Language-Based Security for Web Applications

Angel Luis Scull Pupo

Dissertation submitted in fulfillment of the
requirement for the degree of Doctor of Sciences

June 19, 2021

Promotor:
Prof. Dr. Elisa Gonzalez Boix, Vrije Universiteit Brussel
Co-promotor:
Prof. Dr. Jens Nicolay, Vrije Universiteit Brussel

Jury:
Prof. Dr. Ann Nowé, Vrije Universiteit Brussel (chair)
Prof. Dr. Dominique Devriese, Vrije Universiteit Brussel (secretary)
Prof. Dr. An Braeken, Vrije Universiteit Brussel
Prof. Dr. Walter Binder, University of Lugano, Switzerland
Prof. Dr. Alejandro Russo, Chalmers University Technology, Sweden

Vrije Universiteit Brussel
Faculty of Sciences and Bio-engineering Sciences
Department of Computer Science
Software Languages Lab

Printed by
Crazy Copy Center Productions
VUB Pleinlaan 2, 1050 Brussel
Tel / fax : +32 2 629 33 44
crazycopy@vub.ac.be
www.crazycopy.be

ISBN 9789493079885

NUR 993

Abstract

In support of our daily tasks, web applications are provided with sensitive information such as banking accounts numbers, social security information, etc. Therefore, it is expected that the developers of such applications rely on adequate tools offered by JavaScript and browsers to help them develop secure applications. However, neither JavaScript nor browser security mechanisms fully address modern application security needs.

Many language-based access control and information flow control approaches have been proposed for securing client-side web applications. However, designing a security mechanism supporting the combination of features such as portability, performance, and many awkward features of JavaScript and browsers is still problematic. Furthermore, in the software development life-cycle it is important to verify the same set of access control and information flow policies during development (static) and production (dynamic).

However, the current state of the art does not allow a safe and efficient combination of static and dynamic enforcement of a shared set of security policies, forcing developers to reimplement and maintain the same policies and their enforcement code in both static and dynamic environments.

This thesis explores language-based access control and information flow control policies for securing client-side web applications.

First, we present Guardia, a framework for declaratively specifying and dynamically enforcing application-level security policies for JavaScript web applications without requiring VM modifications. Guardia combines an internal declarative policy specification language with a decoupled enforcement mechanism, making it possible to experiment with different enforcement techniques that do not require VM modifications.

Second, we present Gifc, a permissive-upgrade-based inlined monitoring mechanism to detect unwanted information flow in client-side web

applications. GIFC covers a wide range of JavaScript features that give rise to implicit flows. In contrast to related work, GIFC also handles dynamic code evaluation online, and it features an API function model mechanism that enables information tracking through APIs calls. As a result, GIFC can handle information flows that use DOM nodes as channels of information.

Based on GUARDIA and GIFC, we develop a novel technique for deriving Static Application Security Testing (SAST) from an existing Runtime Application Security Protection (RASP) mechanism using a two-phase abstract interpretation approach. In our approach, the SAST component avoids duplicating the effort of specifying security policies and implementing their semantics. The RASP mechanism enforces security policies by instrumenting a base program to trap security-relevant operations and execute the required policy enforcement code. The first phase of the SAST mechanism computes a flow graph of the application by statically analyzing the base program without any traps. The results of this first phase are used in a second phase to detect trapped operations and abstractly execute the associated and unaltered RASP policy enforcement code. Deriving a SAST component from a RASP mechanism ensures equivalent semantics for the security policies across the static and dynamic contexts in which policies are verified during the software development life-cycle.

Samenvatting

Webapplicaties ondersteunen onze dagelijkse taken, en behandelen gevoelige informatie zoals bankrekeningnummers, informatie over sociale zekerheid, enz. Daarom wordt verwacht dat ontwikkelaars van dergelijke applicaties kunnen vertrouwen op adequate tools die worden aangeboden door JavaScript en browsers om hen te helpen bij het ontwikkelen van veilige applicaties. Echter, JavaScript noch browserbeveiligingsmechanismen voldoen volledig aan de beveiligingsbehoeften van moderne applicaties.

Er zijn veel op taal gebaseerde benaderingen voor toegangscontrole en informatiestroomcontrole voorgesteld voor het beveiligen van webtoepassingen in de browser. Het combineren van functies zoals overdraagbaarheid, snelheid en de ondersteuning van ingewikkelde features van JavaScript en browsers is echter nog steeds problematisch. Bovendien is het in de levenscyclus van softwareontwikkeling belangrijk om tijdens de ontwikkeling (statisch) en productie (dynamisch) dezelfde verzameling van regels voor toegangscontrole en informatiestroomcontrole te gebruiken. De huidige stand van zaken staat echter geen veilige en efficiënte combinatie van statische en dynamische handhaving van een gedeelde set beveiligingsregels toe, waardoor ontwikkelaars worden gedwongen dezelfde regels en de bijbehorende handhavingscode in zowel statische als dynamische omgevingen opnieuw te implementeren en te onderhouden.

Dit proefschrift onderzoekt op taal gebaseerde toegangscontrole en informatiestroomcontrole voor het beveiligen van webapplicaties.

Ten eerste presenteren we Guardia, een raamwerk voor het declaratief specifiëren en dynamisch afdwingen van beveiligingsregels op applicatieniveau voor JavaScript-webapplicaties, zonder dat VM-aanpassingen nodig zijn. Guardia combineert een interne declaratieve taal voor het uitdrukken van regels met een ontkoppeld handhavingsmechanisme, waar-

door het mogelijk wordt om te experimenteren met verschillende handhavingstechnieken die geen VM-aanpassingen vereisen.

Ten tweede presenteren we GIFC, een permissief upgrade-gebaseerd inlined monitoring-mechanisme om ongewenste informatiestromen in webapplicaties aan de clientzijde te detecteren. GIFC kan overweg met een breed scala aan JavaScript-functies die aanleiding geven tot impliciete stromen. In tegenstelling tot gerelateerd werk handelt GIFC ook online dynamische code-evaluatie af en beschikt het over een API-functiemodelmechanisme dat het opvolgen van informatiestromen van en naar API-aanroepen mogelijk maakt. Als gevolg kan GIFC omgaan met informatiestromen die DOM-knooppunten gebruiken als informatiekanalen.

Op basis van GUARDIA en GIFC ontwikkelen we een nieuwe techniek om Static Application Security Testing (SAST) af te leiden van een bestaand Runtime Application Security Protection (RASP) -mechanisme door middel van een abstracte interpretatie in twee fasen. Onze aanpak van de SAST-component vermijdt de inspanning van opnieuw dezelfde beveiligingsregels te specificeren en hun semantiek te implementeren. Het RASP-mechanisme dwingt beveiligingsregels af door een basisprogramma te instrumenteren om beveiligingsrelevante bewerkingen te onderscheppen en de vereiste code voor het afdwingen van regels uit te voeren. De eerste fase van het SAST-mechanisme berekent een stroomdiagram van de applicatie door het basisprogramma statisch te analyseren zonder enige onderschepping van relevante operaties. De resultaten van deze eerste fase worden in een tweede fase gebruikt om de te onderscheppen operaties te detecteren en de bijbehorende en ongewijzigde RASP-handhavingscode abstract uit te voeren. Het afleiden van een SAST-component uit een RASP-mechanisme zorgt voor gelijkwaardige semantiek voor de beveiligingsregels in de statische en dynamische contexten waarin de beveiliging wordt geverifieerd tijdens de levenscyclus van softwareontwikkeling. Bovendien vereist onze benadering van abstracte interpretatie in twee fasen niet dat RASP-ontwikkelaars de handhavingscode voor statische analyse opnieuw moeten implementeren.

Acknowledgements

First, I want to sincerely thank my promotors. I don't have enough words to thank the two of you, Elisa Gonzalez Boix and Jens Nicolay. Without your invaluable support during all this time at SOFT, I wouldn't have reached this special moment. I must say that this thesis is the fruit of your selfless work and effort (especially during the weekends) to point me in the right direction. My sincere and eternal thanks to you!

Second, I would like to thank the members of my jury, Ann Nowé, Dominique Devriese, An Braeken, Alejandro Russo and Walter Binder, for the time you dedicated to read my thesis, in addition to the comments, suggestions and feedback that you gave me during the private defense.

Third, I would like to express my gratitude to all professors at SOFT who in different ways have helped me to grow as a person and researcher. In particular, I want to thank Coen De Roover for his insightful comments and recommendations on my work. To Wolfgang De Meuter, for his constructive recommendations on the study of the language, mainly during the Opinio discussions. In general, I would like to thank all the students and postdocs in the group for the support provided during these 5 years at SOFT. In particular, I would like to thank Laurent Christophe for his help and the fruitful discussions we had related to our research. A special mention of thanks to my colleagues Matteo, Jim and Kevin for the discussions and recommendations on my research within the framework of DisCo group. Special thanks to Isaac, my flatmate almost since the beginning who has helped me in this final sprint proofreading the thesis. Olgita, Cirelda and Patrick have been part of my family here in Belgium, to you my thanks. Also, a special thanks goes to my friends Humberto and Carmen, who, beyond the professional level, have helped me a lot at a personal level.

I can't finish making these remarks without thanking my family, especially all my uncles. To my uncle Roman who has always been like a father to me. To my aunt Celina who has always been at the foot of the canyon with our battery. This result is fundamentally thanks to the education received from my parents. Especially, without the support and sacrifices of my mother Mayte, and my cute sister Leticia, today I would not be writing these words. This title is yours! Last but not least, I want to thank the support received from my beloved wife Acelia, who has been the owner of my heart in the most important part of this quest in my life.

This research has been funded by Innoviris Secloud project (2015-2018) and the Cybersecurity Initiative Flanders (2019-2021).

Contents

List of Figures

List of Tables

Listings

Acronyms

AC Access Control.

API Application Programming Interface.

AST Abstract Syntax Tree.

CDN Content Delivery Network.

CSP Content-Security Policy.

DOM Document Object Model.

DOS Denial-Of-Service.

DSL Domain-Specific Language.

EE Execution Explorer.

EM Execution Monitor.

GPL General-purpose Programming Language.

IFC Information Flow Control.

JSON JavaScript Object Notation.

ML Machine Learning.

NSU No-Sensitive Upgrade.

OWASP Web Application Security Project.

PU Permissive Upgrade.

RASP Runtime Application Self-Protection.

SAST Static Analysis Security Testing.

SME Secure Multi-Execution.

SOP Same-Origin Policy.

SPA Single-Page Application.

SQL Structured Query Language.

SVM Support Vector Machine.

TCB Trusted Computing Base.

URL Uniform Resource Locator.

VM Virtual Machine.

XML Extensible Markup Language.

XSS Cross-Site Scripting.

Chapter 1

Introduction

Modern web applications have become indispensable tools for humanity, helping individuals and organisations decentralise their businesses and services. A fundamental feature of these applications is their ability to compose code and content from different service providers, enabling new products in the process. Consider, as an example, UberEats [1], a food delivery service that connects customers to restaurants through a client-side web application. To build such a system, code from different sources is composed together. For example, the system may use Google Maps [2], as it shows real-time mapping information of the delivery, as well as React [3], a library for building user interfaces. More importantly, UberEats may interact with different services to facilitate the payment process (e.g., PayPal, Google Pay, etc.) and reach as many users as possible.

Despite their success and proliferation, modern web applications are subject to vulnerabilities. Since all code runs in the same execution environment with the same privileges, attackers can exploit third-party code to compromise the application's correct behavior. These exploits may result in the unavailability of the application (e.g., due to a Denial-Of-Service (DOS) attack) or the leaking of sensitive information (e.g., due to a Cross-Site Scripting (XSS) attack).

Web application security is central since web application attacks are reported to be responsible for breaches more than any other method [ZY17]. The average cost of a data breach is $3.86 million in 2020 [PI20].

[1] Uber Eats: https://www.ubereats.com/be-en
[2] Google Maps: https://www.google.be/maps
[3] React: https://reactjs.org/

This thesis focuses on securing web applications written in JavaScript, a dynamically typed language widely used in client-side web applications. JavaScript, originally designed as a scripting language, follows a *no crash* philosophy; the language will try to execute operations and make the necessary type adjustments to avoid program crashes [AGM+17]. While these features make JavaScript suitable for fast prototyping and development, they complicate reasoning about the application's behavior.

The critical issue for web security is that client-side web applications are executed in a web browser residing in the end-user machine. In a nutshell, a browser fetches content and code (i.e., HTML, JavaScript, images, etc.), often from many different origins, and renders a user interface resulting from parsing and executing the HTML and JavaScript code. In addition to displaying a user interface, the browser exposes different Application Programming Interfaces (APIs), enabling the application to interact with the users' machine hardware such as the network, camera, microphone, etc. Exposing such APIs to the application is risky; JavaScript code, including third-party utility libraries, can freely use these APIs because all the code within the application is executed in the same context.

Browsers incorporate the Same-Origin Policy (SOP) [Mozb] (to isolate the content and code originating from different origins), and the Content-Security Policy (CSP) [Moza] (to prevent code injection attacks such as cross-site scripting). However, browser security mechanisms can be omitted, wrongly configured, or bypassed [RHZN+13, LKG+17, CUT+21]. For example, in Chapter 2, we demonstrate how CSP can be bypassed in the context of an existing web application (Juice Shop [Kim]) using modern technologies. As a result, complementary *application-level* security policies are required for effectively securing web applications.

An application-level security policy expresses a program property that must hold during the entire application's execution. Much research has studied application-level security policies. Two well-known types of application-level security policies are Access Control (AC) and information Information Flow Control (IFC) policies [Bie13]. An *access control policy* restricts access to a specific resource, for example "Do not allow the creation of `iframe` elements at runtime". An *information flow control policy* prevents information to flow from specific sources to particular information sinks, for example, "User input must

be sanitised before being displayed on screen" or "Sensitive user data such as passwords must not be logged to the console". Approaches supporting AC and IFC policies have been studied for client-side web applications, including *static* approaches based on source code analysis [GL09, GL10, GPT+11], *dynamic* approaches based on runtime monitoring [PSC09, ML10, MFM10, AF09, AF10, DP10, ASF17], and *hybrid* combinations thereof [CMJL09, WR13, TFP14a].

However, rather than choosing either a static or a dynamic approach, developers use both static and dynamic approaches for verifying, testing, and enforcing security policies at different points in the software development cycle [HL06, BLH08, FBJ+16]. More concretely, in the context of the secure application development life cycle, Static Analysis Security Testing (SAST) refers to tools that statically verify the application against predefined security policies and are used in the early stages of development. Runtime Application Self-Protection (RASP) refers to tools that monitor the application at runtime for detecting and preventing policy violations. This dissertation tackles two main challenges related to RASP and SAST. We study how to devise portable RASP tools for access control and information flow control that can correctly enforce policies in a complete, tamper-proof and transparent manner. More importantly, we tackle the challenge of devising mechanisms to facilitate the development of reusable policies and their semantics for both RASP and SAST tools. Specifically, we envision an approach that, starting from a set of policy specifications and a RASP tool based on meta-programming, can derive a SAST tool with the minimum development effort.

1.1 Problem Statement

In this thesis, we argue that portability should be a key feature of mechanisms for securing client-side web applications due to the existing browser diversity. Moreover, to promote policy understanding and preventing programmers from making unintended mistakes during the policy specification, we argue that policy specification should be declarative. A declarative specification constrains the developer to particular patterns for defining a policy. This leads to less error-prone code and frees the developer from manually writing enforcement code [JH10].

Because developers use both RASP and SAST for securing applica-

tions, such a security mechanism should be reusable across different stages of the application's development. Unfortunately, in the state of the art, there is no approach that supports the combination of these features.

Given this context, this thesis considers the following two hypotheses:

Research hypothesis 1: it is possible to build a runtime monitor for access control and information flow control that is portable and tamper-proof using meta-programming.

Research hypothesis 2: it is possible to derive a SAST tool from an existing RASP tool, based on source code instrumentation, to verify the same set of policies ahead of time without re-implementing the policies' semantics.

1.2 Research Goals and Approach

Given the hypotheses in the problem statement, this dissertation pursues the following goals:

- the integration of application-level security policies in web applications by proposing portable and tamper-proof security mechanisms for AC and IFC policies (see Chapters 3 to 5), and
- the introduction of an integrated RASP and SAST approach (see Chapter 6) to facilitate the static verification and dynamic enforcement of the *same* set of security policies without reimplementing their semantics.

To achieve these goals, we explore language-level techniques for implementing portable and tamper-proof security mechanisms that enable the specification of fine-grained security policies for client-side web applications. We explore two families of security policies, *Access Control* (AC) and *Information Flow Control* (IFC).

Pilar 1 We use JavaScript's *meta-programming* facilities to ensure the portability and tamper-proofness of dynamic techniques for application-level security mechanisms.

Pilar 2 We use the technique of *abstract interpretation* to derive a program's execution model which enables reuse of the semantics of dynamic analyses deployed using JavaScript's meta-programming techniques.

1.3 Contributions

The contributions of this thesis are the following.

- Our first contribution is GUARDIA, a declarative embedded Domain-Specific Language (DSL) to enforce fine-grained access control security policies dynamically. GUARDIA decouples policy specification from their enforcement, allowing developers to deploy policies using either source code instrumentation or JavaScript proxies. We evaluate GUARDIA's expressivity, applicability, and performance.

- Our second contribution is GIFC, a portable IFC enforcement mechanism. In GIFC, the *permissiveness* of its *Permissive Upgrade* (PU) [AF10] monitoring mechanism is improved by upgrading the security label of write targets before the execution takes a branch conditioned by security-sensitive data. This automatic upgrading of variables is possible because the monitoring mechanism has access to the *Abstract Syntax Tree* (AST) during the analysis of the program operations. GIFC's instrumentation platform is written in JavaScript, enabling code instrumentation at runtime, which is needed for tracking IFC on JavaScript constructs like `eval`.

- Our last contribution is an approach to *safely* and *efficiently* derive SAST from RASP, starting from a single set of policy specifications. Specifically, we introduce a *two-phase* abstract interpretation approach that frees developers from re-implementing the policies and, more importantly, the semantics of such policies in a static analysis tool. Splitting the analysis into two phases enables the use of different analysis parameters for each phase, which are used to improve the precision and analysis speed. We build on the work of JIPDA, which provided us with the syntax and abstract semantics of JS_0 [NSDD17] where we describe our approach. We also implemented our approach on top of ARAN for the RASP component

and JIPDA, which implements JS_0's semantics, for the SAST component. Finally, we empirically evaluate the trade-offs of our two-phase abstract interpretation approach with respect to precision and analysis speed.

1.4 Supporting Publications

The following publications directly support this dissertation. Other papers supporting the research were presented at different workshops [SNG16, MSBG21].

- **GUARDIA: specification and enforcement of JavaScript security policies without VM modifications** Angel Luis Scull Pupo, Jens Nicolay and Elisa Gonzalez Boix. *Proceedings of the 15th International Conference on Managed Languages & Runtimes (17:1-17:15).* Association for Computing Machinery (ACM), 2018.

  This paper introduced the first version of GUARDIA. In this version, GUARDIA's enforcement is implemented using JavaScript proxies. However, the specification language was already decoupled from the enforcement.

- **Practical Information Flow Control for Web Applications** Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. *Lecture Notes in Computer Science : Proceedings of the 18th International Conference on Runtime Verification.* Vol. 11237, p. 372-388, Springer, 2018.

  This paper introduced GIFC, a run-time monitoring mechanism for enforcing IFC security policies without browser modifications. This paper shows that the enforcement based on source code instrumentation can cover several practical JavaScript and browsing context challenges such as dynamic code evaluation and built-in libraries while being permissive.

- **Deriving Static Security Testing From Runtime Security Protection For Web Applications** Angel Luis Scull Pupo, Jens Nicolay and Elisa Gonzalez Boix. *The Art, Science, and Engineering of Programming.* 2022, accepted for publication.

This paper introduces an integrated approach for deriving SAST from RASP while reusing the policy specification and semantics. The paper shows that our two-phase approach offers a better trade-off regarding analysis precision and speed compared to analyzing instrumented applications in only one phase.

- **GuardiaML: Machine Learning-Assisted Dynamic Information Flow Control** Angel Luis Scull Pupo, Jens Nicolay, Kyriakos Efthymiadis, Ann Nowe, Coen De Roover and Elisa Gonzalez Boix. *Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2019).* p.624-628, IEEE, 2019.

  This paper integrates GIFC with machine learning in an IDE plugin to help developers find IFC policy violations. Specifically, developers can benefit from the artificial intelligence assisted suggestions of sources and sinks during the application development.

- **Tamper-proof security mechanism against liar objects in JavaScript applications** Angel Luis Scull Pupo, Jens Nicolay and Elisa Gonzalez Boix. *Fourth International Workshop on Programming Technology for the Future Web.* (ProWeb 2020). 2020.

  This workshop paper discusses the challenges regarding tamper-proofness of a dynamic monitoring mechanism based on source code instrumentation.

1.5 Dissertation Outline

In the following, we describe the structure of this dissertation.

Chapter 2: Motivation and Background This chapter provides the research context and motivates the need of application-level security mechanisms for client-side web applications. We also describe the necessary background information related to access control, information flow control, and runtime monitoring mechanisms. Of each policy family, we describe their state of the art in the context client-side web applications in three categories: purely dynamic, static, and hybrid approaches.

Chapter 3: Guardia: Access Control Policies for Web Applications This chapter describes the design and implementation of GUARDIA. First, we explain how programmers can use GUARDIA to specify policies appearing in related work. Then, we describe the main elements of both the JavaScript proxy-based and code instrumentation-based enforcement mechanisms. The evaluation looks into GUARDIA's expressivity, applicability, and performance. The chapter ends with discussing the transparency and correctness properties of both enforcement mechanisms described in the chapter.

Chapter 4: Practical and Permissive Dynamic IFC This chapter discusses the dynamic code evaluation, external libraries support, permissiveness, and portability challenges, posed by both JavaScript and the browser context that influence the practicality of a dynamic IFC mechanism. Then, we describe GIFC's monitoring interface, user API, and the approach taken to solve the challenges discussed at the beginning of the chapter. Finally, the chapter presents a precision and performance evaluation of GIFC. The precision evaluation uses a state of the art benchmark designed for measuring the precision of IFC mechanisms, which we used to compare GIFC with three state of the art dynamic IFC mechanisms. The performance evaluation relates GIFC's performance to the same three dynamic IFC mechanisms in a different set of benchmark programs taken from related work.

Chapter 5: Tamper-proof and Transparent Analysis of Web Applications This chapter focuses on the integrity challenges that a monitoring mechanism for client-side web applications faces when it is implemented based on source code instrumentation. It starts by describing those integrity challenges from the JavaScript and browser context perspective. Then, the chapter describes our approaches to solve each of the challenges. Finally, we compare our proposed solutions to state of the art solutions.

Chapter 6: Deriving Static Analysis for Web Applications This chapter starts by motivating the need for an integrated toolchain to statically verify and dynamically enforce the same set of security policies. Then, the chapter describes the necessary features needed

of a monitoring mechanism based on source code instrumentation to enable our two-phases abstract interpretation approach. Then, we formally describe our two-phase approach using small-step semantics. The evaluation describes our two-phase static analysis' viability by applying it to the benchmark programs used for the evaluation of GUARDIA and GIFC. We also compare our approach to the analysis of the application in one phase in terms of performance and precision.

Chapter 2

Motivation and Background

The previous chapter briefly described the threats to which client-side web applications are exposed. We also mentioned the shortcomings of the browsers' security mechanisms, concluding that developers need additional security mechanisms for their applications.

This chapter explains in detail those threats and the browser's security mechanisms shortcomings. Motivated by these problems, we study state of the art approaches for application-level security mechanisms. Finally, we give a broad description of different design and implementation challenges of application-level security mechanisms.

2.1 Motivating Example

To illustrate vulnerabilities that can be used to harm modern web applications, we will employ an application that is part of the Web Application Security Project (OWASP) foundation [OWAa] called Juice Shop. Juice Shop is a deliberately vulnerable web application used by security researchers and developers to learn the best practices of web application security. It contains above 60 vulnerabilities covering all categories of the OWASP Top Ten [OWAb]. As shown in Figure 2.1, Juice Shop is a shopping application with the typical listing of products and a search bar. Users can create accounts, add items to a basket, etc. We now present two examples of vulnerabilities that an attacker can exploit.

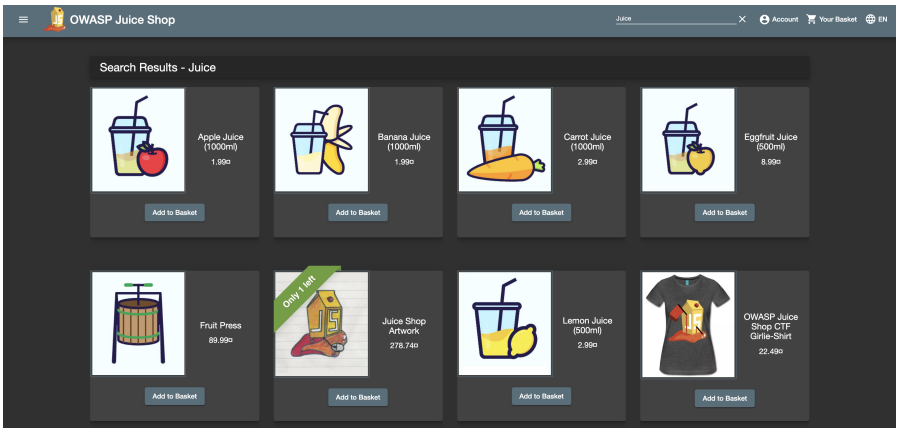


Figure 2.1: Screenshot of the Juice Shop application's welcome page.

Example 1: Cross-Site Script Attack. This example illustrates an *improper data sanitisation* vulnerability. This vulnerability exposes the application to Cross-Site Scripting (XSS) attacks. An XSS attack consists of injecting malicious code into the application's HTML code that will eventually be presented to a victim's browser. Then, when a user requests this "infected" application, the injected code is executed in his/her browser. This injected code, may cause harm, for example, by stealing private information from the user's account or making the application unusable. The underlying problem is that the browser cannot distinguish between data provided by the user and the application's code when the user input is inlined with the HTML code.

As a concrete example, consider the JUICE SHOP's search bar, which allows users to filter the product listings using text. Let us assume that JUICE SHOP is hosted in *http://myjuiceshop.com/*. When the user searches the term "apple", the browser sends *http://myjuiceshop.com/#/search?q=apple* request to the server. When the server answers with the results of the search, the browser parses them and shows an HTML page to the user. In this example, the user interface shown to the user includes the input text along with the search results as shown in the red box in Figure 2.2.

JUICE SHOP includes code to prevent the browser from executing data, such as user input, as code: it removes `<script>` tags from a given data. However, such sanitisation code does not prevent the attacker from using

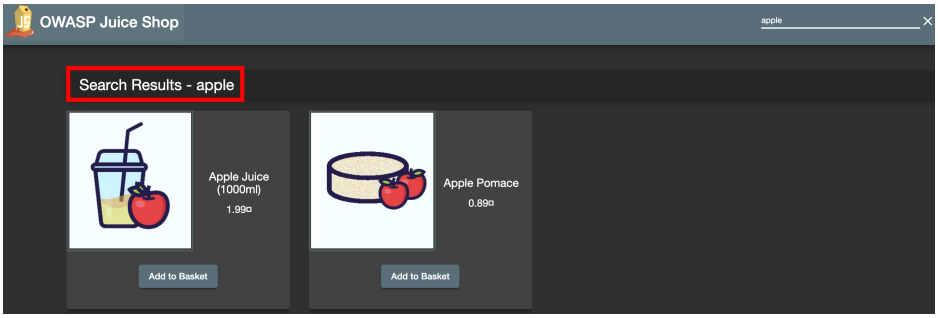


Figure 2.2: Screenshot of the search results.

other elements like images, iframes, etc., to perform the XSS attack. For example, he can trick an application's user to click a link with the following url:

```
1 http://myjuiceshop.com/#/search?q=<img src onerror="fetch('evil.com?q
    ='+document.cookie)">
```

Now the search term becomes the malicious string `<img src onerror="fetch('evil.com?q='+document.cookie)` which includes the `img` HTML element tag. When the victim clicks the link, the application will understand this search term as a valid input from a legitimate user. Therefore, the application will attempt to render the image given as input. However, the image load fails because the missing value of the `src` property, triggering the `onerror` handler to execute the given JavaScript code (i.e., `fetch('evil.com?q='+document.cookie)`). In this way, the attacker can perform an XSS attack on the HTML page served to the user.

Example 2: Third-party library attack. This example illustrates the risks that entail when using third-party source code in web applications.

The developer of a client-side web application can use `script` elements to modularise the application. Script elements allow the browser to download and execute code from a specific `URL` as part of the application. Since the code included using `script` elements is executed under the same privileges, the included code has access to all browser's APIs. This implies that all user data and application specific information are exposed

to potentially malicious code which may, for example, leak user's private information to a third party server.

Consider as example Juice Shop, which includes third party JavaScript libraries from a Content Delivery Network (CDN) including jQuery [1] and Cookie-consent [2]. The Juice Shop developer trusts the included libraries to perform operations using specific APIs. For example, jQuery is expected only to manipulate the Document Object Model (DOM). Changing element styles, adding new elements to the document are side effects expected when using this library. However, making a network request is not. In this case, it would be desirable to prevent jQuery from using any other API besides DOM. Unfortunately, this is not possible.

In the Cookie-consent library case, the developer assumes that the library communicates information through the network about the user's consent regarding the cookies and nothing else. Moreover, it would expect the library to communicate this information to a predefined server. However, this library can communicate any information to any server without restrictions.

In conclusion, browser APIs can be freely used either by the application's code or third-party code, which may have harmful consequences, e.g., hampering user's information privacy and application integrity.

2.2 Browser-Level Security

To help mitigate exploits of the security vulnerabilities explained above, modern browsers provide 2 mechanisms: the *Same-Origin Policy* (SOP) and *Content Security Policy* (CSP).

Same-Origin Policy The goal of SOP is to restrict documents or code with different web origins from interacting deliberatively [Goo09, Mozb]. It helps isolate mutually untrusted documents, reducing the attack surface to which those documents are exposed.

An *origin* is defined by the scheme, host and port of an URL. For example, in the URL `https://myjuiceshop.com` the values `https`, `myjuiceshop.com` and `443` are the scheme, host and port of the domain.

[1] https://jquery.com/

[2] https://www.osano.com/cookieconsent

The previous origin is different from the origin `http://myjuiceshop.com` (http, myjuiceshop.com, 80) because they differ in port and protocol.

Although the SOP isolates content from different origins, it allows embedding images, style sheets, scripts, etc., retrieved from different origins. For example, the `script` element in the snippet below can execute JavaScript code that can be harmful to the application as described in the previous section. This drawback that make it insufficient for securing modern client-side web applications.

```
1    <script src="profile.myjuiceshop.com/util.js"></script>
```

Listing 2.1: Example of importing a third-party library in a client-side web application.

Content Security Policy (CSP) CSP enables developers to specify from which domains the browser can load resources such as images, JavaScript scripts, stylesheets, etc. [SSM10, WSLJ16]. The main goal of CSP is to prevent the injection of malicious code (i.e., XSS attacks). A developer can specify from which origins the browser can evaluate JavaScript code by specifying a CSP policy in the HTML document header. For example, the developer of Juice Shop can mitigate the XSS attack of Section 2.1 using the following policy:

```
1  <meta http-equiv="Content-Security-Policy" content="script-src 'self'
       cdnjs.cloudflare.com">
```

Listing 2.2: Example of the specification of a CSP policy.

For an HTML document containing the policy above, the browser will only evaluate JavaScript code originating from the document's domain (i.e., the `self` pseudo-variable) or from `cdnjs.cloudflare.com`. The code originating from domains other than the one configured in the policy are blocked by the browser. More importantly, the browser will block the execution of any inlined JavaScript code, including event handlers such as the `onerror` handler from the XSS example of Section 2.1. Even though CSP offer some form of access control, it also has some limitations:

- Once a CSP policy grants accesses to a third-party script, nothing prevents the script from accessing and possibly leaking sensitive information within the browsing environment [WSLJ16, LKG+17].

- The sandboxing directive offered by the CSP is limited to a small set of privilege restrictions. It only applies to a small set of concerns (e.g., popups, script elements within the included document, top-level navigations, etc.) which makes it insufficient to address modern applications' security needs. For example, CSP cannot be used for enforcing privacy policies where precise tracking of how the information flow during the execution, or to enforce fine-grained access control policies. For example, CSP cannot prevent the use of a web API based on the arguments of the call of such an API.

In short, the browser's security mechanisms, SOP and CSP, are insufficient to secure client-side web applications. A large scale study of over 1 billion hostnames done by Weichselbaum, Spagnuolo and Lekies [WSLJ16] found that 78.8% of distinct policies use script whitelists that allows attackers to bypass CSP and that 99.34% of hosts with CSP use policies that do not prevent XSS attacks. Moreover, their implementation has inconsistencies across browsers reported previously [YL16, WSLJ16, LKG+17, CRB16, SBR17]. Finally, CSP and SOP lack the granularity required to secure modern client-side web applications. For example, they do not offer developers a mechanism to express fine-grained access control over the browser APIs. Moreover, they do not prevent the JavaScript code within a specific page to communicate information to a third-party server once the code accesses the information. Therefore, these mechanisms should be complemented with application-level security policies.

2.2.1 Attacker Model

In this dissertation, we assume an *attacker model* in which the attacker can execute arbitrary code in the context of a client-side web application. This corresponds to a scenario where the attacker may take advantage of vulnerabilities present in the application such as, improperly sanitised input, a wrong configuration of a CSP policy, etc. We assume that exploiting such a vulnerability enabled the attacker to store JavaScript in the application's database as part of a user input mechanism (e.g., a user comments, search box, etc.). For example, he/she can use a post comment (containing JavaScript code) about an item. When a victim (i.e., benign user) visits a page that loads and executes the attacker code as part of page evaluation, this attacker code is executed in the browser with the

same privileges as the code of the page. Especially if the victim is an authenticated user, the attacker can obtain sensitive information. Similarly, an attacker can cause the application misbehaviour by, for example, exhausting its resources.

2.3 Application-Level Security Policies

As mentioned before, an application-level security policy expresses a program property that must hold during the entire application's execution. In other words, a security policy restricts application behavior to prevent *vulnerabilities* from occurring or being exploited. Schneider et al. [Sch00] classifies application-level security policies in *access control policies*, *information flow control policies* and *availability policies*. In this dissertation, we focus on access control (AC) and information flow control (IFC) policies. In the following sections, we describe background information about the main concepts and elements of AC and IFC policies.

2.3.1 Access Control

An Access Control (AC) policy restricts what operations principals can perform on objects. The specification of an AC policy must express who can access specific information and under what circumstances. There are three main elements that can be identified in an AC policy:

- **object** represents the entity being secured by the AC system. It can be an object, a property field, a function, etc.
- **subject** refers to the principal of the program execution. During program execution, the principal is accountable for the privileges the user of the program has. A principal can be a process, a thread, an object in the program, etc.
- **privilege** is the abstract notion of *access* that a subject has to an object.

An example of an AC policy in the context of the Juice Shop application is to "*deny setting the* `innerHtml` *on the HTML elements of the page*". In this example, the object being protected is the element's `innerHtml` property, the subject is the script being executed, and *deny*

setting the `innerHtml` value express the privilege given to the principal to *write* to the element's property.

In general, access control checks place restrictions on the release of information, but not its propagation [SM03]. When information is released from its container, the program may attempt to improperly leak released information.

2.3.2 Information Flow Control

Trusting that programs (or third-party libraries used by those programs) in a large system are trustworthy is unrealistic. For example, third-party libraries used by Juice Shop do not give evidence that proves these libraries do not leak information. Therefore, it is necessary to control how the information flows through all these untrusted entities to prevent its unwanted release.

In 1973, Lampson [Lam73] described the problem of information leakage as the *confinement* problem. A confined program supplied with sensitive data must ensure that the data remains confidential during its execution. Later, in 1977 Biba [Bib77] defines *integrity* as the concern within a computer system that ensures that the system behaves as its developer intended it. Both integrity and confidentiality can be expressed as information flow problems [HS12a].

An IFC policy restricts what subjects can infer about objects from observing system behaviour. IFC can be used to enforce *confidentiality*, preventing trusted inputs from leaking to public outputs, and *integrity*, preventing untrusted inputs from affecting trusted outputs [HS12a, BSS17].

The semantic foundations of Information Flow Control (IFC) are based on the concept of *noninterference* [GM82, HS12a]. For example, the notion demands that public outputs do not depend on secret inputs for a confidential program. Conversely, to ensure a program's integrity, noninterference demands that the public inputs cannot interfere with the trusted outputs. There are four main elements in an IFC policy:

- **label** expresses the security level of program values. For example, a *low* label `L` can be associated with non-sensitive program values that are allowed to be publicly observable. In the Juice Shop application, a low value can be the names and description of the products sold in the store. In contrast, *high* labels `H` can be associated with

sensitive values that should remain private to the application. For example, the email *address* and *password* in the JUICE SHOP application are considered security sensitive.

- **information source** refers to the entities in a program that produce values with a particular label. Examples of sources in JUICE SHOP include the page's input elements, the document's `cookies` and the geolocation API. A security label (usually `H`) is associated with the values produced by such sources.

- **information sink** is an information *channel* that allows observing information communicated through this channel. For example, the `fetch()` function, which allows making network requests, is considered a sink because it allows writing to the network channel. An IFC policy identifies information sinks in a program and associates them with a label as well.

- **ordering** establishes how different security levels are related, for example, through the use of a total or partial order (lattice) between labels [Den76, DD77]. In our example, we would have `L` $\subset$ `H`, expressing that `H` is more sensitive than `L`, so that `H` values are not allowed to flow to `L` sinks. For example, given the email and password elements as `H` and `fetch()` as a `L` sink, the IFC enforcement must prevent either password or email values from flowing to any `fetch()` call. However, during program execution, it is not trivial to know how program operations' interaction produces information flows.

Sources and sinks together with the labels and the ordering enable the specification of an IFC policy. In the following sections we describe different aspects relevant to the enforcement of IFC policies.

2.3.2.1 Explicit and Implicit Flows

An information flow from a *source* to a *sink* can be classified as *explicit* or *implicit* [Den76, DD77, HS12a, BSS17].

Explicit flows

Explicit flows arise from the direct copy of the information. For example, the assignment expression `y = x` causes an explicit flow from variable `x` to `y`, and after the assignment `y` will have the same security label as `x`. In general, explicit information flow to variable `y` occurs from any expression `x` (e.g., binary, I/O expressions, etc.) that directly assign information to `y` derived from the expression operands [Den76].

Implicit flows

Implicit flows arise when the execution of an explicit flow statement depends on a security-sensitive value (`H`). For example, after executing the statement `if (z) y=0 else y=1` the value of variable `y` depends on the value of `z`. This results in an implicit flow from `z` to `y`, and after the `if` statement the value of `y` will have the same label as `z`. Language constructs that produce implicit flows include control flow structures such as `while`, `for` and `switch` statements.

Other language constructs like `return` in a non-final position, `break`, `continue`, and `throw`, etc., can extend the implicit flows beyond the enclosing syntactic boundaries. To illustrate this problem, consider Listing 2.3 in which function `g` prints its argument. The `try-catch` block (line 4-9) catches the exception thrown conditionally on line 6. Note that throwing the exception depends on the trustworthiness of the `secret` value. Moreover, the execution `g(false)` on line 8 also depends on the secret, even though the call is outside the syntactic boundaries of the `if` statement. In this case, the dependency on the secret is extended to the first exception handler i.e., catch block found after throwing the exception.

```
1  function g(x){
2    print(x);
3  }
4  try{
5    if(secret){
6      throw new Error();
7    }
8    g(false);
9  } catch(e){}
10 g(true);
```

Listing 2.3: Non-structured implicit flow example.

Implicit flows are further classified as *observable* or *hidden* implicit flows [BSS17, SSB+19]. An *observable implicit flow* arises when a variable depends on a higher-label value. Consider the program in Listing 2.4 using an `if` statement that conditionally assigns to variables `x` or `y`.

```
1 let h = true;   // h -> H
2 let x = 0;      // x -> L
3 let y = 0;      // y -> L
4 if(h){
5   x = 1;
6 }else{
7   y = 1;
8 }
```

Listing 2.4: Implicit flow example 1.

Variable `h` is considered secret `H` (for example, a boolean flag indicating if the current user is an administrator or not) while `x` and `y` are public values `L`. There is an observable implicit flow at line 5 because the assignment is conditioned by a higher context label. This is, `x`'s label is lower than `h`'s label.

A *hidden implicit flow* arises from not executed assignments conditioned by higher security expressions. In Listing 2.4, the lack of the assignment of `y` at line 7 causes a hidden implicit flow from `h` to `y` because in a different execution `y` may have a different value after the execution exits the `if` statement.

2.3.2.2 Flow Sensitivity

In IFC, flow sensitivity is a property that specifies whether variables can hold values of different security levels (e.g., `L` and `H`) during the execution of the program [RS10].

Flow insensitivity In a flow insensitive information flow tracking, the program variables' security label is static, meaning that a variable's label cannot be changed during the program execution.

```
1 let h = false;   // h -> H
2 if(h){
3   x = 1;
4 }
```

Listing 2.5: Implicit flow example 2.

For example, in a flow insensitive IFC enforcement of the program shown in Listing 2.5 the variable `h` is statically assigned the security level `H`. At line 1, the program attempts to assign `h` to `false` which will cause an *explicit flow* from `false`(`L`) to `x`(H). The enforcement will classify the program as insecure because the security level of the variable is not lowered. However, in this case this is too restrictive because, after overriding `x` with an `L` value, it is not possible for the program to leak information.

Flow sensitivity In a flow-sensitive enforcement, the security level of variables can be dynamically changed during the enforcement, except for *sinks*. In our example shown in Listing 2.5, this means that the assignment `h = false` is allowed, changing the security level of `h` to `L`. Therefore, the analysis will *precisely* accept the program as secure.

A dynamic flow sensitive IFC enforcement may miss some information flows and therefore become *unsound* (i.e., report false negatives) if the label of variables used in conditional statements (e.g., `if`) are also conditioned by `H` values [AF10, BHS12, HBS15]. Soundness is warranted if the enforcement ensures that observable outputs comply with a given information flow control policy [BR16, LBW05].

To illustrate how a dynamic flow sensitive IFC enforcement can be unsound, consider a program shown in Listing 2.6. It turns out that the execution of that program can release information about secrets using public (`L`) variables. First, consider that `secret H` holds a trusty value. During the program execution, the `if` test in line 2 will pass, which entails the assignment at line 3. After the assignment `temp` holds `1` as value and `H` as security label because of the assignment in a high `H` context. Then, the `if` test at line 5 fails, which means the assignment at line 6 does not get executed, and `public` holds `1` as a value at the end of the program. The program terminates successfully (i.e., the analysis does not halt the execution).

```
1 public = 1, temp = 0; // public, temp -> L
2 if(secret){  // secret -> H
3   temp = 1;
4 }
5 if(temp != 1){
6   public = 0;
7 }
```

Listing 2.6: Implicit flow example 3.

Whenever `secret` has a false value, the `if` test at line 2 will fail and `temp` remains `0` (`L`). Then, the test at line 5 will pass, and the statement at line 6 will assign `0` to `public`. Note that the assignment happens in a *low* context because the variable `temp` was never assigned a *high* value. Therefore, the analysis will not render the execution invalid, allowing the program to leak information about `secret`. At the end of the two examples' execution, the `public` variable will thus hold information about `secret`.

2.3.2.3 Permissiveness

Permissiveness is the ability of an IFC enforcement to allow the execution of semantically secure programs [HBS15, CN15]. To overcome the soundness problems of flow sensitive dynamic IFC described in Section 2.3.2.2, [Zda02, AF09] proposed the *No-Sensitive Upgrade (NSU)* technique. Under this technique, any side effect that depends on secret information will terminate the execution. To illustrate this, consider the program in Listing 2.7. An NSU enforcement of this program terminates it when its execution reaches the assignment to `y` at line 5 because the occurrence of this side-effect depends on the secret value of variable `x`. Although this behavior is *sound*, the termination of the program execution is premature since the value of `y` is never used afterwards. From a practical point of view, an NSU enforcement will reject too many valid programs, which may render the enforcement useless.

```
1 let x = true; //H
2 let y = false;//L
3 let z = true; //L
4 if (x) {
5   y = false; //P
6 }
7 print(z);
```

Listing 2.7:

```
1 let x = true; //H
2 let y = false;//L
3 let z = true; //L
4 if (x) {
5   y = false; //P
6 }
7 print(y);
```

Listing 2.8:

```
1 let x = false;//H
2 let y = false;//L
3 let z = true; //L
4 if (x) {
5  y = false
6 };
7 print(y);
```

Listing 2.9:

Permissive Upgrade (PU) [AF10] is an alternative to NSU that provides a more permissive approach for handling implicit flows. A PU enforcement keeps track of secret-dependent values using a special label `P` that indicates that the information is *partially leaked*, i.e., it is currently `H` but in alternative executions may remain `L`. The execution is terminated

only when a partially leaked value is used in a conditional statement or flows to a public sink. Therefore, at the assignment to `y` in Listing 2.7, instead of stopping the execution as an NSU monitor would, a PU monitor tags the value of variable `y` with `P` and execution continues until the end. However, a PU monitor will halt the program's execution in Listing 2.8 when reaching the `print` statement as a result of the flow of a partially leaked `y P` to a sink `L`.

2.3.2.4 Taint Analysis

Taint analysis [TPF+09] is a lightweight form of information flow control. Taint analysis establishes sources of tainted information, information sinks and the means of tracking tainted information. In a taint analysis enforcement, values are either tainted or not. Also, some sources of information flows are not considered. For example, implicit flows caused by side effects conditioned on tainted information are usually discarded. However, explicit flows are handled in a similar way as described in Section 2.3.2.1.

In the context of client-side web applications security, this technique has been mainly used to detect integrity vulnerabilities. For example, an attacker can exploit a vulnerability in the application that allows user-provided data to reach the DOM without sanitisation (e.g., to perform an XSS attack). In this scenario, the data generated from user input is tainted while all DOM APIs are tagged as sinks. The task of the taint analysis then is to track how the user-supplied data flows through the program. Whenever tainted data reaches a DOM API call, an alarm is raised.

2.4 Deployment of Application-Level Security Policies

This section describes the state of the art of *deployment techniques* for application-level security mechanisms for client-side web applications. We focus the discussion on techniques deploying access control and information flow control mechanisms. In the following, we discuss background information on deployment techniques, and later we review the literature on static analyses for securing client-side web applications.

2.4.1 State of the Art of Dynamic Techniques for Client-Side Web Application Security

A dynamic deployment mechanism enables the monitoring of program operations at run-time. A *runtime monitor* also called reference monitor, observes the execution of a target system and raises an alarm when a security-relevant *event* is about to violate a security policy [Bib77, SMH01]. Raising the alarm usually implies reporting the violation, halting the current application event or halting the current execution. An application event or program operation can be reading a variable, applying a function, a binary operation, etc.

To correctly enforce the desired security policies, a runtime monitor must exhibit the following properties [And72, Bib77, ES00]:

Complete mediation. The runtime monitor must observe the program execution and enforce all security-relevant events. Complete mediation is achieved when all events that can cause a policy violation are observed, and the policy checks are enforced before executing the event.

Tamper proofness. The runtime monitor must be protected from accidentally or maliciously tampering. If the validation mechanism can be tampered with by third-party code, then the application's execution can be altered in impredictable ways.

Correctness. The runtime monitor must faithfully enforce the security policies. The correctness condition implies the possibility of being able to prove the reference monitor correct. Correctness is linked to the intended semantics of the security policy. Therefore, having a small execution monitor codebase helps proving the monitor both correct and complete.

Transparency. The runtime monitor must not alter the application's behaviour other than to raise an alarm. For example, the monitor should not perform side-effects on the application's values.

A common technique for observing the application execution is by *inlining* the monitor within the application execution [SMH01]. According to Erlingsson and Schneider [ES00], the specification of an inlined runtime monitor requires defining the following aspects:

- **security relevant events/operations** are the program operations that must be intercepted by the reference monitor;
- **security state** is the information of the current application execution (e.g., a counter or boolean flag) used by the monitor to check the validity of security relevant events; and
- **security updates** are small fragments of code that are executed in response to security relevant events that update the security state or raise security policy violations (e.g., stop the application's execution).

2.4.1.1 Implementation Techniques for Runtime Monitors

The most common implementation techniques for inlining a runtime monitor is by either modifying the execution engine of the target runtime, or using the meta-programming facilities of the host language's runtime to modify the application execution. Below we describe them and analyse the advantages and disadvantages of each technique.

VM-based Implementation A runtime monitor can be implemented as part of the target runtime by relying on *VM modifications* [RHZN+13, VADRD+11, ML10, HV05]. In this case, the execution engine's language semantics are modified to call the monitor at security-relevant operations. This approach benefits the monitor's integrity (i.e., tamper-proofness) as the monitor resides in a different address space from the application that the VM is executing. Another advantage of VM modification is the freedom to access all resources from the VM (call stack, etc.), which may not be available for other approaches.

An essential consequence of browser vendors' diversity for client-side applications is that VM-based security mechanisms need to be somehow supported by all those browser vendors. This diversity also carries all the maintenance and assurance of all properties that the monitor should exhibit. Therefore, requiring VM modifications constraints the *portability* of the resulting monitoring mechanism, which must be reimplemented and customised for each target runtime.

Meta-programming-based Implementation Alternatively, the implementation of such a monitor can be achieved by modifying the appli-

cation through meta-programming. This makes the monitor portable to different runtime. However, its integrity could be compromised if it is not well designed.

Many approaches provide policy enforcement on the fly by employing the host language's runtime *reflective* capabilities [MFM10, AVAB+12, SNG18, SCN+18, PSC09, RDW+07, YCIS07, KYC+08, CN15, STA18]. However, the reflective capabilities of a language may be too limited to monitor all security-relevant operations. Limited reflective capabilities, may also restrict the types of policies that can be enforced. In our context, in JavaScript, security policies can be implemented using proxies. However, JavaScript proxies cannot be used to track primitive values or operations on them, preventing the implementation of an IFC enforcement mechanism relying on JavaScript proxies alone.

A second implementation option based on meta-programming is *code instrumentation*. Using code instrumentation, the program's source code is modified by injecting code to protect the security-relevant operations. The result of this process is an equivalent application with security checks inlined within the source code. Code instrumentation may affect the instrumented program's transparency, particularly if the instrumented program makes extensive use of reflection.

Finally, an important aspect to discuss about the implementation techniques for runtime monitors is performance.

Implementing the reference monitor at the VM level, in general, may introduce low performance overhead as the monitor can have direct access to the implementation details of the language. Moreover, hooking into the language operations needs fewer levels of indirection as the implementor has the freedom to change the VM. As a downside, the portability is lost as the implementation cannot be ported to a different VM without substantial implementation efforts, as we already mentioned.

Inlining the monitor within the application source code using meta-programming has performance implications because of the extra level of indirection added by the program's operations emulation. Despite these performance implications, the instrumented application can still benefit from program optimisations at runtime.

In the remainder of this section, we survey dynamic state of the art mechanisms for enforcing AC and IFC for client-side web applications.

2.4.1.2 State of the Art of Mechanisms for Dynamic Enforcement of Access Control Policies

Dynamic enforcement of access control policies has been approached in different ways in the form of *access mediation* [RDW+07, ML10, MFM10, PSC09, YCIS07, VADRD+11], *sandboxing* [MSL+08, AVAB+12] and *ad-hoc* [HV05, MSR+19, PPA+20].

Access Mediation Reis et al. [RDW+07], proposed BrowserShield, a framework for vulnerability filtering on client-side web applications. In BrowserShield, HTML code and all included JavaScript code are instrumented in order to enforce filtering policies. A filtering policy in this approach has access to the operation information (e.g., arguments and function pointer of a function call) and can modify it according to the policy semantics. Policies are registered using *hooks* that allow programmers to specify which program operations the policy must enforce. For example, deploying a policy for intercepting function calls is done by means of the `addJSFunctionPolicy(function, policy)` hook, where `function` is the pointer of the security-sensitive function and `policy` is the function that implements the policy logic. Then, during the instrumented program execution, all function call operations trigger the registered `policy` to enforce the policy logic.

In [PSC09, MPS12], the authors propose light-weight self-protection wrappers for JavaScript. A wrapper is an object that encapsulates a target object to prevent unauthorised operations on its target. Similar to BrowserShield, in this approach, policies are specified in an *aspect-oriented* manner using wrappers to register policies defined as functions triggered by security-sensitive operations. The wrappers instrument the application at runtime by adding a function that acts as a proxy of the security-sensitive function. Listing 2.10 shows an example of a policy specification and deployment as proposed in [PSC09]. The implementation of `enforcePolicy` wraps the sensitive function, `document.createElement` in this case. The semantics of the policy is then specified as a function from line 2 to 9.

```
1 enforcePolicy({target: document, method:'createElement'},
2                function(invocation){
3                  var str = stringOf(invocation, 0);
4                  if(str.indexOf('iframe')>=0){
```

```
5                   return;
6                 }else{
7                   invocation.proceed();
8                 }
9               })
```

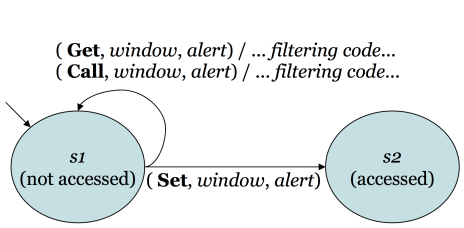
Listing 2.10: Policy implementation example: *Prevent dynamic creation of 'iframe' elements.* (Source: [PSC09])

Like BrowserShield and [PSC09, MPS12], in ConScript [ML10] policies are expressed as functions. In contrast to previous approaches discussed in this section, ConScript achieves access mediation by modifying the VM. More concretely, it extends the JavaScript engine to weave the policy enforcement code to the function pointer being protected. While this implementation choice threatens the portability of the approach, the challenge of implementing a *deep advice* that fully mediate accesses to the secured resources motivated this choice. In a deep advice system, the function representation is extended with a pointer to its advice function. Whenever the function is called, its advice is executed to enforce the security policy.

Inspired by the deep advice approach of ConScript, Van Acker et al. [VADRD+11] proposed WebJail, a client-side architecture that enables *least-privilege* integration of components into a web mashup. In contrast to the previous approaches, where policies are imperatively specified using functions, in WebJail, policies are specified as mapping web API categories to *restrictions*. As shown in Listing 2.11, a policy is a JavaScript Object Notation (JSON) object where properties identify categories of web APIs, and their values are the restriction setting the access that the component will have over the APIs on a particular category. For example, `"extcomm"` allow web APIs for networking (e.g., `XMLHttpRequest, WebSocket, fetch, etc.`) to have communication *only* with `"google.com"` or `"youtube.com"`.

```
1 {
2   "framecomm" : "yes",
3   "extcomm" : ["google.com", "youtube.com"],
4 }
```

Listing 2.11: Policy implementation example to restrict web API usage. (Adapted from: [VADRD+11])




```
<template name = "FuncReplacement">
  <object> obj </object>
  <property> prop </property>
  <states> s1, s2 </states>
  <replacementAction>
    ... filtering code ...
  </replacementAction>
</template>
```

Figure 2.3: Policy example to *prevent resource abuse*. (Source: [KYC+08])

The CoreScript [YCIS07, KYC+08] approach uses source code instrumentation as implementation technique for policy enforcement. In CoreScript, the authors propose `XML` as specification language for encoding policies based on edit automata [LBW05]. Figure 2.3 shows a police to prevent attackers from exhausting the machine resources by calling `window.alert()` many times. In the example, the policy defines an initial state and which program operations force a transition of the automata, which triggers the policy enforcement code. Whenever the `window.alert` is read (Get) or executed (i.e Call) the policy triggers the enforcement code. However, if the `window.alert` is assigned a new value (Set) the enforcement code is not triggered anymore. The malicious alert is overridden, and the new value is not considered dangerous.

The authors of ObjectViews [MFM10] propose an abstraction based on proxies for securely sharing sensitive objects between frames in a client-side web application. An object view is the composition of a proxy and a policy, implemented as an aspect system. In addition to the aspect based policies, ObjectViews propose document sharing policies. The declarative policy specification language allows developers to specify a set of DOM elements, a set of restrictions to apply to those elements, and a set of object interactions rules to be applied to those elements.

```
1 var m = makePolicyView(makeView(document));
2 var policy = [{"selector": "(//*[@class='example'])
3                             (//*[@class='example']//*)",
4               "enabled": true,
5               "defaultFieldActions": {read: permit}}];
6 m.applyPolicy(policy);
7 return m.view;
```

Listing 2.12: Policy example to restrict a subtree to read only operations if the root's class name includes `example`. (Adapted from: [MFM10])

Listing 2.12 shows an example policy for sharing the `document` element between frames. In the code, the `selector` property at line 2 uses XPath [W3C99] expressions to select the HTML elements to which the restrictions in lines 4 and 5 will be applied. Specifically, the policy enforces that any node containing a `class` property named `example` will be `enabled` (accessible to the receiver of the view), allowing only `read` operations on such nodes' properties.

Sandboxing Access control of security sensitive resources can be implemented by *isolating* the untrusted source code [MSL+08, AVAB+12]. Untrusted code is executed in a controlled environment, where the code is given the least privileges (and is subject to security policies) needed to perform its computation.

A representative approach of sandboxing is Caja [MSL+08], a JavaScript subset that enables object-capability security [Mil06]. Untrusted third-party libraries are statically analysed to insert security checks preventing the application from accessing global variables. When the application is executed, the code does not have ambient authority.

Another approach for isolating untrusted code is JSand [AVAB+12], a framework that enables secure confinement of third-party scripts through an object-capability environment. Such an environment allow developers to isolate third-party scripts from security-sensitive objects. Isolation is achieved by implementing a membrane pattern that installs policies around security-sensitive objects. Any object originating from a membrane is transitively wrapped and therefore subject to the membrane's policy.

Ad-hoc approaches Hallaraker and Vigna [HV05] proposed an audit system for client-side web applications which enables the implementation of auditing policies as a state transition model using JavaScript. The audit system monitors the application's execution and reports any event that violating the audit policies. The authors do not propose a policy language for expressing such policies, having policies and their enforcement mixed together. This makes policy understanding and maintainability difficult.

Richards et al. [RHZN+13], propose the use of the delimited histories with revocation to enforce access control policies. In this approach, the objects in the application are extended with ownership annotations related

to the *origin* of the code. Then, in a particular computation (e.g., function call), a history of the computation is recorded until it reaches a decision (i.e., a call to a native function) or suspension point (i.e., the end of the computation). Whenever one of these program states is reached, a policy is enforced that either allows the computation, or rolls back all the write operations within that computation.

ScriptProtect [MSR+19] prevents the exploits of XSS vulnerabilities in third-party libraries by instrumenting APIs that enable code injection. In this approach, policies are specifically designed to sanitise the values passed to APIs. The authors of ScriptProtect assume that third-party libraries are benign. Therefore they do not tackle challenges like tamper-proofness.

Phung et al. [PPA+20] propose MyWebGuard, a user-centric tool that allows the enforcement of privacy policies based on the origin (scheme, host, port) of the code. The implementation of MyWebGuard installs wrappers around sensitive objects to monitor operations on those objects. Policies are installed at a predefined set of sources and sinks of sensitive information. When a sink is called, the system can enforce a policy based on the caller's origin. These policies based on code origins can, for example, prevent specific third-party libraries from accessing specific APIs.

2.4.1.3 State of the Art of Mechanisms for Dynamic Enforcement of Information Flow Control Policies

In this section, we discuss dynamic IFC approaches for JavaScript based on the seminal survey by Bielova et al. [BR16].

Fine-grained IFC Zdancewic [Zda02] introduced the concept of *No-Sensitive Upgrade* (NSU) which has then been applied to JavaScript in JSFlow [AF09, HS12b, HBBS14]. To relax NSU, JSFlow uses upgrade instructions for public labels before entering a more sensitive context. However, this requires programmer intervention to specify where and what the interpreter should upgrade, which can lead to misconfigurations. JSFlow is not portable, because it needs to be adapted for each JavaScript engine.

Santos and Rezk [SR14] were the first that developed an IFC inlining compiler for a core of JavaScript and developed a practical implementation of it. Bichhawat et al. [BRGH14] implemented a dynamic NSU-based

IFC mechanism for the JavaScript bytecode produced by Safari's WebKit engine. They formalize the Webkit's bytecode syntax and semantics, and their instrumentation mechanism, and prove non-interference.

Coarse-grained IFC Another approach for enforcing information flow control policies is by restricting the flow of information at the level of untrusted components [YNKM09, SYM+14]. In contrast to monitoring mechanisms that track flows through all program operations, the enforcement of policies is done coarsely at communication boundaries between components.

For example, to offer flexible development and privacy of sensitive data, Stefan et al. [SYM+14] propose COWL, a confinement system for untrusted code on client-side web applications. COWL is based on mandatory access control. Browsing contexts are extended with labeled pairs of boolean formulas that express which browsing contexts may read (*secrecy*) or write (*integrity*) context's data. COWL labeled policies are enforced by allowing a browsing context to send data to another context if the receiver's context label is restrictive (subsumes) as the sender's label.

Multi-Execution Secure Multi-Execution (SME) [DP10, RS16] takes a different approach than traditional monitoring approaches for IFC. Programs under SME are executed multiple times, once for each security level, using special rules for input and output operations. Executions that are not allowed to access sensitive information are provided with dummy values representing those sensitive values. For example a program calling `fetch L` with an URL `H` will be executed twice, one time for the level `L` and another for the level `H`. Intuitively, executing the program at level `L` will replace the URLs value with `undefined` to prevent a flow from a value `H` to a sink `L`. Executing the program at level `H` will prevent the execution of `fetch L`, preventing the release of any value `H`.

SME served as inspiration for *faceted values* [ASF17, NBF+18]. Values in a faceted execution carry as many values as there are security levels in the system, reducing the required multi-executions and consumption of machine resources.

2.4.2 State of the Art of Static Analysis for Client-Side Web Application Security

Besides dynamic enforcement of AC and IFC, some work has proposed static analysis to verify such policies. In contrast to the runtime enforcement (which bases its decision on program operations alone), a static analysis reasons about the program behavior without executing it. Type systems, abstract interpretation and logic programming through DATALOG are among the most prevalent approaches for statically verifing security policies in client-side web applications.

Type Systems Static analyses for information flow control, have been mostly implemented as a form of security type system [DD77, Den76, VIS96]. In security-typed languages, variables and expressions are annotated with security types to specify the desired policies. Then, the information flow analysis typing rules can be implemented as described in Section 2.3.2.1.

Sabelfeld and Myers [SM03] presented a very comprehensive and extensive study of essential properties and challenges for static analysis of information flow control policies. Hunt and Sands [HS06] investigate the formal properties of a family of flow-sensitive type systems for information flow control for a `While` language.

Deductive Databases Static analyses can be expressed as deductive databases systems using DATALOG. The main idea is to represent the program and the analysis as sets of facts and rules. A solver is then used to derive new facts from the rules until a fixed-point is reached.

Livshits and Lam [LL05] developed a points-to analysis specified in DATALOG for the implementation of a *taint object propagation* strategy. This strategy is then used for detecting different security vulnerabilities in web applications such as SQL injections and XSS attacks.

Guarnieri et al. [GL09] presents GATEKEEPER as a static analysis framework to detect vulnerabilities in JavaScript applications. Because the static analysis of the full set of features of JavaScript is hard, their static analysis approach considers a *safe* subset of JavaScript. Gatekeeper's analysis is based on a points-to analysis implemented using DATALOG where the policies are written as DATALOG queries.

The work on GATEKEEPER was extended by Guarnieri and Livshits

in Gulfstream [GL10], a static analysis framework for web applications that treats applications as a stream of source code. In Gulfstream, a points-to analysis in Datalog is staged between the server and a client. The server statically analyses the code that is available offline. Then, the results are sent to the client that update those results by analysing unknown code that the application may download from third parties.

Taly et al. [TEM+11] developed an automated tool named ENCAP that, given the implementation of an API reference monitor and a set of security sensitive objects, can verify the confinement of such an API. Their static analysis approach uses conventional points-to analysis to build a conservative Datalog model of all API methods, which is then used to verify the API confinement.

Abstract interpretation Trip et al. [TFP14a] propose a string analysis as a refinement of taint analysis for web applications. They represent their static analysis component as an abstract interpretation. Nicolay et al. [NSD16] proposed JS-QL, a static analysis framework for detecting security vulnerabilities in client side JavaScript applications. In JS-QL, the analysis happens over the application's flow-graph computed using abstract interpretation. The users of the framework specify queries using a DSL written in JavaScript based on regular path expressions [dLW03].

2.4.3 State of the Art of Hybrid Approaches for Client-Side Web Application Security

Static and dynamic analyses have their advantages and disadvantages. A static analysis does not affect the performance of the application as the analysis is done offline. Moreover, static analysis can inspect all possible application execution paths enabling the reasoning about the whole program. However, because answering the exact program behaviour is *undecidable*, a static analysis in the context of security typically may predict a larger set of policy violations that never happen during the application execution. In other words, static analyses techniques may face precision problems which can lead to reporting as invalid, too many valid programs [BSS17].

In contrast, most of the dynamic analyses can render precise results as the analysis can access the application's concrete values at runtime. However, a dynamic analysis that monitors the application execution may

add a non-negligible performance impact that may render the monitoring unusable in practice. Moreover, the analysis is limited to reason about the current execution path taken at runtime.

Motivated by the advantages of static and dynamic analyses, some approaches aim to bring together the benefits of both approaches. In what follows, we discuss approach that use a combination of static and dynamic analyses to enforce IFC policies for client-side web applications.

Hybrid approaches for IFC [CMJL09] have used static analysis to analyse as much information as possible offline, limiting the dynamic enforcement to places where the static analysis cannot precisely detect policy violations when additional source code loaded dynamically. For example, Chugh et al. [CMJL09] develop a staged information flow control for JavaScript. To overcome the limitations of statically analysing a web application, where the code is not fully available, the authors propose to perform information flow control in stages. The first stage, where most of the informations flows are computed, is done statically, resulting in a set of *residual checks* to be done at runtime. This static analysis is done through a static constraint-based analysis that computes the set of values of the variables of the available code that can flow to the dynamically loaded (untrusted) code. In the second stage, a syntactic check is done to verify whether any of the variables of the residual policy are read from or written within dynamically loaded code.

Other hybrid approaches for IFC use static analysis to improve the permissiveness of the runtime monitor by computing information related to branches not taken during the program execution [LBJS07]. For example, Moore and Chong [MC11] use a flow-sensitive security type system [HS06] to determine when and which variables cannot cause an information flow. The authors claim that the selective tracking of variables can reduce the overhead associated with storage needed for tracking the variables' security levels, and the performance overhead of the join operation on the security level of those variables.

Jest [CN15] is an IFC monitor for JavaScript that implements NSU. It uses the concept of *boxes* to associate label information with program values. To improve the permissiveness, Jest implements an intra-procedural control flow and exception analysis to determine control dependencies at branching points. Jest also relies on an external process to handle dynamic code evaluation, which degrades the application performance on

calls to `eval()`.

Hedin et al. [HBS15], develop a value-sensitive hybrid IFC monitor for a JavaScript-like language. Their approach use a monitoring mechanism improved with static analysis performed on the fly. The static analysis is used whenever there is an elevation of the security context. This allows to improve the permissiveness of the dynamic monitoring by identifying and elevating the label of write targets conditioned by security-sensitive values. To enable experiments with different static analysis and levels of precision and performance, the soundness of the system is ensured by the monitoring mechanism, which means that the static analysis can be imprecise without compromising the soundness of the hybrid approach.

Sayed et al. [STA18] introduce IF-TRANSPILER, a hybrid flow-sensitive monitor inlining framework for JavaScript applications. At a branching point, the static analysis collects and upgrades the label of all variables that could have been assigned in the untaken branch.

Fragoso et al. [FSJRS16] propose a hybrid analysis for information flow control of a JavaScript-like language where the heavy work is done statically using a security type system, leaving dynamic checks where necessary. The main ingredient of their static component is its ability to wrap statements inside an internal boundary whenever they cannot be precisely analysed (e.g., on dynamic object property access). This boundary identifies which statements need to be verified at runtime.

Vogt et al. [VNJ+07] propose a modified version of the JavaScript engine in Firefox which uses information flow control to prevent XSS attacks. Their monitoring mechanism is complemented with a static analysis that is invoked on-demand. Specifically, a combination of a linear taint analysis and stack analysis is used to ensure non-interference.

Tripp and Weisman [TFP14b] develop a hybrid analysis for performing a security assessment of client-side JavaScript code. In their approach, a web crawler retrieves and executes a web page to record useful runtime information. The resulting JavaScript code and recorded information are passed to a static taint analysis component for performing the actual security assessment. Hybrid analyses such as this one combine static and dynamic analysis so that they rely on each other for their operation.

2.5 Conclusion

This chapter described the main concepts about web applications regarding security.

We show through simple examples that the security needs of modern client-side web applications cannot be solved with browser's security mechanisms alone. In particular, we observe that SOP and CSP are insufficient for expressing fine-grained application-level policies, such as, AC and IFC policies. Motivated by these shortcomings, we studied different language-based approaches and their deployment techniques for access control and information flow control policies as well as implementations techniques for deploying policies.

With respect to AC policies, we observe that dynamic approaches tend to use imperative policy specifications using JavaScript functions. Declarative specification approaches are limited to certain aspects (e.g., enabling or disabling web APIs).

With respect to IFC policies, we observe much work has focused on dynamic enforcement over static analysis, mainly due to the dynamic and reflective nature of JavaScript. There is some consensus on implementing the monitoring mechanism using a variant of the NSU technique. However, there is no principled approach to increase permissiveness.

Given that client-side web applications run on many different browser vendors, we argue that portability is an essential feature of monitoring mechanisms for client-side web applications. In this regard, we observe that AC monitors tend to use meta-programming for the implementation. On the other hand, IFC monitors tend to rely on VM modifications to track and enforce policies.

This thesis explores access control (Chapter 3) and information flow control policies (Chapter 4) in both static and dynamic context. Because of the problems (e.g., portability and maintainability) that entails implementing the VM level monitoring, we explore the JavaScript reflective capabilities and source code instrumentation as deployment techniques to promote portability. Chapter 5 discusses language challenges and solutions regarding the integrity and transparency of our approach. Finally, Chapter 6 explores the challenges of deriving a static analysis from an existing dynamic monitoring mechanism.

Chapter 3

Guardia: Access Control Policies for Web Applications

This chapter focuses on the dynamic enforcement of access control (AC) policies. We start by discussing the advantages and disadvantages of the surveyed approaches' design choices for dynamic enforcement of AC policies described in Chapter 2. We then present the first contribution of this dissertation: Guardia, an internal Domain Specific Language (DSL) for specifying and enforcing AC security policies in JavaScript. Guardia combines a *declarative* policy specification language with a *decoupled* enforcement mechanism, making it possible to experiment with different enforcement techniques that do not require VM modifications. To the best of our knowledge, this combination is unique in the context of JavaScript web applications.

3.1 Motivation

Section 2.4.1.2 surveys the existing solutions for specifying and enforcing AC policies for client-side web applications. Based on this survey, we identified four design choices and associated benefits and shortcomings that motivated us to propose Guardia.

General-purpose vs. domain-specific specification languages

Some approaches express access control policies in a full-fledged *General-purpose Programming Language* (GPL) such as JavaScript or C++ [HV05, RDW+07, YCIS07, PSC09, ML10, AVAB+12]. Using a GPL provides developers with the freedom of using the complete set of features of the host language. However, relying on a GPL for a domain-specific concern (security) may introduce more accidental complexity [Gho11].

Designing a DSL for expressing security policies aims to free policy designers from the accidental complexity of a GPL. Some approaches propose a standalone (external) DSL language to express security policies, different from the host language of the application (e.g., [DNM15, KYC+08]). Relying on a new language potentially results in more freedom of expressiveness, but at the cost of having to learn the language first.

An internal DSL combines the best of the two worlds, as it provides the flexibility of an external DSL, while both the application and its security policy specifications are written in the same host language. This is the approach taken by WebJail for JavaScript and C++ [VADRD+11], and by ObjectViews for JavaScript [MFM10].

Imperative vs. declarative specifications

Access control security policies are usually specified at the granularity of methods and properties of objects. Many approaches propose an *imperative* specification of policies [RHZN+13, AVAB+12, ML10, PSC09, RDW+07, HV05]. Using an imperative specification offers flexibility but can lead to security misconfigurations and inconsistencies that attacks can exploit. The main disadvantage of an imperative specification is that developers are responsible for ensuring that policies are *tamper-proof* (i.e., its integrity cannot be compromised) and free of bugs that result in new vulnerabilities. Additionally, imperative policies are generally challenging to combine and reuse because they can assert various overlapping and conflicting concerns [JH10, HJS12].

Alternatively, security policies can be *declaratively* specified [YCIS07, KYC+08, DNM15]. A declarative approach offers a well-defined interface for specifying policies, constraining developers to particular patterns for defining a policy. Declarative specifications lead to less error-prone code and free developers from manually writing enforcement code [JH10]. How-

ever, a declarative policy specification language usually requires policy developers to use new notations for expressing their policies and additional support for enforcing them in an engine, parser, or compiler. For example, in CoreScript, developers describe policies in XML.

ConScript, ObjectViews and Phung et al. [PSC09] employ a hybrid approach in which policies are specified in an aspect-oriented manner, but security checks are written imperatively. None of these approaches provides a mechanism to combine policies.

Coupled vs. decoupled enforcement

In many imperative approaches, developers *mix* the code specifying security policies with their enforcement [HV05, PSC09, ML10, VADRD+11, AVAB+12, RHZN+13]. Developers have to manually encode or call the enforcement mechanism to perform the security checks. This decreases code reusability and maintainability.

Specifying security policies with a DSL enables a decoupling between the specification language and the enforcement mechanism [YCIS07]. The security policy language then interacts with the enforcement mechanism by means of a well-defined interface that provides runtime information regarding a security-relevant operation. The only approach that provides decoupling is CoreScript, in which the developer has to provide the action that the enforcement mechanism needs to take for a given policy.

3.1.1 Problem Statement

The previous observations have inspired the design of a novel approach for specifying and enforcing application-level access control policies, called Guardia. To the best of our knowledge, Guardia is the first approach to explore an internal DSL embedded in JavaScript for declaratively specifying security policies, featuring a decoupled enforcement mechanism without requiring VM modifications. Table 3.1 summarizes existing approaches and Guardia concerning the analyzed design choices. More in detail, Guardia is the result of the following design decisions.

- The main design choice of our work is to explore language-based security that does not require VM modifications.

- Inspired by [DNM15] and CoreScript [YCIS07], Guardia explores a domain-specific policy specification language.
- In contrast to those approaches, we explore an internal DSL embedded in JavaScript to express and compose complex policies. As both the target application and its security policies are written in the same language (JavaScript), this design choice may reduce the learning curve.
- A declarative specification of policies enables the decoupling between specification and enforcement. Guardia goes one step further than CoreScript and also allows developers to use different metaprogramming APIs for the enforcement mechanism (e.g., JavaScript proxy API, Virtual Values, code instrumentation APIs [SKBG13, CGDD16], etc.).

Section 3.2 gives a brief introduction to the policy language by means of examples. Then, in Section 3.3 we discuss the design choices and the advantages and limitations of two enforcement mechanisms based on metaprogramming. Finally, we compare Guardia with state of the art approaches for application-level access control for web applications.

3.2 Guardia at a Glance

In this section, we describe the specification language of Guardia that allows to declaratively express application-level access control security policies for client-side web applications written in JavaScript. Guardia is designed for ease of use; as such, it is designed as an internal DSL, meaning that its constructs are expressed using host language (JavaScript) constructs.

In Guardia's DSL, policies are written following a fluent style [Fow10] as shown in the examples of this section. Programmers construct a policy object made up out of predefined policy predicates by chaining method calls. Chaining method calls improves policy readability and understanding their purpose.

	GPL or DSL	Imperative or Declarative Specifications	Modified runtime enforcement?	Decoupled enforcement?
HV[HV05]	GPL	imperative	yes	no
ConScript[ML10][HV05]	GPL	imperative	yes	no
Richards et al. [RHZN+13]	GPL	imperative	yes	no
Phung et al. [PSC09]	GPL	imperative	no	no
JSand[AVAB+12]	GPL	imperative	no	no
BrowserShield[RDW+07]	GPL	imperative	no	unknown
CoreScript [YCIS07, KYC+08]	External DSL	declarative	no	yes but only policy code
Drossopoulou et al. [DNM15]	External DSL	declarative	not applicable	not applicable
ObjectViews[MFM10]	Internal DSL	partially declarative (inspired by AOP)	no	no
WebJail[VADRD+11]	Internal DSL	imperative	yes	no
GUARDIA	Internal DSL	declarative	no	yes

Table 3.1: Overview of surveyed approaches with respect to the analysed design choices.

Policy 1: Prevent `document.write()` calls.

```
1 GG.onCall(document.write).deny();
```

Listing 3.1: Policy 1: Deny calls to document.write().

Consider an application-specific policy for "preventing `document.write() calls`". Writing directly to the DOM can enable XSS attacks as discussed in Section 2.1 of Chapter 2. Expressing such a policy is done as shown in Listing 3.1. `GG.onCall()` creates a function call policy for the given argument `document.write`. In this case, the policy will prevent all attempts to call the function `document.write`.

Policy semantics (i.e., enforcement) and the fluent interface are decoupled. This decoupling enables the evolution of policy syntax and semantics separately. It also allows us to introduce syntactic sugar around some policy constructs to improve policy readability. For example, the building blocks `with` and `and` point to a common implementation and, therefore, the same policy semantics.

The current implementation of Guardia has several built-in policies that allow programmers to express AC policies to restrict language operations over sensitive resources. The list of these building blocks is shown in Table 3.2. These building blocks are categorised as *simple* or *higher-order*.

A simple block takes non-policy predicate values as arguments and returns a policy predicate as its result. A policy predicate is a closure that returns a `boolean` value as a result. For example, `onCall` takes a function pointer as argument and returns a predicate that tests if any given operation is a function call.

A higher-order block takes policy predicates as arguments, and returns a policy predicate as its result. The logical operators `and`, `or` and `not` are examples of such higher-order blocks in Guardia. These blocks may accept one or more policy predicates as arguments and return a policy predicate as result. Higher-order blocks are essential for enabling policy composition in a declarative manner.

In the rest of this section, we use Guardia to specify a number of policies appearing in the related work. Afterwards, we describe how Guardia and its application-specific policies can be deployed.

Block	Description
`onCall(target) => P`	Predicate for function calls.
`onRead(obj, prop) => P`	Predicate for object property reads operations.
`onWrite(obj, prop) => P`	Predicate for object property write operations.
`arg(predFn, typeArg, ...args) => P`	Enforces a given predicate over future runtime function calls arguments.
`targ(pos, type) => P`	Coerce a function call argument to a given type.
`and(...pols) => P`	Higher-order block for the logical 'and' operator.
`with(...pols) => P`	Syntatic sugar higher-order block with 'and' semantics.
`or(...pols) => P`	Higher-order block for the logical 'or' operator.
`not(...pols) => P`	Higher-order block for the logical 'not' operator.
`moreThan(times) => P`	Simple block to limit the number of occurrence of certain operations.
`afterRead(obj, prop) => P`	Simple block that remembers the reading of the given object property.
`afterWrite(obj, prop) => P`	Simple block that remembers the writing to an object property.
`deny() => undefined`	Deploys the policy into the policies database that deny the execution of the given policy predicates.
`allow() => undefined`	Deploys the policy into the policies database that only allows the operations that obey the given policy predicates.

Table 3.2: GUARDIA's API

Policy 2: Preventing dynamic creation of *iframe* elements.

Policies can be fine-grained, allowing developers to restrict the arguments given to a function at a call site. As another example, consider the policy for "preventing dynamic creation of *iframe* elements". Dynamic creation of `iframe` elements is considered dangerous because it enables an attacker to get a pointer to original built-in methods by accessing them from the `iframe` browsing context [PSC09].

```
1 //Assume that the application is protected with Policy 1 before the
      code below
2 let iframe = document.createElement('iframe');
3 document.body.append(iframe);
4
5 //copying the write method from the child iframe context
6 document.write = iframe.contentWindow.document.write;
7 document.write(...)
```

Listing 3.2: Example of restoring a pointer to a built-in method.

For example, an attacker could bypass Policy 1 by means of the code

in Listing 3.2. In this attack example, it is assumed that the application is protected using Policy 1 to prevent calls to `document.write` from being executed. However, whenever a *child* browsing context is created (lines 2 and 3), an attacker can use the child context's built-ins to restore the protected method (e.g., `document.write`).

Protecting against this attack vector implies preventing the application from creating child browsing contexts dynamically (e.g., `iframe`, `frame`, etc.). Listing 3.3 shows the GUARDIA implementation of such a policy to prevent the creation of `iframe` elements. Calls to `document.createElement(tagName)` will be only allowed when `tagName` is different from `'iframe'`. In this example, the `.with(...)` construct allows the composition of the policy on the function call with the policy on the arguments of the call (see `GG.arg(...)`).

```
1 GG.onCall(document.createElement)
2   .with(GG.arg(GG.equals(0,'iframe')))
3   .deny()
```

Listing 3.3: Policy 2: Prevent dynamic creation of *iframe* elements.

Policy 3: Limit the number of dynamically opened windows.

Kikuchi et al. [KYC+08], and Meyerovich et al. [ML10] define a policy to limit the number of attempts to open a popup window. Repeated creation of new windows in a web application can lead to the exhaustion of machine resources, and can also be exploited by malicious applications to send fake messages to an unsuspecting visitor.

```
1 GG.onCall(window.open).moreThan(3).deny();
```

Listing 3.4: Policy 3: Prevent opening more than three windows dynamically.

Limiting the number of dynamically opened windows is considered to be a *stateful* policy as it requires counting the number of times the `window.open` function has been called. Listing 3.4 shows how to write a policy that limits the number of dynamically opened windows to three. In this example, `...moreThan(3)` deploys the necessary code (i.e., policy predicate) to check the number of times the `window.open` function has been called and to increment a counter.

Policy 4: Prevent opening a new window without a location bar or for a URL that is not white-listed.

To exemplify the use of higher-order predicates, consider a policy to *"prevent opening a new window without a location bar or for a URL that is not white-listed"* shown in Listing 3.5. This policy was suggested by Phung et al. [PSC09] in order to prevent *forgery attacks*. In a forgery attack, a user is seduced to believe that an attacker's website is legitimate. For example, the attacker may use `window.open(...)` from a legitimate application to open its own web application in a different window without a `location` and `status` bar.

```
1 const hasLocation = GG.arg(GG.contains, GG.targ(2, String), "location=
      true")
2 const isWhitelisted = GG.arg(GG.inList, GG.targ(0, String), allowURLs)
3
4 GG.onCall(window.open).with(GG.and(hasLocation, isWhitelisted)).allow()
```

Listing 3.5: Policy 4: Higher-order policy predicate example. The policy prevents creation of new windows without location or white-listed urls.

In Listing 3.5, the policy predicates `hasLocation` and `isWhitelisted` (see lines 1 and 2) check whether `window.open` is being called with `true` for the status bar option, and with a *url* contained in `allowURLs`. In line 4, the policy predicates are combined using the higher-order block `and` to enforce that both predicates must hold to allow the `window.open` call. In this case, `.with(...)` is syntactic sugar for better policy readability. Finally, the policy is deployed using the `allow` block, which *only* calls `window.open` if the stated policy predicates hold.

Deployment: The first step to protect an application is by importing GUARDIA. This import can be done within the `head` tag of the HTML document, as shown below:

```
1 <html>
2 <head>
3  <script src="path/to/guardia.js"></script>
4  <script src="path/to/policies.js"></script>
5 </head>
6 ...
7 <html>
```

Importing Guardia is a similar process to importing any other JavaScript library. However, Guardia should be imported *before* any third-party library. Importing Guardia first ensures that Guardia is executed in a fresh (trusted) environment to prevent the introduction of vulnerabilities that may enable exploits to the application or Guardia itself (see Chapter 5 on tamper-proofness). After the import, programmers can access Guardia constructs using the `GG` global variable.

Depending on the type of application in which Guardia is being deployed, other minor changes may be required. For Single Page Applications, including Guardia in the initial page suffices to secure the entire application. For traditional web applications, in which the browser reloads the window on each request, Guardia can be added by using a proxy mechanism in the server that modifies each response.

3.3 Guardia's Enforcement Mechanism

Guardia's enforcement mechanism is decoupled from its policy specification API. Decoupling policy specification from enforcement enables us to investigate different meta-programming techniques to enforce policies without VM modifications. The monitoring mechanisms described in this section are based on intercepting *program operations* such as function calls, object property access, and so on. These mechanisms use a form of *intercession* [KdRB93], in which *traps* conditionally inject behavior that either *halts* or *proceeds* with program operations. In this context, a trap is a function that encapsulates the behavior that must be executed when a specific program operation occurs.

In a monitored application, we distinguish between *base code* and *meta code*. Base code is essentially the target program code, while meta code refers to the code that captures security-relevant operations and enforces the security policies. The rest of this section explores the policies implementation followed by the implementation of Guardia's enforcement mechanism employing the host language's reflective capabilities and an alternative based on source-code instrumentation.

Policy implementation

In GUARDIA, a security policy is represented as an object shown in Listing 3.6. A policy object has an `enforce` function, which defines its semantics.

```
1 policy: {
2     ps: [],
3     enforce: function(op) {
4       return this.ps.every(function(p) {
5         return p.enforce(op);
6       });
7     }
8 },
```

Listing 3.6: GUARDIA's policy object implementation.

The `enforce` method (lines 3-7) takes a *program operation object* `op` as argument. A program operation object (i.e., `op`) holds the information of a particular program operation. For example, in a function call operation like `document.createElement('div')`, the `op` object will have the following fields:

```
1 {
2   type: 'funCall',
3   target: document.createElement,
4   ths: document,
5   args: ['div']
6 }
```

The argument `op` must have a `type` field representing the program operation that is being enforced (a function application in this example). The remainder of the properties are the values of the objects used in the operation: `target` is the function being applied, `ths` is the `this` object for the function call, and `args` is the array of arguments.

The `enforce` function applies the set of predicates (`ps`) configured for the policy. Note that all predicates must return `true` in order to uphold the policy. Values contained in `ps` are similar in structure, i.e., they must have an `enforce` function and optionally a `ps` property, allowing policy composition. In GUARDIA, policy predicates can be stateless, stateful, or higher-order.

Stateless predicate: A stateless predicate does not perform any side-effects during its enforcement. For example, Listing 3.7 shows an approximate implementation of the `onPropertyRead` policy predicate. The constant `pred` is the predicate object, which in this case is configured with the `object` and the `property` used during the enforcement. The `enforce` method takes a program operation object and implements the predicate semantics. In this example, the predicate only compares previously stored information (`object` and `property`) with information contained in the operation object `op`.

```
1  ...
2  onPropertyRead: function(object, property) {
3     const pred = {
4         object,
5         property,
6         enforce: function(op) {
7            return op.type === 'get' &&
8                   op.ths === this.object &&
9                   op.key === this.property;
10     }
11   };
12   ...
13 }
```

Listing 3.7: Example implementation of a stateless policy predicate.

Stateful predicate: A stateful predicate performs side-effects during its enforcement. The `moreThan` predicate is an example of a stateful predicate. Listing 3.8 shows the implementation of `moreThan`. The `times` property stores the number of times the execution of the predicate is valid. The `counter` property is updated whenever the predicate is enforced. As a convention, side-effects are moved out of the `enforce` method and are grouped in the `notify` method, with the former calling the latter at some point. For example, in line 6, the predicate tests whether the `counter` is equal to or exceeds the given `times`.

```
1 moreThan: function(times) {
2    const pred = {
3      times: times,
4      counter: 0,
5      enforce: function(op) {
6        var r = this.counter >= this.times;
7        if(r) this.notify(op);
```

```
8         return r;
9       },
10      notify: function(op) {
11        this.counter++;
12      }};
13  }
```

Listing 3.8: Example implementation of a stateful policy predicate.

Higher-order predicates: Higher-order predicates take one or more policies as arguments to specify the desired semantics. For example, the `or` predicate (see Listing 3.9) takes one or more policies to test whether the result of enforcing some of the given policies is true. Higher-order predicates are fundamental for composing policies, as shown in Listing 3.5.

```
1  or: function(...pols) {
2      const pred = {
3        ps: pols,
4        filter: function(op) {
5          return this.ps.some(function(p) {
6            return p.policy.enforce(op);
7          });
8        }
9      };
10 ...
11 }
```

Listing 3.9: Example implementation of a higher-order policy predicate.

3.3.1 Proxy-based Enforcement

In this section, we focus on GUARDIA's enforcement mechanism based on JavaScript's reflective capabilities using proxies [Ecm15]. We provide a brief introduction to JavaScript proxies before explaining how they are used to enforce GUARDIA policies.

```
1  let obj = { foo: "Hello World!"};
2  let handler = {
3    get: function(obj, field){
4       return 42;
5    }
6  }
7
```

```
8 let proxy = new Proxy(obj, handler);
9 console.log(proxy.foo); //prints 42
```

Listing 3.10: Proxy API usage example.

A JavaScript *proxy* is an object that acts as a wrapper of another JavaScript object. A proxy can intercept and change the semantics of operations on the wrapped object. A proxy object is created using the `Proxy(target, handler)` constructor. The `target` argument represents the object to be wrapped (i.e., the object being secured in our context). The `handler` argument is another JavaScript object that defines the semantics of the modified operations on the target object. For example, a handler object can change the semantics of getting a property value on a target object by providing an implementation of a `get` trap, as shown in Listing 3.10. In this example, the handler reimplements the semantics of the `get` operation for the object `obj`. Specifically, it returns `42` whenever a property is being read on the `proxy` object, as shown in line 4.

In the following, we describe the design of a reference monitoring mechanism using JavaScript proxies to enforce GUARDIA policies.

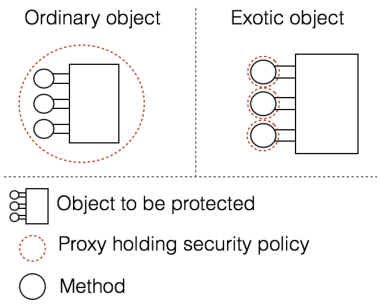


Figure 3.1: Proxy-based enforcement approach in GUARDIA.

A proxy-based monitor

For the enforcement mechanism to adhere to the *complete mediation* requirement, policies should be deployable on all types of JavaScript objects. However, browser host environments provide *exotic objects* such as `window`, `document`, `location`, etc. Exotic objects differ from ordinary objects in that they do not implement the default behavior for one

or more of the essential internal methods that must be supported by all objects [Ecm15]. Exotic objects add extra difficulties to an enforcement mechanism based on proxies, as they may require different monitoring strategies for policy enforcement. In what follows, we detail the enforcement using proxies on both ordinary and exotic JavaScript objects.

Enforcement for ordinary objects: A reference to an ordinary JavaScript object can be freely reassigned with a proxy that secures it. This is shown on the left-hand side of Figure 3.1. The enforcement mechanism must secure sensitive objects right after their creation to prevent problems with aliasing and therefore to ensure that attackers only have access to the sensitive object's secured version.

Listing 3.11 illustrates how object proxy handlers are implemented in GUARDIA. Of particular interest are the `get` and `set` properties, which reify the semantics of how object properties should be read and written, respectively.

```
1  let handler = {
2     get: function (trgt, prop, rcvr) {
3          if(policy.enforce({type: 'get', obj: trgt, key: prop})){
4             return Reflect.get(trgt, prop, rcvr);
5          }else{
6             throw new Error('Policy violation: [GET]!');
7          }
8     },
9     set: function(trgt, prop, value, rcvr){
10     ...
11    }
12 };
13
14 let target = new Proxy(target, handler);
```

Listing 3.11: GUARDIA proxy handler's implementation.

Whenever a property read occurs on a secured object, the proxy intercepts this and forwards the operation to the `get` method of its handler. Lines 2–8 specify the semantics of GUARDIA for verifying whether the property read is allowed. This verification is done by calling the `enforce` method. The argument for the `enforce` call is the program operation object. First, GUARDIA notifies the `policy` object, which holds the policy predicates for securing the target object. If the policy is violated, the handler throws an exception to prevent the actual read operation on the

underlying secured object (line 7). Otherwise, the read operation on the underlying object is performed (line 5). A similar approach is used for property write operations, which are intercepted by the `set` method on the handler.

Enforcement for exotic objects: Exotic objects pose a challenge to a reflective enforcement approach because they are read-only references according to the HTML5 standard [WHA17]. Developers can modify these objects by adding or deleting properties, but it is forbidden to change their references[1].

Instead of wrapping the entire object as with ordinary objects, it is necessary to wrap each *method* of the exotic object with a proxy enforcing the relevant security policies. This wrapping is shown on the right-hand side of Figure 3.1. This approach respects the exotic object's invariants, while still introducing the necessary checks on security-sensitive operations on these objects.

To illustrate this approach in a concrete example, we take Policy 2 (Listing 3.3), which prevents dynamic creation of `iframe` objects by disallowing the call expression `document.createElement('iframe')`. Instead of wrapping the entire `document` object, GUARDIA only wraps the `document.createElement` function object as shown in Listing 3.12. The handler has to intercept a function call and therefore implements an `apply` operation.

```
1  let handler = {
2     apply : function (target, thisArg, argumentsList){
3       //Check the security policies
4       return Reflect.apply(target, thisArg, argumentsList)}
5  };
6
7  Object.defineProperty(document, 'createElement', {
8      configurable: false,
9      writable: false,
10     value: new Proxy(document.createElement, handler)}
11 );
```

Listing 3.12: Function proxy handler's implementation.

[1] An exception to this rule is the `location` object that, when assigned with a location, causes the browser to navigate to that location.

Line 3 in Listing 3.12 is a placeholder for the enforcement code that verifies whether the function call is allowed. Checking the predicates in the relevant policy object uses a similar mechanism as the one used for ordinary objects. The method `Object.defineProperty` adds or modifies a property on an object. Lines 7–10 replace the original `document.createElement` function object with the wrapped one on `document`. The properties `configurable` and `writable` are set to `false` to prevent any subsequent modification of the `document.createElement` property.

GUARDIA's enforcement mechanism based on proxies is not fully complete because of the `location` object, an exotic DOM object that is non-configurable, including its methods. It is impossible to wrap the `location` object with proxies to intercept security relevant operations such as changing the location by invoking `location.assign(url)`.

3.3.2 Source Code Instrumentation-based Enforcement

An alternative to proxies that enables behavioral intercession is source-code instrumentation. Source code instrumentation rewrites the source code of an application by inserting additional code. In the context of application-level security, the goal of source code instrumentation is to insert traps around security-sensitive operations. In what follows, we describe the main trade-offs in the design of a reference monitor based on source code instrumentation for GUARDIA.

The target program interacts with the policy code (i.e., GUARDIA) by including the library providing the policy specification language and enforcement mechanism [2]. Similar to an enforcement approach based on proxies, including GUARDIA means that its fluent interface becomes available in the target application environment. However, program operations in the base program still must be forwarded to GUARDIA's enforcement. Forwarding these program operations is where source code instrumentation plays its role. Specifically, traps around the target program operations are part of a meta program and are inserted by a rewriter [3].

As an example of source code instrumentation-based enforcement, consider that the policy in Listing 3.3 has been included, and that the snippet

[2]Note that the base program may also perform additional initialization and configuration before executing any code.

[3]We use Aran as rewriter: `https://github.com/lachrist/aran`

below is part of the original base program:

```
1 document.createElement('div');
```

Listing 3.13: Base program example.

For enforcing the example policy, and in a similar way to proxy traps, traps added as instrumentation must forward to GUARDIA's enforcement mechanism that deals with function call operations. This is illustrated below in an instrumented version of the base program code snippet:

```
1 META.apply(document.createElement, document, ['div']);
```

Listing 3.14: Base program example.

The `META` object is a handler object that defines an `apply` trap for trapping function calls. This trap is invoked whenever a function call is performed in the base program. The trap's body, shown in Listing 3.15, provides the meta behavior of calling a function in the base program.

```
1  const META = {
2      apply: function(target, ths, args){
3         if(GG.notify({ type: 'funCall', target, ths, args})){
4           return Reflect.apply(target, ths, args);
5         }else{
6           throw new Error('Policy violation detected!');
7         }
8      },
9      get: function(obj, prop, rcvr){
10       ...
11     }
12     ...
13 }
```

Listing 3.15: Example implementation of the META handler object.

First, the information related to the operation is collected (i.e., target function, arguments, and the `this` pseudo variable). At line 3, GUARDIA's enforcement is notified with the operation's information to be enforced against the deployed policies. If the enforcement returns `true`, it means that the operation being executed does not violate any of the stated policies, and the "original" function application is performed. Otherwise, an error is thrown, indicating that the operation violates some security policy.

3.4 Evaluation

This section reports on three kinds of experiments that we performed to evaluate GUARDIA concerning the design decisions detailed in Section 3.1.1. These experiments use the enforcement mechanism based on proxies described in Section 3.3. The goal of this evaluation is to assess GUARDIA in three different aspects: expressiveness, applicability, and performance.

In a first experiment, we expressed 13 different security policies in GUARDIA extracted from the literature, including approaches listed in Table 3.1. Specifically, in Section 3.4.1 we compare the expressivity of GUARDIA with the related approaches. Not all surveyed approaches (e.g., MYWEBGUARD [PPA+20]) target developers as end-users of the approach, or fit into our attacker model [MSR+19]. Other approaches, such as OBJECTVIEWS [MFM10], focus on policies that mediate the sharing of information between frames, while others do not contain policy specification examples [AVAB+12]. Therefore, this evaluation will consider a reduced set of approaches related to the dynamic enforcement of AC policies described in Section 2.4.1.2.

A second experiment consisted of applying GUARDIA to three types of programs: synthetic benchmarks, three experimental web applications, and ten real-world web applications. Finally, Section 3.4.3 describes a third experiment to evaluate the performance implications of GUARDIA on both synthetic benchmarks and the experimental applications.

3.4.1 Expressivity Compared to Related Work

This section evaluates the expressiveness of GUARDIA's DSL by expressing 13 policies found in related work [HV05, YCIS07, PSC09, ML10]. Table 3.3 gives an overview of these policies and their origin.

Table 3.3 extends the table presented in [Bie13] with the type of attack that each policy aims to prevent. In contrast to the original table, we consider only 11 distinct policies (denoted as Policy 1–11) because several policies could be combined into a single policy. Table 3.3 shows that 10 out of the 11 policies analyzed in related work can be expressed in GUARDIA. For each policy, we compared the specification in GUARDIA with the specification in related work.

We report on this comparison for 4 out of the 11 policies below. We

Attack type	Security policy	HV [HV05]	Yu et al. [YCIS07]	Phung et al. [PSC09]	ML [ML10]	GUARDIA
Forgery	Limited number of popup windows opened (Policy 3)	✓	✓	✓	✓	✓
Forgery	No popup windows without location and status bar (Policy 4)	✻	✻	✓	✻	✓
Resource abuse	Prevent abuse of resources like modal dialogues (Policy 5)	✓	✓	✓	✓	✓
Restoring built-ins from frames	Disallow dynamic iframe creation (Policy 2)	✻	✻	✓	✓	✓
Information leakage	Disable page redirects after `document.cookie` read (Policy 8)	✓	✓	✓	✓	✓
Information leakage	Only redirect to whitelisted URLs (Policy 12)	✻	✻	✓	✓	✓
Information leakage	Restrict `XMLHttpRequest` to secure connections and whitelist URLs (Policy 11)	✻	✻	✻	✓	✓
Information leakage	Disallow setting of `src` property of dynamic images (Policy 13)	✻	✻	✓	✻	✓
Impersonation	`XMLHttpRequest` is restricted to HTTPS connections (Policy 11)	✻	✻	✻	✓	✓
Impersonation / Information leakage	Disallow open and send methods of XHR object (Policy 6)	✻	✻	✓	✓	
Man in the middle	`postMessage` can only send to the origins in a whitelist (Policy 9)	✻	✻	✻	✓	✓
Run arbitrary code	Disallow `string` arguments to `setInterval` & `setTimeout` (Policy 10)	✻	✻	✻	✓	✓
Information Leakage	Disable geoposition API (Policy 7)	✻	✻	✻	✻	✓

Table 3.3: Comparison of approaches in security policies. Policy numbers 1–11 refer to the policies discussed in Sections 3.2 and 3.4.1 and appendix A. A (✓) means that the approach implements the policy in the paper, (✻) that the policy is not described in the paper but can be expressed with the approach, and (✗) that the policy cannot be expressed with the approach.

compare a GUARDIA policy specification with a different approach selected from Table 3.3 for each of the 4 comparisons described in this section. Our intention with this comparison is to give a flavor of the limitations and advantages of using GUARDIA for the specification of AC policies in a declarative manner. For completeness, Appendix A includes the implementation of the seven remaining policies in GUARDIA listed in Table 3.3.

ConScript

```
around(document.createElement, function (c : K, tag : U) {
  let elt : U = uCall(document, c, tag);
  if (elt.nodeName == "IFRAME") throw 'err'; else return elt;
});
```

Listing 3.16: (Policy 2) Prevent dynamic creation of `iframe` in ConScript (extracted from [ML10]).

Listing 3.3 (Section 3.2) introduced Policy 2 in GUARDIA to prevent dynamic creation of `iframe` elements. We compare this policy to the equivalent CONSCRIPT policy specification [ML10] given in Listing 3.16. As mentioned in Chapter 2, CONSCRIPT policy specification follows an aspect-oriented approach in which a pointcut is declared to intercept relevant calls, in this case to the `document.createElement` function. CONSCRIPT forces programmers to write code for *both* policy specification *and* enforcement. As a result, programmers have to manually ensure the completeness, transparency, and tamper-proofness of the enforcement mechanism. In contrast, GUARDIA developers only have to declare security policies, without programming their enforcement.

CONSCRIPT relies on VM modifications and can only be applied to Internet Explorer 8, while GUARDIA runs in any browser that implements the ECMAScript 2015 standard.

CoreScript

We compare Policy 5 specified in CORESCRIPT (Figure 2.3 in Page 31) to the specification in GUARDIA (Listing 3.17). This policy prevents resource abuse by denying the creation of `alert` windows.

```
GG.onCall(window.alert).deny();
```

Listing 3.17: Policy 5: Prevent showing alert dialogs.

Originally, CoreScript security policies were described using a formalism based on edit automata [YCIS07], but a follow-up paper presents an approach in which developers can also encode policies by writing XML files [KYC+08]. The XML in Figure 2.3) specifies Policy 5. In CoreScript, developers identify the object, property, or method to which code rewriting must be applied. Instead of forcing the developer to think in terms of state and transitions, which may not be common knowledge among developers and security engineers, Guardia uses a declarative and arguably more descriptive approach for specifying security policies.

In contrast to Guardia, CoreScript forces developers to know how to write the *replacement action* code. A replacement action should not only perform the "original" behavior it is replacing, but also the actual enforcement of the policy. In our view, it is less error-prone to specify a policy that prevents specific behavior than to manually write code that should behave similarly to the replaced code, while at the same time taking care of the enforcement and transparency concerning the normal program execution. In Guardia, developers are not burdened with writing enforcement code. The semantics of the operations and their properties, such as transparency and tamper-proofness (with the limitations that we discuss later in Chapter 5), are provided by the underlying enforcement used by Guardia.

3.4.1.1 Hallaraker and Vigna's Auditing System for JavaScript

Listing 3.18 shows Policy 3 specified in Hallaraker and Vigna's auditing system for JavaScript (HVAS) [HV05]. This policy limits the number of popups that a window can open. The equivalent Guardia policy specification was given in Listing 3.4 on page 46. Policies in HVAS are specified as a state transition model. While both specifications are expressed in more or less the same amount of code, the number of allowed popup windows is hardwired in the HVAS specification. Additionally, in HVAS, the policy designer has to write as many `if` statements as the number of popups that are allowed, which hampers code maintainability and reusability. In contrast, Guardia's specification parametrizes the maximum allowed number of popup windows as an argument of the policy. Furthermore, the Guardia specification makes it straightforward to add and combine additional policy predicates for imposing additional restrictions as part of the policy.

```
if((event.method.name==open) && (event.method.object=="window")){
  if(stateW4.includes(event.host)){
   log("Script has opened 5 windows. Possibly a malicious script!")
  }
  else if(stateW3.includes(event.host)){
    stateW3.delete(event.host);
    stateW4.add(event.host);
  }
  else if(stateW2.includes(event.host)){
    stateW2.delete(event.host);
    stateW3.add(event.host);
  }
  else if(stateW1.includes(event.host)){
    stateW1.delete(event.host);
    stateW2.add(event.host);
  }
  else{
    stateW1.add(event.host);
  }
}
```

Listing 3.18: (Policy 3) Limit number of popup windows in HVAS (extracted from [HV05]).

3.4.1.2 Lightweight Self-Protecting JavaScript

We compare Lightweight Self-Protecting JavaScript (LWSPJS) [PSC09] to GUARDIA by means of Policy 6, which prevents impersonation attacks using the `XMLHttpRequest` (XHR) object by disallowing calls to its `send` method based on the arguments of `open`.

```

var XMLHttpRequestURL = null;
enforcePolicy ({ target : XMLHttpRequest , method: 'open' },
     function(invocation){
       XMLHttpRequestURL = stringOf(invocation ,1);
       return invocation.proceed();
     });

enforcePolicy({ target : XMLHttpRequest , method: 'send'},
    function(invocation){
    XMLHttpRequestPolicy(invocation);
});

```

```
14 var XMLHttpRequestPolicy = function(invocation){
15      //allow the transaction if the URI is in the whitelist
16      if (AllowedURL(XMLHttpRequestURL))
17         return invocation.proceed () ;
18      policylog('XMLHttpRequest is suppressed:'+
19       'potential impersonation attacks') ;
20 }
```

Listing 3.19: (Policy 6) Prevention of impersonation attacks in LWSPJS (extracted from [PSC09]).

Listing 3.19 shows the specification of Policy 6 in LWSPJS, in which the URL passed to the `open` method is forced to be a `String`. The policy deployed on the `send` method verifies that the URL string is contained in the whitelist of URLs. Developers have to manually specify the enforcement code (lines 3–7, 9 –11, and 13–18), and consequently, most of the code in Listing 3.19 is dedicated to enforcement of the policy.

Despite the fact that Guardia's enforcement mechanism has the appropriate underlying infrastructure to express Policy 6, Guardia's DSL does not allow to express it. What it missing is a stateful predicate enabling a shared state accessible from different policies.

Listing 3.20 shows an alternative implementation for Policy 6 in GUARDIA. In this variant of the policy, the execution is allowed to proceed if the `XMLHttpRequest` instance calls its `open` method with a whitelisted URL.

```
1 const noWhiteListedURL = GG.arg(GG.notIn, GG.targ(0, String), URLsList)
2 GG.onCall(XMLHttpRequest.prototype, "open")
3     .with(noWhiteListedURL)
4     .deny();
```

Listing 3.20: Policy 6: Prevention of impersonation attacks in GUARDIA.

The equivalent policy expressed in the LWSPJS approach is shown in Listing 3.21. Similar to CONSCRIPT, users of LWSPJS express policies in an aspect-oriented manner. Therefore, these users need to ensure the transparency, completeness, and integrity of the policy enforcement code themselves.

```
1 enforcePolicy ({ target : XMLHttpRequest , method: 'open' },
2      function(invocation){
3        var XMLHttpRequestURL = stringOf(invocation ,1);
4        //allow the transaction if the URI is in the whitelist
```

```
5       if (AllowedURL(XMLHttpRequestURL))
6        return invocation.proceed () ;
7       policylog('XMLHttpRequest is suppressed: potential impersonation
      attacks') ;
8      });
```

Listing 3.21: (Simplified version of Policy 6) Prevention of impersonation attacks in LWSPJS.

3.4.2 Applicability

This section describes three experiments designed to assess the practical applicability of GUARDIA. To this end, we designed three experiments that gradually increase the complexity of the program to which GUARDIA was applied. The experiments use small synthetic benchmarks, experimental web applications, and real-world web sites.

Correctness on synthetic benchmarks

In the first experiment, a suite of synthetic benchmarks was used that also served to drive forward the implementation of GUARDIA itself by testing new functionality and avoiding regressions. Each program in the set of synthetic benchmarks is implemented so that it is straightforward to determine whether a vulnerability (or some other kind of behavior) is present or absent. We then developed GUARDIA policies targeting these benchmarks, and verified for each synthetic benchmark whether policy enforcement results agreed with the expectations.

The approaches in Table 3.3 are strictly more expressive than GUARDIA, because the policy specification and enforcement in these approaches is imperatively written in JavaScript, which allow developers to use any language feature for this purpose. However, we show that those 13 policies can be expressed in a declarative manner, thereby freeing developers from writing enforcement code.

Practicality and transparency

In the second experiment, GUARDIA was tested on three experimental applications: Juice Shop, NodeGoat, and SoundRedux. Juice Shop and NodeGoat are part of the OWASP, which serves as a learning resource for

application security. By design, both applications have security holes that developers and penetration testers can inspect and exploit to learn how to protect their applications. SoundRedux provides a fully functional application in a more complex scenario. All three applications use modern JavaScript libraries and frameworks. Therefore, these applications provide a good notion of how practical it is to secure them with GUARDIA. It also enables us to assess the transparency of GUARDIA's enforcement mechanism based on proxies in real-world scenarios that involve state-of-the-art libraries and frameworks. The remainder of this section describes the results of securing each experimental application with GUARDIA.

OWASP Juice Shop: We use JUICE SHOP that was introduced in Chapter 2 during the applicability and transparency evaluation. As mentioned before, GUARDIA is implemented as a JavaScript library and can therefore be deployed in any standard ECMAScript 5 (or more recent) runtime environment, including web contexts, using standard mechanisms. Juice Shop is an Single-Page Application (SPA), and therefore GUARDIA must only be included once in this application.

We applied GUARDIA's implementation of the policies described in Table 3.3 to Juice Shop to protect the application from Reflected Cross-Site Scripting attacks [VNJ+07, Pan14]. We found that GUARDIA is able to enforce all policies except Policy 12, which targets the `location` object. As explained in Section 3.3.1, the `location` object imposes strong invariants that make it impossible to protect it without relying on VM modification or source code instrumentation.

OWASP NodeGoat: NodeGoat [4] is a vulnerable web application that manages employee retirement savings. The application offers typical functionalities such as user login and registration. Registered users have a private dashboard page in which they can modify their preferences and manage their benefits.

NodeGoat has similar security vulnerabilities to those found in JUICE SHOP. It is developed using current technologies and includes libraries such as JQuery and Twitter Bootstrap. We therefore applied the same set of security policies to NodeGoat as to JUICE SHOP, and obtained the same results in terms of security.

[4] https://github.com/OWASP/NodeGoat

Vue SoundCloud: Vue SoundCloud[5] is a client-side web application that serves as an interface to the SoundCloud[6] application, which enables the exploration of the SoundCloud music database. In contrast to NodeGoat and Juice Shop, Vue SoundCloud is a fully functional web application that is not deliberately made insecure.

Vue SoundCloud is developed using popular software libraries such as Vue[7]. To deploy GUARDIA in Vue SoundCloud, the application's index page was modified by adding a `script` tag for including GUARDIA itself. Furthermore, a second `script` tag was added pointing to our set of security policies.

In contrast to the previous two applications, we did not perform any attack on Vue SoundCloud through its interface because the application does not have any apparent security breaches, and it is not the aim of our work to *discover* security holes. Instead, by running code that attempts to bypass the deployed policies in the browser's developer console, we found that the deployed policies were fully and correctly enforced by GUARDIA.

Transparency in web applications

This section describes an experiment in which the set of GUARDIA policies appearing in Table 3.3 is applied to 10 real-world web applications (Table 3.4). The motivation of this experiment is to verify whether these web sites continue to perform as expected in the presence of GUARDIA. This allows us to assess the transparency of GUARDIA.

The selection of the 10 applications is based on the Alexa Top 500 ranking[8]. The experiment sites were selected based on their purpose (i.e., *news, shopping, entertainment, social network, etc.*). Although the selected web sites vary in their intended use, all involve substantial amounts of complex JavaScript code that runs in the browser.

The first step in the experiment was deploying the security policies, which was done using the Burp Suite[9]. Burp Suite enables the interception of HTML responses from web sites and the subsequent injection of GUARDIA policies to modify these responses. As a result, when an ap-

[5]https://github.com/soroushchehresa/vue-soundcloud.git
[6]https://soundcloud.com/
[7]https://facebook.github.io/react/
[8]https://www.alexa.com/topsites
[9]https://portswigger.net/

plication's page was rendered in the browser, it contained the deployed policies. Because the applications listed in Table 3.4 do not have evident security holes, the policies of Table 3.3 were tested by writing code in the browser's console that attempted to bypass these policies.

The result of the experiment was that all sites, except *YouTube*, continued to function as designed in the presence of GUARDIA. Closer inspection revealed that *YouTube* attempts to override properties secured and sealed by GUARDIA policies. The *Vimeo*, *eBay*, *Reddit* and *BBC* web sites also did not render correctly at first. Inspecting the produced error trace indicated that these applications were attempting to create `iframe` elements dynamically and that GUARDIA was preventing this behavior. These web sites executed normally after removing Policy 2, which disallows the dynamic creation of `iframe` elements.

Application	Type	Deployed
google.com	Search Engine	✓
baidu.com	Search Engine	✓
bbc.com	News Site	✓
reddit.com	News Site	✓
youtube.com	Entertainment	
vimeo.com	Entertainment	✓
amazon.com	Online Shopping	✓
taobao.com	Online Shopping	✓
ebay.com	Online Shopping	✓
linkedin.com	Social Network	✓

Table 3.4: Real-world applications tested with GUARDIA.

3.4.3 Performance

To assess GUARDIA's performance impact, we measured the runtime overhead of deploying GUARDIA policies in the three types of benchmark programs we experiment with: small synthetic benchmarks, experimental web applications, and real-world web sites. These experiments were performed on a MacBook Pro equipped with a 2.3 GHz Intel Core i9 processor and 16

GB of DDR4 RAM. All experiments were executed using Google Chrome version 81.0.4044.

Performance on synthetic benchmarks

We developed a synthetic benchmark using policies with three levels of complexity to estimate the slowdown introduced by GUARDIA in the performance of 4 built-in functions. Table 3.5 shows the description of the complexity of each policy, together with the built-in functions used in the experiment. A policy with a single predicate only contains a point-cut predicate such as `GG.onCall(document.write)`. A policy with a combined predicate is composed by a point-cut predicate and a higher-order predicate such as `GG.onCall(document.write).with(...)` Finally, a policy with 10 combined predicates is composed by a point-cut predicate and a higher-order predicate composed of 9 simple predicates.

Methodology Each experiment measured the execution time of a baseline program and 3 secured programs. A baseline consist of calls to one of the functions in Table 3.5. A secured program consists of one policy specification using one of the complexities described, and calls to the protected function.

To measure the performance overhead, we ran 100 fresh process executions and 1000 in-process iterations for each combination of <policy, function> from Table 3.5. In each in-process iteration, we recorded the execution time of the protected function. All measurements were done with just-in-time (JIT) compilation enabled. Therefore, to measure the execution time of the protected functions in a steady state of the program, we dropped the first 200 in-process iteration results.

Results After removing the first 200 in-process iteration results, our data set consisted of 80000 measurements of each <policy, function> combination. Table 3.5 shows the performance overhead introduced by the policies. The slowdown factor ranges from 1.01x to 3.5x with respect to the baseline program's execution time.

In addition to the slowdown factor, we computed the average (mean) execution time and the confidence intervals for each <policy, function> combination from Table 3.5. These statistics provide us with an intuition of whether the overhead added by the policies is statistically significant

Policy	`document.createElement()`	`document.write()`	`window.setTimeout()`	`window.setInterval()`
Simple Predicate	1.25x	1.20x	1.19x	1.01x
Combined Predicate	1.76x	1.34x	1.23x	1.14x
10 Combined Predicates	3.15x	1.37x	1.32x	1.31x

Table 3.5: Overhead of GUARDIA on synthetic benchmarks.

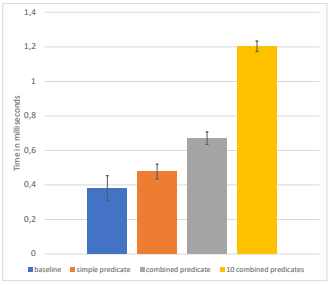
with respect to the execution time of the baseline programs. Figures 3.2a to 3.2d show the average execution time of each benchmark program. The error bars in the figures represent a 95% confidence interval.

Confidence intervals shown in Figure 3.2 suggest that there is a significant statistical difference on the performance overhead introduced by policies with combined predicates with respect to the performance of the original application. However, for policies with a single predicate, the confidence interval suggests that this difference may be due to random effects.

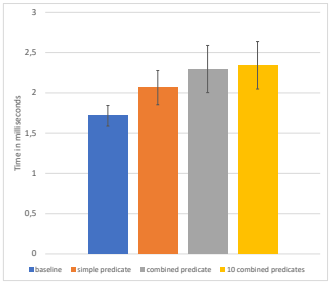
Performance on experimental applications

We measured the performance impact of using GUARDIA for deploying and enforcing Policy 2 and Policy 10 in Juice Shop, NodeGoat, and SoundRedux. Other policies are not triggered by these applications (e.g., Policies 6, 7, 11), or would be difficult to time because they require user interaction to open or close popup windows (e.g., Policy 3 and Policy 5).

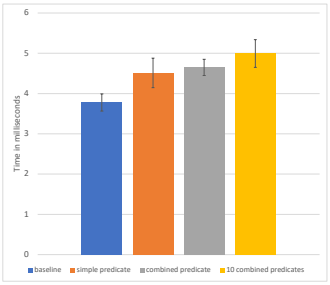
Methodology Each application was loaded 100 times to calculate the average load time of the *home* page of each application. The time spent by the browser to load the home page was measured by computing time differences using JavaScript's `performance` API. Table 3.6 relates the lines of JavaScript Code (LOC), the page load time without and with GUARDIA, and the overhead added by GUARDIA. *Policy checks* represents the number of calls to the policy enforcement code of Policy 2 and Policy 10 during



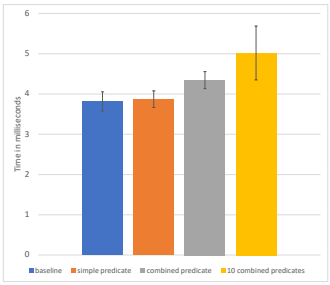
(a) Run-time overhead of policies for `document.createElement`.



(b) Run-time overhead of policies for `document.write`.



(c) Run-time overhead of policies for `setTimeout`.



(d) Run-time overhead of policies for `setInterval`.

Figure 3.2: Comparing the performance overhead introduced by policies using *one predicate, a combined predicate* and `10 predicates` to the execution of the methods `document.createElement`, `document.write`, `setTimeout` and `setInterval`. Vertical bars represent the mean of 100000 executions of each configuration of policy and method. The error bars indicate the 95% confidence interval.

a page load.

Results Figure 3.3 shows the average time taken by the browser for loading the applications. The bars in blue represent the load time of the original application without Policy 2 and Policy 10 being deployed in the application. Orange bars correspond to the load time of the applications with Policy 2 and Policy 10 deployed in the application. In these cases, the confidence interval and the mean suggest that there is no statistical difference between loading the application with and without Policy 2 and Policy 10. From the results in Table 3.6 we also can conclude that

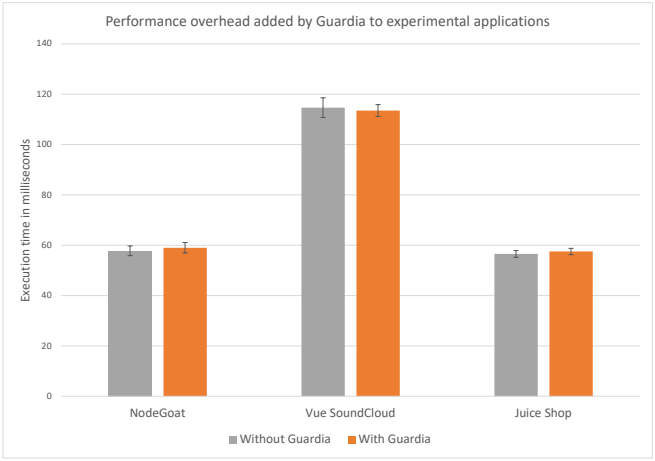


Figure 3.3: Run-time overhead introduced by the deployment of Policy 2 and Policy 10 in the experimental applications. The bars show the average time of opening each application 100 times in Google Chrome. Error bars indicate the 95% confidence intervals.

there is negligible overhead when enforcing Policy 2 and Policy 10 during each page load. Although the policy checks are triggered several times in each application, this does not significantly impact those applications' performance.

Application	Description	LOC	Load time(ms)	Load time with GUARDIA (ms)	Overhead	Policy checks
Juice Shop	Online Shop	22649	56.55	57.53	1.01x	41
NodeGoat	Social Security App	3730	57.78	59	1.02x	23
Vue Sound-Cloud	SoundCloud Client	1969	114.67	113.48	0.99x	14

Table 3.6: Experimental applications tested with GUARDIA.

Performance on real-world applications

We attempted to measure the performance overhead introduced by GUARDIA for the applications listed in Table 3.4. Measuring this overhead required performing a *man-in-the-middle* manipulation of the HTML re-

sponse from the applications' servers. To this end, we used *mitmproxy* [10] to cache the responses of applications. Next, we recorded page load times with and without GUARDIA policies. We found that the performance impact introduced by GUARDIA is negligible compared to the variance introduced by the number of resources (images, scripts, styles, etc.) loaded by these applications. The fluctuations in the measurements did not allow us to measure the performance overhead added by the policies in a useful manner.

3.5 Discussion

In this dissertation, it is assumed that the meta code, implemented either by proxy handlers or `META` object traps, does not influence the base program's behavior in any way except for potentially halting program execution by throwing an exception. The implementation of meta code should not, for example, change the state of the base program by changing the value of variables or object fields.

Transparency of the proxy-based enforcement: To test the transparency of the proxy-based enforcement of GUARDIA, we conducted experiments to investigate how proxies behave in real-world applications running in different browsers when using popular libraries such as JQuery. The first experiments revealed some issues. In particular, JQuery presented errors when methods on the `window` or `document` objects were wrapped. Further investigation showed that JQuery uses the `toString` function of methods on host objects to assert whether the containing host objects are native or not. However, this check fails when proxies wrap these functions. GUARDIA overcomes this problem by binding the wrapped `toString` function to the target object instance instead of the proxy.

Our experiments also revealed that proxies do not behave transparently on DOM `Node` objects. The `node.appendChild(child)` function, for example, checks whether the argument value is of type `Node`. When this method receives a proxy, the type check fails and the node is not added to the tree. To overcome this problem, GUARDIA treats `Node` instances

[10] https://mitmproxy.org/

as exotic objects: instead of wrapping the entire object, every function on the object is wrapped.

Transparency of the source code instrumentation-based enforcement: The code instrumentation-based enforcement does not have the transparency problems that arise from wrapping objects with proxies, as discussed above. However, a naive code instrumentation will not be fully transparent to the base application execution under certain circumstances. Perhaps the most obvious way of losing transparency is through reflective operations in the base application. In an instrumented application, the `META` handler object lives in the same environment as the rest of the base application objects. This environment pollution with meta code can be perceived in reflective operations such as `Object.keys(window)`. Making JavaScript's reflective operations aware of the instrumentation can mitigate this problem. It implies the redefinition of those reflective operations.

Correctness: Weaving GUARDIA's enforcement mechanism (using either proxies or code instrumentation) yields a `correct` inlined-reference monitor for the access control of policies programmers can express using GUARDIA. However, formally proving correctness boils down to proving that the behavior of the injected JavaScript proxies and the result of the rewriting process performed by the source code instrumentation platform (ARAN in our case) are correct, which is beyond the scope of this work. Instead, GUARDIA's enforcement mechanism is designed to be small, so that a manual review of it remains tractable.

3.6 Conclusion

This chapter presented GUARDIA, an internal DSL for the specification and enforcement of access control security policies. GUARDIA enables the specification of composable security policies that combine the flexibility of imperative specification languages with the ease of development provided by more declarative solutions.

GUARDIA's decoupled implementation allows the evolution of the language and its enforcement individually. This flexibility enabled us to develop two enforcement mechanisms based on meta-programming. First,

GUARDIA can use JavaScript proxies that act as wrappers that intercept operations on security-sensitive objects and functions. Alternatively, applications can be instrumented using source code instrumentation to insert traps that mediate the execution of sensitive program operations.

To evaluate GUARDIA's declarative policy specification language, we implemented 13 access control security policies from related work and found that the specification language can express most of them.

We also evaluated the applicability and performance impact of GUARDIA's dynamic enforcement mechanism on three experimental applications and ten real-world web sites. Our experiments indicate that the reflection-based enforcement mechanism of GUARDIA is correct and transparent. The overhead introduced by deployed GUARDIA policies is negligible compared to the application performance during page loading. All of this was achieved without changing the VM, which ensures portability of GUARDIA across all ECMAScript-compatible browser implementations.

Chapter 4

Practical and Permissive Dynamic IFC

In this chapter, we focus on dynamic information flow control (IFC) analysis for client-side web applications. As we already discussed in Chapter 2, implementing a runtime monitoring mechanism for IFC using source code instrumentation promotes portability. However, it is challenging for such a monitor to enforce the desired IFC policy while supporting the complex language features such as interactions with the DOM and web API, `eval`, prototype inheritance and implicit flows (cf. Section 2.3.2.1).

This chapter presents GIFC, a dynamic IFC mechanism for client-side web applications which exhibits the following properties:

Portable: GIFC does not require modifications to the underlying JavaScript interpreter or rely on a specific JavaScript runtime environment, but instead works with any ECMAScript 5 compliant JavaScript interpreter.

Support for dynamic code evaluation: GIFC handles dynamic code evaluation online. This is possible because we leverage on an instrumentation platform running alongside the instrumented program.

Support for libraries: GIFC features an API function model mechanism that enables information tracking through APIs calls. To handle external function calls we took inspiration from the specification of function models described in [HSPS17].

Permissive: The monitor of GIFC is based on the *permissive upgrade* (PU) technique of Austin and Flanagan [AF10] (cf. Section 2.3.2.3). GIFC's monitor further extends the PU permissiveness by means of *automatic upgrade annotations*. The annotations dynamically upgrade the security label of write targets in all branches of a statement conditioned by a security sensitive value.

Performant: The monitor of GIFC is inlined in the source code, so that the instrumented program (including the monitor) can still benefit from the optimizations offered by the underlying JavaScript runtime.

To the best of our knowledge, the combination of these properties is novel. Therefore, we consider that GIFC is suited to perform practical IFC for modern client-side web applications.

The remainder of this chapter is structured as follows. Section 4.1 discusses important language and technological challenges that need to be addressed to build a portable IFC enforcement mechanism. Next, Section 4.2 describes the design aspects of GIFC's implementation. Finally, Section 4.3 evaluates our dynamic IFC enforcement mechanism.

4.1 Challenges for Portable and Permissive IFC in Web Applications

This section describes the overall challenges faced when implementing an IFC monitoring mechanism. Specifically, it describes challenges related to JavaScript and the browsing environment from the perspective of using source code instrumentation for implementing the monitor.

4.1.1 Implicit Coercions

Implicit coercion happens when the interpreter converts a given value to a value of a different type. In JavaScript, implicit coercions are performed within the implementation of different operators and built-in functions to convert a given value to a value of one of the primitive types (e.g., string, number, etc.). The programmer is allowed to partially hook into the coercion applied to a specific object by adding `toString` or `valueOf` methods to the object. However, the specific semantics of the coercion is implemented by the interpreter in the abstract operation `[toPrimitive]`,

which can choose between `toString` or `valueOf` in order to perform the coercion. Note also that `toString` and `valueOf` can perform arbitrary side-effects.

The problem that implicit coercions pose to a source code instrumentation-based IFC mechanism is that it cannot instrument `[toPrimitve]` since it is an abstract operation. Therefore, the monitor cannot track precisely the information flows originated during implicit coercions.

To illustrate this problem, consider Listing 4.1 as an example. The example, defines the object `obj`, which implements the `toString` and `valueOf` methods. Again, these methods can selectively be called by the interpreter. In the example, `obj L` is added to `x` (`L`), and the result is assigned to `y`. Even though both `obj` and `x` are labelled `L` the label assigned to `y` is determined by the chose made by `[toPrimitive]` which cannot be instrumented.

```
1 let obj = { // has label L
2     valueOf(){return public} // public has label L
3     toString(){return secret} // secret has label H
4  }
5  let y = obj + x; //x has label L
```

Listing 4.1: Example of information flows originating from implicit coercions.

4.1.2 External Libraries

JavaScript web applications do not live in isolation in the browser, but they instead interact with the rest of the system to do something useful like processing user input/output, sending data over the network, displaying a web form, etc. All these interactions performed by the application are done through calls to web APIs, implemented by the browser in other languages (e.g., C++). We refer to all APIs that are not implemented in JavaScript but provided as APIs as *external libraries*.

Listing 4.2 shows an example of an external library function call, `Math.pow`. When executing that code with a monitor to track information flow, the application runs with augmented semantics, e.g., values are labelled and monitored. Since external libraries do not understand the values used in the augmented semantics, the monitor should not pass la-

bel information to `Math.pow`. However, after the external library call, the monitor cannot know which label assign to the result (i.e. `x`'s value).

```
1 let y = 13; //H
2 let x = Math.pow(y,2);
```

Listing 4.2: Example of an external library call.

A conservative approach to solve this problem is to label the result with the most sensitive label of the library call's values. However, this is considered to be conservative because the return value may not depend on any sensitive data during the library call [HBBS14]. To illustrate the implications of such an approach, consider the code in Listing 4.3 as a conservative external library implementation. Assume that parameters `x` and `y` are labeled as secret and public, respectively. The labelling of the return value as sensitive depends on the `choice` parameter. Therefore, tagging the value assigned to `z` at line 8 as sensitive is imprecise and may hamper the permissiveness of the monitor.

```
1 fun choice(choice, x, y){
2   if(choice > 10){
3      return x;
4   }else {
5      return y;
6   }
7 }
8 z = choice(9, secret, public);
```

Listing 4.3: Example of a conservative approach for external library calls.

4.1.3 Document Object Model

The Document Object Model (DOM) is the main web API offering page rendering and input/output facilities [LH05]. DOM elements are exposed to JavaScript as objects; however, they are not *ordinary* objects. Their semantics are implemented in the browser's host language, which may vary between browser vendors. Properties of DOM elements are implemented as pairs of getter/setter functions provided by the browser; thus, their implementation cannot be instrumented. As a result, DOM elements should be treated as external libraries.

DOM elements form a tree to compose the HTML document representing the page shown to users. From a security perspective, the `document`

object can be used by attackers as a storage channel of information. Monitoring flows from and to the DOM – or other web APIs – is crucial as attackers could store secret information as a DOM element or as part of their properties or attributes to then later retrieve them and leak that information.

To illustrate this problem, consider the example of Listing 4.4, where an attacker managed to inject malicious code as part of the application login page. Lines 1-4 add a *keypress* event listener to the input *password* element. Whenever the user types in the component, the element's value is added to the `body.classList` (lines 2 and 3). As a result, the password value is stored within the document tree. Lines 6 to 9 register a *click* event listener into the page's login button. When the user clicks the button to access the application with its credentials, the code creates an image element dynamically and sets its `src` attribute pointing to the attacker's server. Additionally, it sets the value of `body.className` as part of that URL leaking the user's password.

```
1 document.querySelector("#user-password").onkeypress = function(){
2    let x = document.querySelector("#user-password").value;
3    document.body.classList.add(x);
4 }
5
6 document.querySelector("#login-btn").onclick = function(){
7       let img = document.createElement("img");
8       img.src = "http://evil.com?pass="+ document.body.className;
9 }
```

Listing 4.4: Example of using DOM tree as storage channel.

The example shows some challenges for tracking information flow precisely. As mentioned before, the DOM element's methods are external to the language. Therefore, they need to be treated as *external libraries*. Moreover, some of these function calls (e.g. `img.src = ...` at line 8) perform side effects within the document tree, which need to be precisely modelled. Finally, side effects on some DOM elements' properties can be reflected elsewhere in the tree; this also needs to be modelled. For example, the call to `classList.add(x)` at line 3, adds `x`'s value to an internal `TokenList`. This list can be accessed in several ways like `body.className`, `body.classList`, and `body.getAttribute("class")`.

Although the implementation details of DOM and web APIs may differ between browser vendors, they mostly adhere to an HTML stan-

dard [HTM], whose description can be used to model the behavior of these libraries.

4.1.4 Dynamic Code Evaluation

Functions like `eval`, `setTimeout` and the `Function` constructor, allow the execution of arbitrary code represented by a string value. For example, calling `eval("z = secret ? x : z ;")` creates an implicit flow from `secret` to `z`. For a source code instrumentation IFC monitor to support `eval()` with minimum performance implications, the instrumentation mechanism must run alongside the instrumented program.

4.1.5 Permissiveness

As explained in Section 2.3.2.3, a PU strategy is more permissive than NSU. However, PU still may stop the execution of too many valid programs (i.e., a relatively high number of false positives). For example, consider the variable `secret` (`true`) from Listing 4.5 as a sensitive value. Also consider that an attacker can only observe information by means of `print` calls as in line 9. The `if` statement at line one elevates the security context of the PC to `H` as consequence of evaluating `secret`. As a result, the assignment to `x` at line 2 is executed under a `H` context. This assignment is prevented under NSU. Under PU, the assignment to `x` is valid and results in `x` being labeled as *partially* (`P`) leaked. However, the PU halts the execution at line 6 when using `x`, a partially leaked variable, in a conditional statement. As shown in the example, NSU and PU stop a correct execution of the program prematurely.

```
1 if(secret){
2    x = 1; //stopped under NSU
3 }else{
4    y = 1;
5 }
6 if(x){ //stopped under PU
7    y = 2;
8 }
9 print(z);
```

Listing 4.5: Example of sensitive value flow.

Permissiveness can be increased by identifying and upgrading possible target variables of write operations on all branches of a security-sensitive

control-flow statement [AF10, HS12b, HBS15]. The problem with upgrading write target variables relies on their precise identification. This problem has been approached by providing *privatization operations* [AF10], *upgrade annotations* [HS12b] or a hybrid combination combination of static information collected before the program execution [HBS15]. However, providing write targets annotations manually does not scale to large programs. Moreover, these approaches do not scale for web applications where large parts of code originate from third-party servers, preventing manual annotation or static analysis.

4.2 Gifc

This section describes GIFC[1], a flow-sensitive dynamic IFC monitor for client-side web applications. GIFC extends the permissiveness of purely dynamic PU monitors by analysing untaken branches at run-time. Information flows between the JavaScript code and DOM elements and other web APIs are handled using API models. GIFC inlines the IFC monitor by instrumenting the source code of the application.

In this section, we build on the source code instrumentation-based monitoring mechanism described in Chapter 3 for the enforcement of IFC policies. Recall that inlining the monitor means that JavaScript code is instrumented to trap security-relevant operations and call the monitor through a well-defined interface.

We assume that developers tag the sources and sinks in the input program and provide the specification of function models to handle external libraries. In what follows, we will first introduce the IFC monitor details, how it deals with implicit flows and external libraries and the rest of the challenges introduced before. Then, we describe the necessary modifications or extensions of the instrumentation platform (i.e., ARAN) used in Chapter 3, needed for enforcing IFC policies.

4.2.1 Gifc Monitor Interface

Figure 4.1 shows our IFC runtime monitor's interface based on the semantics of the PU monitor presented by Austin et al. [AF10]. GIFC is flow-sensitive, meaning that security levels of program values can change

[1] https://gitlab.soft.vub.ac.be/ascullpu/guardia-ifc

	Monitor function	Description
implementation	`pushContext(x, t)`	Push a context label given a type
	`popContext(t)`	Pop a context label given its type
	`join(x,y)`	Returns the least upper bound of the labels
	`permissiveCheck()`	Determine if there is no PU violation in a branching point
	`enforce(y,...xs)`	Enforce IFC if `y` is a sink and some of `xs` is a source
	`leave(fn)`	Remember all values' labels of an external function call before its execution
	`enter(fn, val)`	Attach a computed label to the return value of an external function
user interface	`tagAsSource(x)`	Tags `x` as source (i.e., sensitive data)
	`tagAsSink(x)`	Tags `x` as sink (i.e., produce a public observable data)
	`addFnModel(fn, md)`	Registers a model `md` for an external function `fn`

Figure 4.1: GIFC monitor interface

during the program execution. The instrumented code interacts with the monitor using a well-defined interface shown in Figure 4.1, distilled from the PU monitor's semantics presented by Austin et al. [AF10].

Similar to GUARDIA's enforcement in Section 3.3, GIFC decouples its implementation from the instrumentation platform, which enables changing parts of the monitoring mechanism independently.

GIFC's interface distinguishes two categories of functions that can be called from the monitor. Calls to the functions categorized as "implementation" are called by the enforcement and will be described in Section 4.2.3. These functions track the information flows during the program execution and enforce the IFC policy. The IFC policy is defined using the functions marked as "user interface", which are explicitly called by the GIFC user, i.e., developer implementing the IFC analysis. Those calls refer to the tagging functions for sources and sinks and to add a function model and are described in the next section.

4.2.2 Gifc User API

GIFC provides functions `tagAsSource` and `tagAsSink` that developers have to insert into a program to identify sensitive sources and sinks. For example, the program in Listing 4.6 shows the required tagging for enabling the IFC monitor to prevent the user password from flowing to the browser console output. The `console.log` function is tagged as a sink, and the `value` property of the HTML element with id `#pass` as a source.

```
1 tagAsSink(console.log);
2 const onClickHandler = () => {
3  const $ = document.querySelector;
4  let pass = tagAsSource($('#pass').value);
5  ...
6  console.log(pass);
7 }
```

Listing 4.6: Prevent password leakage

Although developers currently have to manually tag sources and sinks in the code, it would be possible to devise a more declarative (external) manner for specifying sources and sinks, which the code instrumenter can then use to introduce the tag functions in the target program where appropriate. To help developers with the specification of sources and sinks, we have built GUARDIAML [SNE+19], a VSCODE plugin that uses Machine Learning (ML) to suggest which functions are sources or sinks. We will give further details in Section 4.2.2.1.

API Function Models: Besides identifying sources and sinks, GIFC also expects that external functions are registered using `addFnModel(fun, γ)`. Function γ has to approximate the flow of information of function `fun` in terms of the labels of the arguments. For example, for `Math.pow(x,y)` shown in Listing 4.2, we would register `γ(x,y) = x⊔y`, correctly capturing the notion that if `Math.pow` is called with one or two sensitive argument values, then the resulting value is also sensitive. We implemented models for some objects of the standard libraries including `Math`, `Array`, and `String`. However, the monitor fall back to a default conservative model for functions calls that do not have a precise model implementation. Section 4.2.4 gives the function models implementation's details.

4.2.2.1 Automated Suggestions of Information Sources and Sinks

As mentioned in Section 4.2.2, we developed a VSCode plugin called GuardiaML to suggest information sources and sinks based on ML . More concretely, GuardiaML uses a training set of sources and sinks based on the Node.js API, which is used by a Support Vector Machine (SVM) algorithm to train a recommendation model. This recommendation model can be called from VSCode's JavaScript editor to predict which function calls are considered to be sources and sinks. Figure 4.2 shows an example of using GuardiaML in VSCode.

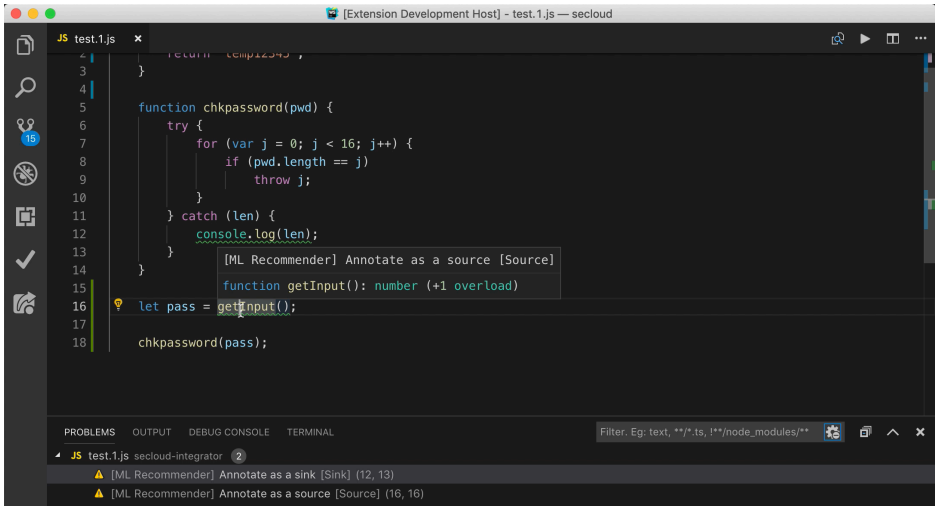


Figure 4.2: Example interface of automatic classification of sources and sinks using GuardiaML.

For the ML component, a sink is defined as a call to a resource method that either creates or overwrites a previous value, and a source is defined as a call to a resource method that reads and returns a value at the application code.

We draw inspiration from [RAB14], in which ML is used to classify sinks and sources in the Android API. This approach is, however, specific to the Android API and its conventions, which is reasonable as the approach is application-specific. GuardiaML focuses on the Node.js API and deals with sinks and sources specific to JavaScript.

Dataset creation and prediction To encode the labelled dataset, we used a JSON-formatted text that includes the necessary information to apply ML. Our approach defines features of inputs that are fed to the SVM algorithm for training based on naming characteristics, such as whether a method starts with a specific keyword such as `get` or `set`.

```
{
    "cl":0,
    "textRaw": "assert(value[,message])",
    loc: {...}
}
```

Listing 4.7: Input format for SVM.

An example of the training data fed to the SVM algorithm is shown in Listing 4.7. From this specification, features are automatically extracted from the `textRaw` field. Each input's respective class is denoted in the `cl` field. The inputs come in the form of a binary representation based on the presence or absence of a feature. Table 4.1 shows a partial list of features used as input for the classifier assuming only 3 keywords: `get, set, log`. If a particular feature is present then its respective bit is set to 1.

Table 4.1: Sample features assuming three keywords: get, set and log.

Function	Binary Representation
`getPassword(u)`	001
`setEmail(e)`	010
`console.log(t)`	100

We devised a list of common keywords for sinks and sources for Javascript as a driving force, as this would increase the applicability of our approach to other APIs. This design decision makes the algorithm sensitive to particular naming conventions. However, if coding conventions are followed, our algorithm can be applied to unseen instances of different APIs. In case an API uses a very specific naming convention, then features need to be re-designed and a new SVM trained. This does not limit applicability of our approach, as it is a straightforward process.

Annotating examples is important in order to use supervised methods to tackle this problem. We performed labelling on some of the methods of the API manually in order to create a training and test set. Then,

we applied machine learning on the training dataset in order to learn how to handle new unseen method specifications, or entire APIs, and automatically classify each method into one out of three possible classes: sink, source or neither.

4.2.3 Gifc Implementation API

This section describes the monitor functions which are internally used by GIFC to enforce IFC policies. GIFC uses a shadow stack to maintain the *pc* label. The `pushContext()` function pushes a security label into the stack every time the program encounters a control flow statement. The label value is the `join` of all values that influences control flow in a control flow statement. The `popContext()` function removes the top element of the stack when the execution reaches the end of a control flow structure body.

JavaScript programs can have non-structured control dependency due to `break`, `continue`, exceptions, etc. Therefore, in addition to *pc* stack, our monitor maintains a stack for these language features. More concretely, our monitor maintains an *exception* stack that keeps track of implicit flows that arise from throwing exceptions in sensitive contexts. We push into the *exception* stack when the execution of a `throw` statement depends on sensitive information because there is no syntactic way to know when an exception will be handled. Then, when a `catch` handler is reached, the monitor pops all values from the *exception* stack.

The `join(a,b)` operation is used whenever the label of a value depends on multiple values (i.e., the *least upper bound* of the elements). As a concrete example, consider `let z = x + y;`. The label of `z` depends on the more sensitive label involved in the values of the binary operation (also, in the label of the pc context, etc.).

The `permissiveCheck()` enforces the PU invariant at the branching point of control flow structures to avoid a total leak of information. `enforce()` is then used at code locations (e.g., function application, setters) where information can leak the system to prevent information flow violations. It checks if there is any sensitive value flowing to a setter or function annotated as a sink.

As mentioned in Section 4.2.2, external functions need to be registered using `addFnModel(fun, md)`. During program execution, upon the call to an external function, function `leave` looks up the corresponding γ function

and saves the labels from the call information (i.e., the arguments, `this`, and function pointer of the call). Next, the actual external function is called with the unlabeled argument values. Finally, function `enter` calls γ which computes the information flow for the given external function call. The resulting value is the label ℓ of the value returned from the external call.

4.2.4 Handling External Libraries

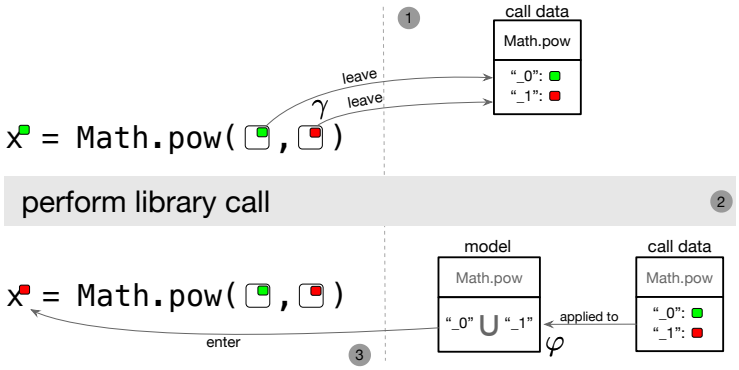


Figure 4.3: Example diagram of the interaction of external library call with its API function model.

This section explains how GIFC handles information flows between JavaScript code and external libraries. Specifically, we describe an API function model with two functions φ and γ, inspired by the ones presented by Hedin et al. in [HSPS17]. The φ knows how to marshal the values from the monitored program to the external function. Those stored labels are then used by γ to decide which label should be attached to the return value of the API function call.

Consider `Math.pow(x,y)` as example of an external library call in JavaScript. Its approximate function model implementation is given in Listing 4.8. The object property `getLabel` is a method that computes the return value label resulting from the library call. Note that at line 3 the function refers to an internal `state` variable. This `state` property stores the labels of the values involved in the function call. In this example, the call's values are the `Math` object, the `pow` method and the two arguments of the call. In this example, the call arguments are the only

values that affect the label of the result. Therefore, we do not consider labels of `Math` and `pow`.

```
addFnModel(Math.pow, {
  getLabel: function(){
    return IFC.join(this.state["_0"], this.state["_1"]);
  }
})
```

Listing 4.8: Implementation example of the `Math.pow` function model.

The behavior of marshalling and storing the `Math.pow` arguments' labels is shown in Figure 4.3 ①. For example, whenever `Math.pow` is about to be called, the monitor fetches the function model for `Math.pow` and saves the call arguments' labels.

After marshalling, the unlabeled values are applied to the `Math.pow` in ② which computes the `pow` as in a non-instrumented JavaScript application. After the function application, a label needs to be attached to the return value. This is done by applying the stored argument labels in ① to the function model (i.e., `getLabel` in Listing 4.8) implementation as shown in Figure 4.3 ③.

Note that the labelled values can be structured data like objects and arrays. This implies marshalling the entire object structure (i.e., the object and its properties). Specifically, the marshalling creates a mirror of the object reflecting the label information of its properties. For example, consider an external function call `join([x, secret, z])` which takes an array of string values and returns a concatenated string of the array elements as a result. The array values are first marshalled to an array of the same length but only containing the element's label for a given index.

Document Object Model: To be able to reason about DOM element methods without VM modifications, one could turn to JavaScript proxies [Ecm15] to enhance DOM element methods with IFC semantics. However, the DOM cannot handle proxified nodes because type checks that inspect actual DOM elements will fail for proxies. Also, our function model above is stateless, while many DOM elements model state.

To monitor the DOM API, GIFC associates a meta-object with each DOM element. This meta-object keeps track of the element properties' labels and is stored in its target DOM object as an "anonymous" property, using a symbol property key. Note, however, that this approach is

transparent but not tamper-proof. This is because the attacker can gain access to the meta-object by mean the language reflective features (i.e., `Object.getOwnPropertySymbols()`).

When a getter or setter is executed on a DOM element, the instrumentation ensures that each element property write operation updates its corresponding label in the meta-object. Every value resulting from a property read operation will be labelled with its corresponding label. For handling DOM elements methods, function models associated with the method is used.

4.2.5 Dynamic Code Evaluation

This section explains how GIFC tracks information flows on language constructs that enable dynamic code evaluation.

Some language functions can interpret string values as code. The `eval` function is one example of these functions that evaluates a given string as code. Because this code is only known at run-time and therefore not instrumented, all information flows within this specific code are lost during the call to `eval`.

To solve the problem above, `eval` is specialised to track information flow by instrumenting its argument before evaluation. In GIFC, the metadata associated to program values is managed by an access control system provided by the instrumentation platform [CGDD16]. Such an access control system enables the precise tracking of values and their metadata during the program execution, which in our case is the security label. Whenever the execution creates a new value, the access control system creates a security label (metadata) and associated it with the value. This security label is available to the monitoring mechanism during the lifetime of the value.

4.2.6 Permissiveness

GIFC's monitor is a flow-sensitive variation on the PU strategy presented by Austin et al. [AF10]. GIFC proposes to use AST information of the program to extend the *pc* label context of language constructs such as `return`, `break`, `throw`, etc., when their execution depends on secret values. This information is crucial and must be handled carefully by approaches like `NSU` or `PU` to ensure soundness and permissiveness guarantees. If the

language features mentioned are not handled, the monitor will potentially leak information and become unsound. On the other hand, if they are used with an approach like `NSU`, the monitor could become excessively restrictive. For example, consider the code snippet in Listing 4.9 in which `h` is secret. The execution of lines from 2 to 4 depends on the value of `h`, given the `throw` statement will execute based on the value `h`. Therefore, a `NSU`-based monitor will stop the execution at the assignment statement (line 2). In this example, this problem is extended until the program encounters the first error handler.

```
1 if (!h) {
2   throw new Error()
3 };
4 y = z;
5 f();
6 g();
```

Listing 4.9: Example of non-structured implicit control flow due to exceptions.

Listing 4.10 shows an approximate implementation of the handling of conditional non-structured control flow statements in GIFC. Whenever an `if` statement is reached, the `test` trap is executed. Within the trap, the security context is extended with the label resulting from the test expression, as shown in line 3. Then, from lines 6 to 8, the trap collects the statements that cause non-structured implicit control flow and extends the *statement context* (e.g., `throw, return`, etc.) with the value of the current context. Notice that at line 7, as a rule for handling exceptions, the context for conditional exceptions (i.e., `throw` statements) is saved; this is because different statements have different scoping rules. Applying the statement context rule for exceptions to the program in Listing 4.9 implies that all code starting at line 4, until the first exception handler (i.e., `catch`) will be influenced by the label of `h` joined with the security level of the context at line one.

```
1 META.test = function(value, idx){
2   ...
3   const context = IFC.join(IFC.currentContext(), value.label);
4   const node = aran.node(idx);
5   if (node && (node.type === 'IfStatement')) {
6      if (IFC.hasNode("ThrowStatement", node)) {
7       IFC.pushContext(IFC.THROW, context, idx);
8      }
```

```
9       ...
10    }
11    ...
12  }
```

Listing 4.10: Example implementation non-scructured control flow.

Automatic write target annotation: In GIFC, we propose to extend the permissiveness of our PU based monitor with *upgrade annotations* computed at run-time using AST information. The annotations upgrade the security label of write targets of assignment expressions found in the branches of control-flow structures dependent on sensitive data.

Collecting write targets of assignment expressions in the branches requires access to the AST. In GIFC, this information is available at run-time through the instrumentation platform. Each trap has access to the AST node, which triggered the trap shown in Listing 4.11 at line 3.

```
1  ...
2    if(value.label === IFC.PARTIAL){
3       const node = aran.node(idx);
4       const exps = IFC.writeTargets(node);
5       upgradeWriteTargets(exps, scope);
6    }
7  ...
```

Listing 4.11: Example implementation of the *upgrade annotation* strategy for improving permissiveness.

Automatic upgrade annotations are inserted on control-flow statements such as `if` statements. Listing 4.11 shows an approximate implementation of the upgrade annotation strategy used in GIFC. The `test` function traps the test expression of control flow statements, specifically, `test` receives as arguments the `value` resulting from the test expression and the AST node index (`idx`) of the control flow statement (e.g., `if`). The node's index is used to access the AST during the trap's execution.

In the example, the upgrade annotation strategy is performed if the control-flow statement's test expression depends on sensitive information, as shown in line 2. In this case, the current control-flow statement's write targets are computed as shown in line 5. Finally, write targets' labels are upgraded at line 6 through `upgradeWriteTargets(exps, scope)`. During the program execution GIFC mirrors the environment. Therefore,

traps can access the `scope` on which the expression that triggered the trap is being executed.

Extending the PU strategy with upgrade annotations is sound if all possible write targets in alternative branches (paths) of the execution are identified. If this is the case, then it is safe to upgrade values in the taken branch because all possible information flows in alternative branches are ensured. However, upgrade annotations can be unsound in the presence of dynamic code evaluation expressions within alternative branches because side-effects within the dynamic code cannot be observed. To explain this problem, consider the code snippet in Listing 4.12. In this example, `x` is a security sensitive variable. The dynamic evaluation of the variable `code` in line 5 is conditionally executed in an elevated security context. Whenever `x` is `false`, our automatic upgrade annotations strategy will analyse the non-taken (`else`) branch to identify the write targets in order to upgrade their label. Because this branch is not executed, the value of code is unknown and as such, write targets cannot be identified precisely.

```
1 if(x){ // label of x -> H
2   doSomething(x);
3 }else{
4   let code = foo();
5   eval(code);
6 }
```

Listing 4.12: Example of the use of dynamic code evaluation to bypass automatic upgrade annotations.

To keep the enforcement sound, the monitor should not perform the upgrade annotations strategy whenever dynamic code evaluation is present in an alternative branch. Discarding the upgrade annotations means that the default PU strategy is enforced before taking any branch of the control flow statement. In future work, we plan to include the check of expressions that dynamically evaluate code into our automatic upgrade annotations.

4.2.7 Code Instrumentation Platform

Similar to GUARDIA in Chapter 3, we use ARAN as a source code instrumentation platform. GIFC also uses *Linvail* [CGDD16] for augmenting the program values with security information. In particular, it allows adding meta-data information to runtime values (including primitive values), enabling dynamic analysis such as IFC.

GIFC's monitor is thus implemented by means of ARAN's traps. Listing 4.13 shows the implementation strategy of one of the traps for tracking IFC in GIFC. The *get* trap is a function with two responsibilities: (i) perform the base program operation transparently and (ii) call the IFC monitor to track the information flow specific to the current program operation being executed.

```
1 META.get = (o, k, idx) => {
2   const res = linvail.advice.get(o, k, idx);
3   res.label = IFC.join(o.label, res.label, k.label);
4   return res;
5 }
```

Listing 4.13: Example of the get trap implemented as part of GIFC monitor.

In Listing 4.13 the base program operation is performed at line 2. Note that, `o` and `k` are labeled values. This implies that `linvail.advice.get(o, k, idx)` needs to unwrap those values before applying the `get` operation. The `res` value will be a labeled value. However, it has the label of the context at the point of its creation. Therefore, in line 3, its security label is appropriately adjusted.

Method invocations in GIFC are handled by the *invoke* trap implemented as shown in Listing 4.14. The implementation of this trap has three responsibilities: (i) to enforce the IFC policy, (ii) perform the method call and (iii) simulate information flow whenever the method being executed is a built-in. Enforcing the IFC policy (lines 2 – 12) involves arguments' labels as well as the security context's label of the call (line 9). The security context of the call includes the *pc* label extended with the label of the non-syntactic control flow constructs (exceptions, `break`, etc.) that may influence the current method call.

If the method being invoked is a built-in (e.g.`Math.pow(2,3)`), the information flow within the built-in call is handled by our function models as shown in lines 13 to 15 and in lines 19 and 20. This implementation corresponds to the tracking of information flow during external libraries (built-in) calls (explained in Section 4.2.4).

```
1 META.invoke = (o, k, xs, idx) => {
2     const ths = o.inner; // unwraps the object
3     const key = k.inner; // unwraps the property
4     const fn = ths[key];
5     let isSink = IFC.isSink(fn);
```

```
6
7     if (isSink) {
8       const labels = Array.from(xs).map(arg => arg.label);
9       labels.push(IFC.currentContext());
10      ... //push the labels of o and k...
11      IFC.enforceIFC(isSink, aran.node(idx), ...labels);
12    }
13    let isExternal = extlib.hasFunction(fn)
14    if (isExternal)
15      extlib.leave(fn, o.inner, xs);
16    //Invoking the target function
17    let res = linvail.advice.invoke(o, k, xs, idx)
18    //labeling the new value based on the libraries models.
19    if (isExternal)
20      res = extlib.enter(fn, res);
21    return res;
22  }
```

Listing 4.14: Example of the invoke trap implemented as part of GIFC monitor.

4.3 Evaluation

To evaluate our approach, we performed a qualitative and quantitative evaluation of our GIFC implementation. The qualitative evaluation shows how effective our approach is in detecting illicit information flows. The quantitative evaluation shows our approach's performance implications concerning an uninstrumented baseline and compares it to related approaches. This evaluation only considers IFC approaches with an available implementation for JavaScript applications.

4.3.1 Qualitative Evaluation

To evaluate the effectiveness of GIFC in a practical way, we compare it with IF-TRANSPILER, JSFLOW, and ZAPHODFACETS by determining whether or not illicit flows are detected in a suite of benchmark programs[2]. The benchmark suite was designed by Sayed et al. [STA18] and consists of 33

[2]Unfortunately we were unable to set up a functional test environment for FLOWFOX and JEST. In the case of JEST, specific models are required that are undocumented and not trivial to reproduce.

programs designed explicitly for testing information flow control. It contains a wide variety of (combinations of) language features that challenge any IFC approach.

We extended the original benchmark suite with 5 new programs to test features such as `eval`, API function calls, and property getters/setters not present in the original one. Appendix B.2 includes a table describing the 28 programs included in the original benchmark suite. The last entries in the table describe the new 5 additions. Each benchmark program takes as input a secret string value, which the program attempts to leak explicitly or implicitly in various ways. We ran all tools on all benchmark programs in Node.js, except for ZaphodFacets, of which the experiments were performed in *Mozilla Firefox 8.0* as required by the tool. This setup is similar to the one used by the authors of IF-transpiler in their benchmark.

Table 4.2 shows how Gifc compares to the other three IFC approaches. The ✓ means that a tool could detect the illicit information flow, while ✗ indicates that a tool was unable to detect the illicit flow. *R.Err* indicates that a tool threw a runtime exception and was unable to execute the program correctly. *In.Err* indicates that the tool was unable to inline the monitor into the original program source code. *Exp* indicates that the tool threw an exception at a point where an illicit information flow could be, but it was premature because there was no invalid information flow at that point. This observation was also made in [STA18].

The results in Table 4.2 show that Gifc can detect and prevent illicit information flows in all test programs. For the 28 programs from the original suite, we were able to reproduce the findings reported by Sayed et al. [STA18] for IF-transpiler, JSFlow, and ZaphodFacets. For the 5 test programs that we extended the suite, Gifc and JSFlow successfully detected all illicit flows.

Both IF-transpiler and ZaphodFacets were able to successfully detect an illicit flow in only one out of 5 new test programs. Adding online support for `eval()` (Test 29) in IF-transpiler needs the static analysis component and the transpiler in the same application process. Supporting APIs (Test 31) in IF-transpiler will require the refactoring of the transformation rules to include function models. It will also require implementing the mechanism that allows assigning models to APIs functions that need to be configured at runtime. Finally, a revised implementation of IF-transpiler transformation rules for inlining getters/setters (Test

33) is needed.

From this experiment, we conclude that GIFC is on par with the existing tools in terms of detecting illicit information flows in the presence of different JavaScript features.

4.3.2 Quantitative Evaluation

We conducted performance benchmarks to measure the impact of GIFC on the original application's performance (the baseline) and gauge how our approach compares with IF-TRANSPILER, JSFLOW, and ZAPHODFACETS in this regard.

Methodology The benchmark programs consist of 9 different algorithms used in Sayed et al. [STA18]. These programs originated from the Rosetta Code [3] project which contains a collection of different programming tasks and their solution in different languages.

For executing the performance benchmarks, we used the same interpreter for the tools as in the qualitative evaluation in Section 4.3.1. Each algorithm was executed 10 times in one process with JIT enabled. After, we computed the average execution time (in milliseconds) for each algorithm and tool combination. We did not compare the performance overhead between the tools statistically because they were executed using different JavaScript interpreters (i.e., Node.js and Mozilla Firefox).

Results Table 4.3 shows the execution time of each algorithm and tool combination. Both JSFLOW and ZAPHODFACETS failed to execute the AES algorithm, which was also reported in [STA18]. The results in Table 4.3 show that the approaches that rely on code instrumentation (GIFC and IF-TRANSPILER) have a performance impact which is one or more orders of magnitude smaller than the performance impact of approaches that rely on an additional interpreter (JSFLOW and ZAPHODFACETS). IF-TRANSPILER performs better than GIFC, but performance is still comparable. Essential sources of performance overhead in GIFC's dynamic monitor are the wrapping and unwrapping of values before and after API calls and the emulation of implicit calls to functions `toString()` and `valueOf()` due to implicit value coercion in the target program. However, this over-

[3] Roseta Code: `http://rosettacode.org/wiki/Rosetta_Code`

Program	JSFlow	ZaphodFacets	IF-transpiler	Gifc
Test 1	✓	✓	✓	✓
Test 2	✓	✓	✓	✓
Test 3	✓	✓	✓	✓
Test 4	✓	✓	✓	✓
Test 5	✓	`R.Err`	✓	✓
Test 6	`Exp`	`R.Err`	✓	✓
Test 7	`Exp`	`R.Err`	✓	✓
Test 8	`Exp`	`R.Err`	✓	✓
Test 9	`Exp`	`R.Err`	✓	✓
Test 11	`Exp`	`R.Err`	✓	✓
Test 11	`Exp`	`R.Err`	✓	✓
Test 12	`Exp`	`R.Err`	✓	✓
Test 13	✗	`R.Err`	✓	✓
Test 14	✓	`R.Err`	✓	✓
Test 15	✓	`R.Err`	✓	✓
Test 16	✓	`R.Err`	✓	✓
Test 17	✓	`R.Err`	✓	✓
Test 18	✓	`R.Err`	✓	✓
Test 19	✓	`R.Err`	✓	✓
Test 20	✗	`R.Err`	✓	✓
Test 21	`Exp`	`R.Err`	✓	✓
Test 22	✓	`R.Err`	✓	✓
Test 23	✓	`R.Err`	✓	✓
Test 24	✓	`R.Err`	✓	✓
Test 25	✗	`R.Err`	✓	✓
Test 26	✗	`R.Err`	✓	✓
Test 27	✗	`R.Err`	✓	✓
Test 28	✗	`R.Err`	✓	✓
Test 29	✓	✗	✗	✓
Test 30	✓	`R.Err`	`In.Err`	✓
Test 31	✓	`R.Err`	✗	✓
Test 32	✓	✗	✓	✓
Test 33	✓	✓	✗	✓

Table 4.2: Effectiveness comparison

head provides better precision to GIFC compared to IF-TRANSPILER as discussed in Section 4.3.1.

4.4 Conclusion

This chapter introduced GIFC, a practical and portable dynamic IFC monitoring mechanism for client-side web applications. GIFC implements the PU strategy extended with upgrade operations to improve the permissiveness of the monitoring. It offers support for DOM and external libraries, enabling the practical use of IFC. Having static information at runtime makes it possible to develop a more precise model of implicit flows. GIFC is the first inlining mechanism that supports dynamic code evaluation online.

Benchmark results show that the performance impact is better than approaches that rely on an IFC-aware interpreter, but still is non-negligible. Nevertheless, we believe that the approach can be used in settings where security plays a key role. Also, GIFC can aid developers when used as an IFC testing tool at development time.

Approach	FFT	LZW	KS	FT	HN	24	MD5	SHA	AES
IF-TRANSPILER	14ms	11ms	363ms	10ms	327ms	126ms	33ms	29ms	284ms
GIFC	23ms	34ms	747ms	35ms	1238ms	1233ms	31ms	35ms	780ms
JSFLOW	404ms	421ms	5206ms	661ms	5165ms	4371ms	491ms	566ms	fails
ZAPHODFACETS	100ms	188ms	15563ms	145ms	12657ms	6403ms	124ms	197ms	fails

Table 4.3: Performance benchmarks

Chapter 5

Tamper-proof and Transparent Monitoring for Web Applications

In Chapters 3 and 4 we presented GUARDIA and GIFC, two mechanisms for declaring and enforcing AC and IFC security policies. A fundamental property of GUARDIA and GIFC is their portability across different JavaScript runtimes. This property was achieved by inlining GUARDIA and GIFC within the application's code.

This chapter discusses the trade-offs involved with the design of an inline monitor based on source code instrumentation with respect to the 4 properties inline monitors should uphold (Chapter 2, Section 2.4.1): completeness, correctness, transparency, and integrity. However, when the monitor is part of the application to be secured, the tamper-proofness of the monitor becomes an issue. Malicious code may tamper with the monitor or with the environment in order to alter the monitor's behavior to, for example, bypass the security checks. This problem is especially critical in client-side web applications written in JavaScript, as the language is highly dynamic and features strong reflective capabilities. Furthermore, browser runtimes feature additional, non-standard means for dynamically executing code, such as the `innerHTML` property of DOM elements. The combination of these language and browser features makes it challenging to obtain complete mediation and integrity of an inlined monitor based on source code instrumentation such as those included in GUARDIA and

GIFC.

We analyse these challenges in the context of a concrete source code monitor, the one included in GUARDIA, and propose solutions to these challenges. GIFC's monitor uses the same technology as GUARDIA's monitor, and therefore faces the same issues. The contributions of this chapter are:

- A thorough analysis of all challenges and existing solutions for the integrity of an inlined monitor for client-side web applications.
- A novel strategy to deal with integrity concerns introduced by implicit coercions on function arguments.
- A qualitative comparison of different language-based approaches concerning the desired features of an inlined reference monitor in the context of client-side web applications.

5.1 Integrity Challenges of Inlined Runtime Monitors

This section provides the necessary background to understand the integrity issues that inlined reference monitors must tackle so that attackers cannot tamper with the monitor's behavior. We first discuss integrity concerns introduced by JavaScript features, and then discuss additional concerns specific to the web application context.

5.1.1 Integrity Concerns Introduced by JavaScript

This section discusses four challenges that JavaScript features pose to runtime monitors to ensure their integrity: its weakly-typed nature, prototype-based object model, dynamic code evaluation, and higher-order built-in.

5.1.1.1 Challenge 1: Automatic Value Coercion

JavaScript is a *dynamically-typed* language, meaning that variables can store values of different types during a program's lifetime. Allowing variables to store different values of different types implies that *type-checking* of program values is done at runtime. JavaScript is also considered to be *weakly-typed*, as it will try to "find" compatible values so that a program

operation can happen. Determining compatible values implies a *coercion* of the given value to its counterpart in the required type. For example, the ECMAScript semantics for the addition operator specifies an algorithm to implicitly *coerce* the operand values to a common compatible primitive (i.e., boolean, string, or number) value types before performing the addition. For example, consider the code snippet in Listing 5.1.

```
1 let name = "John";
2 let length = 5;
3 let flag = true;
4 name = { foo: "bar" };
5 length(flag + name);
```

Listing 5.1: Weakly type checking example.

The addition expression in line 5 in Listing 5.1 will coerce its *boolean* and *object* operands to *string* values before performing the addition. Value coercions ensure that the program rarely crashes because of the types of values used by a specific JavaScript construct, such as the addition operator or a built-in method like `Array.prototype.filter`. However, value coercion can introduce semantic errors and, more importantly for this work, security vulnerabilities. To illustrate this problem, consider the example in Listing 5.2.

```
1  let liar = {
2    value: 'div',
3    toString(){
4      let temp = this.value;
5      this.value = 'iframe';
6      return temp;
7    }
8  }
9
10 ...
11 createSafeElement(liar);
12
13 function createSafeElement(tagName){
14  if(tagName != 'iframe'){
15     document.createElement(tagName);
16  }
17 }
```

Listing 5.2: Coercion example.

Suppose an attacker is in control of the `liar` object defined on lines 1 – 8. This `liar` object can be, for example, provided by an attacker as result of an XSS attack. When `createSafeElement(liar)` is called, the `!=` semantics will coerce the `tagName` to string. To coerce `tagName`, its `toString()` method is called implicitly by the interpreter. In this case, the coercion of `tagName` results in the value `'div'`. After the check on line 14, the `document.createElement(tagName)` is called on line 15, and the *tagName* object is coerced to a string again, resulting in the value `'iframe'`. In this example, both the test on line 14 and the secured operation on line 15 perceive different views of the same object, which is dangerous for a security mechanism.

Let us illustrate how this coercion problem could happen in GUARDIA using a concrete example. Whenever a security-relevant program operation is about to happen, the reference monitor checks the validity of the data involved in the operation. For example, the instrumented program in Listing 5.3 attempts to create an *iframe* element using the `liar` object from Listing 5.2.

```
1 META.apply(document, document.createElement, [liar]).
```

Listing 5.3: Implementation of an instrumented call to `createElement`.

Assume that the `META.apply` trap calls `createSafeElement` to enforce that `document.createElement` does not violate any policy. Note that the function and arguments (i.e., `document.createElement` and `'iframe'`) are outside the Trusted Computing Base (TCB), and as such they cannot be trusted. In this example, the attacker can use the `liar` object from Listing 5.2 to bypass the security mechanism. Because the behavior of the call to `META.apply(document, document.createElement, [liar])` is the same as the call to `createSafeElement(liar)`, the security is bypassed.

In conclusion, the implicit coercion problem arises from unwanted side-effects by attacker-provided data during policy enforcement.

5.1.1.2 Challenge 2: Prototype Inheritance Model

JavaScript's prototype-based object-oriented model can also be subject to security vulnerabilities. In JavaScript, objects are associative arrays in which each key represents the name for an object property. Properties can be added and deleted at run-time.

Figure 5.1: Example of prototype inheritance chain of a JavaScript object.

Objects are created with a private property called *prototype* that points to a *super* object. An object's prototype has also its own prototype, which results in a chain of objects. This chain is known as the prototype chain. The prototype chain ends when an object's prototype points to `null`.

Looking up a property in an object implies looking up the property in the object's own properties. If the property is not found within the object's own properties, then it is looked up in the object's prototype, and so on. This process is repeated until the property is found or a `null` prototype is reached. Any change in the prototype chain of an object, such as adding or deleting a property, can affect the object's behavior.

Similar to the rest of the properties, an object is allowed to change its prototype during program execution, which can potentially lead to a family of exploits known as *prototype poising* [MFM10, MPS12]. A prototype poisoning attack consists of changing any object in a target object's prototype chain in order to tamper with the target object's behavior, specifically when the attacker does not have direct access to the target object itself. These changes may be:

- adding, changing or deleting a property in the object's prototype chain, or
- assigning or replacing any of the prototypes objects in an object's prototype chain.

Prototype poisoning attack Figure 5.1 shows the prototype chain of the `user` object after its creation at line 1 in Listing 5.4. In this example, the prototype of the `user` object points to `Object.prototype`, which in turn points to `null` that represent the end of the prototype chain.

As a concrete example for a prototype poisoning attack, consider the code snippet in Listing 5.4.

```
1 let user = {};
2 ...
3 Object.prototype.isAdmin = true; //Attacker's code
4 ...
```

```
5 if(user.isAdmin){
6   doAdmin();
7 }
```

Listing 5.4: Prototype poisoning example.

Line 3 results in the addition of a new property `isAdmin` to the `Object.prototype`. Because `Object.prototype` is in the `user`'s prototype chain, the addition of this property is reflected in `user`'s behavior. As already mentioned before, the attacker does not need direct access to the `user` objects for affecting its behaviour. Instead, by tampering with the Object.prototype he has compromised the integrity of the `user` object. When the, property `isAdmin` is looked up in line 5, the value of the poisoned `Object.prototype` will be retrieved and used for the test. This results in the call to the "security sensitive" function `doAdmin()` in line 6.

5.1.1.3 Challenge 3: Dynamic Code Evaluation

A *code evaluation sink* is a language construct that enables the execution of data (usually strings) as code. Different language constructs are considered code evaluation sinks. Functions `eval, Function` and `setTimeout` are examples of such evaluation sinks in JavaScript. The DOM API also introduces evaluation sinks such as `Element.innerHtml` and `document.write`.

Evaluation sinks are challenging for source code instrumentation-based monitoring mechanisms because the instrumentation in such monitors is done statically. Therefore, any code provided to an evaluation sink will not be trapped by the monitoring mechanism.

If user-controlled data reaches an evaluation sink, this constitutes a security vulnerability that can be exploited to perform XSS attacks as discussed in Section 2.1. For example, Listing 5.5 shows setting the `Element.prototype.innerHTML` property. This assignment allows developers to add an HTML fragment expressed as a string to a component. The string representing the HTML fragment is then parsed as a DOM node and added to the document's tree. The attacker can craft a node containing JavaScript code that is executed when the node is added to the tree. Specifically, the assignment in Listing 5.5 results on parsing the fragment as an `Image` element with an `onerror` event handler and a `null src` property. In this case, the `onerror` event handler is always triggered

because the image always fails to load.

```
element.innerHTML = "<image src onerror='foo()'/>";
```

Listing 5.5: Dynamic HTML parsing example.

5.1.1.4 Challenge 4: Higher-Order Built-in Functions

JavaScript has built-in functions that are higher-order. A higher-order function can receive one or more functions (callback) as arguments, or/and can return a function as a result. The built-in `Array.prototype.map` is an example of such higher-order function in JavaScript. When called, `Array.prototype.map` *implicitly* applies the callback to the elements of the array. The values resulting from the call to the callback are collected in an array and returned as the result of the call to `Array.prototype.map`.

Unfortunately, built-in functions' bodies cannot be monitored using source code instrumentation, which means that the monitor cannot mediate the *implicit* calls of the callback function given as the argument to the built-in.

To explain why this can be problematic, consider that Policy 2 from Chapter 3 Section 3.2 that *prevents the dynamic creation of iframe elements* is governing the code shown in Listing 5.6. In this example, the application attempts to bypass the Policy 2 by creating an *iframe* element by mapping the `document.createElement` over an array that contains the single value `'iframe'`. By using `Array.prototype.map`, the application exploits the inability of GUARDIA monitor of instrumenting the call to `document.createElement`, which is done implicitly as we explained before. As shown in the instrumented version of the application in Listing 5.7 `document.createElement` is not wrapped by META.apply, and as such the monitor cannot mediate its calls.

```
//Application is governed by the policy that prevents creation of '
    iframes'
...
let iframe = ['iframe'].map(document.createElement)[0];
...
```

Listing 5.6: Example of higher-order function call in JavaScript.

```
...
let iframe = META.apply(['iframe'].map,
```

```
3                                   ['iframe'],
4                                   [document.createElement]);
5 ...
```

Listing 5.7: Example of an instrumented higher-order function call.

5.2 JavaScript Security Mechanisms

This section describes *strict mode* and the *built-in functions for hardening objects* that can be used to help remove unsafe features from JavaScript, and prevent prototype poisoning attacks.

5.2.1 Strict Mode

One of the first and most important mechanisms for securing a modern JavaScript application is *strict mode*. Executing an application in strict mode changes the language syntax and semantics by adding more warnings and removing some unsafe features [Rau14]. Below we describe security aspects that the strict mode can help with:

- Strict mode disallows the use of the `caller` property of a function. The caller property allows a function to obtain its caller. Accessing a caller of a function can be used by an attacker to inspect the arguments, which is considered insecure [TEM+11, ML10].
- Strict mode disallows the use of `with` keyword. `with` adds an object to the scope chain when evaluating a statement. The properties of this object shadow any variable with the same name in the scope, making it hard to know which variable a given object points to within the statement.

5.2.2 Built-in Functions for Object Hardening.

The ECMAScript 5 specification adds `Object.freeze`, `Object.seal`, and `Object.preventExtensions`, to provide developers with fine-grained control over object properties [VCM10]. Specifically, these functions manipulate *property descriptors* of object properties.

A property descriptor is an object that allows the specification and manipulation of an object's property and its *attributes* [Rau14, Ecm15].

Properties of a property descriptor define the set of attributes for the property. Manipulating the property's attributes allows developers to protect an object's property.

A property descriptor can represent either a *data property* or an *accessor property*. A data property describes a property that has a value. An accessor property describes a property by means of a `get` and `set` functions.

For example, the property descriptor {`value: 'Hello', configurable: false, writable: false`} defines a read-only *data property* whose value is 'Hello'. In this example, `value, writable` and `configurable` are the attributes of the property.

Listing 5.8 defines an *accessor property* with a *getter* method.

```
{
  get() { return 'Hello' },
  enumerable: true,
  configurable: true
}
```

Listing 5.8: Accessor property descriptor example.

JavaScript adds three functions to `Object` for convenient manipulation of property descriptors of object properties. Specifically, the following three functions can be used for hardening objects:

- `Object.preventExtensions(obj)` prevents the addition of new *own* properties to `obj`. However, existing properties can be deleted and new properties can be still added to `obj`'s prototype.

- `Object.seal(obj)` prevents the addition or deletion of properties on `obj`. It also prevents changes to the configuration of existing properties, preventing for example the change of a *value property* to a *data property*, or vice versa.

- `Object.freeze(obj)` prevents addition, deletion, and change of any of the `obj` properties. It also forbids changing the prototype of `obj`.

This object hardening mechanism is limited to the object's own properties. After hardening an object, changes on its inherited properties are still possible. Therefore, the object's integrity may still be compromised.

Challenge	Challenge description	Our Solution
Challenge 1	Automatic Value Coercion	Memoizing property access and calls to toString and valueof.
Challenge 2	Prototype Inheritance Model	Deep-freezing prototype chain of built-ins.
Challenge 3	Dynamic Code Evaluation	Disallowing eval, creation of Function objects and dynamic parsing HTML.
Challenge 4	High-order Built-in Functions	Dynamic instrumentation of High-order Built-in Functions using proxies

Table 5.1: Description of the solutions to the challenges in GUARDIA.

```
1 let proto = { ratio: 42}
2
3 let obj = Object.create(proto);
4 let obj.calculate = function(x){ return this.ratio * x;}
5
6 Object.freeze(obj);
7 proto.ratio = 0;
8 obj.calculate(12);
```

Listing 5.9: Example of compromising the integrity of frozen objects.

To illustrate this problem, consider Listing 5.9. In this example, on line 3 the variable `obj` is assigned a new object with `proto` as its prototype. On line 4, the own property `myOwnProp` is created on `obj`. `myOwnProp` is a method that uses the inherited property `ratio` in its body. On line 6, `obj` is frozen. Freezing `obj` does not have any effect on its inherited property `ratio`. An attacker may compromise the `obj`'s integrity by changing `proto.ratio`'s value as shown on line 7.

5.3 Boosting the Integrity of an Inlined Reference Monitor

This section proposes a solution for each of the 4 identified challenges. Table 5.1 gives an overview of the challenges and solutions proposed and

implemented in GUARDIA. We first describe (partial) existing solutions, and then we introduce the approach that we adopted for GUARDIA. To the best of our knowledge, the solutions to Challenge 1 and Challenge 2 are novel and have not been used before in a runtime monitor based on source code instrumentation.

We will cover the concerns starting from program instrumentation to policy enforcement. We consider that the attacker's code is part of the base application, or that it is dynamically executed using an evaluation sink. The attacker code is instrumented along with the rest of the base program code, or it is a program value dynamically executed by an evaluation sink. We also assume that the applications secured by GUARDIA run in strict mode, preventing the problems discussed in Section 5.2.1.

5.3.1 Dealing with Implicit Value Coercion

Assuming that the reference monitor provides complete mediation, an attacker may attempt to bypass the security policies using information subject to the enforcement mechanism. For example, the arguments given sensitive calls, which are subject to the enforcement mechanism, can be forged to lie to the enforcement mechanism as previously described in Section 5.1.1.

Protecting the enforcement against implicit coercions (challenge 1) has been previously studied in literature [PSC09, MPS12, MFM10]. Phung et al. [PSC09] propose *call-by-primitive-value* to protect the enforcement against implicit coercions. In this approach, the policy developer specifies the type of policy arguments. During enforcement, the arguments are explicitly coerced to the specified primitive types. Coercing the arguments before enforcing the policies and performing the subsequent call ensures that both, the enforcement and the call use the same value. However, *call-by-primitive-value* is limited to primitive arguments only.

Magazinius et al. [MPS12] extended the *call-by-primitive-value*, with *inspection types*. Inspection types enable developers to specify the types of the coercion applied to properties of untrusted objects inspected during policy enforcement. After the enforcement, similar to Phung et al. [PSC09], those coerced arguments are used by the subsequent (wrapped) call. However, inspection types are not intended to enforce (type-check) function arguments like in [KT15]. Nevertheless, in [MPS12] they are used to *cast* a given value to the predefined type.

In this work, we identify that inspection types are a hassle for policy developers, and more importantly, they do not support union types as previously suggested in [MPS12].

Inspection types specification problem We argue that inspection types specifications can become a hassle for the policy developer to write. For example, consider the TypeScript typing specification of `fetch()` shown in Listing 5.10 and a policy over `fetch()` that inspects the `header` property of the `init` argument object. The policy developer needs to know the specific definition of the `HeadersInit` that is the type signature of the `header` property. Otherwise, implicit coercion will happen. This is a burden for developers as the set of policies grows. Developers need to provide inspection types for all arguments passed to all functions subject to the enforcement.

```
1
2  fetch(input: RequestInfo, init?: RequestInit): Promise<Response>;
3
4  interface RequestInit {
5     body?: BodyInit | null;
6     cache?: RequestCache;
7     credentials?: RequestCredentials;
8     headers?: HeadersInit;
9     integrity?: string;
10    keepalive?: boolean;
11    method?: string;
12    mode?: RequestMode;
13    redirect?: RequestRedirect;
14    referrer?: string;
15    referrerPolicy?: ReferrerPolicy;
16    signal?: AbortSignal | null;
17    ...
18 }
```

Listing 5.10: Example of type definitions of web APIs in TypeScript.

Inspection types do not support union types Inspection types do not support union types. While adding union types to the specification language may not be problematic, enforcing polymorphic functions that use them is. Consider the function `doSomething` in Listing 5.11. This function is considered "polymorphic". The developer of `doSomething` expects that the function is called with a `number` or a `boolean` as argument.

However, the function's behavior is conditioned by the coercion applied by `>` and `==` on the argument.

To demonstrate why inspection types cannot be used for preventing implicit coercions in polymorphic functions, consider securing the function `doSomething` with the wrapping mechanism proposed in [MPS12]. The signature of the wrapped function is `wrap(global, 'doSomething', policy, ['number | boolean'])`. The first two arguments are the object and the function being wrapped, the `policy` argument is the enforcement code, and the last argument is the inspection type. In this example, we require a union type (`number | boolean`) because the argument can only be coerced to a number or to a boolean.

```
1 //doSomething(foo: 'number' | 'boolean') : void
2 function doSomething(foo){
3  if(foo > 10 ){
4     console.log( foo * foo);
5    }else if(foo == true) {
6     console.log(`It is ${foo}!`)
7  }
8 }
9 doSomething('9');
```

Listing 5.11: Polymorphic function example.

When `doSomething` is called with the string `'0'`, it is unclear for the reference monitor to *choose* between `number` or `boolean` as the coercion type for `'0'` because both conversions are valid. Therefore, it is not possible to determine which one is appropriate for the argument.

Solution of Challenge 1: Cached Values of Arguments

To overcome the problem that implicit coercions on arguments poses to a reference monitor, we have explored an alternative approach: we *cache values* to have a constant view of the untrusted values given to the enforcement.

The key idea behind cached values is to make call arguments fixed. Arguments will be perceived as constant values, and inspecting (reading) them will not cause any unwanted side-effects. Listing 5.12 shows a revised version of GUARDIA's enforcement mechanism explained in Section 3.3.2.

GUARDIA differentiates between primitive values and objects. Primitive values are cached without wrapping because there is no risk of un-

wanted side-effect when inspecting primitive values, even though they can be implicitly coerced during the enforcement.

Cached values Objects are wrapped in a proxy that memoizes any property access, or any implicit coercion resulting from a `toString` or `valueOf` call. The goal of a `CachedProxy` is to memoize the result of accessing any of its target object properties and to return the memoized values on subsequent accesses. This means that every access to a property during the enforcement returns the same value. Calls to `toString` and `valueOf` cached because the object may be coerced by a built-in, or a language operator. Furthermore, the wrapping is recursive to prevent implicit coercions on the object properties. Finally, cached values cannot escape to the subsequent function call because the program's transparency is affected.

```
1  META = {
2    apply: function (targetFn, ths, args) {
3      const _xs = [];
4      args.forEach(x => {
5        if (typeof x === 'object' && x !== null) {
6          _xs.push(new CachedProxy(x));
7        } else {
8          _xs.push(x);
9        }
10     });
11     if (GG.notify({ type: 'call', targetFn, ths, _xs })) {
12       return Reflect.apply(fn, ths, args);
13     } else {
14       throw new Error('Not allowed!');
15     }
16   }
17 }
```

Listing 5.12: Example of a trap's implementation example using cached values.

Listing 5.12 shows how cached values are used in Guardia's enforcement. As shown in lines 4 – 10, our approach wraps an object into a JavaScript proxy [Ecm15] using a `CachedProxy`.

Using proxies to wrap arguments comes at a price for the policy library developer. Operations relying on pointer equality such as the strict equality operator (`===`) will not produce the necessary result as the proxy

and its target are two different objects. In GUARDIA, `CachedProxy` instances provide an `unwrap()` that enable the access to the proxy's target. Unwrapping should only be used for operations on pointers (e.g., `a === aProxy.unwrap()`) to avoid bypassing the invariants of property accesses. Values returned by `unwrap` are never assigned neither passed as arguments during the enforcement.

5.3.2 Preventing Prototype Chain Poisoning

The starting point for securing a web application is the security of the set of JavaScript *built-in* objects (Object, Function, Array, etc.). We consider built-in objects part of the program's TCB. If the objects that are part of the TCB are not appropriately secured, an attacker can influence the behavior of the reference monitor or any policy-related code by tampering with them. Prototype poisoning attacks applied to built-in objects have been previously identified as *built-in subversion* in literature [MPS12, PSC09].

```
1 //Implementation of the enforce method property
2 enforce: function(inv) {
3          return this.ps.every(function(p) {
4              return p.enforce(inv);
5         });
6  ...
7 }
```

Listing 5.13: Example implementation of the policy enforcement functionality.

We now illustrate how a prototype poisoning attack can be used to bypass GUARDIA's runtime monitor. In particular, Listing 5.13 shows a snippet of GUARDIA's enforcement implementation which can be subject to prototype poisoning attacks by subverting a built-in object. In this example, `this.ps` at line 2 is an `Array` instance. It has the `every` method from the `Array.prototype` object. An attacker can bypass GUARDIA's enforcement by redefining the implementation of `Array.prototype.every` as shown in Listing 5.14. As shown in that listing, any call to `every` returns `true`, preventing the enforcement from behaving correctly.

```
1 Array.prototype.every = () => true
```

Listing 5.14: Example of the re-definition of a built-in.

Preventing built-in subversion has been partially addressed by Magazinius et al. [MPS12]. This work creates copies of all built-ins used by the policy enforcement. For example, essential built-in functions such as `Function.prototype.apply` are copied to avoid the enforcement code from being affected by redefining them. These built-in copies are then used locally by the enforcement mechanism (only), preventing their subversion by an attacker.

However, this approach has some drawbacks in terms of *transparency* and *complete mediation*. First, storing local copies on an object is not transparent for the base program behavior. This is because built-in functions (like `Object.getOwnProperties`, `Object.hasOwnProperty`, etc.), when used by the enforcement on objects with local copies, will render different results with respect to the original application's behavior. Second, the approach is not complete because the programmer needs to specify which functions will be copied manually. However, it is unclear what functions may be called implicitly by the enforcement, so the reference monitor may not control all security-relevant events, which may leave open doors for attackers to perform other subversion attacks.

Solution of Challenge 2: Preventing Prototype Poisoning of built-in in Guardia.

Instead of locally creating the built-in part of the TCB, we ensure the integrity of built-ins by deep-freezing them before the application code is executed. Deep-freezing has previously been implemented in Secure ECMAScript (SES) [1]. Our goal with deep-freezing is to freeze the target object and its prototype chain. Listing 5.15 shows an approximate implementation of the `deepFreeze` function in Guardia.

Starting with the target object, the `while` loop on lines 2–6, freeze all objects in the prototype chain of the target, including the target. With deep-freezing, the copy of built-in functions to descendant objects is not necessary. Also, because methods are not moved around, the semantics of the program is not affected when the object is used by `Object.getOwnProperties, Object.hasOwnProperty`, etc., and hence the transparency is preserved.

[1] Secure ECMAScript: `https://github.com/endojs/endo/tree/master/packages/ses`.

```
1 const deepFreeze = (obj) => {
2     while (obj) {
3         Object.freeze(obj);
4         obj = obj.prototype
5     }
6 }
7 deepFreeze(Array);
8 ...
```

Listing 5.15: Deep freeze implementation for protecting against built-in prototype poisoning.

Deep-freezing the prototype chain of built-ins has, however, a transparency consequence. Applications using polyfill libraries cannot be used as they rely on changes to built-ins' prototypes.

5.3.3 Preventing Dynamic Code Evaluation

The use of dynamic code evaluation is discouraged in nowadays client-side web application development. However, it is not strange to find applications using such a JavaScript feature, so runtime monitors should have a mechanism to solve their issues.

GUARDIA takes a preventive measure and disallows dynamic code execution as suggested by the literature. However, different approaches are required for different dynamic code evaluation construct types. More concretely, our approach disallows the following:

- Dynamic evaluation of code by removing `eval` from the global object `delete window.eval`.
- Dynamic creation of `Function` objects by installing two policies by default when GUARDIA is loaded. Preventing dynamic creation of `Function` objects implies preventing calls to `Function` either as function or a constructor. Listing 5.16 shows the implementation of these policies in GUARDIA.
- Dynamic parsing of HTML by preventing the use of evaluation sinks on the DOM API. Listing 5.17 shows this in GUARDIA. In this example, any attempt to set a string value to be parsed as HTML raises an exception. To prevent further changes to the `innerHTML`, we make it non-whatconfigurable as shown in line 5. Note that the

property can still be read and remains *enumerable* to reduce the impact on the transparency of this action.

```
GG.onCall(Function).deny();
GG.onConstruct(Function).deny();
```

Listing 5.16: Example implementation for preventing calls to `Function` in Guardia.

```
Object.defineProperty(Element.prototype,'innerHTML',{
   set(val){
       throw new Error('Forbidden setting of innerHTML property');
   },
   configurable: false,
   enumerable: true
});
```

Listing 5.17: Example of prevention of dynamic HTML parsing.

5.3.4 Dynamic Instrumentation of Higher-Order Built-in Functions

The mediation problem introduced with higher-order functions was addressed by Magazinius et al. [MPS12] from the perspective of an enforcement mechanism that works by wrapping security-sensitive objects. Clients of security-sensitive objects are provided with a wrapper that mediates the operations performed on the security-sensitive object. Passing wrappers to higher-order functions is safe because the higher-order function always gets the wrapper to the security-sensitive object. This is, implicit calls within the higher-order function body are always mediated by the wrapper.

However, when using a runtime monitor through source code instrumentation as Guardia does, the monitor does not wrap security-sensitive objects. Higher-order functions always get the reference to the security-sensitive callback as discussed previously in Section 5.1.1.4.

Solution of Challenge 3: Dynamic Instrumentation of Higher-Order Functions in Guardia.

In Guardia built-in higher-order functions are dynamically instrumented using JavaScript proxies as shown in Listing 5.18.

In Listing 5.18, higher-order functions are wrapped with a function proxy. The proxy's handler implements the `apply` trap that wraps the callback argument. As shown in line 4, the wrapper of the callback calls the `META.apply` trap from the instrumentation platform, which ensures the policy enforcement.

```
1 Array.prototype.map = new Proxy(Array.prototype.map, {
2     apply: (targetFn, ths, args) => {
3         //Mediating the access to the callback
4         args[0] = (...xs) => META.apply(args[0], [...xs])
5         return Reflect.apply(targetFn, ths, args)
6     }
7 });
```

Listing 5.18: Example implementation of dynamic instrumentation of higher-order built-in functions in GUARDIA.

5.3.5 Securing the Instrumentation Platform

In the context of GUARDIA, there is one sensitive object which should also be controlled to avoid integrity issues: the `META` object. The `META` object needs to be globally accessible, which may introduce security vulnerabilities. More concretely, an attacker may use one or a combination of the following attacks to break the `META`'s integrity:

- The attacker's code may attempt to perform a prototype poisoning attack on `META`.
- The attacker may create a function to inject a malicious implementation of a trap (e.g. apply).

Controlling access to the enforcement platform is critical for the overall functioning of the application as it is part of the TCB. As such, any access to the `META` object should be controlled, and the object itself must be deep-freeze.

5.4 Comparison with the State of the Art

Having introduced GUARDIA and its mechanisms to deal with the 4 identified integrity challenges, we now discuss the runtime monitor of GUARDIA

Approach	Complete mediation	Tamper-proofness	Transparency	Portability
LWSPJS [PSC09, MPS12]	✗	✻	✻	✓
ObjectViews [MFM10]	✓	✻	✻	✓
ConScript [ML10]	✓	✓	✓	✗
WebJail [VADRD+11]	✓	✻	✓	✗
Guardia	✓	✓	✻	✓

Table 5.2: Comparison of security approaches according the features of a reference monitor. A (✓) means that the approach fully supports the characteristic, (✻) partially, and (✗) means it is not supported or not mentioned in the paper.

to related work in terms of the properties of an inlined reference monitor described in Section 6.1.2, complete mediation, tamper-proofness, transparency and portability. The comparison, however, does not include the *correctness* property because it is hard to assess if a reference monitor is correct without a formal specification.

Table 5.2 compares Guardia with related work describing reference monitors in terms of complete mediation, tamper-proofness, transparency and portability. None of the approaches offers a complete answer for all the characteristics. In what follows, we discuss the different trade-offs.

5.4.1 Portability

Portability is an essential aspect of client-side web application security mechanisms. In this regard, approaches that rely on browser modifications such as WebJail and ConScript are not portable. Guardia, as the rest of the approaches, does not rely on browser modifications, and therefore they can secure applications running on different browser. However, portability is not only a matter of changing or not changing the browser interpreter, because browsers often have non-standardized features, which can hinder the portability of an approach that relies on those features.

5.4.2 Complete Mediation

Complete mediation is only achieved if the monitor can control all security-relevant events. In the context of client-side web applications, complete mediation is affected by the different aliases to the same security-sensitive built-in resource in the browsing environment. As discussed in Section 5.3.1, approaches based on shallow object wrappers (i.e., proxies) do not have a good solution for the aliasing problem. Complete mediation of those approaches can be only achieved if all accesses to the sensible resource are wrapped, which cannot be automated.

ObjectViews [MFM10] proposes a recursive wrapper approach (i.e., membranes) which can only achieve complete mediation for user-defined objects. This system focuses on the safe sharing of user-defined objects between untrusted principals (e.g. frames) within a web application. However, the paper does not tackle how to handle sharing built-in functions.

Note that complete mediation is easier to achieve if the reference monitor is part of the interpreter. ConScript [ML10] and WebJail [VADRD+11] develop a *deep* advice system for enforcing security policies added to JavaScript by modifying the browser interpreter. These approaches extend JavaScript with new built-in functions to support aspect-oriented policy definitions. Because their approaches have access to the interpreter internals, their advice system can easily weave heap objects representing sensitive functions instead of wrapping an access path like LWSPJS [PSC09, MPS12]. By doing that, they ensure that all accesses to the secured function are mediated.

In our approach, all source code is instrumented, and the *unsafe* features (i.e. `eval, innerHTML`, etc.) of the browsing context that allows the addition of code dynamically are blocked, disabling the possibility of evaluating new (non-instrument) code. Because all access paths to possibly aliased function calls are instrumented (if used in the application), our approach can ensure complete mediation without requiring modifications to the interpreter.

5.4.3 Tamper-proofness

Section 5.3 discussed solutions found in related work for the 5 integrity challenges and introduced our approach for each of them. Table 5.3 summarizes our findings regarding the tamper-proofness of the studied ap-

Approach	Prot. poisoning	Aliasing	Imp. Coercion
LWSPJS [PSC09, MPS12]	✓	✻	✻
ObjectViews [MFM10]	✻	✓	✓
ConScript [ML10]	✓	✓	✻
WebJail [VADRD+11]	✻	✓	✗
Guardia	✓	✓	✓

Table 5.3: Comparison approaches according attack vectors. A (✓) means that the approach fully covers the vulnerability, (✻) partially and (✗) means it is not supported or not mentioned in the paper.

proaches described in Section 5.3 according to the different kinds of attacks that can be performed. As shown in the table, our approach can deal with the three of them. In what follows, we compare the different works per attack.

Prototype poisoning

Perhaps the most discussed attack in the literature is prototype poisoning due to its severity in a security context. LWSPJS [PSC09, MPS12] address the problem by creating local copies of built-in functions used by the wrappers, therefore, avoiding attacks targeting the wrappers. The authors of WebJail [VADRD+11] the same approach as LWSPJS for securing their advice function. However, their code is still vulnerable to prototype poisoning by tampering with the `Array.prototype` object. In ConScript [ML10], the authors extend the language with a type system to enforce two properties: reference isolation (i.e., kernel objects should flow to user code) and access path integrity of explicitly invoked functions (i.e., functions invoked by policies must be statically known to avoid prototype poisoning attacks). ObjectViews [MFM10] cannot offer a solution for poisoning attacks by themselves. Instead, the authors rely on the existence of a trusted platform to do so. In our work, we developed a solution to the prototype poisoning problem using modern JavaScript's security primitives. Instead of copying functions around, we keep all of them in their respective objects. However, the prototype chain of the

objects (i.e. built-in) is frozen.

Aliasing

Regarding the *aliasing*, only the shallow wrapping mechanisms LWSPJS [PSC09, MPS12] suffer from this problem. This is because their interposition mechanism is done on a per-object basis. In the case of Object Views, their approach recursively wraps the objects in the system to achieve complete mediation and therefore prevents the problem of unsecured aliases. Aliasing is not a problem for GUARDIA's enforcement as it mediates all program operations. Therefore, any operation on an alias of a secured object is subject to enforcement.

Implicit Coercions

Section 5.3.1 elaborated on the issues of LWSPJS to deal with implicit coercions. Our work solves them by caching values.

In CONSCRIPT, the authors partially addressed implicit coercions because they do not conceive objects passed as arguments. Moreover, it is not clear how their `toPrimitive` utility decides the type of its result without giving any context.

The authors of WEBJAIL mention the problem of calling `toString` on untrusted values during the enforcement. However, they do not propose mitigation for this problem.

OBJECTVIEWS [MFM10] protect against parameter forgery by recursively wrapping call arguments values. Those wrappers have two policies: (i) they prevent any trusted value from flowing into the wrapped argument to avoid leaking sensitive information, and (ii) any method of the argument called within the secured function will be executed in the local scope of the argument value. Our recursive wrapper approach is different to this approach since our `CachedProxy` creates a constant view of the argument value. Furthermore, during the enforcement, we do not call any function on the cached values, avoiding any leak from the reference monitor to the untrusted argument.

5.4.4 Transparency

Transparency of the application execution can be affected from different angles while ensuring system security. For example, in LWSPJS, the application transparency may be affected while copying built-ins. This is because the behavior of reflective functions like `Object.getOwnPropertyNames` (if called upon the receiver of those built-in) is affected by those property additions. Transparency is also affected during automatic coercion of function arguments. This problem also affects our approach, but it is localized to the enforcement to prevent liar objects. In this work, we gave up transparency for coercing argument values to support complete mediation and tamper-proofness.

5.5 Conclusion

This chapter presented an analysis of the 4 challenges that JavaScript and the browsing context pose to an inlined reference monitor approach for a client-side web application. The chapter discusses current approaches to solve those issues and presents *cached values* as a novel alternative to protect a reference monitor based on source code instrumentation against parameter forgery attacks using implicit coercions.

In conclusion, having a reference monitoring for the client-side covering complete mediation, integrity, and transparency, while being portable is hard –if not impossible– to achieve in a browsing environment. To uphold important properties such as integrity, an inlined reference monitor needs to make a trade-off with transparency.

Chapter 6

Deriving Static Analysis for Web Applications

So far, this dissertation focused on runtime enforcement of access control (AC) and information flow control (IFC) security policies. However, AC and IFC policies can also be *statically* analysed. As discussed in Section 2.4.3, static and dynamic analyses each have their own advantages and disadvantages, which makes them complementary tools in a secure application development life cycle. In a client-side web application development context, Static Application Security Testing (SAST) and Runtime Application Security Protection (RASP) tools are used to ensure application's security. Developers use SAST tools (such as GATEKEEPER) in early stages of development for the static analysis of applications to detect XSS injection vulnerabilities, etc. However, since static analysis may not catch all policy violations of JavaScript applications using dynamic features, RASP tools (such as GUARDIA or JSFLOW) are used to detect the same set of vulnerabilities [FBJ+16, HL06]. As such, both SAST and RASP tools are used in the development life cycle for detecting vulnerabilities, and more importantly, they are often used to detect the same kind of vulnerabilities (i.e., policies). Therefore, it could be reasonably expected to find approaches able to statically verify and dynamically enforce the *same set* of security policies in an *efficient* and *safe* way. Unfortunately, there is no such approach.

In this chapter, we propose an approach to *safely* and *efficiently* derive a static analysis from a given dynamic analysis. Starting from a

dynamic analysis component that relies on source code instrumentation, our approach derives a static analysis component for statically verifying the *same* set of policies, thereby avoiding the re-implementation of policy specifications and, more importantly, *enforcement* code. Reusing the policy enforcement code prevents semantic mismatches between the static and dynamic context in which the policies are enforced. Moreover, it offers developers a static analysis implementation for free, relieving them from the hurdles of writing any specification and code on top of static analysis tools themselves.

Specifically, we derive the SAST components from two RASP components based Guardia (cf. Chapter 3) and Gifc (cf. Chapter 4) which enforce Access Control and Information Flow Control security policies, respectively. The core idea of our approach is to use a *two-phase abstract interpretation* in the static component (SAST) that analyses the target application in the first phase, and in a second phase abstractly executes any required policy enforcement code. Splitting the analysis into two phases avoids the complexity of analysing the complete instrumented application in one go. More importantly, it enables developers to use separate and different analysis configurations for each phase for striking the right balance between performance and precision. To keep the static analysis tractable, the base program typically is analysed with lower precision than the precision with which the policy enforcement code is abstractly executed. Applying our *two-phase* static analysis results in a set of code locations of expressions that violate the policies. These code locations can, for example, be integrated into a "security linter" to assist developers with verifying their application.

In summary, the key contributions of this chapter are:

- An approach to safely and efficiently derive a static analysis from dynamic analysis for a single set of policy specifications (which are enforced via code instrumentation).
- A two-phase abstract interpretation for statically analysing a base program (phase 1) and its instrumentation (phase 2) that enables a better trade-off between precision and performance than a single static analysis of the instrumented code.
- The instantiation of the two-phases approach for specifying, enforcing, and verifying AC and IFC security policies for JavaScript web

applications from related work.

6.1 Motivation

This section motivates the need for deriving a static analysis from the security analysis perspective. To this end, Section 6.1.1 introduces an example client-side web application to illustrate the need for both RASP and SAST tools. Finally, we describe the main challenges faced when integrating SAST and RASP tools (cf. Section 6.1.2).

6.1.1 Running Example

Consider a client-side web application that uses a password strength checker component to enforce a password policy. Listing 6.1 shows the JavaScript code corresponding to such a component. [1] The `chkpass` function (lines 5 to 17) enforces that a password has a minimum length, and contains both numbers and symbols. When the password meets the requirements, the checker returns `true`, indicating a quality password; if not, then the checker returns false.

Clearly, the application developer expects the component to assess the password strength, and nothing else. For example, developers assume that the password is processed locally, and that the component does not communicate information to the network. However, the component leaks sensitive information since it also makes a request that sends the password to a third-party server using function `fetch`(line 9).

Preventing the component from making this request can be achieved by specifying a security policy that *"disallows calling `fetch`"* and using a RASP tool to enforce this policy. However, RASP enforces policies at runtime, and therefore it can only cover certain execution paths. For example, function `fetch` is only executed when the length of the password is greater than 8 and `symFlag` and `numFlag` are `true` (lines 12-13). The RASP component will stop the application's execution if line 13 is reached, and report a policy violation, but it will not stop and report it if the control flow does not reach that line (e.g., a run of the application in

[1] For clarity, the component is implemented as part of the application, but it could be included through a `script` tag pointing to the component's implementation on a third-party server.

which a user inputs a 7-character password), even though the security vulnerability is present in the source code of the application. Although RASP will prevent the component from leaking the password, it will also stop the application's execution at line 9, which implies that clients of the application will not be able to use the application at all.

```
1  <html><body><script>
2        const hasDigit = p => /\d/.test(p);
3        const hasSymbol= p => /\W/.test(p);
4
5        function chkPass(pass) {
6          if (pass.length >= 8) {
7            const flags = hasDigit(pass) && hasSymbol(pass);
8            if (flags){
9                fetch("http://evil.com?payload="+pass);
10               return true;
11           }
12           else {
13               return false;
14         }
15       }
16         return false;
17       }
18       function check(event){
19          chkPass(document.getElementById("pass").value);
20       }
21     </script>
22      ...
23         <input type="password" id="pass" onchange="check(event)" />
24      ...
25 </form></body></html>
```

Listing 6.1: Password checker component in JavaScript.

On the other hand, a SAST tool is capable of exploring *all* application execution paths, and therefore report the vulnerability to the developers, who can fix it before deploying the application. However, SAST alone cannot always precisely detect errors, as static analyses typically over-approximate. For example, changing the `fetch` expression on line 9 by the expression `window['f'+(+[])[4]+'t'+'c'+'h']`, where the value of the property being accessed cannot be statically determined, may result in a false negative. Worse, most static analysers for JavaScript are intentionally unsound to some degree in order to remain tractable [LSS+15]. Since a SAST tool may miss certain policy violations, policies need to be

also enforced using RASP components. Nevertheless, static analyses help developers detect and fix as many security vulnerabilities as possible in the early stages of the software development cycle. In conclusion, developers need both RASP and SAST tools for the verification and enforcement of application-level security policies.

6.1.2 Challenges for RASP and SAST Integration

The main challenge for integrating SAST and RASP tools is ensuring that security policies have *identical semantics* in both the static and dynamic contexts in which they are verified. Specifying the same policies in two different tools, once for SAST and once for RASP, may unnoticeably introduce subtle differences in semantics. A policy specification using different tools is also cumbersome, as developers must learn different policy specification languages. More importantly, the dynamic policy enforcement code and its static counterpart have to be maintained in parallel.

Reimplementing SAST and RASP tooling, or attempting to reuse parts of their underlying implementation, is also not a viable option: the result of static analysis is some finite, abstract model of the runtime behavior of an input program, which is significantly different from the information available in a browser runtime. The static analysis code is implemented using abstractions and data structures specific to the static analysis tool and its APIs, making it hard to reuse this information by a JavaScript runtime enforcement mechanism. Furthermore, the reimplementation may also introduce semantic mismatches or other errors between implementations for the two different contexts.

In the context of security, some approaches decouple policy specification from actual verification and enforcement through the use of a *security policy language*. However, this decoupling does not facilitate the development of complementary SAST and RASP tools, because any additional or reused implementation still faces the same problems mentioned above.

Approaches that rely on source code instrumentation [ADF11, SKBG13, SNG18] could allow the derivation of SAST from RASP by analyzing the base program with the runtime enforcement code included. However, analyzing the instrumented application makes the static analyzer's task even harder, as in this case the code to be analyzed contains both the policy enforcement code and the target application. More importantly, both the logic contained in the target application *and* the en-

forcement code are analyzed under the same configuration, precluding the experimentation with different analysis configurations to obtain a suitable trade-off between soundness, precision, and speed. All of this makes the static analysis of an instrumented application impractical.

6.2 Deriving SAST from RASP

This section describes a novel *two-phase abstract interpretation* technique that enables developers to configure a suitable trade-off between speed and precision when using the derived SAST component. Deriving a static analysis through abstract interpretation is *safe* because both the dynamic analysis and the (derived) static analysis are based on the *same* specification code in JavaScript, so no semantic mismatches between the two arise. It is also *efficient* because analysis developers do not need to reimplement the analysis, as the dynamic analysis implementation is reused without requiring developers to adapt it for static analysis.

6.2.1 RASP Through Meta-programming

Before delving into the details of our two-phase abstract interpretation technique, we will describe the RASP component's main features. We use Guardia as RASP component to derive a SAST component for AC policies in Section 6.5. Unfortunately, Gifc uses JavaScript proxies, which are not supported by the current implementation of the abstract interpreter used for implementing our two-phase approach. As such, we reimplemented an IFC runtime monitor that only uses source code instrumentation which we describe later in this section.

In what follows, we will refer to the policy enforcement mechanism as *meta code*, and to the target application to be secured as the *base program*. In our RASP component, the base program includes the meta program providing the enforcement mechanism. [2] The meta-program also defines a set of *traps* on a handler object. We assume that the handler object can be accessed in some well-defined manner; in our implementation, we use a property named `META` on JavaScript's global object. The traps are equivalent to the ones described in Chapter 3 (i.e., a method that encapsulates

[2]Note that the base program may also perform additional initialization and configuration (i.e., add security policy specifications) before executing any code.

the behavior that must be executed when a specific program operation occurs).

Even though the base program includes and configures the meta-program that defines traps for program operations, these traps still need to be explicitly linked to the base program at run time. Linking the traps is done with source code instrumentation. As a result of the instrumentation, the base program is translated into an equivalent instrumented program with an inline Execution Monitor (EM). The EM is responsible for calling the traps and performing the base program operations.

We assume that the meta program does not influence the base program's behavior in any way except for halting the application's execution when policies are violated. The meta-program should not, for example, change the state of the base program by changing the value of variables or object fields. However, the meta-program is allowed to modify its own state (the meta state) and perform additional side effects such as logging, assigning a variable, etc.

In what follows, we describe the necessary details of the two RASP components used for deriving SAST components for verifying AC and IFC policies. Those components are later used for the experiments and evaluation of our approach in Section 6.5.

6.2.1.1 Access Control RASP

The access control RASP component employs GUARDIA's source code instrumentation-based enforcement explained in Section 3.3. In what follows, we will use a concrete policy example to illustrate the output of a RASP component which is given to our two-phase approach.

Consider a more interesting version of the policy described in Section 6.1, *"Disallow calling `fetch` more than three times"*, and its enforcement code shown in Listing 6.2 and Listing 6.3, respectively. The `META` object is the handler object that defines an `apply` trap for intercepting function calls in GUARDIA. This trap is invoked whenever a function call is performed in the instrumented base program, and the trap's body provides the meta behavior for enforcing the policy specification. In this example, the meta behavior checks whether the `fetch` function is being applied, and if so it increments the `counter` property, which is part of the internal state of the policy. If `counter` is equal or larger than 3, then the meta code signals that program execution should *halt* (line 7), otherwise

the execution *proceeds* (line 9).

```
1 GG.onCall(fetch).moreThan(3).deny();
```

Listing 6.2: Implementation using GUARDIA of *"Disallow calling `fetch` more than three times"* policy.

```
1  const META = {
2   PROCEED: true,
3   HALT: false,
4   counter: 0;
5   apply: function (fn, ths, args) {
6     if(fn === fetch && this.counter++ >= 3){
7        return META.HALT;
8     }
9     return META.PROCEED;
10  }
11 }
```

Listing 6.3: Example enforcement code for the policy declared in Listing 6.2.

We now illustrate the result of applying the sample AC policy to the password checker application from Section 6.1. Consider as base program a subset of this password checker application (shown in Listing 6.4). Applying GUARDIA results in the instrumented base program as shown in Listing 6.5.

```
1 ...
2 if (flags) {
3   fetch("evil.com?payload="+pass);
4   return true;
5 }
6 ...
```

Listing 6.4: Snippet from Listing 6.1.

```
1 ...
2 if (flags) {
3   EM.apply(fetch, window, ["evil.com?payload="+pass]);
4   return EM.return(true);
5 }
6 ...
```

Listing 6.5: Instrumented version of Listing 6.4.

Instrumenting the base program syntactically links the program operations to the operations defined on the Execution Monitor(EM). The EM is responsible of calling the corresponding trap on the `META` object. For example, Listing 6.6 defines the "monitoring" operation `apply` for function calls. The function is responsible of applying the trap (line 3) and executing the base program operation (line 4).

Note that the enforcement for the policy in Listing 6.2 performs side-effects to maintain its internal state as shown in Listing 6.3 at line 6. However, those side-effects are transparent to the base program, except when the policy is violated.

```
1 const EM = {
2     apply: function(fn, ths, arg){
3         if(META.apply(fn, ths, args)){
4             return fn.apply(ths, args);
5         }else{
6             throw new Error();
7         }
8     }
9 }
```

Listing 6.6: Example implementation of the EM.

6.2.1.2 Information Flow Control RASP

We now describe the main features of our IFC RASP component that is only based on source code instrumentation. The code of Listing 6.7 shows an excerpt of an IFC policy library for tracking information flow for binary operations, function applications, and variable writes based on taint analysis (cf. Section 2.3.2.4). In the shadow execution (i.e., meta program execution), values carry the taint (i.e., the security level) of their corresponding concrete values in the base program execution. Booleans are the only shadow values allowed, representing secret and public information, respectively. The `binary`, `apply` and `write` functions are examples of the traps that are called when the corresponding program operation is about to be executed in the base program. Functions are the only values that can be considered sinks in this RASP component. Therefore, the `apply` trap is crucial to enforce IFC policies on each function call.

In our RASP component for enforcing IFC policies, the base program interacts with the policy library (i.e., meta code) using an interface con-

```
const META = {
  stack: [],
  binary: function (op, l, r) {
    let right = this.stack.pop();
    let left = this.stack.pop();
    this.stack.push(left || right);
  },
  literal: function (l) {
    this.stack.push(false)
  },
  write: function (vName, value) {
    this.writeVar(vName, this.stack.pop());
  },
  apply: function (fn, ths, args) {
    let taint = false;
    for (let a of args) {
      taint = taint || this.stack.pop();
    }
    return !(isSink(fn) && taint);
  }
}
```

Listing 6.7: IFC policy library example.

sisting of two functions: `taint(x)`, and `sink(x)`. The `taint(x)` function tags its argument as sensitive, while the `sink(x)` function registers its argument as a public sink of information. For example, `sink(fetch)` will prevent the release of any tainted data as part of an argument of a `fetch` call. Similar to the AC RASP component previously described, the state of an IFC policy is maintained as internal state of the library. Since our IFC RASP component is based on taint analysis, it does not cover JavaScript features that can cause hidden implicit flows, as GIFC does. We further discuss implicit flows in Section 6.6.

6.2.2 Deriving SAST From RASP Using a Two-Phase Abstract Interpretation Approach

The previous section explained how AC and IFC policies can be enforced by runtime monitors that intercept program operations to determine whether they violate a policy. This section introduces our novel two-phased abstract interpretation approach to derive a SAST compo-

nent from a RASP component. Our approach consists of a static analysis of the base program in the first phase, and the triggering and abstract execution of the associated meta program (i.e. enforcement code) in the second phase. The key benefit of our approach is that the enforcement source code from the RASP component is reused *without modification* within the SAST implementation.

In what follows, we describe the two phases in more detail, using the password checker from Section 6.1 as a running example. We use Listing 6.4 as base program and the AC policy library from the previous section as meta program (the core of which was shown in Listing 6.5). Section 6.3 and Section 6.4 formally describe each phase using a small-step operational semantics.

The first phase of our approach performs an abstract interpretation of the base program, resulting in a *control-flow graph* of the base program called flow graph in the remainder of the chapter. To illustrate the concept of a flow graph, consider Figure 6.1 and Figure 6.2 showing the flow graph of the concrete and abstract evaluation of the base program shown in Listing 6.4, respectively. The graph nodes (depicted as ovals) denote the different program states visited by the concrete machine to evaluate the base program. Pink ovals represent states where the abstract machine is about to perform a program operation. Green ovals represent states where the machine just computed a value and is ready for continuing evaluation with that value. Yellow ovals are terminal states holding program result values. Each graph state in a flow graph can be considered a snapshot of the program (syntactic node being evaluated, store, stack, ...) resulting from the application of the different transition rules from our small-step operational semantics (detailed later). For example, the figures include the rules E-Simple and E-Fun-Call responsible for the machine transitions.

The second phase explores the flow graph resulting from the first phase to detect security-relevant operations that must be trapped, and for which the appropriate handlers must be triggered. In our running example in Listing 6.4, this entails detecting function calls of `fetch` to enforce the AC policy (from Listing 6.2). We call the component that examines the flow graph for detecting policy violations the *Execution Explorer* (EE).

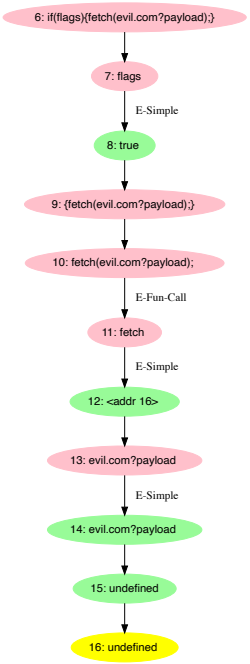


Figure 6.1: Flow graph schematics for the concrete evaluation of Listing 6.4.

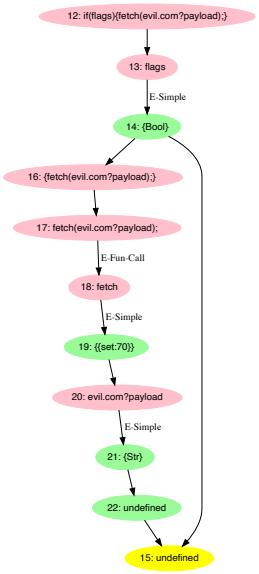


Figure 6.2: Flow graph schematics for the abstract evaluation of Listing 6.4.

From a flow graph, security-relevant program operations can be identified by inspecting the syntactic information contained in the states. Because the base program includes the analysis library (i.e., the meta code) and a graph state is a snapshot of the base program execution, the handler object (`META` in our implementation) is contained and available in each state. This means, that the analysis can inspect the state of the interpreter at any particular state transitioned by the abstract interpreter. In this particular case, the analysis can access to the abstract value to which `META` points-to.

To ensure that the abstract interpretation of meta code results in a useful approximation of the concrete execution of the meta code in the handlers at run time, the Execution Explorer (EE) must be modeled after the Execution Monitor (EM). Therefore, every operation that is intercepted at runtime by the EM should be statically detected by the EE as well.

The identification of security-relevant operations and the availability of `META` are necessary conditions to fulfill our safety property. However, to reach a sufficient condition the meta program semantics (i.e., the policy library enforcement) must be identical in the static and dynamic contexts.

Whenever the EE reaches a trap, its abstract interpretation is triggered, corresponding to concretely executing the trap in a RASP mechanism. This abstract interpretation is parameterized with the program operation information that is extracted from the current state. For example, Figure 6.3 shows a procedural view of the second phase of our abstract interpretation for the example of Listing 6.4 (Figure 6.1). When the EE reaches a function call state, a new abstract interpretation of `META.apply` is triggered (see ② in Figure 6.3). This interpretation is given the function pointer (`fetch`), the `this` value, and the arguments of the call ($[url]$), resulting in a new flow graph in which the terminal state represents the value resulting from the abstract execution of the meta program. When a policy is violated, the result subsumes *META.HALT*, and the EE collects the source code location from the current state's syntactic information.

Central to our approach is the fact that the meta program is abstractly interpreted using a different configuration than the one used for the analysis of the base program. This is crucial for enabling suitable trade-offs between analysis speed and precision than when analyzing the instrumented application in one single phase. Precision and speed can vary according

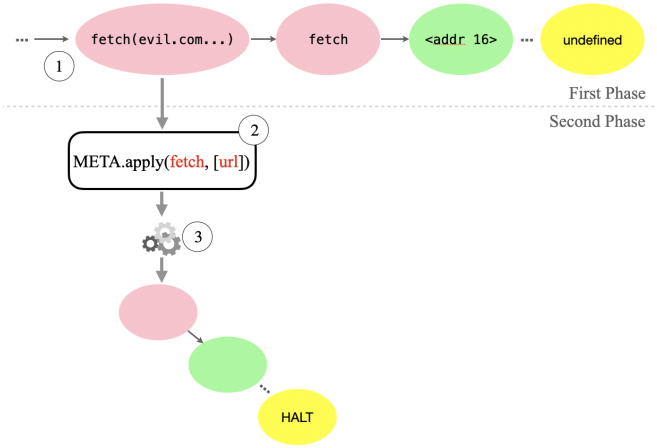


Figure 6.3: Procedural view of the second phase abstract interpretation.

to the development stage in which the static analysis component is used. For example, when used in an IDE, developers would like violations or bugs to be reported faster than when analysis is applied during a nightly build, when higher precision may be desired instead.

6.3 Phase 1: Static Analysis of Base Programs

This section elaborates on our two-phase abstract interpretation approach. Specifically, the work on this section is based on the calculus of my co-promotor Prof. Dr. Jens Nicolay described in [NSDD17]. That work presented JS_0, a core functional language that models a subset of JavaScript, and a static analysis that models the execution of JS_0 programs as a flow graph from which information about control and value flow, and effects can be extracted. This section presents the syntax and semantics of JS_0 based on its original specification in [NSDD17]. We focus on the relevant features to understand the contributions of this work, and include the details on the components and operations of the abstract machine in Appendix C.2.

6.3.1 Syntax of js_0

JS_0 is a core functional language that features objects as maps, higher-order functions, assignment, and prototype-based inheritance. The origi-

Figure 6.4: Input language JS_0.

$$
\begin{aligned}
e \in \mathsf{Exp} ::= & s \mid f \mid v\texttt{(}s\texttt{)} \mid s_0\texttt{.}v\texttt{(}s_1\texttt{)} \mid \texttt{new } v\texttt{(}s\texttt{)} \mid \texttt{return } s \mid v\texttt{=}e \mid s\texttt{.}v \\
& \mid s\texttt{.}v\texttt{=}e \\
s \in \mathsf{Simple} ::= & v \mid \texttt{this} \\
f \in \mathsf{Fun} ::= & \texttt{function (}v\texttt{)\{var } v_h\texttt{; } e\texttt{\}} \\
v \in \mathsf{Var} = & \text{ a set of identifiers}
\end{aligned}
$$

nal syntax of JS_0 is depicted in Figure 6.4. Focusing on essential JavaScript features simplifies the presentation of our approach, although the features of JS_0 still are sufficient to show the applicability and generality of our approach. For validating our approach (Section 6.5) we used an implementation of JS_0 that supports a substantially larger subset of JavaScript features such as conditionals, variable declarations, loops, exceptions, and type coercion. This implementation also defines parts of the standard built-in Javascript objects and functions.

In JS_0, all program elements have a unique label to distinguish between different occurrences of the same syntactic expression (for example, to differentiate between the two references to `pass` in line 7 in Listing 6.1).

6.3.2 Semantics of js_0

We define the small-step semantics of JS_0 as an abstract machine [FF87] that transitions between states. This machine, based on the CESIK$^\star\Xi$ abstract machine introduced in Johnson and Van Horn [JVH14], operates on abstract values but can be configured to express concrete semantics.

State-space Components. The space-state of the abstract machine semantics is shown in Figure 6.5. We now describe its main components. A machine state is either an evaluation state (e) or a continuation state (**ko**). In an evaluation state, the machine evaluates an expression e in an environment ρ. In a continuation state (**ko**), the machine is ready to continue evaluation with a value d it has just computed. An environment ρ maps variables to addresses. A store (σ) maps addresses (a) to values.

Figure 6.5: State-space of the abstract machine semantics.

$$
\begin{array}{rll}
\varsigma \in \mathit{State} & ::= \mathbf{ev}(e, \rho, \sigma, \iota, \kappa, \Xi) & \text{[eval state]} \\
& \mid \quad \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi) & \text{[kont state]} \\
\rho \in \mathit{Env} & = \mathsf{Var} \rightharpoonup \mathit{Addr} & \text{[environment]} \\
\sigma \in \mathit{Store} & = \mathit{Addr} \rightharpoonup (D + \mathit{Obj}) & \text{[store]} \\
d \in D & = \mathcal{P}(\mathit{Addr} + \mathit{Prim}) & \text{[value]} \\
\delta \in \mathit{Prim} & = \{\mathbf{undef}, \mathbf{true}, \mathbf{false}\} & \text{[primitive value]} \\
\omega \in \mathit{Obj} & = (\mathsf{Var} \rightharpoonup D) \times (\text{“proto”} \mapsto D) & \\
& \quad \times (\text{“call”} \mapsto \mathcal{P}(\mathit{Callable})) & \text{[object]} \\
c \in \mathit{Callable} & ::= (f, \rho) & \text{[callable]} \\
\iota \in \mathit{LKont} & = \mathit{Frame}^* & \text{[frame]} \\
\phi \in \mathit{Frame} & ::= \mathbf{as}(v, \rho) & \text{[assignment frame]} \\
& \mid \quad \mathbf{st}(s, v, \rho) & \text{[property store frame]} \\
\kappa \in \mathit{Kont} & ::= (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma) & \text{[meta-continuation]} \\
\Xi \in \mathit{KStore} & = \mathit{Kont} \rightharpoonup \mathcal{P}(\mathit{LKont} \times \mathit{Kont}) & \text{[stack store]} \\
a \in \mathit{Addr} & \text{ is a set of addresses} & \text{[address]}
\end{array}
$$

In our formalism, we only consider two types of values: pointer values that correspond to addresses (i.e., pointers to an object), and primitive values **undef**, **true**, and **false**. Other primitive value types such as numbers, strings, etc., can be added by extending the set *Prim*. $\mathcal{P}(X)$ denotes the *power domain* of set X. An object (ω) is represented as a map from properties to values. Two internal object properties "call" and "proto", distinct from regular properties, exist to implement function objects and object prototypes, respectively.

The stack is modeled as a combination of an intraprocedural continuation (ι) and either an interprocedural continuation (κ) or the empty continuation (ϵ). Interprocedural continuations play the role of *execution contexts* that are generated at call sites. An intraprocedural continuation also serves as a stack address pointing to underlying stacks in a stack store (Ξ). The empty continuation corresponds to the root context, which is created at the start of program evaluation.

Transition Relation. The semantics of the different syntactic elements of JS_0 are implemented as transition rules from evaluation states (e). For example, the semantics of a method call, implemented by the E-METHOD-CALL rule in Figure 6.6, is applied whenever the machine reaches a state where e is a method call expression. Transition rules may also use auxiliary relations or evaluation functions. For example, the function *evalSimple* is used to evaluate different types of simple expressions in JS_0, e.g. literals, references and the *this* expression. The full specification of JS_0 rules, auxiliary functions, and relations can be found in Appendix C.2.

Figure 6.6: E-METHOD-CALL rule implementation.

E-METHOD-CALL

$$\frac{\begin{array}{c} d_{\text{this}} = \mathit{evalSimple}(s_0, \rho, \sigma, \kappa) \qquad d_{\text{arg}} = \mathit{evalSimple}(s_1, \rho, \sigma, \kappa) \\ a_{\text{this}} \in d_{\text{this}} \qquad d_f \in \mathit{lookupProp}(v, a_{\text{this}}, \sigma) \\ a_f \in d_f \qquad \omega_f = \sigma(a_f) \qquad c \in \omega_f(\text{“call”}) \qquad \kappa' = (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma) \end{array}}{\mathbf{ev}(\underbrace{[\![s_0\,.\,v(s_1)]\!]}_{e}, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa')}$$

6.3.3 Concrete and Abstract Evaluation

A program can be evaluated by calling the function `eval` with three arguments: `eval(`e`, `α`, `*alloc*`)`. e is the expression or program to evaluate. This expression is injected into an initial evaluation state, from which all other reachable states will be computed by the abstract machine. Function α converts concrete values into abstract values. Function *alloc* is used for generating addresses for allocation into the value store (σ). The result of evaluating e is a flow graph in which nodes are reachable states, and edges are transitions between states. The flow graph therefore represents the steps taken by the machine during the abstract interpretation.

Functions α and *alloc* can be used to control the precision of evaluation, and enable the machine to express both concrete and abstract semantics.

For *concrete semantics*, function α is the identity function, so that the machine operates on concrete values. Additionally, store allocator function σ must always return fresh (i.e., unused) addresses. This configuration yields an abstract machine that computes a concrete interpretation of the given program. Analyzing a program using concrete semantics is equivalent to the execution of the same program in a standard interpreter.

Because concrete semantics can yield large or infinite executions (and, therefore, large or infinite flow graphs), *abstract semantics* are used when performing static analysis to guarantee finite and tractable flow graphs. In abstract semantics, abstraction function α maps concrete values onto elements of a (bounded) lattice, which represents the (finite) abstract domain of those values. For example, function α can map concrete values to their type. The store allocator function σ additionally must choose addresses from a finite set, and may therefore return addresses that are already used in the store.

To illustrate parameterization for concrete and abstract semantics, consider again the base program in Listing 6.4 as example. Analyzing this program using concrete semantics with variable `flags` equal to `true`, results in the graph shown in Figure 6.1. The flow graph only includes the states reached by the machine during the analysis of `if` statement test expression and the consequent branch (which is the branch taken by the machine after the test expression). As the figure shows, transition rule E-SIMPLE (defined in Figure C.1 in Appendix C) is applied to state 7, meaning that the program is about to perform an evaluation of a simple expression (i.e., identifier operation in this case). The transition relations

of our abstract machine (described in Appendix C.2.2) are expressed in small steps, allowing us to obtain a very detailed graph in which program operations can be easily identified. Figure 6.2 is the flow graph resulting from the analysis of Listing 6.4 using abstract semantics. As Figures 6.1 and 6.2 make clear, different machine parameterizations influence the number of reachable states and the transitions between these states, but not the kind of program operations that are supported, and the manner in which they can be detected in the resulting flow graph.

6.4 Phase 2: Static Analysis of Meta Operations

The second phase of the proposed *two-phase* (2PH) approach explores the flow graph resulting from the first phase to perform the *target analysis* embedded in the meta program (in our case, policy enforcement code for verifying AC and IFC policies). Exploration of the flow graph for evaluating the appropriate meta program operations is the responsibility of the Execution Explorer (EE), first described in Section 6.2.2. The *same* abstract machine evaluator used for generating the flow graph in the first phase is used for abstractly executing the meta code (which is also JavaScript). The semantics of JS_0 therefore not only form an *operational* foundation for a static analysis of the base program, but also for a *result-oriented* abstract interpretation of the meta code in this second phase to determine the outcome of the overall analysis.

Although the same abstract machine is used for performing abstract interpretation, the key idea of splitting the abstract interpretation in two phases is to enable the use of a different parameterization for each individual phase. In terms of code size and complexity, the base program is usually significantly larger, and therefore typically is analyzed with lower precision than the meta program. The meta program containing the analysis code, on the other hand, is usually significantly smaller and less complex than the base program, and is also under tighter control of the analysis developer. To remain faithful to the intended analysis semantics, the meta code therefore typically is evaluated with significantly higher precision than the base code. In fact, in our experiments we configured the second phase to be as precise as possible. More concretely, this configuration is as close as possible to concrete semantics (e.g., with full precision), only losing precision due to control and abstract values

introduced in the first phase.

The rest of this section describes how our analysis intercepts base program operations and invokes meta program operations (i.e., performes the target analysis) from a given flow graph. We also discuss the treatment of a meta store for enabling stateful analyses (for expressing stateful security policies).

6.4.1 Intercepting Base Program Operations and Invoking Traps

The second phase involves exploring the flow graph that resulted from the first phase. The EE, which is the static counterpart of the EM for RASP, visits every state in the flow graph. For every state, the EE checks whether it represents a program operation that is trapped. In our running example of a password strength checker application, these are calls to functions `check`, `fetch`, etc. in Listing 6.1. Other examples include reading and writing object properties.

Intercepting base program operations is relatively straightforward when looking at the transition relation for JS_0 (shown in Appendix C.2.2). Considering the interception of function calls as an example, we can observe that states in which state transition rules E-FUN-CALL, E-METHOD-CALL, and E-CTR-CALL apply, are states in which a function is about to be called. For example, state `17` in Figure 6.2 is such a state. In our semantics, we define a relation *handle* that the EE applies to every state when exploring a base program's flow graph. This relation is formalized in Appendix C.3.2. Relation *handle* takes a state and a meta store (we explain the latter below), and checks whether there is a trap method that corresponds to the evaluation step represented by that state; if so it executes the trap method. The result of executing a trap method is a tuple consisting of a return value (`META.HALT` or `META.PROCEED` for our AC and IFC security analyses) and a meta store. We discuss obtaining and executing trap methods next.

6.4.1.1 Obtaining Trap Methods

If a trapped operation is detected for a state, relation *handle* first obtains the reference to the `META` handler object that represents the access point to the analysis code. Recall that in our implementation we make `META`

a property of the global object, so obtaining this reference amounts to performing a straightforward property lookup on the global object within a state. Next, `handle` looks up the trap method that corresponds to the trapped operation by using the name of the trapped operation (e.g., `apply`, `get`, `set`, etc.). Looking up this method again amounts to a property lookup, this time on the `META` handler object that is reachable in each state. Continuing the previous example, if in state `17` in Figure 6.2) a function call is intercepted, then `handle` will look up method `META.apply` in that state. The step of obtaining a trap method is formalized as the *trap* relation, given in Appendix C.3.1. In particular, Trap-Callable specifies how to obtain the trap method `META.apply`.

The formalisation of phase 2 in JS_0 is included in Appendix C.3.

6.4.1.2 Executing Trap Methods

As the final step, relation `handle` will abstractly execute the trap method, which corresponds to invoking the associated meta operation of the handler object. Remember that, even though the input base program application already *included* `META` as a library, the meta operations were never *called* during the first phase. It is only in this second phase that the abstract execution of the meta code (i.e., the enforcement of a security policy) is performed by obtaining and abstractly executing the appropriate trap methods. If the value resulting from executing a trap method subsumes the abstract value `META.HALT`, then this indicates that, according to the target analysis, a base program operation was intercepted that must halt the execution.

Since abstract interpretation of trap methods is not an "analysis", but rather a result-oriented abstract execution, trap methods are always executed with the highest possible precision that is bounded by the precision obtained during the abstract interpretation of the first phase. The reason for this is that the resulting precision of each abstract execution of each trap method is affected by the imprecision of the base program analysis. For example, analyzing `META.apply({{set:70}}, {Str})` in the second phase may introduce imprecision if the second argument ($\{Str\}$) is used. In this case, $\{Str\}$ represents any string, which makes the result of the operations using this abstract value imprecise.

The abstract execution of the same trap methods that are used in the dynamic analysis is what makes our approach *safe* and *efficient* be-

cause the same analysis specification (meta code) and semantics (abstract machine) are used for both the static and dynamic analysis.

6.4.2 Maintaining Analysis State

So far we have ignored the issue of stateful analysis code, i.e. a meta program that maintains state of its own by performing side-effects on it. The AC policy enforcement code in Listing 6.3 is an example of a stateful analysis because it performs side-effects to update a counter variable. Likewise, the IFC policy enforcement from Listing 6.7 is stateful because it maintains a shadow stack.

In case the analysis is stateless, it suffices for the Execution Explorer (EE) to visit all flow graph states once in an unspecified order, and handle these states as described in Section 6.4.1. In case the analysis is stateful, however, meta state has to be maintained as well. The meta store is the component that is responsible for maintaining the meta state in our approach. As mentioned before, the relation `handle` takes a state and a meta store as input, and returns a value and a meta store. The input meta store represents the meta state that a trap method has access to, and any modifications that a trap method makes to this meta state are captured in the returned meta store. At the start of a trap method's execution, the meta store is merged into the "base" store contained in the state that represents the trapped operation. When the trap method returns, the meta store is obtained by collecting all heap information reachable from the `META` object. Function $\mathcal{R}$ in Appendix C.3.2 formalizes the concept of reachability in a store.

Any meta state changes resulting from the execution of trap methods in a certain state ς must be propagated to subsequent executions of traps in states reachable from ς. Because the flow graph may contain cycles, this means that handling states using `handle` now must be expressed as a fixed point computation over the flow graph. The fixed point is reached when handling each state adds no new information to either the return value or the meta store for that particular state in comparison to the previous call to `handle`.

In our semantics, the transition rule EE-Trans, detailed in Appendix C.3.2, formally describes a transition from a reachable triple representing a state, a trap's return value, and a meta store, to triples for successor states in the flow graph. The transitive closure of this relation

represents the fixed point that the EE computes.

6.5 Evaluation

We now evaluate the applicability of our approach (Section 6.5.1) and we compare it to a single-phase approach in terms of precision and performance (Section 6.5.2).

6.5.1 Evaluation of Applicability

Our approach's applicability is evaluated by statically verifying a set of AC and IFC policies that were previously also used for evaluation of Guardia and Gifc (in Chapter 3 and Chapter 4 respectively). The goal is to demonstrate that our approach is general enough to dynamically enforce and statically verify the two most well-known types of security policies starting from only the policies' specification and without requiring the enforcement code's reimplementation of the static analysis.

6.5.1.1 Access Control Policies

For each of the policies in Table 3.3 we designed a test program that attempts to bypass the policy. Each test program was first executed with the RASP component based on Guardia, using the source code instrumentation-based enforcement described in Section 3.3.2. Then the SAST component derived using our two-phase approach was executed on the same set of programs to determine the number of policy violations it would detect. We found that the SAST component effectively detected all the intended policy violation for all tests.

6.5.1.2 Information Flow Control Policies

Table 6.1 shows the results of analyzing the 14 programs without hidden implicit flows, out of 33 IFC test programs in Table 4.2. The derived SAST component was able to detect policy violations in all 14 test programs. We believe that support for implicit control flows would improve the security guarantees of our IFC monitor. However, it has been shown previously [SSB+19] that tracking explicit flows is sufficient to enforce *integrity* IFC policies, while for privacy related IFC policies tracking hidden

Test case [STA18]	Features	Violation detected
1		✓
2	if	✓
3	lp	✓
4	lp	✓
5	lp	✓
6	lp, if, arr, oprop	✓
7	lp, if, cb, oprop	✓
14	oprop, this	✓
15	oprop, this, new	✓
17	oprop, this, new	✓
19	ret, oprop, this, new	✓
22	oprop	✓
24	ret, oprop, oproto, this, new	✓
28	lp, oprop	✓

Table 6.1: Result of applying the SAST component derived from our RASP IFC monitor on 13 test cases without hidden implicit flows from of Sayed et al. [STA18]. Each test case contains an IFC policy violation, and a checkmark in column *Violation detected* signifies that the static verification correctly detected this. Column *Features* lists the set of notable features are present in each test program: *if*—`if` statement, *lp*—`for` or `while` statement, *ret*—(conditional) `return` statement, *thr*—`throw` statement, *this*—`this` expression, *new*—`new` expression, *arr*—arrays, *oprop*—access or modification of object property, *oproto*—access or modification of prototype property.

flows may be required. Therefore, from a software development point of view, the application developer can use our two-phase approach to know when sensitive components in an application may be affected by untrusted data (for integrity).

6.5.2 Evaluation of Performance and Precision

In this section we validate our *two-phase* static analysis approach (2PH) by comparing it with the analysis of the instrumented version of the application (1PH) in terms of precision and analysis speed. The goal of this evaluation is two-fold:

- Validate a key property of our approach: when the runtime enforcement of security policies is provided through metaprogamming in JavaScript, the static verification can be automatically obtained through a two-phase abstract interpretation approach, without the need to analyze the entire instrumented program.
- Confirm our hypothesis that under the same analysis configuration, a two-phase 2PH performs better than 1PH in terms of speed and precision.

Setup. We use JIPDA [NSDD17] as a configurable abstract interpreter to perform static analysis of JavaScript programs and abstractly execute policy code. We employed JIPDA because it is possible to steer the abstract interpreter for implementing our two-phase approach, and because it implements a version of the syntax and semantics presented in Section 6.3 and Appendix C. Therefore, the evaluation of a JavaScript program with JIPDA outputs a flow graph that approximates the behavior of that program for all its possible execution paths. JIPDA and the flow graphs it produces fulfil the assumptions discussed before. For RASP, we use GUARDIA with the source code instrumentation-based enforcement described in Section 3.3.2.

Methodology. We evaluate the performance and precision based on 8 small experimental client side web applications, ranging from 46 to 110 LOC of HTML and JavaScript. All applications were subject to the policy previously described in Listing 6.2. Despite the applications being small compared to real world applications, they do contain a substantial set of features from the JavaScript language and browsing environment (including DOM elements and events). Because the applications are not deliberately vulnerable, we randomly inserted calls to `fetch()` in their source code to perform the experiments. For each experimental application we build an instrumented version counterpart, in which all function calls are instrumented.

For the analysis of uninstrumented applications using our 2PH, we use a tunable lattice that allows to change the precision of the abstraction function α (i.e., *high* and *low*) during the analysis. However, during an application analysis, JIPDA will lower the lattice's precision based on a threshold to produce values that are more abstract to ensure termination.

Each uninstrumented application was analyzed twice using the 2PH approach. First, the application was analyzed using a high precision α for the first stage of our approach. Second, the application was analyzed with a low precision α for the first stage of our approach. Independent of the α function used during the first stage of the analysis, we always use a high precision α function for the second phase of our 2PH. The instrumented applications were also analyzed twice. For the first analysis, we use a high precision α for the lattice, while for the second analysis we used a low precision α.

6.5.2.1 Evaluation of Precision

Table 6.2 shows the results of the analysis of all experimental applications using the configurations explained before. Each row of the table is the result of analyzing a *program*, using an *approach* (i.e 1PH or 2PH). For the 1PH approach, the instrumented version of the benchmark program was analyzed. As shown in the table, 2PH outperforms 1PH for all pairs of equivalent applications using the same lattice configuration. This is because our 2PH approach runs with a high precision configuration for the lattice in the second phase. We did not observe any false negatives during our experiments, but they can occur given the fact that our analysis is unsound. As mentioned before, practical static analyses of non-trivial JavaScript applications using dynamic features are always unsound to some degree so we envision our 2PH approach to be used early during the development phase, or as part of a building pipeline to help catch vulnerabilities introduced by developers of an application or by third-party code it includes.

6.5.2.2 Evaluation of Performance

To measure performance of our 2PH approach, we use the same setup as for measuring precision. Figure 6.7 shows the analysis speed in seconds for all applications using the four different combinations ($2PH_H, 2PH_L$, $1PH_H, 1PH_L$). The broken bars show the analyses that did not finish in a predefined time window (430 seconds).

As shown in the figure, the 2PH approach performs better than the 1PH approach for both lattice configurations. Performance is influenced by the way the fixed point is computed, which in the second phase is

Table 6.2: Precision comparison between the single-phase approach (*1PH*) and our two-phase approach (*2PH*) for statically detecting AC policy violations. Column *Precision* indicates the analysis precision: *H* for high precision, *L* for low precision. Columns *TP*, *FP*, and *FN* denote the number of true positives, false positives, and false negatives, respectively, with respect to reported policy violations by each approach. "-" denotes the absence of a value due to analysis timeout.

		1PH			2PH		
Program	Precision	TP	FP	FN	TP	FP	FN
`sequential`	H	3	0	0	3	0	0
	L	-	-	-	3	0	0
`branches`	H	0	4	0	0	4	0
	L	0	7	0	0	4	0
`iterative`	H	0	3	0	0	0	0
	L	0	4	0	0	2	0
`safe`	H	0	4	0	0	0	0
	L	0	6	0	0	2	0
`recursive`	H	-	-	-	1	0	0
	L	1	3	0	1	0	0
`fib`	H	-	-	-	1	0	0
	L	1	3	0	1	0	0
`passStrength`	H	-	-	-	2	1	0
	L	-	-	-	2	3	0
`steal`	H	0	3	0	0	0	0
	L	-	-	-	0	1	0

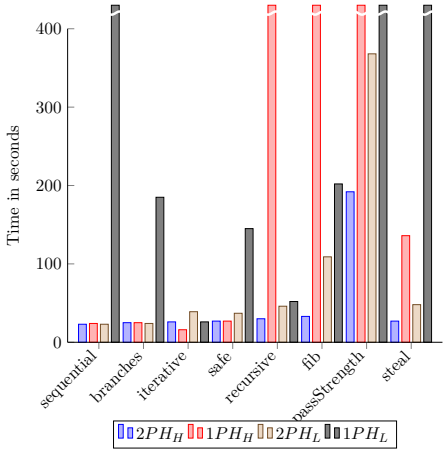


Figure 6.7: Speed comparison between the *1PH* approach and our (*2PH*) approach for statically detecting AC policy violations with high precision (H) and low (L) precision. Each application is thus analysed using four different configurations ($2PH_H, 2PH_L$, $1PH_H, 1PH_L$).

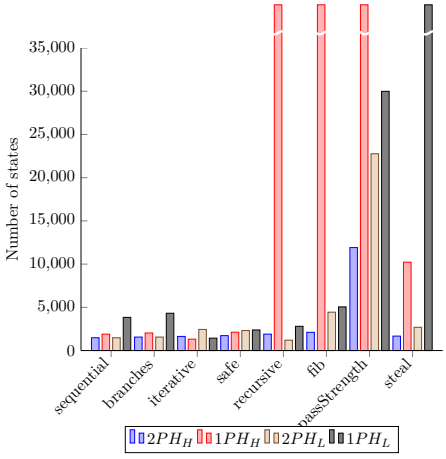


Figure 6.8: Comparison between the number of states generated by 1PH approach and our 2PH during analysis using *low* (L) and *high* (H) precision lattice configurations. Each application is thus analysed using four different configurations ($2PH_H, 2PH_L$, $1PH_H, 1PH_L$).

affected by only the meta store. There is a similar performance for the configurations $1PH_H$ and $2PH_H$ for the *sequential, branches, iterative* and *safe* programs. Most likely this is because the values remained under the configured threshold for the lattice during the analyses. From our experimental data, we also observe that high analysis precision often results in better performance than low analysis precision. Although the speed–precision trade-off for a static analysis depends on many factors and is generally unpredictable, lowering precision tends to slow down an analysis because this increases the number of spurious paths explored. However, more experimental data is needed to draw a general conclusion about this.

The performance can also be measured in function of the number of states generated during the analysis. Figure 6.8 and Table C.1 show the relation of states produced by the 1PH and 2PH approaches for our set of programs evaluated in both high and low precision lattice configurations. As the figure shows, 2PH performs better than 1PH for most of the programs in terms of states generated (with more states representing more explored paths). For the examples *iterative* (H and L), *recursive* (L) and *fib* (L) the 1PH performs slightly better than 2PH, but this at the expense of precision, however. The broken bars indicate that the analysis generated more than 100000 states.

6.6 Discussion

This section discusses some of the properties of our two-phase abstract interpretation approach for deriving SAST from RASP.

Efficiently deriving SAST Our goal is to derive a static analysis from an already existing dynamic analysis mechanism with the least effort. Although our two-phase approach avoids the reimplementation of the meta program (i.e., policy enforcement code) when deriving the static analysis from the dynamic enforcement, the triggering mechanism used in the second analysis phase is not reused from the dynamic enforcement mechanism and therefore still has to be reimplemented. However, typically the policy code is much larger and more complex than the triggering code. For instance, the code for monitoring a certain application behavior, such as function calls and value creation, is similar in structure in both the runtime

and the static implementation. Therefore, we believe that the advantages of performing static analysis in two phases as described in Section 6.2.2 outweighs the downside of having to reimplement (only) the triggering mechanism.

Performing static analysis in two phases can also bring benefits in terms of speed and precision (as we showed in Section 6.5). Since precision is one of the most important properties of any static analysis, it deserves some discussion. In this regard, we would like to emphasize that our goal is not to improve precision of our static analysis over other custom static analyses. Instead, the goal is to avoid writing the policies' semantics for the static analysis, and this by reusing the policies' semantics already implemented by the runtime monitor. As explained in Section 6.4, the precision of our analysis depends on the flow graph resulting from the first phase. As such, abstracted base program values in the first phase can introduce imprecision to our security analysis during the second phase.

What the meta code does is independent of the static analysis. In contrast, it does matter *how* the meta code is implemented. Therefore, it is a reasonable assumption that analysis developers should avoid features that the abstract interpreter cannot soundly or precisely handle (e.g., `eval`, the `Function` constructor, etc.).

6.7 Conclusion

This chapter presented an approach for performing both Static Application Security Testing (SAST) and Runtime Application Security Protection (RASP) using a single security policy library in a safe and efficient manner. Our approach starts from a set of declarative security policies, from which a runtime enforcement mechanism is generated by instrumenting the target application for trapping security-relevant program operations and forwarding them to policy enforcement code. Next, a static analysis mechanism is derived from the runtime enforcement mechanism without reimplementing any of the policy enforcement code.

Our approach reduces the effort of combining SAST and RASP, enabling that policy semantics are identical between their static verification and their runtime enforcement. We demonstrated the applicability of our approach by detecting and enforcing 12 access control and 14 information flow control policies originating from related work.

Chapter 7

Conclusion

This dissertation focused on security issues affecting modern client-side web applications. We studied application-level security mechanisms for JavaScript, the lingua franca in web development. From the study of the state of the art in language-based approaches for security, we distilled a set of essential properties of practical dynamic mechanisms for securing modern client-side web applications. We subsequently propose GUARDIA and GIFC: two dynamic mechanisms for enforcing access control (AC) and information flow control (IFC) security policies, respectively. Finally, motivated by the use of security tools at different stages of the software development lifecycle and the lack of integrated tools, we proposed a novel two-phase static analysis approach to safely and efficiently derive a Static Application Security Testing (SAST) component from an existing Runtime Application Security Protection (RASP) component based on source code instrumentation.

This chapter concludes this dissertation by revisiting the research contributions, and by discussing limitations and avenues for future research.

7.1 Summary

This section summarises what each chapter contributes to this dissertation's research goals:

Chapter 1 gave the context of this dissertation, giving a high-level overview of the shortcomings of the SOP and CSP browser-level secu-

rity mechanisms. We described the research hypotheses and the goals of this thesis related to (i) the possibility of building portable and tamper-proof runtime monitors, and (ii) the possibility of deriving a SAST tool from a RASP tool based on meta-programming in a safe and efficient way. Finally, we described the contributions and supporting publications of this thesis.

Chapter 2 In this chapter, we studied the theoretical aspects of AC and IFC policies, and how they are implemented and deployed in the client-side web application ecosystem. Our study shows that these policies are enforced at run-time, statically analysed, or verified using a hybrid combination of a static and a dynamic approach. We concluded that dynamic IFC monitoring approaches often modify the VM to implement the enforcement mechanisms, while AC monitoring mechanisms are quite often implemented using metaprogramming. We also concluded that there is no scientific consensus on how to implement a hybrid combination of static and dynamic analyses of IFC and AC policies for web applications.

Chapter 3 started by discussing the portability, mediation, tamper-proofness, and transparency as desired properties of a runtime monitor for security. From this discussion, we concluded that portability of a security mechanism is a crucial property to support in a web development context. We also concluded that decoupling the policy declaration from its enforcement is a good fit to enable the experimentation with different techniques to enforce security policies. Then, we presented Guardia's declarative policy language by example, with AC policies from related work. We described the language's main design elements, followed by two enforcement implementations based on JavaScript proxies and source code instrumentation, respectively.

Chapter 4 started by discussing the properties that an IFC monitor needs to exhibit for client-side web applications. The discussion is grounded in the challenges that JavaScript and the browsing context bring to an IFC monitor based on source code instrumentation, including implicit coercions, external libraries, and dynamic code evaluation. We paid special attention to the permissiveness aspect of the monitoring mechanism, as this is considered to be an essential property for IFC monitors'

usability. Then, we introduced GIFC, an IFC monitor for JavaScript applications. GIFC instruments the source code of the application to track information flows and employs JavaScript proxies to associate program values with their respective security labels.

Chapter 5 extensively discussed the challenges for implementing tamper-proof and transparent AC and IFC runtime monitors based on source code instrumentation. This discussion started with the investigation of JavaScript features that pose challenges for a security mechanism, namely implicit coercions, its prototype inheritance model, dynamic code evaluation, and built-in higher-order functions. We discussed the solutions proposed in the literature for tackling these challenges and subsequently proposed cached values for handling untrusted objects within the enforcement code.

Chapter 6 motivated the need for an integrated approach for policy verification across all program development stages by means of a web application scenario. We argued that it is possible to offer an integrated SAST and RASP solution in which the same set of policies are verified and enforced by starting from a RASP component based on source code instrumentation, from which a SAST component can then be derived. We introduced the main ideas to safely and efficiently derive a SAST component from a RASP one through a two-phase abstract interpretation approach. We specified our approach for a substantial subset of JavaScript as a small-step operational semantics. This semantics also includes evaluation rules for abstractly executing meta code during our approach's second phase. Subsequently, we instantiated our approach using GUARDIA and an IFC monitor described in Chapter 6. We measured the ability of the derived SAST components to detect the same policy violations as their RASP counterparts, and evaluated the precision and performance of analysing applications using our two-phase approach with respect to analyzing the instrumented application in one phase.

7.2 Contributions

This dissertation has developed novel language-based security mechanisms for client-side web applications: GUARDIA, GIFC, and a two-phase ab-

stract interpretation approach to support integrated static and dynamic analysis of security policies. Together, they form the main contributions of this dissertation.

Guardia GUARDIA is a novel declarative internal domain-specific language for enforcing access control security policies dynamically. Central to GUARDIA is the possibility for developers to declaratively specify fine-grained AC policies without the need to learn a new language, as policies are specified in JavaScript. A declarative approach also allows GUARDIA to decouple policy specification from its enforcement, enabling the use of different enforcement mechanisms. Chapter 3 presented the implementation of enforcement mechanisms based on JavaScript proxies and source code instrumentation. We evaluated GUARDIA's expressivity, applicability, and performance by expressing 13 policies appearing in the state of the art. We use 10 real-world applications in which we deployed GUARDIA to corroborate the transparency of our approach.

Gifc GIFC is a portable IFC enforcement mechanism that improves the precision of existing permissive upgrade-based monitors by dynamically upgrading the security label of write targets before the execution takes a branch that is conditioned by security-sensitive data. The automatic annotation of variables is possible because our monitoring mechanism has access to the abstract syntax tree during the enforcement of program operations.

We evaluated the precision of GIFC using a set of 28 programs designed to benchmark IFC monitors [STA18]. This benchmark was then extended with 5 additional programs to test additional features missing from the original benchmark, such as dynamic code evaluation and external libraries. The evaluation shows that for the 33 programs, our approach demonstrates better precision and permissiveness than the other benchmarked IFC monitors. However, the performance impact of tracking information flow using source code instrumentation is considerable.

An Integrated RASP and SAST Approach Our final contribution presents a novel approach to safely and efficiently derive a static analysis from a dynamic analysis, starting from a single set of policy specifications. Specifically, we introduced a *two-phase* abstract interpretation approach

that frees developers from re-implementing security policies and, more importantly, the semantics of such policies in a static analysis tool. Splitting the analysis into two phases enabled us to apply different analysis parameters for each phase, so that the overall analysis precision and speed can be improved. We formally described our approach through small-step operational semantics and implemented them on top of JIPDA, a static analyzer for JavaScript. Finally, we empirically evaluated the trade-offs in precision and analysis speed of our two-phase abstract interpretation approach. The results show that our two-phase approach outperforms the one-phase analysis of the instrumented applications for our set of benchmark programs.

7.3 Limitations

We now highlight limitations of our approaches in the broader context of this dissertation.

Meta-programming vs. VM modifications Our work departs from the observation that securing applications using the JavaScript meta-programming facilities promotes the portability of the monitoring mechanism. Unfortunately, either the transparency or the integrity guarantees of the monitoring mechanism will be affected. In this dissertation, we trade-off the transparency in order to promote the integrity of the monitoring mechanisms.

Our automatic write target annotation approach for improving the permissiveness of GIFC computes a set of write targets (e.g., variables being assigned) in order to upgrade their label. Having access to such information necessarily needs dynamic scope information. Having access to the dynamic scope in which a function is being called using meta-programming implies mirroring the execution's scope chain. Moreover, the set of write targets within alternative branches needs to be constructed in a per branch basis. Computing the set of write targets can be done statically to avoid the performance impact of doing so at runtime, as GIFC currently does. However, this approach implies changes to the instrumentation platform in order to provide the set of write targets to the monitoring mechanism. Alternatively, the monitoring mechanism could memoize this information after computing it dynamically.

Policy specification language In this dissertation, we explored a declarative approach for specifying policies since it promotes decoupling between specification and enforcement of policies. GUARDIA policies and GIFC policies are limited to application wide policies. For example, GIFC's policy specification language limits developers to the specification of a general IFC policy. In client-side web applications, it may be beneficial to give developers sufficient granularity to specify more fine-grained IFC policies to differentiate, for example, between the origins of the information [PPA+20].

Evaluation The runtime monitors developed in this dissertation do not handle all JavaScript and browser features, including promises, async/await, classes, generators, events, and import/export. This, in turn, limited the type and size of the programs that could be used during evaluation in this dissertation. Once the language operations related to these features can be instrumented, additional research will be needed. Specifically, it will be necessary to analyse how these features influence the enforcement of information flow control policies.

Supporting some of these features only requires more engineering effort, while others requires additional research. The latter is specially true for enforcing information flow control policies of async/await, and promises.

7.4 Future Work

We now discuss how our research could be extended or studied in a different context.

This dissertation has focused on the use of metaprogramming for securing modern client-based web applications. While the proposed mechanisms are portable, the performance overhead added to the application by GUARDIA and GIFC is still very high, mainly due to the source code instrumentation. Therefore, as future work, we are investigating more efficient technologies such as WebAssembly to lower the overhead added by runtime monitoring. In this direction, we have performed experiments towards an instrumentation platform for web applications where analyses are compiled to WebAssembly [MSBG21]. However, due to the context switches between JavaScript and WebAssembly, the performance of the

overall instrumented application is not improved. Therefore, having an efficient instrumentation strategy requires thinking from ground up how to tackle this problem.

With respect to our approach for deriving SAST from an existing RASP, it would be interesting to explore a solution to support IFC policies that at runtime also require a static approximation of the program runtime behavior to correctly deal with hidden implicit flows. As mentioned in Section 6.2.1, the IFC monitor used in Chapter 6 cannot handle language features that cause implicit flows. Handling implicit flows is not a limitation of our two-phase abstract interpretation approach but a limitation of the employed IFC monitor. Other work that has explored the precise handling of hidden implicit flows in an IFC monitor relies on a static analysis for control flow statements prior to the execution of the program [CN15, BRGH14]. To incorporate such a technique into our approach requires to add yet another static analysis phase, which is not trivial and remains to be studied.

Finally, while integrated SAST and RASP is essential for secure web development, we believe that our two-phase abstract interpretation approach presented in Chapter 6 is more broadly applicable to other domains besides security. As future work, we would like to explore its applicability to other dynamic analyses that can be implemented with runtime monitors based on meta-programming, akin to the RASP components used in this work. For example, Gong et al. [GPSS15] develops a dynamic linter (DLint) to detect violations of code quality rules at runtime. In their framework, a rule is specified as a predicate over program events (i.e., function calls, variable writes, etc.) to check common quality issues in JavaScript, such as inheritance and type inconsistency problems, APIs misuse, etc. Another kind of dynamic analysis that could be implemented using our two-phase approach are profilers. For example, JITProf [GPS15] is a profiling framework to dynamically identify code locations that prohibit JIT optimisations. In JITProf, the information of program events is used to maintain metadata associated with the source location of those events. The metadata is then used to pinpoint those locations that prohibit JIT (i.e., just-in-time) optimisations.

Appendix A

Additional Access Control Security Policies

Policy 7: Disable geoposition API

Geo-location API allows to gather the physical location of the device. In spite of that browsers have a policy that asks user explicitly for using the geolocation information, it is desirable to deactivate the use of this feature programmatically.

```
1 GG.onCall(navigator.geolocation, ['getCurrentPosition', 'watchPosition'
      ]).deny()
```

Listing A.1: Policy 7: Disable geoposition API in GUARDIA.

Policy 8: Disable page redirects after `document.cookie` read

Cookies are commonly used by web servers to store data regarding to a user session. If an attacker is allowed to make a request after reading information stored in cookies, this could cause leakage of valuable information [KYC+08, PSC09, ML10]. There are different ways to make a request to an external site, but here we present a policy that disallows changing the `location` property of the window to avoid such an attack.

Listing A.2 shows how to construct such a policy. The first predicate (line 1), checks whether the operation sets `window.location` to a new location. The second predicate at line 2, has two responsibilities. First, it registers a listener for *read* operations on `document.cookie` during policy

deployment. Second, it checks whether the cookie property was already read.

```
GG.onWrite(window, 'location')
  .afterRead(document, 'cookie')
  .deny();
```

Listing A.2: Policy 8: Disable page redirects after `document.cookie` read in GUARDIA.

Policy 9: Allowing a whitelist cross-frame messages

Cross-origin communication using `window.postMessage` can lead to attacks such as Cross Site Scripting and Denial of Service. The policy below is intended to prevent these kinds of attacks by checking that the origin URL of the message is white-listed. The predicate of the policy verifies, by means of `ParamInList`, that the second parameter of the invocation of `postMessage` is contained in a whitelist of URLs. If this is not the case, then the invocation of `postMessage` is denied.

```
const urls = ['http://google.com', 'http://facebook.com'];
GG.onCall(window, 'postMessage')
   .not(GG.arg(GG.inList, GG.targ(1,String), urls))
   .deny()
```

Listing A.3: Policy 9: Allow whitelisted cross-frame messages in Guardia.

Policy 10: Disallow *string* arguments to *setInterval* and *setTimeout* functions

This policy aims to disallow the execution of arbitrary code as described in [ML10]. Functions *setTimeout* and *setInterval* can accept a closure or string as callback argument. As such, these functions can be abused to run malicious code.

listing A.4 shows how we express a policy to restrict the execution of these functions to closures. In the policy below the execution of `setTimeout` and `setInterval` is permitted only if the first parameter of the invocation is a function.

```
GG.onCall(window, ['setInterval', 'setTimeOut'])
   .with(GG.arg(GG.typeOf, GG.targ(0), 'function'))
   .deny()
```

Listing A.4: Policy 10: Disallow *string* arguments to *setInterval* and *setTimeout* functions in GUARDIA.

Policy 11: Restrict `XMLHttpRequest` to secure connections and whitelist URLs

Phung et al. [PSC09] prevent impersonation attacks using the *XMLHttpRequest* object by restricting its `open` method to whitelist URLs. Meyerovich et al. [ML10] propose a policy that enforces an HTTPS request when user and password arguments are supplied to the `open` method. Here we implement a security policy that compose these approaches.

```
1 GG.onCall(XMLHttpRequest.prototype, 'open')
2    .not(GG.or(GG.arg(GG.inList, GG.targ(0,String), urls)
3                    GG.arg(GG.startsWith, GG.targ(0, String), 'HTTPS') )
     )
4    .deny()
```

Listing A.5: Policy 11: Restrict `XMLHttpRequest` to secure connections and whitelist URLs in GUARDIA.

Policy 12: Only redirect to whitelisted URLs

Both Pungh et al. [PSC09] and Meyerovich et al. [ML10], propose a policy to prevent redirection to another web site by means of changing the location property of the *window* and *document* objects.

listing A.6 illustrates this policy in GUARDIA. Redirections and setting of source locations are allowed only for URLs that are contained in a whitelist.

```
1 const urls = ['http://google.com', 'http://facebook.com'];
2 GG.onWrite(window, 'location')
3    .not(GG.arg(GG.inList, GG.targ(0,String), urls))
4    .deny()
```

Listing A.6: Policy 12: Only redirect to whitelisted URLs in GUARDIA.

Policy 13: Disallow setting of `src` property of images

This policy was studied by [PSC09] with the aim of preventing leakage of information by changing the source location of images, forms, frames, and iframes.

```
1 GG.onWrite(HTMLImageElement.prototype, 'src').deny()
```

Listing A.7: Policy 13: Disallow setting of `src` property of images in Guardia.

Appendix B

Information Flow Control Benchmark Programs

B.1 Description of IFC Benchmark Programs for Performance

The following list describe the algorithms used for the performance benchmarks in Chapter 4.

- FFT: Fast Fourier Transform algorithm.
- LZW: Lempel Ziv Welch (LZW) lossless data compression algorithm.
- KS: Knapsack, combinatorial optimization algorithm.
- FT: Floyd Triangles algorithm.
- HN: Hamming Numbers algorithm, that generates 5-smooth and 7-smooth of hamming numbers.
- 24: 24 Game algorithm that takes four digits as an input, each from one to nine, with repetitions allowed and generates an arithmetic expression that evaluates to 24 using just those four digits, and all of those four digits are used exactly once.
- MD5: MD5 hash function implementation in JavaScript.

B.2 Description of Test Programs for Benchmarks of IFC Precision

The following table provides a description for each test program used in GIFC. Each program has the structure of the listing bellow. It takes as input a secret string value, which the program attempts to leak explicitly or implicitly in various ways as described in the table.

```
var pass;
pass = 'temp1234';
chkpassword(pass);
function chkpassword(pwd) {
   //Test program code
}
```

Approach	FFT
Test 1	Direct information flow and leakage of the whole password.
Test 2	Usage of an IF-Statement and a direct information flow leakage of the whole password.
Test 3	Usage of the For-loop that leaks the password in the first iteration.
Test 4	Usage of a While-loop that leaks the password in the first iteration.
Test 5	Usage of a For-in-loop that leaks one character of the password on each iteration.
Test 6	Usage of an empty Array that leaks the password length by assigning a boolean true at the index `password.length`.
Test 7	Usage of a For-loop with a conditional `break` statement. The loop assumes the maximum password length to be 16. The If-statement breaks the loop when the length of the password is equal to the loop counter. Then leaks the loop counter which is the same as password length.
Test 8	Usage three For-loops in combination with a `continue` statement. The first loop fills a 16 elements array with `true` values. The second loop iterates over the array setting all values to `false`, continuing conditionally if the length of the password is equal to the loop counter. The third loop iterates all elements of the array leaking the loop counter at the index on which the array has a `true` value.
Test 9	Combines a Try-catch statement with For-loop. The loop iterates 16 times and conditionally, if the password length is equals to the counter, throws an error with the program content value. At the `catch` block it leaks the error value which is equals to the length of password.
Test 10	Combines a For-loop with a conditional `return` statement in the function. When the loop counter reaches the password length, the function returns the loop counter value. The returned value reflects the password length.
Test 11	Combines a For-loop with a conditional `return` statement. When the loop counter reaches the password length, the functions returns. The loop counter of the function is a global variable. After the function returns the variable reflects the password length.
Test 12	Similar to the previous test, however, the program counter is referred as `this.j` instead of `j`. The goal is to test whether the monitor implements correctly the scoping rules.

Table B.1: Description of IFC benchmarks programs 1.

Approach	FFT
Test 13	In this test the password length is assigned to an object property. Then, in a method, the object try to leak the reference to `this` using a print statement. The idea is to test if the monitor implements a correct scoping rules for `this`.
Test 14	In this test the password length is assigned to an object property. Then, in a method, the object try to leak the value of property using a print statement.
Test 15	Creates a function constructor which receives a password and sets `this.i` to the length of the password. Then, the program calls the function using `new` to create a new object. Later the program leaks the property "i" of the created object.
Test 16	Similar to the previous test. However the function is called as a regular function and not as a constructor. Therefore the assignment to `this.i` happens on the global object.
Test 17	Similar to test 15 but the function returns an object. The trick is that functions called with `new` must return an object, but if the function has a return statement and returns an object, that object will be used instead of the one created using `new`.
Test 18	Combines two functions one inside the other and the `this` variable scoping. The outer function when called executes the inner function passing the password as argument. The inner function declares a variable `i` and assigns the password length to `this.i` and returns the number 5. Then the outer function is executed and finally, the program prints the value of the variable `i`.
Test 19	Similar to 15 but the function returns the value 5. Therefore the object created with `new` is used instead of the returned value.
Test 20	Combines 3 nested If-statements with a conditional `return` statement. At the inner most `if`, the function tests if the password length is less than 10 and return if it is true. At the end of the consequent of the outer most `if` the function assigns a global variable to false. Before the function execution the global variable was initialized with `true`. After the function call the variable is printed leaking the approximate length of the password.

Table B.2: Description of IFC benchmarks programs 2.

Approach	FFT
Test 21	Combines If-statement with object property deletion. The program deletes a property from an object if the length of the password is equal to 8. Then the program test if the property is undefined and prints 8. The idea is to leak information about the password length using the existence of a property.
Test 22	Uses object aliases to set a property and leak the password length.
Test 23	Uses object prototypes to store information. The program defines a function, and then adds a property to the function's prototype. Then, it creates an object out of the function using `new` and leaks the password length.
Test 24	Similar to test 23 but instead of a property a method is added to the prototype of the function.
Test 25	Defines a function with multiple conditional return statements that none of them get executed. The goal is to test the analysis of the untaken control-flow branches.
Test 26	Similar to test 25 but replaces the conditional assignments by function calls.
Test 27	Similar to tests 25 and 26 but uses object property assignments instead of function calls or variable assignments.
Test 28	Uses an If-statement to conditionally create two functions inside the consequent and the alternate. The functions prints an string indicating the name of the function being called.
Test 29	Uses the `eval()` function to perform a binary operation and a variable assignment that is then leaked.
Test 30	Uses a For-loop to fill an array with closures containing one character of the password. Later the program iterates the array and prints the result of the execution of each closure to leak the password.
Test 31	Uses the `Math.pow()` using the password length. Then the program calls `Math.sqrt()` on the previous value to leak the password length. The idea is to test if the analysis is able to track information that flows to non-instrumented code like the standard library.
Test 32	Defines an object with a `valueOf()` method that returns the password. Then the test concatenates the object with an empty string. The concatenation implicitly calls the `valueOf()` method.
Test 33	Defines an object with a getter function on it. The implementation of the getter function assigns the password variable to another global variable that it is leaked after accessing the property on the object.

Table B.3: Description of IFC benchmarks programs 3.

Appendix C

Additional Material for Deriving SAST from RASP

This appendix contains additional material for Chapter 6. First, we provide a comparison of numbers states generated by our two-phase approach in comparison to 1PH (cf. Section 6.5). Subsequently, Appendix C.2 and Appendix C.3 formally describe each phase our approach for deriving SAST from RAST (cf. Section 6.2.2) using a small-step operational semantics.

C.1 Comparison of Number of States Generated by 1PH and Our 2PH Approaches

Table C.1 relates the number of explored states of the 1PH and our 2PH under low (L) and high (H) lattice configurations.

Table C.1: Comparison the between number of states generated by (*1PH*) and our (*2PH*) during the analysis of experimental applications using low (*L*) and high (*H*) precision lattice configurations.

		States generated	
Program	Precision	1PH	2PH
`sequential`	H	1899	**1481**
	L	3839	**1481**
`branches`	H	2033	**1560**
	L	4317	**1560**
`iterative`	H	**1316**	1636
	L	**1435**	2445
`safe`	H	2122	**1726**
	L	2383	**2312**
`recursive`	H	-	**1902**
	L	**1199**	2842
`fib`	H	-	**2111**
	L	**4439**	5052
`passStrength`	H	-	**11912**
	L	29991	**22767**
`steal`	H	10266	**1673**
	L	-	**2692**

C.2 Phase 1: Semantics of $\mathbf{js}_0$

We now formally describe each phase of our approach using a small-step operational semantics. Before delving into the details on the transition rules, auxilary functions and relations, we introduce the notation and conventions used.

Notation and Conventions We use $\uplus$ to denote *disjoint union*: if $X = Y \uplus Z$, then $Y = X \setminus Z$. The notation $X = x : X'$ deconstructs a *sequence* X into its first element x and the rest X'. We write $\langle\rangle$ for the empty sequence. The *power domain* of set X is denoted as $\mathcal{P}(X)$. The *empty function* is denoted as $[]$, and for all inputs returns the bottom element $\bot$ of its range. The notation $f[x \mapsto y]$ denotes *function extension*

Figure C.1: Evaluation rules for simple expressions.

$$\text{E-LIT}\ \frac{}{evalSimple(\delta, \rho, \sigma, \kappa) = \alpha(\delta)} \qquad \text{E-VAR}\ \frac{v \in \mathrm{Dom}(\rho) \qquad a = \rho(v)}{evalSimple(v, \rho, \sigma, \kappa) = \sigma(a)}$$

$$\text{E-GLOBAL}\ \frac{v \notin \mathrm{Dom}(\rho) \qquad \omega_0 = \sigma(a_0)}{evalSimple(v, \rho, \sigma, \kappa) = \omega_0(v)}$$

$$\text{E-THIS}\ \frac{}{evalSimple([\![\texttt{this}]\!], \rho, \sigma, (e, c, d_{\mathrm{arg}}, a_{\mathrm{this}}, \sigma)) = \{a_{\mathrm{this}}\}}$$

and yields a function f' such that:

$$f'(z) = \begin{cases} y & \text{if } z = x, \\ f(z) & \text{else.} \end{cases}$$

We write the *function restriction* (or narrowing) of a function f to domain X as $f|_X$, such that $(f|_X)(x) = f(x)$ if $x \in X$ and $(f|_X)(x) = \bot$ else. *Function joining* happens in a pointwise fashion. If $\sqcup$ is the join operator for the range of the function, then $[x \mapsto y_1] \sqcup [x \mapsto y_2] = [x \mapsto y_1 \sqcup y_2]$. In particular, $\bigsqcup\{[x_0 \mapsto y_0], \ldots, [x_n \mapsto y_n]\} = [x_0 \mapsto y_0] \sqcup \ldots \sqcup [x_n \mapsto y_n]$.

C.2.1 Auxiliary Evaluation Functions and Relations

The evaluation rules for simple expressions are shown in Figure C.1. Function *evalSimple* : $\mathsf{Simple} \times \mathit{Env} \times \mathit{Store} \times \mathit{Kont} \mapsto D$ evaluates three types of simple expressions in JS_0: literals, references to either a global or non-global variable, and `this` expression.

Relation *lookupProp* looks up a property by traversing the prototype chain of an object. If the property is not found in the chain, `undefined`

is returned.

$$
\begin{aligned}
&\mathit{lookupProp}(v, a, \sigma) \\
&\qquad = \begin{cases} \omega(v) & \text{if } v \in \mathrm{Dom}(\omega) \\ \{\mathbf{undef}\} & \text{if } \omega(\text{“proto”}) = \varnothing \\ \mathit{lookupProp}(v, a', \sigma) & \text{else} \end{cases} \\
&\quad \text{where } \omega = \sigma(a) \\
&\qquad\quad a' \in \omega(\text{“proto”})
\end{aligned}
$$

Function *evalCall* applies a callable (or closure) (f, ρ) to an argument d_{arg} in a certain program state. We assume a single parameter and a single local variable declaration. Therefore, *evalCall* extends the function's static environment ρ and the store σ by binding parameter to its argument value and the variable is hoisted to the beginning of the function scope and bound to `undefined`.

The continuation κ is the execution context of the caller and parameter κ' is the execution context for the call itself. The stack store Ξ is extended by allocating the caller stack (ι, κ) at stack address κ'. The function's body evaluation occurs in the static environment and store extended with the binding of the argument (ρ' and σ'), with an empty local stack $\langle\rangle$, and the execution context for the call κ'.

E-CALL

$$
\frac{\begin{array}{c} f = [\![\texttt{function (}v\texttt{)\{var }v_h\texttt{; }e\texttt{\}}]\!] \qquad \rho' = \rho[v \mapsto a] \\ \sigma' = \sigma \sqcup [a \mapsto d_{\text{arg}}, a_h \mapsto \{\mathbf{undef}\}] \qquad a = \mathit{alloc}(v, \rho, \sigma, \iota, \kappa) \\ a_h = \mathit{alloc}(v_h, \rho, \sigma, \iota, \kappa) \qquad \Xi' = \Xi \sqcup [\kappa' \mapsto \{(\iota, \kappa)\}] \end{array}}{\mathit{evalCall}((f, \rho), d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa') = \mathbf{ev}(e, \rho', \sigma', \langle\rangle, \kappa', \Xi')}
$$

C.2.2 Transition Relation

We define the transition relation $\mapsto$ of our abstract machine using the functions defined in the previous sections.

$$(\mapsto) \sqsubseteq \mathit{State} \times \mathit{State}$$

Rules for transitions from evaluation states (**ev**) correspond with the different syntactic cases, while rules for transitions from continuation states (**ko**) correspond with the different kinds of continuations. Figure C.2 and Figure C.3 show the transitions rules.

$$\textsc{E-simple}\quad \frac{d = \mathit{evalSimple}(s, \rho, \sigma, \kappa)}{\mathbf{ev}(s, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi)}$$

$$\textsc{E-fun-call}\quad \frac{d_f = \mathit{evalSimple}(v, \rho, \sigma, \kappa) \qquad d_{\text{arg}} = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \qquad a_f \in d_f \qquad \omega_f = \sigma(a_f) \qquad c \in \omega_f(\text{“call”}) \qquad \kappa' = (e, c, d_{\text{arg}}, a_0, \sigma)}{\mathbf{ev}(\llbracket \underbrace{v\texttt{(}s\texttt{)}}_{e} \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathit{evalCall}(c, d_{\text{arg}}, \sigma, \iota, \kappa, \Xi, \kappa')}$$

$$\textsc{E-assign}\quad \frac{\phi = \mathbf{as}(v, \rho)}{\mathbf{ev}(\llbracket v\texttt{=}e \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi)}$$

$$\textsc{E-load}\quad \frac{d_r = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \qquad a \in d_r \qquad d \in \mathit{lookupProp}(v, a, \sigma)}{\mathbf{ev}(\llbracket s\,\texttt{.}\,v \rrbracket, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma, \iota, \kappa, \Xi)}$$

$$\textsc{K-assign-var}\quad \frac{v \in \mathrm{Dom}(\rho) \qquad a = \rho(v) \qquad \sigma' = \sigma \sqcup [a \mapsto d]}{\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma', \iota, \kappa, \Xi)}$$

$$\textsc{K-assign-global}\quad \frac{v \notin \mathrm{Dom}(\rho) \omega = \sigma(a_0)[v \mapsto d] \qquad \sigma' = \sigma \sqcup [a_0 \mapsto \omega]}{\mathbf{ko}(d, \sigma, \mathbf{as}(v, \rho) : \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma', \iota, \kappa, \Xi)}$$

$$\textsc{K-store}\quad \frac{d_r = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \qquad a \in d_r \qquad \omega = \sigma(a)[v \mapsto d] \qquad \sigma' = \sigma \sqcup [a \mapsto \omega]}{\mathbf{ko}(d, \sigma, \mathbf{st}(s, v, \rho) : \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma', \iota, \kappa, \Xi)}$$

$$\textsc{K-ctr-return}\quad \frac{(\iota', \kappa') \in \Xi(\kappa) \qquad d' = \{a_{\text{this}}\} \qquad (\llbracket \texttt{new}\ v\texttt{(}s\texttt{)} \rrbracket, \_, \_, a_{\text{this}}, \_) = \kappa}{\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) \mapsto \mathbf{ko}(d', \sigma, \iota', \kappa', \Xi)}$$

$$\textsc{K-fun-return}\quad \frac{(\iota', \kappa') \in \Xi(\kappa)}{\mathbf{ko}(d, \sigma, \langle\rangle, \kappa, \Xi) \mapsto \mathbf{ko}(d, \sigma, \iota', \kappa', \Xi)}$$

Figure C.2: Transition rules of the abstract machine 1.

E-FUN

$$\frac{\begin{array}{c} a = \mathit{allocFun}(f, \rho, \sigma, \iota, \kappa) \\ a' = \mathit{allocProto}(f, \rho, \sigma, \iota, \kappa) \qquad \sigma' = \sigma \sqcup [a \mapsto \{\omega_f\}, a' \mapsto \{\omega_{\text{proto}}\}] \\ \omega_f = [\text{“call”} \mapsto \{(f, \rho)\}, \text{“proto”} \mapsto \varnothing, \texttt{prototype} \mapsto \{a'\}] \\ \omega_{\text{proto}} = [\text{“proto”} \mapsto \varnothing] \end{array}}{\mathbf{ev}(\underbrace{[\![\texttt{function (}v\texttt{)\{var }v_h\texttt{; }e\texttt{\}}]\!]}_{f}, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ko}(\{a\}, \sigma', \iota, \kappa, \Xi)}$$

E-CTR-CALL

$$\frac{\begin{array}{c} d_f = \mathit{evalSimple}(v, \rho, \sigma, \kappa) \qquad d_{\text{arg}} = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\ a_f \in d_f \qquad \omega_f = \sigma(a_f) \qquad c \in \omega_f(\text{“call”}) \\ a_{\text{this}} = \mathit{allocCtr}(e, \rho, \sigma, \iota, \kappa) \qquad \omega = [\text{“proto”} \mapsto \omega_f(\texttt{prototype})] \\ \sigma' = \sigma \sqcup [a_{\text{this}} \mapsto \{\omega\}] \qquad \kappa' = (e, c, d_{\text{arg}}, a_{\text{this}}, \sigma) \end{array}}{\mathbf{ev}(\underbrace{[\![\texttt{new }v\texttt{(}s\texttt{)}]\!]}_{e}, \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathit{evalCall}(c, d_{\text{arg}}, \sigma', \iota, \kappa, \Xi, \kappa')}$$

E-STORE

$$\frac{\phi = \mathbf{st}(s, v, \rho)}{\mathbf{ev}([\![s\,.\,v\texttt{=}e]\!], \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ev}(e, \rho, \sigma, \phi : \iota, \kappa, \Xi)}$$

E-RETURN

$$\frac{d = \mathit{evalSimple}(s, \rho, \sigma, \kappa)}{\mathbf{ev}([\![\texttt{return }s]\!], \rho, \sigma, \iota, \kappa, \Xi) \mapsto \mathbf{ko}(d, \rho, \sigma, \langle\rangle, \kappa, \Xi)}$$

Figure C.3: Transition rules of the abstract machine 2.

C.2.3 Program Evaluation

Function $\mathcal{I} : \mathsf{Exp} \to \mathit{State}$ injects an expression into the state-space. It returns an initial evaluation state with empty environment, initial store, empty local continuation, and the root context (κ_0) as interprocedural continuation.

$$\frac{\kappa_0 = (e, \bot, \bot, a_0, \sigma_0) \qquad \sigma_0 = [a_0 \mapsto []]}{\mathcal{I}(e) = \mathbf{ev}(e, [], \sigma_0, \kappa_0, \epsilon, [])}$$

The initial store σ_0 contains the global object at address a_0, which we assume to be available throughout the semantics.

If ς_0 is the initial state for program e, then the evaluation of this program corresponds with computing the transitive closure of $\mapsto$ starting from ς_0.

$$\frac{\kappa_0 = (e, \bot, \bot, a_0, \sigma_0) \qquad \varsigma_0 = \mathcal{I}(e) \qquad \varsigma_0 \mapsto^* \mathbf{ko}(d, \_, \langle\rangle, \kappa_0, \_)}{d \in \mathcal{E}(e)}$$

Our static analysis requires a finite model (i.e flow graph) for every possible (finite) program. This can be guaranteed by parameterizing the abstract machine with an address allocator that draws addresses from a finite set *Addr*. When both Var and *Addr* are finite sets, then the entire state-space is finite as well and $\mapsto$, which is monotonic, has a least fixed point.

C.3 Phase 2: A Posteriori Abstract Interpretation of Meta Operations

During the second phase, the appropriate meta program operations must be triggered by exploring the output of the abstract interpretation from the first phase. This is the responsibility of the Execution Explorer (EE) (cf. Section 6.2.2). The semantics of JS_0 therefore not only form an *operational* foundation for a static analysis of the base program, but also for a *result-oriented* abstract interpretation of the meta code in this second phase. Both objectives are fulfilled by representing the semantics of JS_0 as an abstract machine that models evaluation in small steps.

Intercepting operations is relatively straightforward when looking at the transition relation for JS_0 (cf. Appendix C.2.2). Again, taking the interception of function calls as an example, we can observe that rules E-FUN-CALL, E-METHOD-CALL, and E-CTR-CALL are states in which a function is about to be called.

Upon detection of an operation that must be intercepted, the EE has to invoke the associated meta operation on the handler object. However, instead of duplicating the behavior of the run time meta operation by extending the abstract machine of the static analysis, the EE relies on the meta operations defined on the run time handler (i.e `META`). At this point, the EE will initiate an abstract interpretation invoking the appropriate trap on the handler. This is possible because the base program already included the code of meta program, and the base and meta language

are the necessarily the same (because the EM is based in source code instrumentation). This forms the crux of our approach.

C.3.1 Obtaining the Callable Object

Suppose ς is a state in which an operation v_m must be intercepted. The EE first has to obtain the handler object `META` from this state, which, as before, we assume is a property of the global object. Rule Trap-Callable obtains a reference d_M for the handler object by looking up the $[\![\texttt{META}]\!]$ property on the global object with address a_0. The trap method is looked up as a property with name v_m on the handler, resulting in a reference d_m to a function object. Finally, the value of the "call" special property is returned.

Trap-Callable

$$\frac{\begin{array}{c} d_M \in \mathit{lookupProp}([\![\texttt{META}]\!], a_0, \sigma_\varsigma) \\ a_M \in d_M \qquad d_m \in \mathit{lookupProp}(v_m, a_M, \sigma_\varsigma) \\ a_m \in d_m \qquad \omega_m = \sigma_\varsigma(a_m) \qquad c_m \in \omega_m(\text{“call”}) \end{array}}{c_m \in \mathit{trap}(\varsigma, v_m)}$$

C.3.2 Intercepting Base Program Operations and Invoking Traps

We define a relation *handle* that the EE uses when exploring the resulting base program's flow graph. Relation *handle* takes a state and a meta store, and invokes the required trap if required. It returns the result of a trap invocation and a resulting meta store. The meta store is required for maintaining meta state, and is explained below. Our explanation here focuses on interception of operations and trap invocation, and as an example we illustrate the rule for intercepting method calls and invoking the corresponding apply trap.

Suppose ς is a state that, upon transition, results in a method call, i.e., transition rule E-method-call in Figure C.3 applies to ς. From the specification of the EM the apply trap has to be invoked. The EE encode this behavior by means of the rule Handle-Method-Call for relation

handle.

$$\textsc{Handle-Method-Call}\quad \frac{\begin{array}{c}\mathbf{ev}(\llbracket s_0\,.\,v(s_1)\rrbracket,\ldots)=\varsigma \qquad c_m\in \mathit{trap}(\varsigma,\llbracket\texttt{apply}\rrbracket)\\ d_{\text{this}}=\mathit{evalSimple}(s_0,\rho,\sigma,\kappa) \qquad d_{\text{arg}}=\mathit{evalSimple}(s_1,\rho,\sigma,\kappa)\\ \kappa_m=(\bot,c_m,d_{\text{arg}},a_0,\sigma_m) \qquad \kappa_r=(\bot,\bot,\bot,a_0,\sigma_m) \qquad \sigma_m=\sigma_\varsigma\sigma_M\\ \mathit{evalCall}(c_m,d_{\text{arg}},\sigma_m,\langle\rangle,\kappa_r,\Xi,\kappa_m)\mapsto^*\mathbf{ko}(d_r,\sigma_r,\langle\rangle,\kappa_r,\_)\\ \sigma'_M=\sigma_r|_{\mathcal{R}_{\sigma_r}(a_M)}\end{array}}{(d_r,\sigma'_M)\in\mathit{handle}(\varsigma,\sigma_M)}$$

$$\textsc{Handle-No-Intercept}\quad \frac{\text{no intercept for }\varsigma}{(\bot,\sigma_M)\in\mathit{handle}(\varsigma,\sigma_M)}$$

Rule Handle-Method-Call applies when state ς is effectively a method call, which is the case when transitioning from an **ev** state with a method call as control component. Relation *trap* is used to obtain the callable *apply* trap. The remainder of the rule specifies the arguments for the call to the semantic *evalCall* function, which actually invokes the trap. Like a regular function application in JS_0, the trap function is called with an empty local continuation. The trap function is called with the empty meta-continuation as if its body were top-level code. As a consequence, upon return of the trap function there is no continuation possible and a final state is reached.

Side effects can occur during the abstract interpretation of the handler trap in ς. Therefore, the EE maintains a meta store (σ_M) and propagates those changes thorough the exploration of the base program's flow graph states. Before the *handle* apply call, the rules computes a new store σ_m for ς where the information of the meta store σ_M is "merged" with the ς store. After the abstract interpretation a new meta store σ'_M is computed containing the possible side effects that happened during call.

Rule Handle-No-Intercept applies when no intercept is required for a state, i.e., when no other rules for *handle* apply. It returns an absent trap invocation result $\bot$ and the unmodified meta store.

Function $\mathcal{R} : \mathcal{P}(\mathit{Addr}) \times \mathcal{P}(\mathit{Addr}) \times \mathit{Store} \to \mathcal{P}(\mathit{Addr})$ computes the set of all addresses that are reachable from a given root set of addresses. In general terms, the overloaded function $\mathcal{T} : X \to \mathcal{P}(\mathit{Addr})$ returns the

set of addresses directly referenced by components in the state space.

$$\begin{aligned}\mathcal{T}((f,\rho)) &= \mathcal{T}(\rho)\\ \mathcal{T}(a) &= \{a\}\\ \mathcal{T}(\omega) &= \mathcal{T}(\text{Range}(\omega))\\ \mathcal{T}(\rho) &= \text{Range}(\rho)\\ \mathcal{T}(\delta) &= \varnothing\\ \mathcal{T}(\{x_0,\ldots,x_n\}) &= \bigcup_{i\in 0..n} \mathcal{T}(x_i)\end{aligned}$$

Reachable addresses

$$\frac{a' \in \mathcal{T}(\sigma(a))}{a \rightsquigarrow_{\mathcal{T},\sigma} a'} \qquad \frac{a \in \mathcal{T}(d) \qquad a \rightsquigarrow^*_{\mathcal{T},\sigma} a'}{a \in \mathcal{R}_\sigma(d)}$$

Figure C.4 shows the other rules for trapping operations which are specified in a similar fashion. Rules Handle-Property-Read and Handle-Property-Write handle object property access and updates, respectively. Rule Handle-New-Obj-Expressions handles `new Obj()` expressionsm and rule Handle-Var-Assignment-Expressions handles variable assignment.

C.3.3 Execution Exploration While Maintaining Meta State

For stateless meta code it suffices to visit all explored states once in an unspecified order and passing them to relation *handle*, ignoring the meta store from that relation. If, for a particular state, the value returned by *handle* subsumes the abstracted `META.HALT`, then this indicates that, according to the static analysis, a base program operation was intercepted that should halt the execution.

In case the meta code is stateful, then the meta state has to be maintained as well. In this case, detecting traps that halt the execution of a program must be expressed as a fixed point computation over the flow graph. Let $(\hookrightarrow) \sqsupseteq D \times \mathit{State} \times \mathit{Store} \times D \times \mathit{State} \times \mathit{Store}$ be the relation that operates on triples representing a state and the result of "handling" a state through *handle*, i.e., a meta value and meta store. The single rule below describes a transition from a reachable triple for one state to another triple for its successor state based on an edge in the flow graph

HANDLE-PROPERTY-READ

$$\frac{\begin{array}{c}\mathbf{ev}(\llbracket s\,.\,v \rrbracket, \ldots) = \varsigma \qquad c_m \in \mathit{trap}(\varsigma, \texttt{get}) \\ d_{\text{this}} = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \qquad d_v = \mathit{evalSimple}(v, \rho, \sigma, \kappa) \\ \kappa_m = (\bot, c_m, \bot, a_0, \sigma_m) \qquad \kappa_r = (\bot, \bot, \bot, a_0, \sigma_m) \qquad \sigma_m = \sigma_\varsigma \sigma_M \\ \mathit{evalCall}(c_m, [d_{\text{this}}, d_v], \sigma_m, \langle\rangle, \kappa_r, \Xi, \kappa_m) \mapsto^* \mathbf{ko}(d_r, \sigma_r, \langle\rangle, \epsilon, \_) \\ \sigma'_M = \sigma_r|_{\mathcal{R}_{\sigma_r}(a_M)}\end{array}}{(d_r, \sigma'_M) \in \mathit{handle}(\varsigma, \sigma_M)}$$

HANDLE-PROPERTY-WRITE

$$\frac{\begin{array}{c}\mathbf{ev}(\llbracket s_0\,.\,v\texttt{=}s_1 \rrbracket, \ldots) = \varsigma \qquad c_m \in \mathit{trap}(\varsigma, \texttt{set}) \\ d_0 = \mathit{evalSimple}(s_0, \rho, \sigma, \kappa) \qquad d_1 = \mathit{evalSimple}(s_1, \rho, \sigma, \kappa) \\ \kappa_m = (\bot, c_m, d_0, a_0, \sigma_m) \qquad \kappa_r = (\bot, \bot, \bot, a_0, \sigma_m) \qquad \sigma_m = \sigma_\varsigma \sigma_M \\ \mathit{evalCall}(c_m, [d_0, v, d_1], \sigma_m, \langle\rangle, \epsilon, \Xi, \kappa_m) \mapsto^* \mathbf{ko}(d_r, \sigma_r, \langle\rangle, \epsilon, \_) \\ \sigma'_M = \sigma_r|_{\mathcal{R}_{\sigma_r}(a_M)}\end{array}}{(d_r, \sigma'_M) \in \mathit{handle}(\varsigma, \sigma_M)}$$

HANDLE-NEW-OBJ-EXPRESSIONS

$$\frac{\begin{array}{c}\mathbf{ev}(\llbracket \texttt{new } v\texttt{(}s\texttt{)} \rrbracket, \ldots) = \varsigma \qquad c_m \in \mathit{trap}(\varsigma, \texttt{construct}) \\ d_0 = \mathit{evalSimple}(v, \rho, \sigma, \kappa) \qquad d_1 = \mathit{evalSimple}(s, \rho, \sigma, \kappa) \\ \kappa_m = (\bot, c_m, d_0, a_0, \sigma_m) \qquad \kappa_r = (\bot, \bot, \bot, a_0, \sigma_m) \qquad \sigma_m = \sigma_\varsigma \sigma_M \\ \mathit{evalCall}(c_m, [d_0, d_1], \sigma_m, \langle\rangle, \kappa_r, \Xi, \kappa_m) \mapsto^* \mathbf{ko}(d_r, \sigma_r, \langle\rangle, \epsilon, \_) \\ \sigma'_M = \sigma_r|_{\mathcal{R}_{\sigma_r}(a_M)}\end{array}}{(d_r, \sigma'_M) \in \mathit{handle}(\varsigma, \sigma_M)}$$

HANDLE-VAR-ASSIGNMENT-EXPRESSIONS

$$\frac{\begin{array}{c}\mathbf{ev}(\llbracket v\texttt{=}e \rrbracket, \ldots) = \varsigma \qquad c_m \in \mathit{trap}(\varsigma, \texttt{write}) \\ d_0 = \mathit{evalSimple}(e, \rho, \sigma, \kappa) \qquad d_M \in \mathit{lookupProp}(\llbracket \texttt{META} \rrbracket, a_0, \sigma_\varsigma) \\ \kappa_m = (\bot, c_m, d_{\text{arg}}, a_0, \sigma_m) \qquad \kappa_r = (\bot, \bot, \bot, a_0, \sigma_m) \\ \sigma_m = \sigma_\varsigma \sigma_M \qquad \mathit{evalCall}(c_m, [v, d_0], \sigma_m, \langle\rangle, \epsilon, \Xi, \kappa_m) \mapsto^* \mathbf{ko}(d_r, \sigma_r, \langle\rangle, \epsilon, \_) \\ \sigma'_M = \sigma_r|_{\mathcal{R}_{\sigma_r}(a_M)}\end{array}}{(d_r, \sigma'_M) \in \mathit{handle}(\varsigma, \sigma_M)}$$

Figure C.4: Additional rules for trapping program operations during the second static analysis phase.

and the result of handling the successor state. The initial triple for $\hookrightarrow$ is the initial state ς_0 of the flow graph and the result of $handle(state_0)$.

$$\text{EE-TRANS}\quad \frac{\varsigma \to \varsigma' \in \mathcal{G}(e) \qquad (d'_r, \sigma'_M) \in handle(\varsigma', \sigma_M)}{(\_, \varsigma, \sigma_M) \hookrightarrow (d'_r, \varsigma', \sigma'_M)}$$

Computing the transitive closure of $\hookrightarrow$ then enables detecting states for which a trap returns `META.HALT`.

$$\frac{(d_0, \sigma_0) \in handle(\varsigma_0, []) \qquad (d_0, \varsigma_0, \sigma_0) \hookrightarrow^* (d_r, \varsigma, \sigma_M) \qquad \alpha([\![\texttt{META.HALT}]\!]) \sqsubseteq d_r}{halt(\varsigma)}$$

Bibliography

[ADF11] Thomas H Austin, Tim Disney, and Cormac Flanagan. Virtual Values for Language Extension. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 921–938, New York, NY, USA, 2011. Association for Computing Machinery.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS'09, pages 113–124, New York, NY, USA, 2009. Association for Computing Machinery.

[AF10] Thomas H. Austin and Cormac Flanagan. Permissive Dynamic Information Flow Analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS'10, pages 1–12, New York, NY, USA, 2010. Association for Computing Machinery.

[AGM+17] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A Survey of Dynamic Analysis and Test Generation for JavaScript. *ACM Computing Surveys*, 50(5):1–36, 2017.

[And72] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, 10 1972.

[ASF17] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. Multiple facets for dynamic information flow with exceptions. *ACM Transactions in Programming Languages and Systems*, 39(3), 2017.

[AVAB+12] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. JSand: Complete Client-Side Sandboxing of Third-Party JavaScript without Browser Modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC'12, pages 1–10, New York, NY, USA, 2012. Association for Computing Machinery.

[BHS12] Arnar Birgisson, Daniel Hedin, and Andrei Sabelfeld. Boosting the Permissiveness of Dynamic Information-Flow Tracking by Testing. In *Computer Security – ESORICS 2012*, volume 7459 of *ESORICS 2012, Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2012.

[Bib77] Ken Biba. *Integrity Considerations for Secure Computer Systems*. Defense Technical Information Center, 04 1977.

[Bie13] Nataliia Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming*, 82(8):243–262, November 2013. Automated Specification and Verification of Web Systems.

[BLH08] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'08/FSE-16, pages 36–47, New York, NY, USA, 2008. Association for Computing Machinery.

[BR16] Nataliia Bielova and Tamara Rezk. A Taxonomy of Information Flow Monitors. In *Principles of Security and Trust*, volume 9635 of *POST 2016, Lecture Notes in Computer Science*, pages 46–67. Springer, Berlin, Heidelberg, 2016.

[BRGH14] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information Flow Control in WebKit's JavaScript Bytecode, 2014.

[BSS17] Musard Balliu, Daniel Schoepe, and Andrei Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *Computer Security – ESORICS 2017*, volume 10492 of *ESORICS 2017, Lecture Notes in Computer Science*, pages 124–145. Springer, Cham, 2017.

[CGDD16] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. Linvail - A General-Purpose Platform for Shadow Execution of JavaScript. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 260–270. IEEE, 2016.

[CMJL09] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged Information Flow for Javascript. *ACM SIGPLAN Notices*, 44(6):50–62, 2009.

[CN15] Andrey Chudnov and David A. Naumann. Inlined Information Flow Monitoring for JavaScript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS'15, pages 629–643, New York, NY, USA, 2015. Association for Computing Machinery.

[CRB16] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. Content Security Problems? Evaluating the Effectiveness of Content Security Policy in the Wild. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 1365–1375, New York, NY, USA, 2016. Association for Computing Machinery.

[CUT+21] Stefano Calzavara, Tobias Urban, Dennis Tatang, Marius Steffens, and Ben Stock. Reining in the web's inconsistencies with site policy. In *Network and Distributed Systems Security (NDSS) Symposium 2021*, February 2021.

[DD77] Dorothy E Denning and Peter J Denning. Certification of

Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, 1977.

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[dLW03] Oege de Moor, David Lacey, and Eric Van Wyk. Universal Regular Path Queries. *Higher-Order and Symbolic Computation*, 16:15–35, Mar 2003.

[DNM15] Sophia Drossopoulou, James Noble, and Mark S. Miller. Swapsies on the Internet: First Steps towards Reasoning about Risk and Trust in an Open World. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, PLAS'15, pages 2–15, New York, NY, USA, 2015. Association for Computing Machinery.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[Ecm15] Ecma International. *ECMAScript 2015 Language Specification*. Geneva, 6th edition, June 2015.

[ES00] Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP'00, pages 246–255, USA, 2000. IEEE Computer Society.

[FBJ+16] Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. Chapter One - Security Testing: A Survey. volume 101 of *Advances in Computers*, pages 1–51. Elsevier, 2016.

[FF87] Mattias Felleisen and D. P. Friedman. A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'87, page 314, New York, NY, USA, 1987. Association for Computing Machinery.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[FSJRS16] José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt. Hybrid Typing of Secure Information Flow in a JavaScript-Like Language. In Pierre Ganty and Michele Loreti, editors, *Trustworthy Global Computing*, volume 9533 of *TGC 2015, Lecture Notes in Computer Science*, pages 63–78. Springer, Cham, 2016.

[Gho11] Debasish Ghosh. *DSLs in Action*. Manning Pubs Co Series. Manning, 2011.

[GL09] Salvatore Guarnieri and Benjamin Livshits. GATEKEEPER - Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, USA, 2009. USENIX Association.

[GL10] Salvatore Guarnieri and Benjamin Livshits. GULFSTREAM - Staged Static Analysis for Streaming JavaScript Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, USA, 2010. USENIX Association.

[GM82] Joseph A Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE, 1982.

[Goo09] Google Inc. Google Code Archive - Browser Security Handbook, part 2, 2009. (Accessed on 03/31/2020).

[GPS15] Liang Gong, Michael Pradel, and Koushik Sen. JITProf: Pinpointing JIT-Unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, New York, NY, USA, 2015. Association for Computing Machinery.

[GPSS15] Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. DLint: Dynamically Checking Bad Coding Practices in JavaScript. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA'2015, pages 94–105, New York, NY, USA, 2015. Association for Computing Machinery.

[GPT+11] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the World Wide Web from Vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA'11, pages 177–187, New York, NY, USA, 2011. Association for Computing Machinery.

[HBBS14] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC'14, pages 1663–1671, New York, NY, USA, 2014. Association for Computing Machinery.

[HBS15] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, CSF'15, pages 351–365, USA, 2015. IEEE Computer Society.

[HJS12] Kevin W. Hamlen, Micah Jones, and Meera Sridhar. Aspect-Oriented Runtime Monitor Certification. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7214 of *TACAS 2012, Lecture Notes in Computer Science*, pages 126–140. Springer, Berlin, Heidelberg, 2012.

[HL06] Michael Howard and Steve Lipner. *The security development lifecycle*. SDL, a process for developing demonstrably more secure software. Microsoft Pr, 2006.

[HS06] Sebastian Hunt and David Sands. On flow-sensitive security types. *ACM SIGPLAN Notices*, 41(1):79–90, 2006.

[HS12a] Daniel Hedin and Andrei Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D:Information and Communication Security*, pages 319–347. IOS Press, 2012.

[HS12b] Daniel Hedin and Andrei Sabelfeld. Information-Flow Security for a Core of JavaScript. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 3–18. IEEE, 2012.

[HSPS17] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *International Conference on Principles of Security and Trust - Volume 10204*, pages 49–70, Berlin, Heidelberg, 2017. Springer-Verlag.

[HTM] HTML Living Standard. `https://html.spec.whatwg.org/multipage/browsers.html`. (Accessed on 05/28/2020).

[HV05] O Hallaraker and G Vigna. Detecting malicious JavaScript code in Mozilla. In *IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 85–94. IEEE, June 2005.

[JH10] Micah Jones and Kevin W. Hamlen. Disambiguating Aspect-Oriented Security Policies. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD'10, pages 193–204, New York, NY, USA, 2010. Association for Computing Machinery.

[JVH14] James Ian Johnson and David Van Horn. Abstracting Abstract Control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS'14, pages 11–22, New York, NY, USA, 2014. Association for Computing Machinery.

[KdRB93] Gregor Kiczales, Jim des Riviéres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1993.

[Kim] Bjoern Kimminich. GitHub - bkimminich/juice-shop: OWASP Juice Shop: Probably the most modern and sophisticated insecure web application. `https://github.com/bkimminich/juice-shop`. (Accessed on 04/12/2021).

[KT15] Matthias Keil and Peter Thiemann. TreatJS: Higher-Order Contracts for JavaScript. *CoRR*, abs/1504.08110, 2015.

[KYC+08] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. JavaScript Instrumentation in Practice. In G Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *APLAS 2008, Lecture Notes in Computer Science*, pages 326–341. Springer, Berlin, Heidelberg, 2008.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[LBJS07] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A Schmidt. Automata-Based Confidentiality Monitoring. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435 of *ASIAN 2006, Lecture Notes in Computer Science*, pages 75–89. Springer, Berlin, Heidelberg, 2007.

[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, February 2005.

[LH05] Philippe Le Hágaret. W3C Document Object Model, January 2005.

[LKG+17] Sebastian Lekies, Krzysztof Kotowicz, Samuel Groß, Eduardo A Vela Nava, and Martin Johns. Code-Reuse Attacks for the Web: Breaking Cross-Site Scripting Mitigations via Script Gadgets. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1709–1723, New York, NY, USA, 2017. Association for Computing Machinery.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, volume 14 of *SSYM'05*, page 18, USA, 2005. USENIX Association.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Communications of the ACM*, 58(2):44–46, January 2015.

[MC11] Scott Moore and Stephen Chong. Static Analysis for Efficient Hybrid Information-Flow Control. In *2011 IEEE 24th Computer Security Foundations Symposium*, pages 146–160. IEEE, 2011.

[MFM10] Leo A. Meyerovich, Adrienne Porter Felt, and Mark S. Miller. Object views: Fine-Grained Sharing in Browsers. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 721–730, New York, NY, USA, 2010. Association for Computing Machinery.

[Mil06] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* PhD thesis, Johns Hopkins University, USA, 2006. AAI3245526.

[ML10] Leo A Meyerovich and Benjamin Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[Moza] Mozilla Developer Network. Content Security Policy (CSP). `https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP`.

[Mozb] Mozilla Developer Network. Same-origin policy. `https://developer.mozilla.org/en-US/docs/`

Web/Security/Same-origin_policy. (Accessed on 03/18/2019).

[MPS12] Jonas Magazinius, Phu H Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In T. Aura, K. Järvinen, and K. Nyberg, editors, *Information Security Technology for Applications*, volume 7127 of *NordSec 2010, Lecture Notes in Computer Science*, pages 239–255. Springer, Berlin, Heidelberg, 2012.

[MSBG21] Aäron Munster, Angel Luis Scull Pupo, Jim Bauwens, and Elisa Gonzalez Boix. Oron: Towards a Dynamic Analysis Instrumentation Platform for AssemblyScript. ProWeb'21, 2021.

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja: Safe active content in sanitized JavaScript. Google white paper, June 2008.

[MSR+19] Marius Musch, Marius Steffens, Sebastian Roth, Ben Stock, and Martin Johns. ScriptProtect: Mitigating Unsafe Third-Party JavaScript Practices. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS'19, pages 391–402, New York, NY, USA, 2019. Association for Computing Machinery.

[NBF+18] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. A Better Facet of Dynamic Information Flow Control. In *Companion Proceedings of the The Web Conference 2018*, WWW'18, pages 731–739, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee.

[NSD16] Jens Nicolay, Valentijn Spruyt, and Coen De Roover. Static Detection of User-Specified Security Vulnerabilities in Client-Side JavaScript. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS'16, pages 3–13, New York, NY, USA, 2016. Association for Computing Machinery.

[NSDD17] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Purity analysis for JavaScript through abstract interpretation. *Journal of Software: Evolution and Process*, 29(12):e1889, 2017. e1889 smr.1889.

[OWAa] OWASP Inc. OWASP Foundation, the Open Source Foundation for Application Security. `https://owasp.org/`. (Accessed on 04/01/2020).

[OWAb] OWASP Inc. OWASP Top Ten. `https://owasp.org/www-project-top-ten/`. (Accessed on 04/01/2020).

[Pan14] G K Pannu. A Survey on Web Application Attacks. *IJCSIT) International Journal of Computer Science and Information Technologies*, 5(3):4162–4166, 2014.

[PI20] Ponemon Institute and IBM Security. Cost of a Data Breach Study — IBM. `https://www.ibm.com/security/data-breach`, 2020. (Accessed on 04/11/2021).

[PPA+20] Phu H. Phung, Huu-Danh Pham, Jack Armentrout, Panchakshari N. Hiremath, and Quang Tran-Minh. A User-Oriented Approach and Tool for Security and Privacy Protection on the Web. *SN Computer Science*, 1(222):1–16, 2020.

[PSC09] Phu H Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS'09, pages 47–60, New York, New York, USA, 2009. Association for Computing Machinery.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[Rau14] Axel Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers*. O'Reilly Media, 2014.

[RDW+07] Charles Reis, John Dunagan, Helen J Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield:

Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web (TWEB)*, 1(3):11–es, September 2007.

[RHZN+13] Gregor Richards, Christian Hammer, Francesco Zappa Nardelli, Suresh Jagannathan, and Jan Vitek. Flexible Access Control for Javascript. *ACM SIGPLAN Notices*, 48(10):305–322, October 2013.

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF'10, pages 186–199, USA, 2010. IEEE Computer Society.

[RS16] Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution - Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.

[SBR17] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. On the Content Security Policy Violations Due to the Same-Origin Policy. In *Proceedings of the 26th International Conference on World Wide Web*, WWW'17, page 877–886, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[Sch00] Fred B Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[SCN+18] Angel Luis Scull Pupo, Laurent Christophe, Jens Nicolay, Coen De Roover, and Elisa Gonzalez Boix. Practical Information Flow Control for Web Applications. In Christian Colombo and Martin Leucker, editors, *Runtime Verification*, volume 1123 of *RV 2018, Lecture Notes in Computer Science*, pages 372–388. Springer, Cham, 2018.

[SKBG13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A Tool Framework for Concolic Testing, Selective Record-Replay, and Dynamic Analysis of JavaScript. In *Proceedings of the 2013 9th Joint Meeting*

on Foundations of Software Engineering, ESEC/FSE 2013, pages 615–618, New York, NY, USA, 2013. Association for Computing Machinery.

[SM03] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

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

[SNE+19] Angel Luis Scull Pupo, J. Nicolay, K. Efthymiadis, A. Nowé, C. De Roover, and E. Gonzalez Boix. GUARDIAML: Machine Learning-Assisted Dynamic Information Flow Control. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 624–628. IEEE, 2019.

[SNG16] Angel Luis Scull Pupo, Jens Nicolay, and Elisa Gonzalez Boix. Declaratively Specifying Security Policies For Web Applications. In *Workshop on Meta-Programming Techniques and Reflection*, 2016.

[SNG18] Angel Luis Scull Pupo, Jens Nicolay, and Elisa Gonzalez Boix. GUARDIA: Specification and Enforcement of Javascript Security Policies without VM Modifications. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang'18, New York, NY, USA, 2018. Association for Computing Machinery.

[SR14] José Fragoso Santos and Tamara Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of JavaScript. In *ICT Systems Security and Privacy Protection. SEC 2014. IFIP Advances in Information and Communication Technology*, volume 428, pages 278–292. Springer, Berlin, Heidelberg, 2014.

[SSB+19] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An Empirical

Study of Information Flows in Real-World JavaScript. In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'19)*, PLAS'19, pages 45–59, New York, NY, USA, 2019. Association for Computing Machinery.

[SSM10] Sid Stamm, Brandon Sterne, and Gervase Markham. Reining in the Web with Content Security Policy. In *Proceedings of the 19th International Conference on World Wide Web*, WWW'10, pages 921–930, New York, NY, USA, 2010. Association for Computing Machinery.

[STA18] Bassam Sayed, Issa Traoré, and Amany Abdelhalim. If-transpiler: Inlining of hybrid flow-sensitive security monitor for JavaScript. *Computers & Security*, 75:92–117, 2018.

[SYM+14] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 131–146, USA, 2014. USENIX Association.

[TEM+11] Ankur Taly, Úlfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP'11, pages 363–378, USA, 2011. IEEE Computer Society.

[TFP14a] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 49–59, New York, NY, USA, 2014. Association for Computing Machinery.

[TFP14b] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid Security Analysis of Web JavaScript Code via Dynamic Partial Evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014,

pages 49–59, New York, NY, USA, 2014. Association for Computing Machinery.

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective Taint Analysis of Web Applications. *ACM SIGPLAN Notices*, 44(6):87–97, 2009.

[VADRD+11] Steven Van Acker, Philippe De Ryck, Lieven Desmet, Frank Piessens, and Wouter Joosen. Webjail: Least-privilege integration of third-party components in web mashups. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC'11, page 307–316, New York, NY, USA, 2011. Association for Computing Machinery.

[VCM10] Tom Van Cutsem and Mark S. Miller. Proxies: Design Principles for Robust Object-Oriented Intercession APIs. *ACM SIGPLAN Notices*, 45(12):59–72, 2010.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2–3):167–187, 1996.

[VNJ+07] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS 2007. The Internet Society, 2007.

[W3C99] W3C. XML Path Language (XPath), 11 1999. (Accessed on 03/10/2021).

[WHA17] WHATWG. *HTML Standard*. The Window object. https://html.spec.whatwg.org/, 2017.

[WR13] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for JavaScript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, 2013. Association for Computing Machinery.

[WSLJ16] Lukas Weichselbaum, Michele Spagnuolo, Sebastian Lekies, and Artur Janc. CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16, pages 1376–1387, New York, NY, USA, 2016. Association for Computing Machinery.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript Instrumentation for Browser Security. *ACM SIGPLAN Notices*, 42(1), 2007.

[YL16] Ming Ying and Shu Qin Li. CSP Adoption: Current Status and Future Prospects. *Security and Communication Networks*, 9(17):4557–4573, November 2016.

[YNKM09] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-Preserving Browser-Side Scripting with BFlow. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys'09, pages 233–246, New York, NY, USA, 2009. Association for Computing Machinery.

[Zda02] S A Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[ZY17] Tin Zaw and Richard Yew. Blog — 2017 Verizon Data Breach Investigations Report (DBIR) from the Perspective of Exterior Security Perimeter — Verizon Media Platform. `https://www.verizondigitalmedia.com/blog/2017-verizon-data-breach-investigations-report/`, July 2017. (Accessed on 04/11/2021).