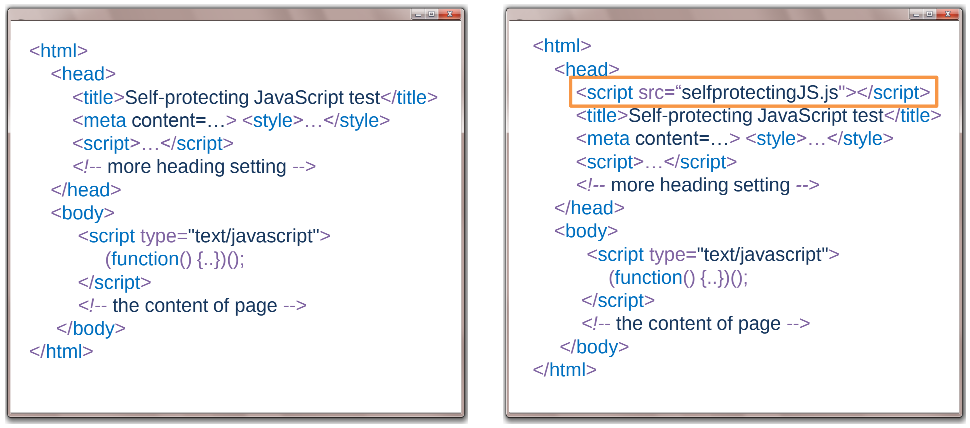
XSS attacks using an application-specific policy. The user would receive this protection without being proactive. The enforcement can be provided as a library by a server or a proxy and the policies are enforced at runtime at the browser. One approach in this area is to transform the original program to embed runtime security check to enforce policies. We refer to these styles as an *invasive* approach since it requires run-time parsing and transformation of the code, therefore can create great runtime overhead (see e.g. BrowserShield [RDW$^+$07]). A number of widely used approaches such as FBJS [Fac], or ADsafe [Cro] work by filtering JavaScript. These approaches first check that the code is in a well-behaved subset of JavaScript to ensure that problematic language features such as `eval` and `document.write` are excluded. A principled perspective on this approach is provided in the work of Maffeis *et al*, e.g. [MMT09].

Differing from the invasive approaches (and the above mentioned mechanisms), we employ our lightweight enforcement approach to enforcing security policies for JavaScript. Our approach is *non-invasive* since it does not require any aggressive code manipulation. The method is called *lightweight self-protecting JavaScript*, where the policy code is written in JavaScript and is safely embedded into web pages to control and modify the behavior of JavaScript code execution. In the next Section we introduce this approach in a JavaScript/browser context.

4.2 The Lightweight Self-Protecting JavaScript Approach

Our proposed *lightweight self-protecting JavaScript* is an enforcement mechanism which is implemented by inserting an enforcement library and policy code into a web page. The code wraps security-relevant events of JavaScript in the web page with pre-defined policies so that the security-relevant behaviour in the web page is monitored and controlled by the policies and thus the page becomes *self-protected*.

The injected self-protecting JavaScript code contains two parts: wrapper code and policy code. The wrapper code is the enforcement mechanism implementation which intercepts security-relevant events defined in policies with a corresponding policy. The policy code can be application-specific, i.e. specific to a particular web page, and therefore can be defined by the web developer who knows the intended functionality of the application so that he can define policies to prevent e.g. XSS attacks by restricting the usage of certain JavaScript features (without adversely affecting the function of the application). A key point of self-protecting JavaScript is that security policies are defined in pure JavaScript language which can fine-grained and stateful policies based on execution history.



```
<html>
   <head>
      <title>Self-protecting JavaScript test</title>
      <meta content=…> <style>…</style>
      <script>…</script>
      <!-- more heading setting -->
   </head>
   <body>
       <script type="text/javascript">
           (function() {..})();
       </script>
       <!-- the content of page -->
    </body>
</html>
```

The original web page

```
<html>
   <head>
      <script src="selfprotectingJS.js"></script>
      <title>Self-protecting JavaScript test</title>
      <meta content=…> <style>…</style>
      <script>…</script>
      <!-- more heading setting -->
   </head>
   <body>
       <script type="text/javascript">
           (function() {..})();
       </script>
       <!-- the content of page -->
    </body>
</html>
```

New version of the web page with embedded self-protecting JavaScript

Figure 4.1: Illustration of injection of self-protecting JavaScript code to a web page. The code is defined in the selfprotectingJS.js file.

In order to deploy the policy enforcement, the self-protecting code is injected into the header of a web page such that the content of the page remains unchanged. Injecting the self-protecting code into the header ensures that the self-protecting code is executed first, so it gets to wrap the security critical events before the attacker code can get a handle on them. The injection of self-protecting code can be performed at any point between client (web browser) and server, e.g. at server, or at a trusted proxy, or even as a browser plug-in. For example, a web developer can inject self-protecting JavaScript code into their web pages to protect their users from XSS attacks. A company can set up a proxy to inject self-protecting JavaScript code into every web pages accessing within the company to prevent some known vulnerabilities. Therefore, the self-protecting approach does not require browser modification. A web page having self-protecting JavaScript code injected is illustrated in Fig. 4.1.

Not very event can be controlled by our method since it is based on wrapping built-in functions. The key idea of wrapping a built-in function is to define a wrapper function such that the actual native code for the built-in function is only accessible via the wrapper function. Policies are run within the wrapper function to control the access to the native code at runtime. Although wrapper functions can be overwritten by e.g. the attacker, the reference to the original built-in function is held uniquely by the wrapper function. This enforcement method is illustrated in Fig. 4.2.

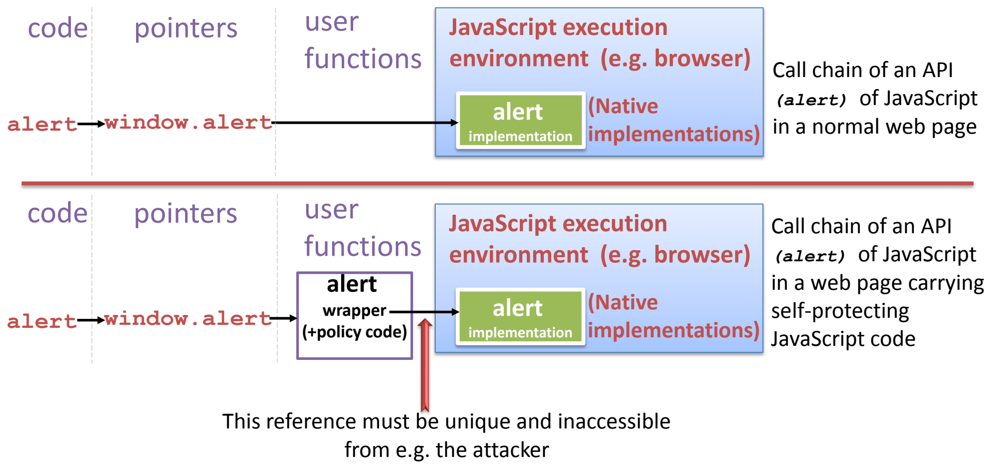

code
pointers
user functions
JavaScript execution environment (e.g. browser)
alert
window.alert
alert implementation
(Native implementations)
Call chain of an API (alert) of JavaScript in a normal web page
code
pointers
user functions
JavaScript execution environment (e.g. browser)
alert
window.alert
alert wrapper (+policy code)
alert implementation
(Native implementations)
Call chain of an API (alert) of JavaScript in a web page carrying self-protecting JavaScript code
This reference must be unique and inaccessible from e.g. the attacker

Figure 4.2: Illustration of the enforcement method

Because the wrapping code and policy code are defined independently from JavaScript code or the body of the page, i.e. only based on security relevant events, parsing or transforming the code of the page is not needed. This ensures low runtime overhead. Moreover, the enforcement method intercepts the behaviour of the code, it can deal with dynamic language features such as on-the-fly code generation which have proved difficult for filtering or code transformation mechanisms.

4.3 Self-protecting JavaScript implementation

The main challenges for implementing self-protecting JavaScript are *completeness*, ensuring that all security relevant events are intercepted, and *tamper-proofing*, ensuring that the malicious code cannot subvert the monitor mechanism itself. JavaScript provides reflection capabilities, in which code can be loaded and executed at runtime by using e.g. `eval` or `document.write` function, that makes it difficult to provide completeness. Tamper-proofing is another problem because the enforcement code is placed within the same code base, therefore it can be overwritten by attacker code. The key point to resolve these problems in self-protecting JavaScript is to ensure the original built-in methods can only be accessible via wrapper methods. Although wrapper methods can be overwritten, the reference to the original built-in method is held uniquely by the wrapper method.

The lightweight self-protecting JavaScript approach has been implemented by adapting an aspect-oriented programming (AOP) library (jQuery AOP [AOPa]) for JavaScript. Using AOP, a security event is defined as a *pointcut*, which defines the point and the condition under which to modify the behaviour of an application, and a policy is defined as an *advice*, which specifies what modifications should be applied. The work has been published in ASIACCS'09 [PSC09] (which is included in this thesis in Paper C). However, the AOP library is not designed for security purposes, there were several security issues that the attacker can exploit to bypass the enforcement[2]. In the continuation of this work, we have identified these issues and proposed solutions to fix the vulnerabilities to make the enforcement safe. In the following subsection, these issues are reviewed briefly.

4.3.1 Safe wrappers and sane policies for self-protecting JavaScript

The implementation of self-protecting JavaScript approach in [PSC09] has not considered all circumstances in which the attacker can exploit to obtain pointers to the original method. These issues include the generic wrapper code and ones relating to the construction of safe policies. We have studied and fixed vulnerabilities of both kinds in the implementation in [PSC09], and propose a way to make it easier to write policies which behave in a way which is not unduly influenced by attacker code.

There are two major issues in the wrapper implementation. First, the attacker can subvert functions or objects that are used in the wrapping function to bypass the policies or extract the original unwrapped methods. If the policy functions were to rely on inherited properties of objects it could be influenced in a similar way. Second, the attacker can use difference aliases of a function to bypass wrapped built-in function since there may be several aliases to the same built-in. These issues are discussed in detail and corresponding solutions are proposed in Paper D [MPS10] to handle the issues.

[2]In fact, we have studied other AOP libraries for JavaScript and these libraries face the same problems (c.f. Paper D, Section 2.6)

We have also provided a way to specify “sane” policies by introducing declarative arguments[3] for policies. The idea is that the policy writer writes a policy and an *inspection type*. An inspection type is a specification of the types of the call parameters that will be inspected by the policy code. This is to ensure that what you see (in the policy checking code) is what you get in subsequent use of the parameters.

4.4 An application of self-protecting JavaScript in the context of untrusted JavaScript

Self-protecting JavaScript can be deployed to monitor the execution of JavaScript in a web page to prevent unintended behaviour, e.g. XSS. One of the appealing points of self-protecting JavaScript is that it does not parse or transform the code and it can enforce fine-grained policies for the code at runtime. Enforcing fine-grained policies at runtime without code transformation is ideal for untrusted JavaScript context since untrusted JavaScript code can be loaded and executed at runtime under the enforcement of fine-grained policies without support of an extra tool to transform or validate the code as in recent approaches. However, deploying self-protecting JavaScript into the context of untrusted JavaScript is not trivial since the enforcement mechanism in self-protecting JavaScript cannot distinguish trusted or untrusted JavaScript. In this section, we introduce our work (Paper E) which proposes a two-tier sandbox architecture in which untrusted JavaScript code can be loaded and executed dynamically within a sandbox environment. The execution of untrusted JavaScript in the environment is monitored under predefined modular and fine-grained policies which can be defined via an adaptation of the self-protecting JavaScript mechanism.

4.4.1 A two-tier sandbox architecture for enforcing modular fine-grained security policies for untrusted JavaScript

Existing approaches to providing security for untrusted JavaScript include isolation of capabilities – a.k.a. sandboxing. Features of the JavaScript language conspire to make this nontrivial, and isolation normally requires complex filtering, transforming and wrapping untrusted code to restrict the code to a manageable subset. The latest JavaScript specification (ECMAScript

[3]Arguments in JavaScript are untyped, and the language applies automatic type coercion when there is a type mismatch. Attacker code can exploit this issue to launch attacks to defeat some enforced policies.

5) has been modified to make sandboxing easier and more widely applicable. ECMAScript 5 (ES5) [Ecm], released by the ECMA committee in December 2009, is a new standard specification of JavaScript language which represents, from a security perspective, a huge improvement over the previous (current) specification, ECMAScript 3. ES5 provides more robust programming to write *secure* JavaScript. Firstly, objects in ES5 can be frozen such that the frozen objects are tamper-proof. Secondly, isolation problems, i.e. *static lexical scope*, and *no encapsulation leak* in ES3 are solved in ES5 strict mode which makes it possible to construct a sandbox environment. This is illustrated in a sandboxing library recently developed by the Google Caja Team which allows untrusted code to interact with a restricted API [TME$^+$11].

However, specifying and enforcing fine-grained policies within an API implementation is complex and inflexible, since each sandboxed application (there may be several within a single web page) may need application-specific policies. Assume that there exists a baseline API that a sandbox environment provides to an untrusted application. Our goal is to specify and enforce modular and fine-grained policies on such an API for specific untrusted applications. By modular, we aim that policy code is separate from API implementation and specific to a particular untrusted application.

We have proposed a two-tier sandbox architecture to define and enforce modular and fine-grained security policies for untrusted JavaScript in SecureECMAScript (SES) environment in the context of ECMAScript 5. Instead of implementing security policies within an API, our mechanism is to intercept a baseline API with modular and fine-grained policies before providing the API to a sandbox environment. Thus the untrusted code running inside this architecture can interact with outside environment through two layers of enforcement: the interaction is first enforced by fine-grained policies on a baseline API, and then enforced by the confinement of the API itself. The implementation of the fine-grained policy part is an adaptation of the self-protecting JavaScript mechanism. More restricted than self-protecting JavaScript, the policy enforcement mechanism in this architecture is also executed within a sandbox so that the policy code cannot expose unprotected resources. The implementation of the architecture is based on client-side JavaScript libraries so that it does not require browser modification.

The two-tier sandbox architecture allows untrusted code to be dynamically loaded and executed without runtime checking or transformation. The execution of untrusted code is enforced by fine-grained security policies which can be specified modularly and specifically to each application.

4.5 Discussion

Web application security has recently received wide attention both in industry and in the research community. Although most web browsers provide security mechanisms for JavaScript, such as sandboxing, same-origin policy and signed scripting [Rud, MDCb], in practice attacks, such as XSS, phishing, or resource abuse, could defeat these protections.

In the research community, there have been a great number of approaches trying to provide security for JavaScript application. Each approach has potential advantages and disadvantages, and each must both overcome numerous technical problems to be practically applicable. Concrete related work has been discussed in Paper C, D, and E. In this section, we highlight the key features of our lightweight self-protecting JavaScript approach. We also discuss some limitations of the approach that the method can be improved to be practically applicable.

In the literature, our self-protecting JavaScript is a unique *lightweight* approach which does not require any aggressive code manipulation nor browser modification. Lightweight self-protecting JavaScript approach [PSC09] has received a quite wide attention in the literature. However, several papers have cited the paper with some imprecise criticisms. For example, in [MMT09], the authors mention that lightweight self-protecting JavaScript approach is not sound for existing browsers. Although our initial implementation had flaws, we believe that the approach is viable and sound (the work of [MMT09] deals with the idealised ECMA-262 Standard which certainly does not model the features of real browsers). More concretely, as we have discussed in [PSC09] (c.f. Paper C, Section 3.5) that in Mozilla deleting a wrapper method exposes the original built-in method. But this problem does not exist in other browsers such as Google Chrome or Safari. In latest browsers such as Firefox 4, and Google Chrome 12.x, the delete problem can be avoided by setting non-configurable to an object thanks to the new features in ECMAScript 5, as we have illustrated in Paper E. Another implementation-specific issue is that we rely on Mozilla-style getter and setter methods to enforce policies property accesses. However, these Mozilla-style methods are included in ECMAScript 5, making our approach to dealing with property access potentially more widely applicable. A revised implementation of our self-protecting JavaScript in ECMAScript 5, described in Paper E, is applicable for the browsers supporting ECMAScript 5. IceShield [HFH11] is a very recent ECMAScript 5 library similar to our lightweight self-protecting JavaScript approach and developed concurrently with the work in Paper E. The library is inlined to a page to detect and prevent malicious behaviour of the page. In IceShield [HFH11], the authors showed how to use the new features of ECMAScript 5 to implement the library such that the attacker cannot subvert the enforcement.

[MFM10], and [ML10] also identified some vulnerabilities of lightweight self-protecting JavaScript. However, these are just implementation-specific issues, not flaws in the method itself. We discovered these and other flaws concurrently with their work, we have revised the implementation in [PSC09] to fix the issues as mentioned briefly in Section 4.3.1.

Chapter 5

Conclusion

Security enforcement mechanisms remain a topic of research for all infrastructure systems [Sch00]. Implementing a security enforcement mechanism at application level is appealing because it can monitor the behaviour of the software at runtime with application-specific policies. Differing from previous approaches to implementing security policy enforcement at application level, this thesis proposes a lightweight enforcement approach by taking the advantages of the *aspect-oriented programming* [KLM+97] paradigm rather than using a security-specific policy language and program rewriting tool.

The approach has been investigated in two different domains to study its effectiveness. In the vehicle application domain, we have shown that the approach is promising and certainly adequate for small examples. We have also applied our approach in implementing the proposed countermeasures in terms of policies for downloaded vehicle applications in order to prevent potential threats in such application.

In the context of JavaScript security, the lightweight enforcement approach has been deployed to prevent script injection attacks. Because the enforcement implementation and policies are provided as a JavaScript library, they can be used to inject into any web page to make the page self-protected without transforming or parsing the page, or modifying the browser. We have also illustrated an application of the lightweight enforcement approach in JavaScript in the context of untrusted JavaScript such as mashup web applications.

Through these studies, we conclude the thesis by answering the research questions raised in the introduction chapter:

What classes of fine-grained security policies can be defined and enforced?

The lightweight enforcement mechanism is based on reference monitor style, therefore the basic class of security policy is monitor-enforceable [Sch00] which is safety properties. Defined in a base language, the security policies can be specified in a flexible way such as a reponse action can be inserted before and/or after an application instruction. We have shown that *edit automata* [LBW05]-based security policies can be specified in the lightweight enforcement approach. These policy classes allow the enforcement mechanism can take richer response actions, such as truncate, insert, or replace an operation, to a security request. Moreover, the policies can be defined specifically to each untrusted software and statful that policy decisions can be depended on execution history of the untrusted software.

How can the approach be integrated with a base system without modifying the base system?

The lightweight enforcement approach can be integrated with a base system running untrusted software without modifying the base system. In particular, in the OSGi framework, the base system running applications in vehicle software architecture, the transformation of untrusted applications is performed at a control center before deploying to the framework. The transformed application can run on the base system as usual.

In the context of JavaScript security, the enforcement code and policy are provided as a library and injected into web pages some where between the server and the browser so that no modification of the browser is needed.

What security assurances the approach can provide to a particular untrusted software system?

Security assurances of the lightweight enforcement mechanism have been provided empirically with examples and experiments. Formal proof of the security assurances needs to be investigated further.

In the OSGi framework for the vehicle application architecture, we have demonstrated that the security assurance provided by the lightweight enforcement mechanism is promising and certainly adequate for small application examples. Security assurance for large and realworld examples remains to be investigated.

In the lightweight self-protecting JavaScript approach, we have demonstrated that the approach can be deployed to defeat a number of known attacks and some realworld applications. We have also conducted experiments attempting to break the enforcement to show that the enforcement mechanism is secure.

What are the shortcomings of the approach?

There are certain shortcomings of the lightweight enforcement approach that are needed to be investigated further.

i, Although the lightweight policy enforcement approach does not need an additional security policy language, this, however, is quite difficult for policy-developers to write security policies. For example, an AOP language does not directly support security aspects, such as history-based or time dependent policies, or policy combination.

ii, Our first attempts are to use off-the-shelf tools to implement the lightweight enforcement approach. However, these tools do not support security aspects directly. Moreover, the design and implementation of these tools do not consider the case that malicious code can be injected to an application which can break the security of the enforcement. For example, [DWPJ06] has identified several security risks in using AspectJ to build secure applications. In JavaScript, we have confirmed that all off-the-shelf AOP tools are vulnerable to the attacks that we have identified in Paper D.

Bibliography

[Ale10] Ruben Alexandersson. *On Aspect-Oriented Implementation of Fault Tolerance.* PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2010.

[AN08] Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Electron. Notes Theor. Comput. Sci.*, 197(1):45–58, 2008.

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

[AOPa] Aspect Oriented Extensions for jQuery. http://code.google.com/p/jquery-aop/.

[AOPb] Aspect-Oriented Programming. http://en.wikipedia.org/wiki/Aspect-oriented_programming. Visited Nov, 2008.

[AS87] Bowen Alpern and Fred B. Schneider. Recognizing Safety and Liveness. *Distributed Comp.*, 2:117–126, 1987.

[Asp] The AspectJ Project. http://www.eclipse.org/aspectj/.

[BLW05] Lujo Bauer, Jay Ligatti, and David Walker. 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, 2005. ACM.

[CR07] Feng Chen and Grigore Roşu. Mop: an efficient and generic runtime verification framework. *SIGPLAN Not.*, 42(10):569–588, 2007.

[Cro] Douglas Crockford. ADsafe – making JavaScript safe for advertising. http://adsafe.org/.

[CW02] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *In Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244. ACM Press, 2002.

[Dij76] Edsger W. Dijkstra. *A discipline of programming.* Prentice-Hall Series in Automatic Computation, Englewood Cliffs: Prentice-Hall, 1976, 1976.

[DJLP09] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded java. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 546–569, Berlin, Heidelberg, 2009. Springer-Verlag.

[DJLP10] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Provably correct inline monitoring for multithreaded java-like programs. *J. Comput. Secur.*, 18:37–59, January 2010.

[DJM$^+$07] Lieven Desmet, Wouter Joosen, Fabio Massacci, Katsiaryna Naliuka, Pieter Philippaerts, Frank Piessens, and Dries Vanoverberghe. A flexible security architecture to support third-party applications on mobile devices. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07, pages 19–28, New York, NY, USA, 2007. ACM.

[DJM$^+$08] Lieven Desmet, Wouter Joosen, Fabio Massacci, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe. Security by contract on the.net platform. In *Information Security Technical Report*. Elsevier, 2008.

[DMM$^+$08] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard. *Security-by-Contract (SxC) for software and services of mobile systems.* At your service - Service-Oriented Computing from an EU Perspective. MIT Press, 2008.

[DS06] Josh Dehlinger and Nalin V. Subramanian. Architecting Secure Software Systems Using an Aspect-Oriented Approach: A Survey of Current Research. Technical report, Iowa State University, 2006.

[DW06] Daniel S. Dantas and David Walker. Harmless advice. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 383–396, New York, NY, USA, 2006. ACM.

[DWPJ06] Bart De Win, Frank Piessens, and Wouter Joosen. How secure is aop and what can we do about it? In *Proceedings of the 2006 international workshop on Software engineering for secure systems*, SESS '06, pages 27–34, New York, NY, USA, 2006. ACM.

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 1–1, Berkeley, CA, USA, 2000. USENIX Association.

[Ecm] Ecma International. Standard ECMA-262: ECMAScript Language Specification. http://www.ecma-international.org/publications/standards/Ecma-262.htm. 5th edition (December 2009).

[EGgC$^+$10] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[ET99] David Evans and Andrew Twyman. Flexible Policy-Directed Code Safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, 1999.

[Eva00] David E. Evans. *Policy-Directed Code Safety.* PhD thesis, Massachusetts Institute of Technology, 2000.

[Evi] Evita Project. E-safety vehicle intrusion protected applications. http://www.evita-project.org/. Accessed in August 2011.

[Fac] Facebook. Facebook JavaScript. http://developers.facebook.com/docs/fbjs.

[FBF99] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening cots software with generic software wrappers. In *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 2–16, 1999.

[Fon98] Philip W.L. Fong. Viewer's Discretion: Host Security in Mobile Code Systems, 1998.

[Fon04] Philip W. L. Fong. *Proof Linking: Modular Verification Architecture for Mobile Code Systems.* PhD thesis, Simon Fraser University, Burnaby, BC, Canada, 2004.

[GRF02] Geri Georg, Indrakshi Ray, and Robert France. Using Aspects to Design a Secure System. In *ICECCS '02: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 117–128, Washington, DC, USA, 2002. IEEE Computer Society.

[GSK10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 126–150. Springer-Verlag, 2010.

[GST] The Global System for Telematics (GST) project. http://www.gstforum.org.

[Ham06] Kevin W. Hamlen. *Security Policy Enforcement by Automated Program-rewriting.* PhD thesis, Cornell University, Ithaca, New York, August 2006.

[HD07] Tobias Hoppe and Jana Dittman. Sniffing/Replay Attacks on CAN Buses: A simulated attack on the electric window lift classified using an adapted CERT taxonomy. In *Proceedings of the 2nd Workshop on Embedded Systems Security (WESS)*, Salzburg, Austria, 2007.

[Her07] Almut Herzog. *Usable Security Policies for Runtime Environments.* PhD thesis, Linköping Studies in Science and Technology, May 2007.

[HFH11] Mario Heiderich, Tilman Frosch, and Thorsten Holz. IceShield: Detection and Mitigation of Malicious Websites with a Frozen DOM. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, RAID'11, 2011. To appear.

[HJ08] Kevin W. Hamlen and Micah Jones. Aspect-Oriented In-lined Reference Monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA, 2008. ACM.

[HMS06] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified In-lined Reference Monitoring on .NET. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 7–16, New York, NY, USA, 2006. ACM.

[Jav] Sun Microsystems, Java Security Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/spec/security-specTOC.fm.html.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[LBW05] Jay Ligatti, Lujo Bauer, and David Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[Lev06] Elias Levy. Worst-case scenario. *IEEE Security and Privacy*, 4(5):71–73, 2006.

[Lig06] Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring.* PhD thesis, Princeton University, 2006.

[LKMB05] Hee-Young Lim, Young-Gab Kim, Chang-Joo Moon, and Doo-Kwan Baik. Bundle Authentication and Authorization Using XML Security in the OSGi Service Platform. In *Proceedings of ICIS '05*, pages 502–507, Washington, DC, USA, 2005. IEEE Computer Society.

[LNJ08] Ulf E. Larson, Dennis K. Nilsson, and Erland Jonsson. An Approach to Specification-Based Attack Detection for In-Vehicle Networks. In *Proceedings of the 12th IEEE Intelligent Vehicles Symposium (IV)*, 2008.

[LPE06] Kerstin Lemke, Christof Paar, and Marko Wolf (Eds.). *Embedded Security in Cars: Securing Current And Future Automotive IT Applications.* Springer, 2006.

[MDCa] Mozilla Developer Center: JavaScript. https://developer.mozilla.org/en/JavaScript.

[MDCb] Mozilla Developer Center: Core JavaScript 1.5 Reference. http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference.

[MFM10] Leo Meyerovich, Adrienne Porter Felt, and Mark Miller. Object Views: FineGrained Sharing in Browsers. In *WWW2010: Proceedings of the 16th international conference on World Wide Web*, New York, NY, USA, 2010. ACM.

[ML10] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *SP '10: Proceedings of the 2010 IEEE Symposium on Security and Privacy.* IEEE Computer Society, 2010.

[MMT09] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with Filters, Rewriting, and Wrappers. In *ESORICS*, pages 505–522, 2009.

[MPS10] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Tuomas Aura, editor, The 15th Nordic Conference in Secure IT Systems, LNCS.* Springer-Verlag, October 2010. (Selected papers from OWASP AppSec Research 2010, June 2010, Stockholm, Sweden). To appear.

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21:527–568, May 1999.

[Nec97] George C. Necula. Proof-carrying code. In *POPL'97*, pages 106–119. ACM Press, 1997.

[NL08a] Dennis K. Nilsson and Ulf E. Larson. Conducting Forensic Investigations of Cyber Attacks on Automobile In-Vehicle Networks. In *Proceedings of the First ACM International Conference on Forensic Applications and Techniques in Telecommunications, Information and Multimedia (e-Forensics)*. ACM Press, 2008.

[NL08b] Dennis K. Nilsson and Ulf E. Larson. Simulated Attacks on CAN Buses: Vehicle virus. In *Proceedings of the Fifth IASTED Asian Conference on Communication Systems and Networks (ASIACSN)*. ACTA Press, 2008.

[NLPJ08] Dennis K. Nilsson, Ulf E. Larson, Francesco Picasso, and Erland Jonsson. A First Simulation of Attacks in the Automotive Network Communications Protocol FlexRay. In *Proceedings of the First International Workshop on Computational Intelligence in Security for Information Systems (CISIS)*, 2008.

[NPL08] Dennis K. Nilsson, Phu H. Phung, and Ulf E. Larson. Vehicle ECU Classification Based on Safety-Security Characteristics. In *Proceedings of Road Transport Information and Control - RTIC 2008 and ITS United Kingdom Members' Conference, IET*, pages 1–7, Manchester, UK, 20-22 May 2008. IET.

[OSG] OSGi Alliance, OSGi - The Dynamic Module System for Java. http://www.osgi.org/About/WhatIsOSGi. Accessed in August 2011.

[OSG07] About the OSGi Service Platform, Technical Whitepaper. http://www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf, June, 2007.

[OWA] OWASP - The Open Web Application Security Projects. Top 10 2010 - Cross-Site Scripting (XSS). https://www.owasp.org/index.php/Top_10_2010-A2. 2010.

[PDL$^+$11] Kailas Patil, Xinshu Dong, Xiaolei Li, Zhenkai Liang, and Xuxian Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *Proceedings of the 31st IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2011.

[PF07] Pierre Parrend and Stephane Frenot. Supporting the Secure Deployment of OSGi Bundles. In *Proceedings of WoWMoM 2007*. IEEE Computer Society, June 2007.

[PH98] Raju Pandey and Brant Hashii. Providing fine-grained access control for mobile programs through binary editing. Technical Report TR98-08, University of California, Davis, 1998.

[Phi] CNN: 'Phishing' scams reel in your identity. http://www.cnn.com/2003/TECH/internet/07/21/phishing.scam.

[Phu11] Phu H. Phung. A Two-Tier Sandbox Architecture to Enforce Modular Fine-Grained Security Policies for Untrusted JavaScript. Technical Report 2011:11, Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden, June 2011. Project URL: http://www.cse.chalmers.se/~phung/projects/jss. ISSN:1652-926X.

[PN10] Phu H. Phung and Dennis K. Nilsson. A Model for Safe and Secure Execution of Downloaded Vehicle Applications. In *Proceedings of Road Transport Information and Control - RTIC 2010 and ITS United Kingdom Members' Conference, IET*, London, UK, 2010. IET.

[PS01] Thomas A. Powell and Fritz Schneider. *JavaScript: The Complete Reference*. McGraw-Hill/Osbone, second edition, 2001.

[PS08] Phu H. Phung and David Sands. Security Policy Enforcement in the OSGi Framework Using Aspect-Oriented Programming. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pages 1076–1082, Turku, Finland, Jul. 28-Aug. 1 2008. IEEE Computer Society.

[PSC09] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight Self-Protecting JavaScript. In *ASIACCS '09: Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 47–60, Sydney, Australia, 10 - 12 March 2009. ACM.

[RDW$^+$07] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[Rob] Robert Lemos, SecurityFocus. Researchers warn over Web worms. http://www.securityfocus.com/brief/229. Published in August 04th 2006.

[Rud] Jesse Ruderman. Same origin policy for JavaScript. http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

[S3M] S3MS Project. Security of Software and Services for Mobile. http://www.s3ms.org. Accessed in June 2010.

[Sam] Samy. I'll never get caught. I'm Popular. http://namb.la/popular/. Published in October 04th 2005.

[SBM08] Andreas Sewe, Christoph Bockisch, and Mira Mezini. Aspects and class-based security: a survey of interactions between advice weaving and the java 2 security model. In *Proceedings of the 2nd Workshop on Virtual Machines and Intermediate Languages for emerging modularization mechanisms*, VMIL '08, pages 3:1–3:7, New York, NY, USA, 2008. ACM.

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[Sec] SecurityFocus. Yahoo!, you've got worms. http://www.securityfocus.com/brief/229. Published in October 19th 2005.

[Sec06] GST Security. The GST Security White Paper. http://gstforum.org/download/White%20Papers/DOC_SEC_White_Paper.pdf, December 2006.

[Set07] Rohit Sethi. Aspect-Oriented Programming and Security. http://www.securityfocus.com/infocus/1895, October 2007. Visited Dec, 2007.

[SH03] Viren Shah and Frank Hill. An Aspect-Oriented Security Framework. In *Proceedings of DARPA Information Survivability Conference and Exposition,*, volume 2, pages 143–145, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[SMH01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A Language-Based Approach to Security. In *LNCS 2000, Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, 2001. Springer-Verlag.

[Ste91] Daniel F. Sterne. On the Buzzword "Security Policy". In *Security and Privacy, IEEE Symposium on*, page 219, Los Alamitos, CA, USA, 1991. IEEE Computer Society.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, SSYM'01, pages 16–16, Berkeley, CA, USA, 2001. USENIX Association.

[SVB$^+$03] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 15–28, New York, NY, USA, 2003. ACM.

[SWE08] Peter SWEATMAN. Vehicle infrastructure integration (vii) for heavy trucks: A new perspective of truck research. In *10th International Symposium on Heavy Vehicle Transportation Technology*, HVTT08, 2008.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2nd edition, 2001.

[TME$^+$11] Ankur Taly, John C. Mitchell, Ulfar Erlingsson, Jasvir Nagra, and Mark S. Miller. Automated analysis of security-critical javascript apis. In *Proc of IEEE Security and Privacy'11*. IEEE, 2011. To appear.

[UE04] Úlfar Erlingsson. *The Inlined Reference Monitors Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, 2004.

[UELX07] Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[UES00a] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 246, Washington, DC, USA, 2000. IEEE Computer Society.

[UES00b] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM.

[VBC01] John Viega, J. T. Bloch, and Pravir Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14:31–39, 2001.

[VN04] Venkatakrishnan Venkatesan Natarajan. *Enforcement techniques for expressive security policies.* PhD thesis, State University of New York at Stony Brook, Stony Brook, NY, USA, 2004. AAI3161076.

[W3C] W3C. Document Object Model (DOM). http://www.w3.org/DOM. Accessed in March 2011.

[Win04] Bart De Win. *Engineering Application-level Security through Aspect-Oriented Software Development.* PhD thesis, K.U.Leuven, Belgium, March 2004.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM.

[XSS] RSnake: XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html.

[Yan10] Fan Yang. *Aspects with Program Analysis for Security Policies.* PhD thesis, Technical University of Denmark, Lyngby, Denmark, 2010.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.

[YLH$^+$05] Huiqun Yu, Dongmei Liu, Xudong He, Li Yang, and Shu Gao. Secure Software Architectures Design by Aspect Orientation. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 47–55, Washington, DC, USA, 2005. IEEE Computer Society.