Information Flow Control for the Web Browser through a Mechanism of Split Addresses

Deepak Subramanian

ANNÉE 2017

CentraleSupélec

UNIVERSITE BRETAGNE LOIRE

N°d'ordre :

THÈSE / CENTRALESUPÉLEC
sous le sceau de l'Université Bretagne Loire

pour le grade de

DOCTEUR DE CENTRALESUPÉLEC

Mention : Informatique

Ecole doctorale 601 « Mathématiques et Sciences et Technologies de l'Information et de la Communication – (MathSTIC) »

présentée par

Deepak Subramanian

Préparée à l'UMR 6074 - IRISA (Equipe CIDRE)
Institut de Recherche en Informatique et Systèmes Aléatoires

Information Flow Control for the Web Browser through a Mechanism of Split Addresses

Thèse soutenue à CentraleSupélec, campus de Rennes
le: 20/12/2017

devant le jury composé de :

Hervé DEBAR
HDR, Professeur, Télécom SudParis / *rapporteur*
Isabelle CHRISMENT
HDR, Professeur, Université de Lorraine, LORIA / *rapporteur*
Thomas JENSEN
Directeur de Recherche CNRS, IRISA / *examinateur*
Erwan ABGRALL
Responsable du laboratoire SSI, DGA Maitrise de l'Information / *examinateur*
Christophe BIDAN
HDR, Professeur, CentraleSupélec / *directeur de thèse*
Guillaume HIET
Professeur associé, CentraleSupélec / *co-directeur de thèse*

Acknowledgements

I would like to once again thank the President of the jury, Dr. Thomas Jensen and the Rapporteurs of my thesis, Dr. Hervé Debar, Dr. Isabelle Chrisment, and the other honorable members of the jury, Dr. Erwan Abgrall, Dr. Christophe Bidan and Dr. Guillaume Hiet for granting me the priviledge of being part of the jury for my thesis defense.

I would like to give special thanks to Dr. Guillaume Hiet, my thesis supervisor, for his constant support and great supervision throughout the course of my work.

I would also like to thank Dr. Christophe Bidan, Dr. Ludovic Mé, Dr. Jean-François Lalande and the rest of the CIDRE team in their constant encouragement and assistance in helping me integrate into the French student life. It was a pleasure to work with them during the course of my PhD.

The staff of CentraleSupelec, including Karine Bernard and Jeannine Hardy deserve special mention in helping me in a variety of situations.

I express my heartfelt gratitude to my father, P.R. Subramanian, my mother, Savithri Subramanian, and my sister, S. Dharini, for their being my pillars during this endeavor. I can also not understate the great support from my fiancé Shruti Mohan and my soon-to-be parents-in-law, Ratnam Mohan and Jalalitha Mohan.

I also thank the rest of my family and friends, with special thanks to Bharath Kumar Venkatesh Kumar, Ha Thanh Le, Dr. Karthik Muthuswamy, Manikantan Krishnamoorthy, Dr. Yogesh Karpate, Dr. Navik Modi, Dr. Hrishikesh Deshpande, Dr. Sumit Darak, Dr. Mihir Jain, Dr. , Dr. Raghavendran Balu, Dr. Surya Narayanan, Dr. Aswin Sridharan, Dr. Manikandan Bakthavatchalam, Dr. Amrith Dhananjayan, Dr. Raj Kumar Gupta, Dr. Dilip Prasad, Dr. Rajesh Sharma, Dr. Paul Lajoie-Mazenc, Dr. Jussi Lindgren, Himalaya Jain, Dr. Regina Marin,

Dr. Chistopher Humphries, Dr. Florian Grandhomme, Dr. Laurent George, Dr. Mounir Assaf and Dr. Radoniaina Andriatsimandefitra

I have profound thanks for my colleagues at Trusted Labs who have been understanding of my needs as I pursued the final completion of my thesis while being employed.

My intership at KU Leuven, Belgium is also one of the most gratifying experiences during this period. Thanks to Dr. Frank Piessens, Dr. Lieven Desmet, Dr. Willem De Groef and the rest of the DistriNet team in KU Leuven.

It has been the greatest pleasure and a most wonderful experience to complete my PhD, I would once again express my thanks to all those who were part of my life during this endeavor.

Thank you !

Contents

Acknowledgements **1**

Table of contents **3**

Introduction **7**

1 Web browser security **11**

1.1 Web browser technologies 11

1.1.1 Working of a web browser 11

1.1.2 JavaScript 13

1.1.3 Typical modern webpage 14

1.1.4 WebRTC 15

1.2 Vulnerabilities on modern webpages 18

1.2.1 Cross-Site Scripting 18

1.2.2 Cross-Site Request Forgery 21

1.2.3 Vulnerabilities on WebRTC 22

1.3 Web security mechanisms 24

1.3.1 Security mechanisms on the server side 25

1.3.2 Security mechanisms on the web browser side 26

1.3.3 Conclusion 29

2 Related work on information flow control **31**

2.1 Background on Information Flow Control 31

2.2 Information flow control in programming languages 33

2.3 Working of IFC 34

2.3.1 IFC models 36

2.3.2 IFC properties 38

2.3.3 Types of IFC analysis 42

2.4 Possibilistic web browser security models using IFC . . . 57
2.4.1 Traditional tainting models . . . 58
2.4.2 SME and Faceted approach . . . 59
2.5 Conclusion . . . 61

3 Address Split Design **63**
3.1 General working of Address Split Design . . . 63
3.1.1 Policy specification . . . 64
3.1.2 Privileges . . . 65
3.1.3 Dictionaries . . . 66
3.1.4 Function privileges . . . 67
3.1.5 Dependency tracker . . . 71
3.2 ASD description and semantics . . . 73
3.2.1 Metavariables and environment . . . 73
3.2.2 Syntax . . . 75
3.2.3 Splitting model . . . 77
3.2.4 Assignment and substitution . . . 78
3.2.5 Functions . . . 81
3.2.6 Example in While language . . . 82
3.2.7 Applying the model to JavaScript . . . 86
3.3 Examples on JavaScript . . . 87
3.3.1 Basic functionalities: variable splitting and policy interpretation . . . 87
3.3.2 Dictionary evolution and rights propagation . . . 89
3.4 Comparison of the approaches . . . 93
3.5 Conclusion . . . 97

4 Implementation and evaluation **99**
4.1 Implementation details . . . 99
4.2 Performance evaluation . . . 101
4.2.1 Performance estimation based on number of dictionaries . . . 102
4.2.2 Comparison of performance with SME and faceted approach . . . 106
4.2.3 Impact of ASD on real websites . . . 107
4.2.4 Standard benchmark tests . . . 114
4.3 Security considerations: handling vulnerabilities . . . 117
4.3.1 Protecting the Cross-Site Request Forgery Token . . . 118
4.3.2 WebRTC . . . 120
4.3.3 Websockets . . . 121

Conclusion **125**

Bibliography **142**

Table of Figures **143**

List of Publications **149**

Introduction

The modern world has evolved to the point where many services are served exclusively through the Internet. It has recently been found that over 73.9 % of Europeans are connected to the Internet[1] and this percentage is growing at a significant rate. Many traditional services such as post are gradually being superseded by the advent of email and other services. Further, even more services such as banking and shopping are also becoming more reliant on the Internet.[2]

The working of these *web applications* depends on server-side as well as client-side software. The major piece of software on the client side that has spearheaded all these web applications is the web browser. This application is in charge of retrieving, preserving and transferring information from the server-side applications. Concretely, the web browser is the interface between the users and server side application: it is used to navigate to the webpages of the server side application and display their contents as intended. Notice that modern webpages often include content from multiple websites so as to personalize its for each user by integrating enriched functionalities such as calendars, advertisements, embedded audio and video as well as feeds from varying sources.

Because these web applications provide to users sensitive services such as banking and shopping, their security is of pivotal importance. From the server side, the range of the security threats includes but is not limited to attacks such as denial of service, security misconfiguration and customer data compromise. Some of these attacks, such as SQL injection, rely on the injection of malicious code on the server side. These security threats still exist and are being addressed by many projects (such as Cloudflare,[3] application security scanners,[4] projects from the web-application security consortium,[5] etc.). From the client side, some of the security issues come with the web browser itself: as any software, it can be subject to attacks such as buffer overflows. In this regard, modern web browsers take great amount of care for their

[1] Internet stats, `http://www.internetworldstats.com/stats.htm`

[2] The Growth of Online Banking, `http://www.wwwmetrics.com/banking.htm`

[3] Cloudflare, `https://www.cloudflare.com/security/`

[4] List of web application security scanners, `https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools`

[5] Web application security consortium – projects, `http://www.webappsec.org/projects/`

source code's security by providing sandboxing that can prevent a webpage from inadvertently accessing system files and other system objects.[6]

However, it is not sufficient to independently prevent security threats from each side, because some security issues of web applications are intrinsic to the web applications themselves. For instance, the modern internet consists of several webpages which are *mashup* webpages. A mashup, in web development, is a web page, or web application, that uses content from more than one source to create a single new service displayed in a single graphical interface [Wikd].

Many websites use a session *cookie* to allow users to access services without requiring them to authenticate each time. In this case, acquiring the session *cookie* is then sufficient to impersonate a user. This can be made possible through the use of a malicious webpage, or a malicious script embedded into a legitimate *mashup* webpage.

More generally, the difficulty of web application security lies in the fact that exploiting a server-side vulnerability can have a client-side impact, and vice versa. It must be noted that many vulnerabilities on the server side such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) have a direct impact on the web browser. Webpages can contain content and scripts from several web application server. A simple example is an advertisement on a webpage. Unless the advertisement provider provides secure content to the webpage, the webpage is inherently vulnerable.

In this thesis, we focus on the client side security of the web browsers. We pay attention to protecting the user's sensitive webpage data from being leaked (confidentiality) and from preventing the modification of sensitive webpage data by unauthorized code (integrity). For this research, we limit ourselves to the context of JavaScript on the web browser. We look into the compilation and execution process of JavaScript and provide a mechanism to secure the variables containing sensitive data on a webpage from being manipulated by unauthorized code.

Motivation for the thesis

Over the years, there has been a lot of improvements of web application security including the use of the Origin header and the key conceptualization of *Same-Origin Policy* (SOP). The term origin refers to the web application server on which a given resource resides or is to be created [FGM+99].

Today, it has become mandatory for web browsers to specify as part of every HTTP request (using the Origin header) the origin web application server for the current webpage. The web application server can then check the Origin header to decide how the request needs to be

[6] Chromium Sandbox, `http://tinyurl.com/ChromiumSandbox`

processed.

The SOP states that a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin, i.e., they comes from the same websites. This policy prevents a malicious script on one webpage from obtaining access to sensitive data on another webpage.

Notice that all the security measures that have been proposed so far tend to be specific to each type of vulnerabilities. Beside, they require that both the client-side, i.e., the web browser, and the server-side web application implement complementary mechanisms. Conversely, if one of the party do not implement the appropriate mechanisms, the security cannot be ensured. And for sure, malicious web sites will not implement these mechanisms. Consequently, the effectiveness of these measures is intrinsically limited.

Let us consider the list of top ten vulnerabilities by the Open Web Application Security Project (OWASP) [Opea] which is regarded as the standard bearer for this domain. According to this list, the main vulnerability of web applications is the cross-side scripting (XSS) that allows a attacker to inject malicious script into a legitimate webpage. Once the malicious script has been injected, any user that accesses the legitimate webpage will have the malicious script executed by his web browser, potentially leading to user's sensitive data theft. Since the malicious script is part of the legitimate webpage, none of the previous security measures allows to really prevent such attack.

An attack based on XSS has even more impact if the webpage containing the malicious script is itself a part of a mashup webpage, as for instance an advertisement that is included into a webpage of an e-commerce website. Hence, vulnerabilities of a webpage cannot be prevented unless all the content provided by the various web application servers are audited for vulnerabilities and subsequently fixed. Of course, this is an unrealistic hypothesis, especially because malicious webpages with malicious scripts will still exist.

In this thesis, we do not consider solving the vulnerabilities themselves but would like to provide a mechanism where user's sensitive information is protected from disclosure as well as unauthorized modifications despite the vulnerability being exploited. Our objective is to propose a preventive enforcement mechanism that helps in maintaining both the confidentiality as well as integrity of the user's sensitive information despite the presence of malicious scripts. Further, we wish our mechanism must not be stuck or not ask for direct user input as a part of making its decision since the web browser is intended for use by all and is not limited to experts. Finally, there is a need to achieve these objectives without causing severe costs to time taken for execution.

For that purpose, we affirm that the vulnerabilities based on malicious script are characterized by illegal information flows. Hence, we propose to develop an approach based on

Information Flow Control (Information Flow Control (IFC)). Indeed, IFC-based approaches are more encompassing in their scope to solve problems and also provide more streamlined solutions to handling the information security in its entirety.

This thesis presents such an approach with an implementation on the v8 JavaScript engine of the Chromium web browser. Our work have been peer-reviewed, published and presented at international venues:

- We analyzed potential risks of WebRTC, a HTML5 communication technology, in joint collaboration with KU Leuven and published the results in the 31st Annual ACM Symposium on Applied Computing 2016 [DGSJ+16].
- The core of our research, the Address-Split Design (ASD) was presented in the 9th International Conference on Security of Information and Networks 2016 [SHB16].
- Further, we have presented a follow up approach to ASD that adds a learning mechanism to auto correct any uncaught information leaks over time in the 9th International Symposium on Foundations and Practice of Security 2016 [SHB17].

Dissertation Outlines

This dissertation is organized as follows. In the Chapter 1 we highlight some of the modern web technologies and the vulnerabilities that exist in the web browsers. We also give a summary of the existing mechanisms that could be used to provide security to web-pages running on the web browsers.

In Chapter 2, we summarize the various related work pertaining to the field of IFC as well as the use of IFC-based approaches for web browser security. We describe the properties and analysis methods that have been used in this area of research. We also provide some insights into how our proposed mechanism compares with these related work.

The core of the Address Split Design (ASD), which is our model for IFC in the web browser, is described in the Chapter 3. ASD is a practical IFC model that relies on modifications to the symbol table mechanism to protect secret variables from disclosure. We show the differences in the working of our model compared to other related work.

In the Chapter 4, we describe the implementation of our model on the v8 JavaScript engine. We provide some highlights on the performance of our solution and the impact of implementing our model on a standard web browser. We also describe how our model can help to tackle the security issues mentioned in Chapter 1.

We provide a conclusion with a contemplation of possible future work at the end of the dissertation.

Chapter 1

Web browser security

In the Section 1.1 we give a brief overview of the main technologies that are used by modern websites and implemented on the client side, i.e. by the web browser. We then discuss vulnerabilities on those modern webpage in Section 1.2. Especially, we present vulnerabilities that we have identified in Web Real-Time Communication (WebRTC) technology [DGSJ+16]. Finally, we introduce in Section 1.3 the classical approaches that have been proposed to enforce the security of web browsers and discuss their limitation.

1.1 Web browser technologies

We first describe the fundamental working of the web browser. This is followed by a brief introduction to JavaScript. Then, we introduce the problem of modern webpages that include scripts of third-parties. Finally, we describe in more details the functionality of WebRTC, a technology that is representative of modern features that are added to the browser.

1.1.1 Working of a web browser

A web browser's responsibility is to first navigate to a web page and display its contents as intended. The web page can be found using the Uniform Resource Locator (URL).[1] The structure of a URL is shown in Figure 1.1. It must be noted that `user information`, while still valid in the URL specification, is no longer supported by many browsers [For]. The reason is there are several malicious sites which use this strategy to trick users. URL such as `http://www.google.com:jkahshsfjkjdfjbjd@kldfdjkhhebahtk.com/` may be confused as a URL leading to the host `www.google.com` while this string is merely the username

[1] RFC 1738, `http://www.ietf.org/rfc/rfc1738.txt`

in the site `kldfdjkhhebahtk.com`. Hence, modern browsers are slowly dropping their support for this part of the URL specification.

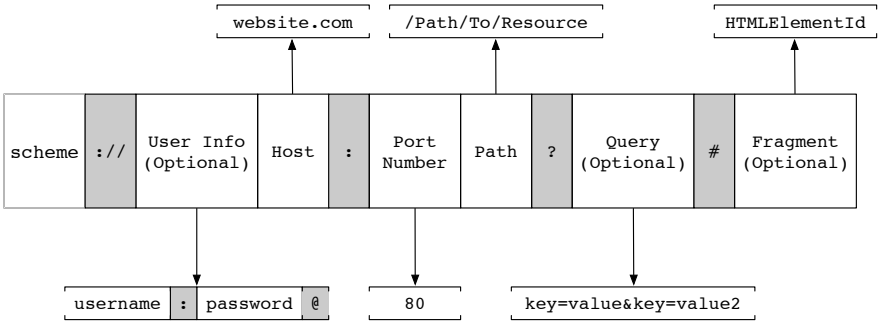


Figure 1.1: Uniform Resource Locator

The process of loading a web page is shown in Figure 1.2. A browser tries to establish a Transmission Control Protocol (TCP) connection to a web-server to access a given webpage resource. When this connection is established, the Hyper-Text Transport Protocol (HTTP) protocol is used to communicate between the web browser and the web application server. The web browser sends to the web server a request that includes a list of headers giving details such as the type of browser it is, (HTTP header: User Agent). The web application server responds to the request with appropriate headers of its own such as the caching policy for the web page, the encoding that has been used, the type of data that is being transmitted, the size of the data, whether the web browser must download the content instead of showing it inline (content-disposition header), etc.

Since the websites try to personalize the content based on the user, they establish a browsing session for the user. These sessions may be authenticated or unauthenticated sessions. They are maintained through the use of a unique string assigned to that session. This string is stored in the form of variable known as a cookie. The browser maintains one cookie per website and automatically sends the cookie for every request made to that website. Conversely, each response of the web application server include a cookie.

When loading a webpage, a web application server response typically contains a HTML (Hyper-Text Markup Language) content. The user interface of the web browser uses a rendering engine to display the elements as required corresponding to the different tags on the given HTML document. The rules on how to align the various elements, colors, or fonts to be used are given in the form of Cascading Style Sheet (CSS). Finally, the dynamic programming language used to run the various functionalities of the web page are usually written in JavaScript.

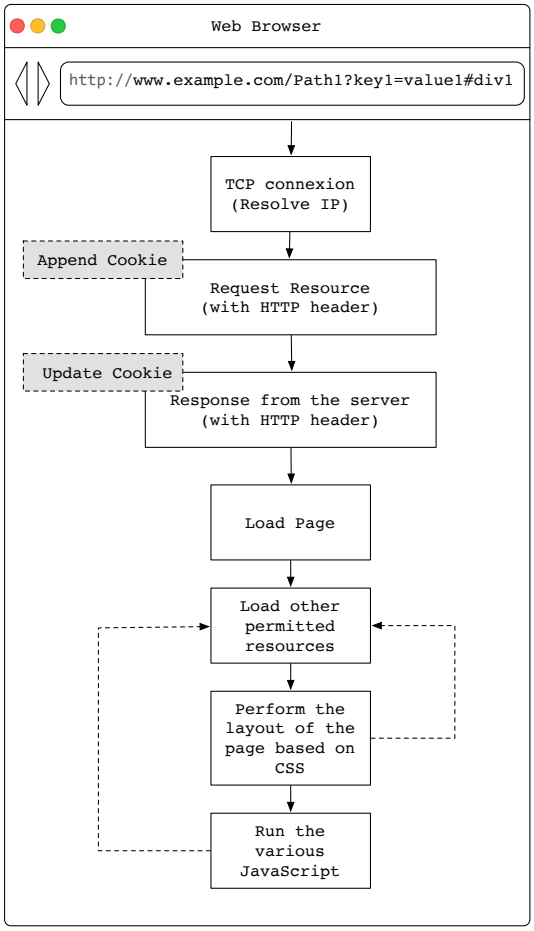


Figure 1.2: The process of loading a page

1.1.2 JavaScript

Client-side script is one of the important aspects of a webpage on a web browser. JavaScript itself is a high-level, dynamic, interpreted language. It has gradually replaced other components such as Java applets and Adobe Flash in terms of functionality and has been ranked among the most used programming languages.[2]

In a typical webpage, JavaScript is used to interact with a remote server and integrate dynamic content into the webpage. This is done with the help of the `XMLHttpRequest` function. This function is used to make a HTTP request to a URL and obtain the response as a JavaScript variable. This variable can then be used to generate the dynamic content. Hence, the content

[2]Techcrunch 2012, `http://tcrn.ch/2nReBSh`

on the page, as well as information from the server, can interact with the aid of JavaScript.

In JavaScript, the `eval` function allows runtime execution. This implies that an arbitrary string can be passed to this function to be executed in the current context. An example of the use of `eval` in JavaScript is shown in figure 1.3. In this example, it can be observed that the code that is executed changes based on the input to the function. The example here is quite simplistic. When the parameter `choice` is equal to `"gt"`, the function checks if the value of the variable `a` is greater than `5`. The actual check happens in the interpretation of eval statements which compiles the string that is passed to the `eval` function. JavaScript is hence a very powerful and flexible language that is more and more used in modern webpages.

```
1  var a = Math.floor((Math.random() * 10) + 1);;
2  function f(x,choice,number)
3  {
4      var y = true;
5      var greaterThan = "if(x<=" + number + "){y = false;}else{y=
           true;}";
6      var lessThan = "if(x>" + number + "){y = true;}else{y=false;}
           ";
7      if(choice === "gt")
8      {
9    eval(greaterThan);
10     }
11     else if(choice === "lt")
12     {
13   eval(lessThan);
14     }
15     return y;
16 };
17 var a_gt5 = f(a,"gt",5);
18 var a_lt5 = f(a,"lt",5);
```

Figure 1.3: Simple eval function example

1.1.3 Typical modern webpage

Today, the modern webpages are complex because they are composed of different components. Indeed, many modern webpages depend on data from multiple web application servers to run as intended by the developer, as illustrated by the Figure 1.4. The third-party websites such as Facebook[3] and Disqus[4] provide content that interact with the page at runtime using scripts. The various scripts provided by advertisement providers can also actively explore the context

[3] `https://fr-fr.facebook.com/`
[4] `https://disqus.com/`

of the page to provide relevant advertisements. All these third-party scripts are added by the developer and are intended to work in the same context of the webpage.

HTML content forms the basis for creating the Document Object Model (DOM) of the current webpage. The JavaScript loaded on the page is part of the `script` tag. The contents of this tag are passed to the JavaScript engine to be executed at runtime. The JavaScript engine can also access the DOM elements and subsequently create, modify as well as delete the DOM elements at runtime to provide dynamic content to the user.

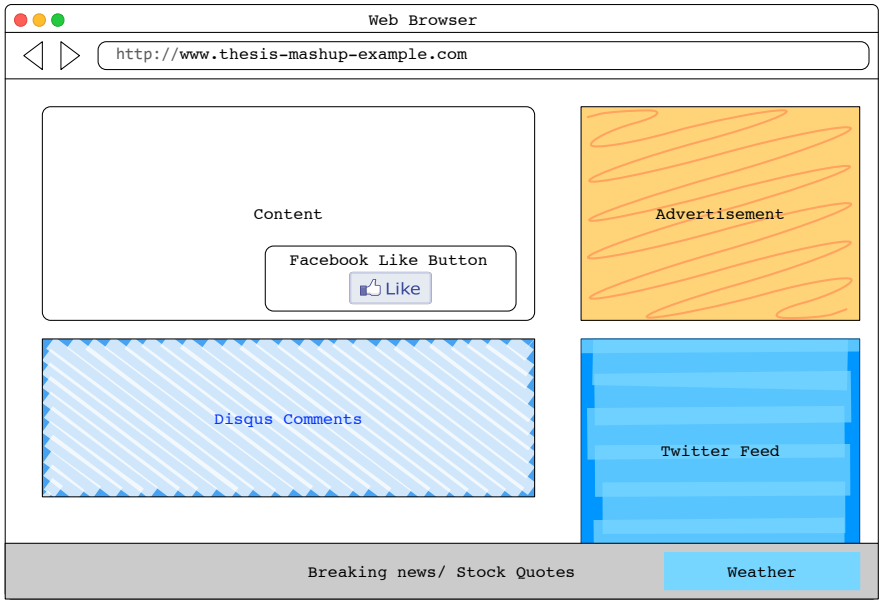


Figure 1.4: A typical webpage

The advent of HTML5 has triggered an array of approaches increasing the feature set of web applications. Some of these novel technologies such as Web Messaging [Hic15b], WebSockets [FM11] or WebRTC [BBJN15], allow for communication on web pages on levels that were not feasible earlier. We give more details on WebRTC technologies in the following sections.

1.1.4 WebRTC

WebRTC is one of the latest additions to the ever-growing repository of Web browser technologies, which push the envelope of native Web application capabilities. WebRTC allows real-time peer-to-peer audio and video chat, that runs purely in the browser. Unlike existing video chat solutions, such as Skype, that operate in a closed identity ecosystem, WebRTC was designed to be highly flexible, especially in the domains of signaling and identity federation.

The high-level architecture of WebRTC can be split into two different *planes* as shown in Figure 1.5. The distinction is made based on the kind of data sent over it. The green layer or the *media plane* delivers the peer-to-peer real-time streams. The top red layer or the *signaling plane* delivers all control- and meta-data between the endpoints.

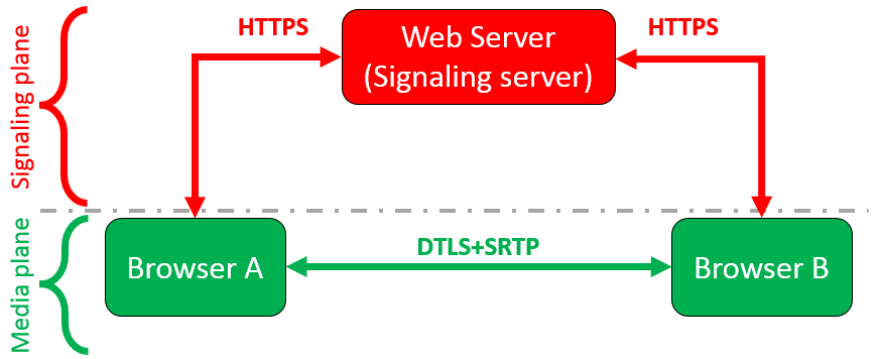


Figure 1.5: Simple architectural view of WebRTC

The signaling plane consists of one or more signaling servers that mediate and route communication, typically over an HTTPS connection, between two or more endpoints. The second task of a signaling server is to serve the initial client-side application-specific code that interacts with the JavaScript API for WebRTC.

The media plane will take care of the peer-to-peer connections between the endpoints using User Datagram Protocol (UDP). Datagram Transport Layer Security (DTLS)-Secure Real-time Transport Protocol (SRTP) is a key exchange mechanism that is mandated for use in WebRTC. DTLS-SRTP uses DTLS [RM12] to exchange keys for the SRTP [BMN+04a] media transport. SRTP is the real-time streaming protocol. This protocol belongs to the application layer of the Open Systems Interconnection (OSI) model. SRTP requires an external key exchange mechanism for sharing its session keys, and DTLS-SRTP does that by multiplexing the DTLS-SRTP protocol within the same session as the SRTP media itself. The protocol is used as the basis for the communication security for the UDP connection.

Due to complex setups of today's network infrastructure, the architectural picture is often far more complex, as shown in Figure 1.6. Services to obtain mapped public IP addresses from within private networks (e.g., STUN and TURN servers, shown in purple), and to provide identity management (shown in blue), are all part of the complete architecture. The Identity Provider (IdP) are an important component for WebRTC since they verify the users who are communicating.

To establish a WebRTC connection, the browser needs to first send a connection request to the signaling server. The signaling server then establishes a TCP connection with the other

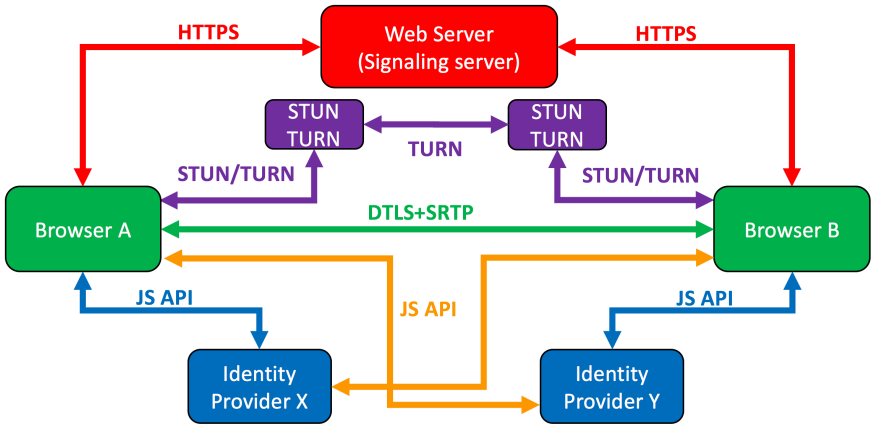


Figure 1.6: WebRTC architecture based on [Res15b]
The HTTPS signaling plane (red), the DTLS/SRTP media path (green), the interaction with STUN and TURN (purple), and the interaction with Identity Providers (blue for the assertion generation and yellow for the assertion verification).

browser and passes the request. The request is actually in the form of a Session Description Protocol (SDP) object. This SDP object is necessary to establish a peer-to-peer connection between the two browsers. The second browser appends its data and creates an SDP answer object which is passed to the first browser by the signaling server. Once both parties have received the SDP objects, the media peer-to-peer connection is established. On the Internet, the IP address of a user is usually dynamic in nature. To keep track of the user over several ISPs and firewalls, WebRTC uses STUN and TURN based proxies.

The WebRTC architecture provides a mechanism to allow applications to perform their own authentication and identity verification between endpoints. These interactions are done via JavaScript APIs within the browser itself. Each endpoint can specify an Identity Provider (IdP) while generating the SDP offer/answer. Based on the content of a received SDP message, the endpoint can check with the IdP to verify the received certificate and thus to validate the identity.

Figure 1.7 provides an architectural overview of the integration of an IdP. In essence, the browser will load a IdP-specific proxy (called *IdP Proxy*) to interact with the Identity Provider and this proxy implements a very generic interface towards the web browser for peer authentication. The web browser first generates an offer and the SDP object is provided to the IdP to be signed. The signed SDP is reflected in the offer. The web browser receiving the offer would then pass the signed SDP to the IdP to get the verification of the sender's identity. It would then generate an answer, which also contains an SDP object, repeating the process of signing using an IdP. Hence the identity of both parties in the peer-to-peer communication can be asserted.

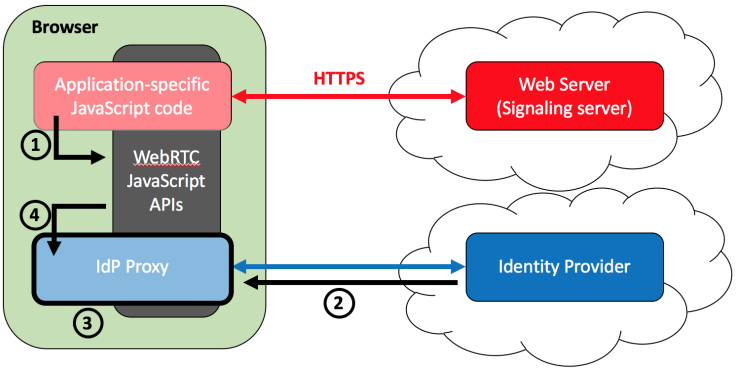


Figure 1.7: WebRTC integration of the Identity Provider.

1.2 Vulnerabilities on modern webpages

As illustrated in previous section, thanks to JavaScript and web technologies, the modern webpages are converging to the functionalities traditionally reserved for desktop applications. This explains the widely use of web applications to provide services that previously required specific software.

However, the widespread use of Javascript and web technologies to provide advanced functionalities to users implies that vulnerabilities in modern webpages are today one of the most critical risks for the user. In this section, we describe two vulnerabilities related to the modern webpages, that are the cross-site scripting (XSS) and cross-site request forgery (CSRF) in Section 1.2.1 and Section 1.2.2 respectively. XSS is part of the list of top ten vulnerabilities on web applications provided by the Open Web Application Security Project (OWASP) [Opea]. I Section 1.2.3, we demonstrate that XSS vulnerabilities can also be exploited on new JavaScript technologies such as WebRTC.

1.2.1 Cross-Site Scripting

Cross-site Scripting (XSS) is one of the most exploited vulnerabilities in the modern web [Opea]. It is caused when untrusted third-party script affects the normal working of a page usually compromising the confidentiality and/or integrity of the user's or server's data in the process. The main problem is when unauthorized scripts are able to run in the same context as that of the webpage. Cross-site scripting is often classified between three types:

- Reflected XSS;
- Persistent XSS;
- DOM-based XSS.

1.2.1.1 Reflected XSS

Reflected cross-site scripting is a vulnerability where an arbitrary script is run because some parameters in the HTTP request are not checked by the server. This is made possible when the GET or POST parameters sent to the web application server are used without sanitizing the variables. Sanitizing is the process of checking if the variable contains any executable strings and modifying them so that they will not be executed by the browser when loaded into the page. Indeed, if left unchecked, an arbitrary script could be sent back to the web browser and it would be executed, thereby exploiting the user. This type of vulnerability is most often exploited in search functionalities of a page.

For example, let us consider, a user 'person A' gets a mail with the following link:

```
http://vulnerablesite.com/search=<script%20src=http:
        //malicioussite.com/script></script>
```

Clicking on this link would send a request to the server of `vulnerablesite.com` with the search string `<script%20src=http://malicioussite.com/script></script>`. The server would then perform the search based on the string. If the resultant webpage that is sent back by the web application server also contains the same search string, and since the search string contains the script, this script is executed by the browser. Consequently, the arbitrary script page is loaded from the URL `http://malicioussite.com/script`. This arbitrary script may then perform a malicious action such as passing the cookie of the user's session for the website `http://vulnerablesite.com`. The attacker can then use this cookie to impersonate the legitimate user on the website `http://vulnerablesite.com`.

This example is a typical reflected XSS. It must be noted that reflected XSS does not infer anything about persistence of the vulnerability. It merely states that the vulnerability is caused because of the way in which the URL parameters are interpreted in the resulting page.

1.2.1.2 Persistent XSS

Persistent cross-site scripting is one of the most dangerous vulnerabilities. It is often a server-side vulnerability that is also caused by lack of sanitizing. As an illustration, let us consider that the website `vulnerablesite.com` is storing all the searches made by person A to aggregate a 'recent searches' list that is loaded at the website's homepage. Let us consider that when the website's homepage `http://vulnerablesite.com` is loaded, the ten last search terms are shown as well. Now, let us consider the same example as in the reflected XSS in Subsection 1.2.1.1. Because of the new functionality (i.e., the display of the ten last search terms), the search string `"<script src=http://malicioussite.com/script></script>"` would be

stored in the database of the web server. The next time person A visits `vulnerablesite.com`, the aggregated list of recent searches is passed as part of the webpage. In this case, the loaded webpage would mandate the web browser to execute the malicious script.

Persistent XSS can be even more serious where one user affects every other user. For example, if the homepage of the website shows a 'trending searches' list. Consider that this is a list of the searches made to the website `vulnerablesite.com` in the last ten minutes to all its users. This would implicate all the users loading the homepage of the website since the script from the URL `http://malicioussite.com/script` would be loaded.

Notice that in modern times, web browser persistence mechanisms have also been targeted. HTML5 introduced a persistence mechanism called LocalStorage[5] which allows a webpage to store key-value pairs in the browser. It is possible to get a persistent XSS if the webpage scripts reading these values are vulnerable. For example, consider the last ten entries are stored in the LocalStorage instead of on the server's database. Let us also assume that the homepage of the website `http://vulnerablesite.com` adds these searches to a list called 'recent entries' that would be shown on webpage when it is loaded. This would similarly execute the malicious script every time the webpage is loaded.

1.2.1.3 DOM based XSS

DOM based XSS was first identified as the third type of XSS by Amit Klein [Weba]. However, DOM based XSS is a variant of persistant and reflected XSS but is different from traditional XSS in a very subtle manner [XSS]. In persistant and reflected XSS, the malicious JavaScript is executed when the webpage is loaded, as part of the HTML sent by the server. In DOM based XSS, the malicious JavaScript is executed at some point after the webpage has been loaded, as a result of the webpage's legitimate JavaScript treating user input in an unsafe way. Thus, a DOM based XSS exploits a vulnerability that exists because of errors in the JavaScript of the webpage.

For example, let us consider the case where person A clicks a link to the following webpage from an email :

`http://vulnerablesite.com/page.html?default=English`

Let us consider that the loaded webpage contains the code fragment of the Figure 1.8. This script allows the user to select the language of the webpage, the default value being provided with the parameter `default` of the URL.

[5] `https://www.w3schools.com/html/html5_webstorage.asp`

```
1 Select the language:
2 <select><script>
3 document.write("<OPTION value=1>"+document.location.href.
    substring(document.location.href.indexOf("default=")+8)+"
    </OPTION>");
4 document.write("<OPTION value=2>English</OPTION></select>");
5 document.write("<OPTION value=3>French</OPTION></select>");
6 </script><select>
```

Figure 1.8: OM based XSS example [Opef]

Now, let us consider that instead of the previous link, A clicks to the following webpage :

`http://vulnerablesite.com/page.html?default=<script>alert(document.cookie)</script>`

In this case, the DOM element `document.location` is assigned following value:

`<script>alert(document.cookie)</script>`

This code is added to the DOM and then executed because of the script that allows to select the language of the webpage.

1.2.1.4 Conclusion on XSS

These are the three main types of XSS. It must be noted that persistent, reflected and DOM based XSS are not mutually exclusive classifications of XSS. An XSS vulnerability can belong on a single, multiple or all categories of the vulnerability, just like in the search example where the vulnerability caused by the `search` term is both reflective as well as persistent. Moreover, persistent XSS and reflected XSS can both result in DOM based XSS.

Finally, notice that in a mashup webpage containing scripts from multiple sources, XSS is dangerous since if any one of the third-party scripts is vulnerable, the entire page becomes vulnerable as well.

1.2.2 Cross-Site Request Forgery

CSRF is one of the most important vulnerabilities on the Internet [Opea]. CSRF occurs when the web application server is unable to distinguish between a legitimate (i.e. as intended by the user) and illegitimate request (i.e. performed by impersonating the user). This is because of the web browser's behavior to append the cookie and session information along with any request made to a given URL. Since websites cannot ask the user to log in for every action to

be carried out, a cookie is used to maintain the user's authenticated state with the site. It must be noted that the cookie is kept active until deleted. The closing of the webpage's tab does not have any impact on the cookie (unless the browser settings explicitly delete the cookie on page close). Similarly, unless explicitly stated, a browser retains the cookies despite being restarted.

Let us remind the usual scenario for a website that uses session cookie. When a user logs on to a website, for instance, through a dedicated authentication webpage, the session cookie is tied to this user. This cookie is stored in the web browser and is sent as part of all subsequent requests to the website enabling the user to be identified (without repeating the login mechanism for every request). Now, let us suppose that another webpage on the same browser makes a request to this website. Then according to the web browser's behavior, this request will automatically be appended with the session cookie and other relevant cookie information when sent to the website by the web browser. This request would appear to be a legitimate request from the user, and the website will subsequently process the request.

Imagine this in the context of sensitive applications such as banking. Another unrelated webpage on the same web browser could send the request to the bank and the web browser would do all things necessary on its behalf. This would directly result in malicious transactions since the bank's web application server would not be able to distinguish between the two requests.

1.2.3 Vulnerabilities on WebRTC

In this section, we discuss the various ways in which the prerequisites for endpoint authenticity can be broken by malicious third-party JavaScript such as injected JavaScript due to XSS vulnerability.

This section covers two different attacks against endpoint authenticity. In Section 1.2.3.1, the integrity of the DTLS certificate is compromised in WebRTC setups where no Identity Provider is present. This first scenario is very plausible as at the time of writing (most) browsers do not yet provide wide support for Identity Provider integration. In Section 1.2.3.2, the second problem where the identity of the user is replaced by an identity under the control of the attacker is demonstrated. This attack is hence quite dangerous since the identity of the user can be interchanged at run-time due to scripts.

1.2.3.1 Compromising the integrity of the fingerprint

The WebRTC specification does not require the use of an Identity Provider within a WebRTC setup. Actually, the default operation of WebRTC instances is without the involvement of an Identity Provider, as the support for IdP integration in web browsers is unfortunately not yet

mainstream.

In the absence of an Identity Provider, the endpoint authenticity is boiled down to the integrity of the DTLS certificate fingerprint within the SDP object. Concretely, this means that in the absence of an Identity Provider the endpoint authenticity can easily be compromised. Every party on the signaling path is able to manipulate the SDP objects and mangle with fingerprints present in the SDP description. In particular, exploiting a XSS vulnerability, the attacker can modify the DTLS certificate fingerprint within the SDP description, or even replace the SDP object by a fake SDP object, retrieved from a website under the control of the attacker. So even in the case of a confidential and integer data channel, it is still not secure as there is no assurance about the other side's identity.

```
1  // pc is an RTCPeerConnection object
2  pc.createOfferOriginal = pc.createOffer;
3  pc.createOffer = function(callback, error){
4     pc.orgCallback = callback;
5     pc.malCallback = function(offer){
6        var newOffer = getAttackerSDPViaXHR(offer);
7      pc.orgCallback(newOffer);
8     };
9    pc.createOfferOriginal(pc.malCallback, error);
10 };
```

Figure 1.9: Example attack showing how to compromise the certificate fingerprint by replacing the SDP offer with an attacker-controlled version.

As an illustration, let us consider for instance the attack scenario, presented in Figure 1.9. This is a fragment of the client-side JavaScript code, that could be pushed by a attacker using a XSS vulnerability. In this code example, the *createOffer* function gets replaced by a wrapper function, which will replace the SDP offer by a fake SDP object, retrieved from the attacker website via XMLHttpRequest (XHR). The SDP offer is represented via a string, and the fake SDP offer will include a new attacker-controlled fingerprint, as well as other vital parameters (e.g. network configuration) to connect to an attacker-controlled endpoint. As this first class of attacks compromises the integrity of the DTLS certificate fingerprint, the endpoint authenticity can not be guaranteed in the absence of an Identity Provider, given the possible presence of XSS vulnerability.

1.2.3.2 Compromising the integrity of the identity assertion

The WebRTC security model stipulates strict requirements about the consent that is required from end-user for access to media devices, such as the camera and the microphone. However,

this is also the only user consent that is required to use WebRTC. No user-interface requirements are stipulated for the browsers to inform the end-user about the fact that a WebRTC connection is being set up, or that an identity assertion is generated or verified by the JavaScript code. Especially the lack of chrome user-interface to select a preferred identity or Identity Provider, and the lack of granting access to a specific identity to set up a remote WebRTC connection undermines the integrity of the identity assertion used in WebRTC.

Even in case an Identity Provider is used to set up the peer-to-peer connection, and the fingerprint is correctly bound to an identity in the identity assertion, this could still compromise the endpoint's authenticity. For instance, exploiting a XSS vulnerability, the attacker can provide a fake identity assertion for an identity and a fake DTLS certificate fingerprint, as well as the code for validating them.

Figure 1.10 illustrates how the current identity assertions can easily be replaced by identity assertions generated for artifacts under the control of the attacker.

```
1  // pc is a RTCPeerConnection object
2  // hjMc is a MessageChannel object
3  hjMc.port1.onmessage = function(e) {
4    newOffer.sdp = changeAllIdentities(e.data,hjMc.offer.sdp);
5    pc.trueCallback(newOffer);
6  };
7  function changeAllIdentities(newIdentity,sdp){
8    identityExtraction = base64(newIdentity);
9    return sdp.replace(/identity:[A-Z0-9]*\n/g, 'identity:'+
       identityExtraction);
10 };
```

Figure 1.10: Example attack showing how to modify the identity string to a fake identity.

In this code listing, the `createOffer` function has already been considered overridden. Hence, even if the offer is completely signed by the identity, the same offer can be signed for a different identity and these identities can be switched. A changed identity would not be detected since the identity is opaque to the signaling server and the identity assertions would result in the same fingerprint. While, this attack in itself cannot cause a hijacked connection, it can be used as a tactic in ensuring problems with the call. For example, if user B has been known to block user A, simply using A's identity will ensure that B would drop the call.

1.3 Web security mechanisms

In the previous section, we have introduced the two most exploited vulnerabilities in the modern web: XSS and CSRF. In this section, we briefly present mechanisms that can be deployed in

the web application server side, and then introduce the security mechanisms that have been proposed to protect web browsers against these vulnerabilities. These mechanisms are mainly based on the use of sandbox, a principle that consists in isolating some parts of a webpage in the web browser so as to prevent the access to sensitive data.

1.3.1 Security mechanisms on the server side

Even if CSRF and XSS target the web browser or lure it to conduct attacks, some defense mechanisms can be implemented on server-side. For example, persistent XSS are made possible because of vulnerability on the web application server side. Persistent XSS can usually be avoided if the web application server uses correct output encodings when storing and displaying the data on the webpage respectively.

CSRF exploits the fact that session cookies are automatically send with every request, even if this request is initiated by a script of another webpage, on a different web browser tab. Because session cookies are not sufficient to protect against CSRF, many modern websites use a CSRF token with every request as another layer of security. The CSRF token is a serializable token that is appended by the website's web application server to every link and JavaScript XMLHTTPRequests on the webpage as a request parameter. Hence, every subsequent request made to the webpage would contain the token. This token must not be stored in the web browser as a cookie or any persistent storage for the mechanism to work. This is because the web browser would automatically append the cookie details to any request made. However, this token is easily obtained by checking the URL strings in the webpage assuming the environment is currently controlled by the attacker. The CSRF token is a secret information that needs to be protected from malicious JavaScript since it provides the last line of defense for CSRF.

Many modern websites also require that the web browsers append an `origin` header as part of every HTTP request they make. This origin header would refers to the origin web application server for the current webpage from which the request originates. The web application server can then check the `origin` header to identify the source of every request. This makes it possible for the web application server to check the webpage that made the request and decide on whether or not it wants to process such a request. Even if it is the browser that has to provide the `origin` header, the check is done by the server.

Origin headers are a reliable mechanism when a request is sent from a third-party server if the request is only supposed to be accessible from the same-origin. However, if is valid for the request to be made from a third-party website, it is necessary to supplement it with the CSRF token. For example, a bank server could block any request to transfer money made from any other origin than the bank's own website. However, an advertisement provider which would typically allow all origins, would need to use a CSRF token since origin header is not useful

for their use case.

Notice that the above mechanisms are bound to fail if the webpage is affected by XSS. Indeed, if a malicious user is able to inject a script that will be loaded and executed in the target webpage, then since the request is from a legitimate webpage, the origin tag is correct and since the script running on the same page has access to the entire page's source code, it can also derive the CSRF token. This token can then be used both directly and indirectly to affect the user. In the direct case, the malicious script can trigger a transaction by activating the event or creating a request to the website with the token appended. Indirectly, it could send the various tokens and cookie elements to a remote web application server, which can then use this information to contrive even more scenarios of malicious use. This emphasizes the need for client-side protection mechanisms; such mechanisms also being useful to protect the security mechanisms enforced on the server side.

1.3.2 Security mechanisms on the web browser side

1.3.2.1 Isolation between different webpages

The premise of sandboxing as a security mechanism in a web browser began through an old concept called Site-Specific Browsers (SSB) [Wiki]. In SSB, the main goal is to prevent the illegitimate use of the cookies. In a typical web browser, the cookie for a given domain is sent along with any request made to the domain across various tabs of the web browser. This is the reason for the exploitation of the CSRF vulnerability. SSB ensures that the web browser is able to open only a dedicated website. Since opening another website implies opening another Site-Specific Browser, there is complete isolation of the cookies. This makes exploitation of CSRF from other websites more difficult. However, though the concept itself is simple and useful, it also severely limited the functionality of the web browser, especially because it is tedious to maintain several individual web browsers capable of browsing only specific websites. Moreover, there are some consequences for this approach which makes it incompatible with modern web pages. For example, third party plugins such as Facebook like button would require the user to login for every website since session cookies cannot be propagated between SSBs.

In recent years, research on web application isolation in a single web browser has gained more traction and Chen et al. [CBR11] have demonstrated one such model with a working prototype on the Chromium browser. Instead of a strict control by SSBs, their approach provides some compromise in restrictions, i.e. allowing some amount of cross-domain access. Their model works on two main concepts namely state isolation and entry-point restriction. The authors aim to create the same impact of using SSB by using these two concepts. The

state isolation is used to maintain the sensitive applications' various data such as cookies and LocalStorage in an isolated manner. Entry-point restriction is a list of website URL patterns that are allowed by a cross-origin webpage to be requested. The list is provided to the browser by the website developer before it requests resources from the particular site. The list is hosted in a well known location (for instance `www.website.com/.well-known/meta-data`) and is automatically retrieved by the web browser before loading the webpage. Using these two mechanisms, the authors are able to enforce isolation in a single web browser. Further, this mechanism has incurred an average performance hit of 0.1ms when tested on the Alexa 100 in 2011.

While this mechanism is very useful to have a proper protection for cookies thereby providing similar protection as SSBs, it also suffers from the same issues. The main issue is that there is no way to prevent a malicious JavaScript on the page from accessing sensitive data. Any JavaScript that is allowed to run needs to be implicitly trusted with any information contained in all variables. It is precisely these issues that information flow control is supposed to tackle in an effective manner.

Most modern browsers come with their own sandbox to prevent the illegitimate use of cookies. This sandbox is often based on the Same-Origin Policy (SOP). The SOP states that the scripts in the web page may only send requests to the current page's domain. This automatically prevents information disclosure to external sites using the `XMLHttpRequest` function. However, the SOP is too restrictive for modern webpages because it deny all the cross-origin requests needed for web applications being able to interact with each other. Thus, the HTML5 specification provides a mechanism called Cross-Origin Resource Sharing (CORS) which allows the developer to override the SOP for specific domains. To enable CORS, it is mandatory for the webpage to contain headers whitelisting the various origins to which it wants to perform CORS requests. Once the webpage is loaded, these headers cannot be changed until a webpage reload. Further, the remote server to which the request is made must contain a whitelist allowing the requesting webpage's domain to access the data contained in the response.

It must be noted that the SOP of the web browser does not apply to components of the webpage such as images, externally hosted scripts, and videos. An example of such a leak is as follows: `<img src='www.bad.com/?cookie=somecookievalue'/>`. In this example, an image tag is used to load an image at `bad.com`. However, the request contains a parameter `cookie` that is passed by making a GET request.

To prevent information disclosure through components, a security mechanism called the `Content-Security-Policy` (CSP) has been envisioned as part of the HTML5 specification. The CSP is effectively an implementation of a whitelist based approach that provides a definitive list of resources that can and cannot be allowed into the webpage. The CSP is passed as

part of the HTTP server headers. CSP hence provides the web browser information on which sources can be accepted to load the various objects such as scripts, images, videos on the webpage. It is also possible to allow or deny in-line JavaScript. In-line JavaScript is loaded as part of the page when the DOM is loaded and is not part of an external JavaScript file. By not allowing in-line JavaScript from executing, the risk of DOM-based XSS can be reduced to a significant extent. This is because even if there is a malicious script in the page due to a prior persistent XSS, the script would not be allowed to run if it is inline. Similarly, providing a whitelist of scripts, images, and other resources decreases the possibility of an exploitation by a large degree. CSP is an effective mechanism in preventing untrusted JavaScript from loading on to the page. However, it is only a mechanism to regulate whether scripts could be permitted execution. It does not provide any means to secure the data from insecure JavaScript running in the same context.

1.3.2.2 Isolation inside a webpage

Notice that sandbox model provide isolation across webpages and not within the same webpage. In a mashup webpage, contents from varying websites find their web applications interacting with each other. These include calendars, date and time indicators, advertisements, embedded audio and video, feeds from varying sources as well as comments and discussion boards. The sandbox mechanism in providing isolation within a webpage is the default `HTML` `iframe` (inline frame mechanism). An `iframe` is a `HTML` tag which loads another webpage in a dedicated inline frame. The JavaScripts of the `iframe` does not run in the same context as the rest of the page. It hence provides a way of sandboxing within a webpage. The `iframe` itself is not bound by the SOP. However, it is not possible for the webpage and the `iframe`, of mutually exclusive origins, to communicate to the iframe. It must be noted that `iframe` was created to embed one website into another and was not envisioned for the scenario of isolation.

A more well defined isolation approach for the context of mashups was defined by Wang et al. [WFHJ07] in MashupOS. This approach introduces a new HTML tags, `<Sandbox>`, to provide for isolation of content. This tag however provides some level of interaction on if the isolated webpage. The webpage that loads the sandbox can call the scripts within the sandbox. However, the objects can only be passed by value and not by reference. The sandbox cannot access any of the DOM elements of the webpage itself. It can however create/modify/delete DOM elements as long as they are within the `<Sandbox>` tag. The authors suggests the use of this mechanism if the webpage developer does not trust the scripts within the `<Sandbox>`. The `<Sandbox>` mechanism is best used in case of third party libraries. This is because while the `<Sandbox>` provides `iframe` like isolation, it also provides a means to communicate securely from the sandbox to the webpage rendering the frame and vice-versa.

It must be noted that with the advent of HTML5 web messaging standard, such communication between an `iframe` and a webpage of different domains are also possible. However, these are a much newer concept and have ben influenced by prior proposal such as the just mentioned MashupOS by Wang et. al. [WFHJ07].

1.3.2.3 Specific mechanisms

Finally, the web browsers provide non-standard mechanisms to prevent some specific vulnerabilities. These mechanisms are not always well-documented and are mostly proprietary. One of the mechanisms found in both the Google Chrome and Chromium browsers as well as Microsoft Internet Explorer is the "XSS Auditor".[6] This mechanism is specifically intended to prevent reflected XSS. It checks if there is a script passed as one of the parameters of the request in URL used to load the webpage. For example: if `www.page.com?search=<script> alert(0);</script>` is the webpage's URL and this parameter is shown as-is in the DOM, it is a reflected cross-site scripting vulnerability. Hence, if the page contains a script string which was part of the parameters passed in the request, this particular script is blocked from execution.

While this mechanism is not well documented, the xss-auditor requires two conditions to be satisfied for working. The first is that the request parameter must contain the `<script>` tag. The second condition is that this parameter must be present as a string in the document body. This condition is verified by a simple string match. If even one character was different due to some server computation or encoding before adding it to the page's contents, the xss-auditor would not work. This is because xss-auditor intends to keep a very low false positive rate. It is very effective against reflected XSS but is a solution only for this problem.

1.3.3 Conclusion

As shown before, there are various approaches that are used to protect against the exploitation of both XSS and CSRF vulnerabilities. However, while these approaches are very efficient and effective, they are tuned to solve very specific problems and are not suitable candidates for a holistic approach towards web browser security. Thus, the use of origin headers or CSRF token allows to prevent CSRF except if malicious script has been injected into the current webpage. Code injection is difficult to detect on the browser side if the attacker is using persistent XSS. The only solution seems to be the SSB approach that ensures physically the isolation of the webpages, but this solution is incompatible with the modern mashup webpages.

[6] `http://www.collinjackson.com/research/xssauditor.pdf`

In this thesis, we defend that there exists another approach. All solutions based on sandboxing have a granularity issue where the least attainable granularity correspond to all the JavaScript code coming from a particular domain or included in a given file. This approach is too coarse-grained to tackle code injection in legitimate web page, for example. We assert that an approach having a variable level granularity can be used. The principle of such an approach is to control the information flows between the script variables and the DOM components so as to ensure sensitive information is not passed to a third party by even scripts from the same webpage.

As an illustration, let us consider the example of the CSRF token. Of course, the value of CSRF token is sensitive, and it should only be accessed by legitimate scripts. Now, let us suppose that the value of the CSRF token is copied into another variable, and that an illegitimate script can access this second variable. In this context, the illegitimate script can access the value of the CSRF token even if it cannot access the CSRF token itself. By controlling the information flows between the script variables, we are able to detect that the value of the variable is equal to the value of the CSRF token, and then deny the access by the illegitimate script.

In the next chapter, we introduce the basic of Information Flow Control (IFC) as well as some web browser security approaches based on IFC.

Chapter 2

Related work on information flow control

We present related work pertaining to the field of information flow control (IFC) in this chapter. In Section 2.1 we give a general overview of IFC. The Section 2.2 gives an introduction to the application of information flow control in programming languages. The Section 2.3 provides an overview of the various considerations to be taken when formulating an information flow control model. Finally, we describe the various prior IFC approaches that have been applied to web browser security in Section 2.4.

2.1 Background on Information Flow Control

In a typical information system, there are subjects (users, programs, etc.) that attempt to access objects (documents, files, variables, etc.) which contains the information. There are various roles that can be given to the subjets based on the requirements of the environment. For example, users can be given the roles such as worker, lower management and upper management. The necessary privileges to access the various objects is given to the subjects based on their role in the environment.

Often, the access control rules are related to the objects. That means the access conditions are defined at the objects level, without considering the type of information that is contained into the objects. However, the access conditions can be related to the type of information into the objects. In this case, we have to classify the objects according to the type of information they contain. Typically, the information is classified based on how sensitive it is and its category. Usually, there can be different classifications of the documents such as $Unclassified$, *Secret*, and *TopSecret* based on the sensitivity of the information contained, and these clas-

sifications are paired with a total order such that $Unclassified < Secret < TopSecret$. The information is also categorized according to the type of information. For instance, the information can be related to *Crypto* or *Nuclear*. These categories are organized by set inclusion. Based on classifications and categories, a partial order can be defined (see Figure 2.1).

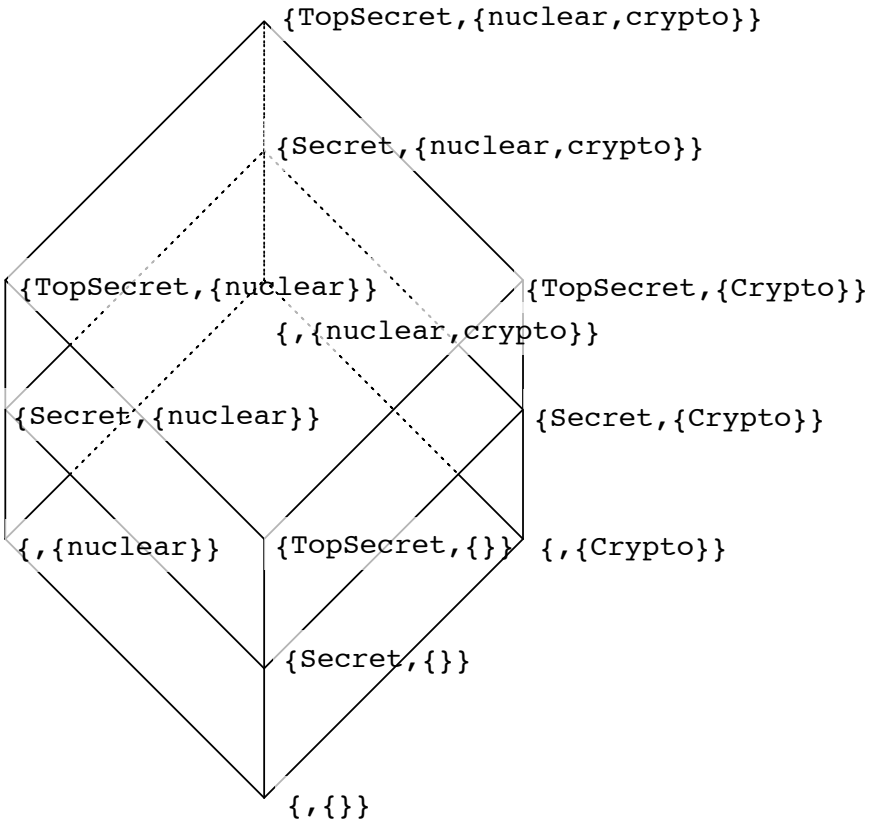


Figure 2.1: Partial order on information

Similar to how the objects are classified, subjects can also be classified according to the type of information they are authorized to access. Then, given the partial order, access control rules can be defined depending on the type of information contained into the objects. For instance, given the partial order of Figure 2.1, a subject with classification $\{TopSecret, \{Nuclear\}\}$ is authorized to access objects that contain information classified $\{Secret, \{Nuclear\}\}$. However, subject with classification $\{TopSecret, \{Crypto\}\}$ cannot access objects that contain information classified $\{Secret, \{Crypto, Nuclear\}\}$. Such access control models are called Multi Level Security (MLS) models which is a form of Mandatory Access Control (MAC).

Several MLS models have been proposed, such as the Bell-La Padula model [BL73] and the Biba model [Bib75]. In the Bell-La Padula model, the access control is quite restrictive. The

object at a particular level cannot be read by the subject at lower levels, and cannot be changed (write access) by the subject at higher levels. This implies that the subjects may write to objects at higher levels than their own, passing on more sensitive information to their supervisors while maintaining confidentiality of the information they pass. Conversely, in the Biba model, the subject of a particular level cannot read objects at a lower level than itself and cannot write information to objects at a higher level. Using such a policy assures data integrity.

In such MLS models, the access control guarantees that the information of any object is in accordance with the classification of that object. This is a main limitation of such approach : the security administrator has to classify all the objects and subjects, at least those that are supposed to manipulate sensible data. This approach clearly lakes of flexibility : the administrator has to know in advance which containers will be used by each subject. On a practical point of view, it is often impossible to specify the classification of all the fine-grained containers such as variables. Thus, such MAC models are typically used with coarse grained containers such as files.

In information flow control approaches, the rules are defined at a fine-grained level, and allow the read or write operations based on the information that flows and not just the static classification of the objects. Thus, if a subject wants to write secret information to an unclassified object, it is permitted but reading this object later is only authorized to subjects with the appropriate classification. In other words, the classification of an object evolves according to the information it contains. Thus, the control is based on the information flow.

The concept of information flow control was clearly described in the seminal work of Denning [DD77]. The information-flow policy of a program is defined using a lattice $(L, \sqsubseteq)$ where L is a set of security classes (i.e., classifications and categories) and $\sqsubseteq$ is a partial order among those classes [KWH11].

2.2 Information flow control in programming languages

Information flow control has found a lot of applications in various domains. In particular, there has been great interest towards using various IFC models at the level of programming languages. In programming languages, it is often important to protect several confidential data, such as secret keys or passwords, and restrict access to some important code components while continuing to use third-party libraries. Thus, IFC at a programming languages level consists of associating labels to the variables that contain sensitive values and then propagating these labels according to the flow of information that occurs during the execution of a program.

The key consideration when it comes to IFC in programming languages has to do with the nature of the information flow. Conventionally, there is a clear distinction between explicit and

```
1 b = a+1;
2 c = d = 0;
3 if (a == 2) {
4   c = 1;
5 } else {
6   d = 2;
7 }
8 print b,c,d;
```

Figure 2.2: Explicit/Implicit Flow

implicit flows [DD77]. An explicit information flow refers to direct assignments. In this case, secret variables only influence the current executing statement. Let us consider the example in Figure 2.2. We consider variable `a` to contain a secret value. Hence, we classify `a` as *secret*. The assignment in line 1, `b = a + 1;` is an example of such an explicit flow. It is clear that after the execution of this statement, the value of variable `b` is dependent on value `a`. Hence, `b` must also be classified as *secret* due to the propagation of the secret value.

In the case of implicit flows, the variables used in one statement influence subsequent statements indirectly. Implicit flows arise due to conditional jumps based on the value of the secret. In this case, the execution path taken by the function was determined by a secret value, thereby determining subsequent assignments. A simple `if-then-else` conditional shown in line 3 of the Figure 2.2, is an example of such implicit flow. Let us consider `a` is classified as *secret*. The value of variable `c` and `d` are indirectly related to the value of variable `a`. The direct assignments to variable `c` and `d` do not contain any secret values. However, the assignment statement that is executed depends on the value of `a`. Hence both variables `c` and `d` must be classified as *secret* due to implicit information flow of variable `a`'s secret value.

2.3 Working of IFC

The working of an IFC can be described by using the diagram shown in Figure 2.3. The system begins with the definition of a policy which in turn contains details about the security lattice and the files/variables holding secret information. Such information is used by IFC models to identify legal information flows. The IFC models could be probabilistic or possibilistic. A probabilisitic model allows for some amount of disclosure as long as the leak is within a permissible limit. A possibilisitic model denies disclosure if there is a slightest possibility of a leak. These models can be implemented using different types of analysis. Static analysis infer details from code before executing it. Dynamic analyses observe runtime environments. Hybrid analysis use a combination of both static and dynamic analysis. These analysis classify

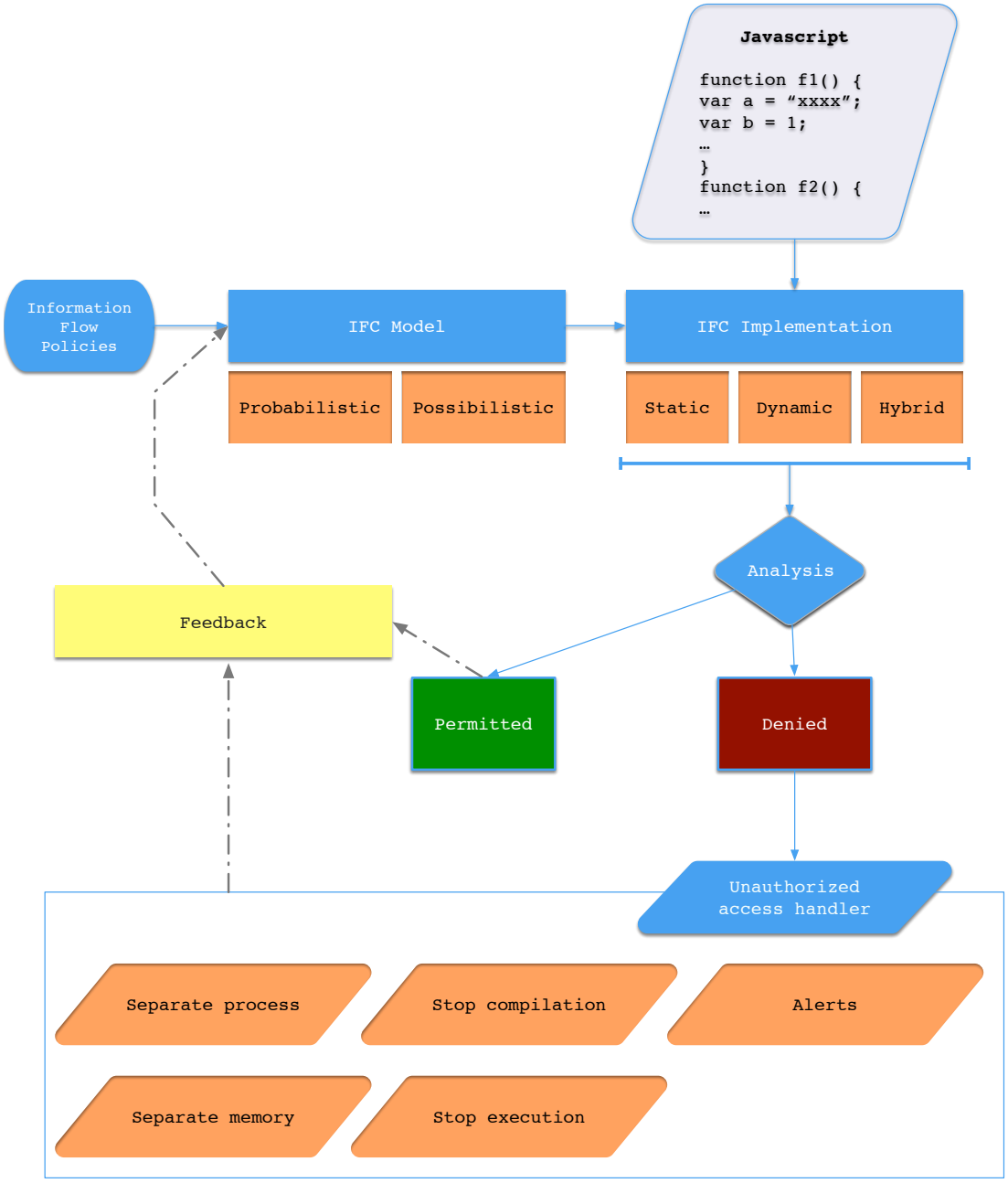


Figure 2.3: Information Flow Working

information flows into legal and illegal flows.

The reaction of the IFC system to illegal information flows depends directly on the implementation. These could be raising alerts, stopping execution, stopping the compilation process, modifying execution or some other customized action. Finally, there could be an optional feedback mechanism that is used to update information in the approaches. For probabilisitic models, these could help in re-computation of their information leakage metrics. This feedback could be based on the past decisions.

This section is organized into three subsections. The Subsection 2.3.1 gives an introduction to possibilistic and probabilisitic IFC models. The Subsection 2.3.2 provides an overview of the various properties that have been used in the domain of IFC. Finally the various types of analysis are described in the Subsection 2.3.3.

2.3.1 IFC models

There are two types of IFC models, namely, the possibilistic IFC and the probabilistic IFC. The former is a coarse-grained approach which considers any possible influence of a variable to another as a leak. The latter tries to evaluate more precisely the amount of information that is leaked.

```
1  function main()
2  {
3    var text = secret; //Gets the value of the secret.
4    var abc = 0;
5    if(text.indexOf('abc')!=-1) { abc = 1; } //the variable abc now
         contains some information about text.
6    var text1 = 'Secret text is: ' + text;
7    var text2 = 'Secret starts with: ' + text.charAt(0); //Just a
        single byte is appended
8    var text3 = 'Contains string \'abc\': ' + abc; //abc is
        appended - this contains some information about text
9    publicOutput(text1);
10   publicOutput(text2);
11   publicOutput(text3);
12 }
13 function publicOutput(x)
14 {
15   console.log(x);
16 }
```

Figure 2.4: Information flow code example

2.3.1.1 Possibilistic IFC

The possibilistic models have been more popular in IFC research [AF09, CMJL09, DG09, BS10, Aus13]. The objective of this approach is to eliminate any possibility of leak, if the leak is considered feasible by the model. In this case, the policies have to be formed to permit or disallow information flows. Consider the code of the figure 2.4. In this case, the variable `text` is tagged with a high-level label since a secret value flows into it. The variables `text1`, `text2` and `text3` become tagged with a high-level label because of information flow. The function `publicOutput()` is not allowed to accept high-level variables. In a possibilistic IFC, the flows resulting from the execution of the lines 9, 10 and 11 will be considered as illegal.

Here, we classify the `text2` at the same level of information leakage as `text`. This is because the possibilistic approaches are coarse-grained making it impossible to distinguish between partial leakage and the leakage of the whole secret. This kind of classification is however necessary in a possibistic IFC to prevent information leak.

2.3.1.2 Probabilistic IFC

The probabilistic information flow control models are an alternative to the possibilistic models. These approaches try to quantify the amount of information that is leaked. In this case, every information flow is not only marked, but also trailed for information leakage and bound to the lattice based models. Once these values reach a threshold, the corresponding information flows are considered illegal. The main motivation of a probabilistic approach is to have a more fine-grained analysis. Such an approach relies on quantifying metrics to capture the measure of information leakage at every possible point in the programs' execution life-cycle. Alvim et al. [AAP10] make a comparative study on the various probabilistic information flow techniques that are currently being used.

Let us consider the previous example illustrated by code of Figure 2.4. In the case of the probabilistic information flow control, information leakage would be estimated for each disclosure. In this case, revealing the presence of "abc" may be acceptable but the string in its entirety should not be disclosed. Enforcing such a policy would be possible using probabilistic information flow control. For example, the creation of the string "text1" at line 6 (see Figure 2.4) would be considered as illegal since this would imply revealing the whole secret variable. However, revealing a much less significant part of the secret in "text2" could be allowed. It must be noted that in this case it involves disclosure of only the first byte. The all or nothing model of the possibilistic model is therefore substituted by the controlled partial disclosure of the probabilistic models.

2.3.1.3 Possibilistic vs. probabilistic IFC

The general idea of probabilistic approaches provide direct advantages over possibilistic approaches by allowing for more fine-tuning thereby reducing the margin for over-approximation. Hoang et al. [HMM+12] formulate a comparison between the probabilistic and possibilistic approaches. One of the inferences made by this research is that possibilistic approaches tend to make approximations which result in a potential loss of precision in their final classification.

However, probabilistic models often require more information to decide the classification. This is because when building a probabilistic model, it can generally be found that several types of probability distributions fit the data [NCC+04, AAP10]. There have been no practical probabilisitic models to date that have seen implementations in a real web browser. While we acknowledge the greater precision that can be obtained by probabilistic models, we see this more as a possible future enhancement.

2.3.2 IFC properties

The IFC models have to satisfy some properties that are needed to formally express the absence of information flow or to quantify leakage.

In case of the possibilistic models, the most important property that has been identified is non-interference [GM82, SS98]. This property states that no secret inputs to the program can influence publicly observed outputs. Formulated in terms of program executions, if the program is run with different secret values, while holding the public values fixed, the public output must not change [HS12a]. Different formulation of this property have been proposed. We will detail those which are related to our context in the following subsections.

The probabilistic metrics differ in their need when compared to the general models followed by the possibilistic approaches. Indeed, those approaches try to evaluate the amount of leakage in order to allow partial information disclosure. We provide some details on these metrics in the subsection on 2.3.2.3.

2.3.2.1 Termination-insensitive non-interference - TINI

Termination-Insensitive Non-Interference (TINI) [AHSS08, Bie13, SM03] only gives a guarantee about terminating programs, ignoring that non-termination may leak some confidential information. This property guarantees that two terminating executions of a program produce output that agrees on public data when started with input that agrees on public data [DP10]. Figure 2.5 provides a general understanding of this property. It can be seen that the public input and public output are the same irrespective of the private output.

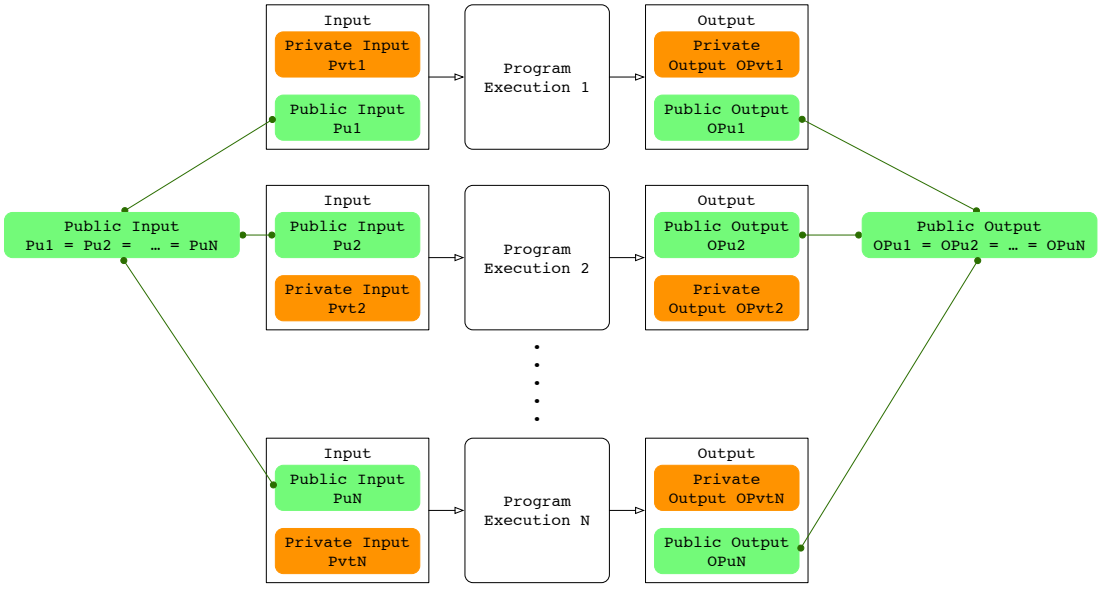


Figure 2.5: Termination-insensitive non-interference

Let us consider the code of figure 2.4. In case of TINI, the observable public output correspond to the execution of the `publicOutput()` function which prints information to the JavaScript `console`. According to the TINI property, `console` output must remain the same regardless of the value of the secret. In this case, the information flow into `publicOutput()` on line numbers 9, 10 and 11 would be deemed illegal. It must be noted that termination-insensitive programs only provide security guarantees for programs which terminate [KWH11]. It hence ignores the possibility of non-termination or of abnormal termination due to unchecked exceptions such as out-of-memory errors.

2.3.2.2 Timing- and Termination-sensitive non-interference - TTSNI

Timing-sensitive non-interference implies that the public output cannot distinguish the secret solely based on the time of the execution. This means that the various values of the secret should not influence the number of steps taken to reach the public output. In case of the example code of figure 2.4, the timing sensitive non-interference is not satisfied. More precisely, the if statement in line number 5 would only execute the assignment operation when the absence of the substring 'abc' in the secret is established. This additional step being executed can be exploited by timing based attacks to infer some information about the secret. In a timing-sensitive non-interference there should be no observable time difference for different values of the secret. Hence, it should take the same time to execute line 5 irrespective of 'abc' being present in the secret or not.

In case of termination-sensitivity, the termination is supposed to be a public output and should not be influenced by secret values. Consider replacing line 5 with the following statement:

`while(text.indexOf('abc')!=-1) { continue; }.`

This would disclose the value of the secret based on termination of the program. A termination-sensitive non-interference makes sure that such an information flow does not influence the public output.

TTSNI states that after any number of execution steps, two executions of a program will have produced output which agrees on public data when run with input that agrees on public data [DP10, KWH11]. This states that the program uses the same number of steps to reach a public output making it timing sensitive. It also states that the termination of the program would not be directly influenced by the secret value hence also guaranteeing IFC for non-terminating programs.

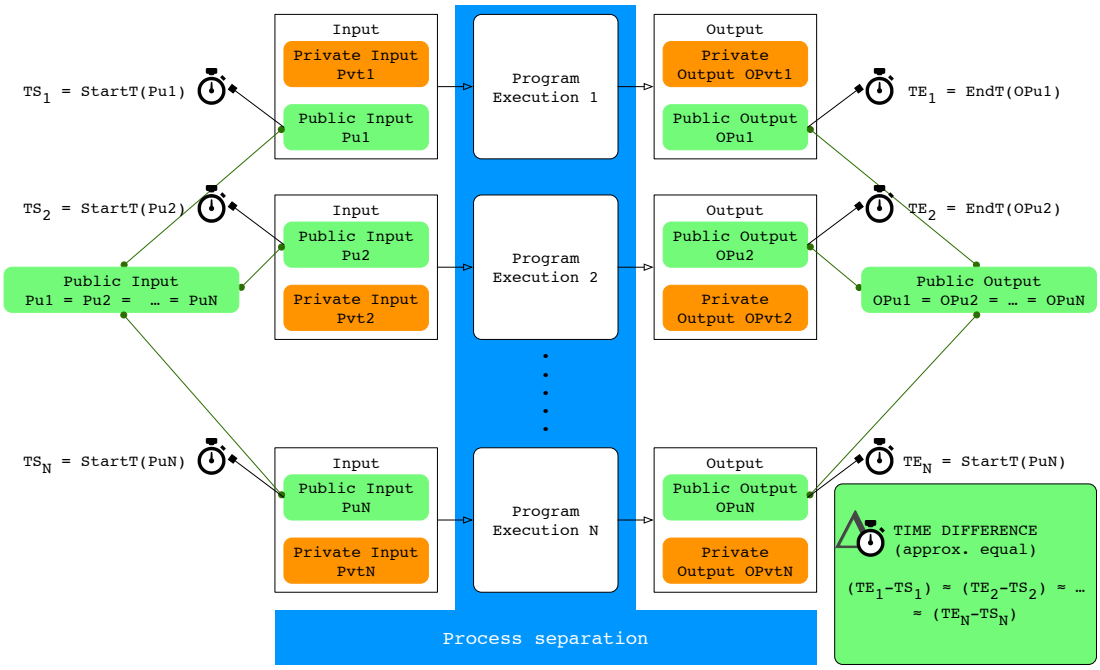


Figure 2.6: Timing- and Termination- sensitive non-interference

Figure 2.6 provides a general understanding of this property. It can be seen that this figure is a superset of Figure 2.5, which represents TINI. TTSNI also needs to satisfy that for the same public input, there is no observable difference in the time taken to generate the public output. In the Figure 2.6, the `StartT` and `EndT` represent the starting and ending time for the execution of the function and `EndT` - `StartT` represents the time taken for the execution. It must be noted

that the time taken for the execution must be the same, baring system noise. Since there is process seperation, i.e. seperate processes for each level in the lattice, the termination (or crash of a process) of the high process would not affect the low processes and vice-versa.

Kashyap et. al. [KWH11] make a distinction between termination-insensitive, weakly termination-sensitive and strongly termination-sensitive programs. Consider a program that consists of two levels, high and low. The approach creates two sub-programs for the program, namely a high sub-program and a low sub-program. In a termination-insensitive non-interference, the execution of a low program's code block is not independent of the high-program preceding it and vice-versa. Hence, if high-program is stuck or causes an abnormal failure of the program because of a particular value of the secret, this value can be inferred. In a weakly termination-sensitive sub-program, the execution of each code block is generally independent of each other. However, it does not take into account that for certain inputs, a high sub-program may over-use the available memory thereby causing abnormal termination is via memory exhaustion to the low sub-program[1]. In a strongly termination-sensitive program, these scenarios are also handled. Notice that while the notion of strongly termination-sensitive models has been formalized by Kashyap et. al., there are no known models that satisfy this property.

2.3.2.3 Probabilistic metrics

In a possibilistic approach, the leak either exists or does not. However, a leak is quantifiable in a probabilistic approach. Various approaches have been proposed to quantify information leakage i.e. partial disclosure. The techniques for calculation of partial disclosure have mathematical roots whose applications go well beyond the realm of information flow control.

These approaches share some common metrics. Information leakage or information exposure is the intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information [Mit, AAP10]. This metric is defined by Alvim et al. [AAP10] using initial uncertainty and remaining uncertainty as:

$$leakage_{info} = uncertainty_{initial} - uncertainty_{remaining} \tag{2.1}$$

Initial uncertainty is the entropy of the initial input. This value takes into consideration the various initial conditions that are taken for the input. Remaining uncertainty is the conditional entropy of the output given the input.

The general idea of probabilistic approaches provide direct advantages over possibilistic approaches by allowing for more fine-tuning thereby reducing the margin for over-approximation.

[1] Out of Memory,`https://en.wikipedia.org/wiki/Out_of_memory`

Hoang et al. [HMM+12] formulate a comparison between the probabilistic and possibilistic approaches in the context of non-interference. One of the inferences made by this research is that by possibilistic approaches tend to make approximations that result in a potential loss of precision in their final classification.

However, probabilistic models also require a lot of initial data to achieve their proposed benefits. The complexity of implementation of such models cannot be underestimated. There have been no practical probabilisitic models to date that have seen implementations in a real web browser. The amount of computation required for every decision can also not be underestimated making their efficiency questionable at best. While we acknowledge the greater precision that can be obtained by probabilisitic models, we see this more as a possible future enhancement rather than a necessity. Hence our model remains a purely possibilistic approach.

2.3.3 Types of IFC analysis

The are two main stages in a program's life-cycle that the IFC analysis could take places namely before the execution (static analysis) or during the execution (dynamic analysis). Some approaches tend to take advantage of both techniques (hybrid analysis).

2.3.3.1 Static Analysis

Static analysis is a greatly explored technique in information flow control and there are several approaches in this context. Models that fall into this category typically do their analysis before the execution of the program begins. Some of them follow the methodology of secure-type systems [SS98, SM03, Mye99, VS97b]. Those approaches work by propagating labels in the form of types associated to the variables. There is a use of a mechanism called the program counter (pc) that is common between these various approaches. This mechanism maintains the current level of the execution based on the information used in the different conditional branch.

A security-type system is a collection of typing rules that describe what security type is assigned to a program, based on the types of subprograms [SM03, VIS96]. This is kept as a meta-data of the static analyzer and is computed for each step of static analysis. These analyses compute the flow of information in almost the same manner as if they were being executed while exploring all possible paths for various variable values. For example, figure 2.7 presents the rules of the secure type system proposed by Sabelfeld and Myers [SM03]. These rules were inspired by the rules proposed in the seminal work of Volpano et. al. [VIS96]. These have been described in the while language [Ald06, CHM07]. While language is a simple imperative language, with assignment to local variables, if statements, while loops, and simple integer and boolean expressions.

[E1] $\vdash exp \ : \ high$

[E2] $\dfrac{h \notin Vars(exp)}{\vdash exp \ : \ low}$

[C1] $[pc] \vdash skip$

[C2] $[pc] \vdash h := exp$

[C3] $\dfrac{\vdash exp : low}{[low] \vdash l := exp}$

[C4] $\dfrac{[pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash C_1;C_2}$

[C5] $\dfrac{\vdash exp : pc \quad [pc] \vdash C}{[pc] \vdash \texttt{while}\ exp\ \texttt{do}\ C}$

[C6] $\dfrac{\vdash exp : pc \quad [pc] \vdash C_1 \quad [pc] \vdash C_2}{[pc] \vdash \texttt{if}\ exp\ \texttt{then}\ C_1\ \texttt{else}\ C_2}$

[C7] $\dfrac{[high] \vdash C}{[low] \vdash C}$

Figure 2.7: Secure-type system [SM03, VIS96]

In these rules, $\vdash exp : \tau$ implies that the expression *exp* has a type τ according to the typing rules. This is an assertion that needs to be satisfied. $[pc] \vdash C$ infers that the program C is typable in the security context *pc*. The rules [E1] and [E2] refer to expressions. [E1] implies that any expression can have a type *high* ($\vdash exp : high$) irrespective of the data it contains. However, Rule [E2] states that expressions can be of type *low* ($\vdash exp : low$) only if none of the variables in the expression have a type *high*. The rules from [C1] to [C7] refer to the program's working. [C1] states that for either *high* or *low* expressions, a `skip` operation is typable. Similarly, in [C2], a *high* variable may take input from any *high* or *low* expressions. However, in case of [C3], there are two restrictions for the rule to apply. The input has to be from a *low* expression and the assignment should be to a *low* variable. If these are satisfied, the *low* variable is assigned the value resulting from the evaluation of the expression.

The rules from [C1] to [C4] are purely targeting explicit flows. Implicit flows are analyzed by rules [C5] and [C6]. These rules simply state that for a conditional branch or loop to be typable in a *low* context, both the expression *exp* of the conditional must be *low* and the loop/branch body C must be individually typable in the *low* context. In all other cases the statement is only typable in the *high* context. This is enforced by the subsumption rule in [C7] which refers to rules where *pc* is used. [C7] states that, in these rules, (namely, [C1], [C2], [C4], [C5] and [C6]), if the rule is typable for the *high* context, it is also typable for the *low* context.

Heintze and Riecke [HR98] proposed a type-system for functional language which is ca-

pable of handling first-class functions. Other models such as the one proposed by Zdancewic and Myers [ZM01] provide more expressive control for functions and references while also providing robust declassification methodologies in their semantics. The context of exception handling has also been explored by various approaches. Volpano and Smith [VS97a] proposed a simple type system for handling exceptions. A more detailed analysis was proposed by Pottier and Simoner [PS03] in which exceptions were handled with great detail. This binding between the exception raised and values is validated and a program is considered typable only if all exceptions that can be raised by the program are typable in that context.

There also exist a set of approaches [CC77, DFST02, Mas05, Zan12, AN16] which follow the path of abstract interpretation in their approach towards static analysis. Abstract interpretation is used to collect approximate information about the runtime behavior of a given program [DFST02]. This implies that it sacrifices standard precise semantics in favor of simpler non-standard semantics known as abstract domains. The program is then interpreted in these abstract domains. Let us consider simple example provided by Cousot and Cousot [CC77] involving mathematical signs. Consider `x = -1515 * 17` in abstract domains denoted by (+), (-), (±) where the semantics of the arithmetic operators are defined by the rule of signs. Abstract execution of -1515 * 17 can be represented as (-)*(+)=(-), thereby proving `x` is a negative number. In case `x = -1515 + 17`, abstract interpretation would result in (-)+(+)=(±). Cousot [Cou] also states that for an abstraction to be sound, the abstract semantics must cover all possible cases of the concrete semantics. Let us consider a simple security lattice (high-low). The *high* and *low* security labels would have their own abstract domains. The main purpose of abstract interpretation is to approximate the concrete semantics of all executions in a finite way. Only information concerning the properties being analyzed is maintained. In information security, this property is the label associated to the program. In general, abstract semantics of a program ignore both values and memories, thereby completely focusing only on the property and the various control flows. In case of a branch, the points of interest for the program are more focused on the beginning or end of the branch.

An example of abstract interpretation semantics for Basic LOTOS as proposed by De Francesco et al. [DFST02] is given in figure 2.8. Basic LOTOS is a process algebra by means of which it is possible to describe the behavior of concurrent processes, concentrating on communications between processes. It includes commands for synchronous communication and parallelism.

In this set of rules, $\natural$ refers to the abstract domain, i is a set of simple instructions such as assignments, sending/receiving messages and `skip` command, op stands for usual arithmetic. $i : exp \rightarrow com$ implies, if expression exp is true, then perform the command com. The expres-

sion $a!e$ implies that the expression e is passed over the message channel a. The expression $a?x$ implies that a message is received from the channel a and saved to variable x. τ is the evaluation of a condition and can be either `true` or `false`. α represents an action. An action can either be a simple command (without sending/receiving messages) or a pair of sending/receiving commands or an evaluation of a condition (τ). λ refers to a 'do nothing' operation. The assignment rule `Assign`, `skip` operation, and $\texttt{While}_{false}$ simply keep track of the abstract domain and do nothing else. The rules $\texttt{Seq}_1$ and $\texttt{Seq}_2$ show how a sequence of commands are handled when they are in the same abstract domain. Message passing is handled by rule `Com` where `r` and `t` represent reception and transmission respectively. The final rule `Par` represents the composition rule for this abstract interpretation.

$$
\begin{array}{lll}
\textit{exp} & ::= & k \mid x \mid \textit{exp op exp} \\
\textit{simple\_com} & ::= & \texttt{skip} \mid x := \textit{exp} \mid a?x \mid a!\textit{exp} \\
\textit{com} & ::= & i : \textit{simple\_com} \mid \texttt{if } \textit{exp} \texttt{ then } \textit{com} \texttt{ else } \textit{com} \mid \\
 & & \texttt{while } \textit{exp} \texttt{ do } \textit{com} \mid \textit{com}; \textit{com} \mid \{\textit{com}\} \\
\textit{proc} & := & \textit{com} \mid \textit{proc} \| \textit{proc}
\end{array}
$$

$$
[\texttt{Assign}]\ \frac{}{i : x := e \xrightarrow{i}{}^{\natural} \lambda}
\qquad
[\texttt{Skip}]\ \frac{}{i : [\texttt{skip}] \xrightarrow{i}{}^{\natural} \lambda}
$$

$$
[\texttt{If}_{true}]\ \frac{}{\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \xrightarrow{\tau}{}^{\natural} c_1}
\qquad
[\texttt{If}_{false}]\ \frac{}{\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \xrightarrow{\tau}{}^{\natural} c_2}
$$

$$
[\texttt{While}_{true}]\ \frac{}{\texttt{while } e \texttt{ do } c \xrightarrow{\tau}{}^{\natural} c; \texttt{while } e \texttt{ do } c}
\qquad
[\texttt{While}_{false}]\ \frac{}{\texttt{while } e \texttt{ do } c \xrightarrow{\tau}{}^{\natural} \lambda}
$$

$$
[\texttt{Seq}_1]\ \frac{c_1 \xrightarrow{\alpha}{}^{\natural} \lambda}{c_1; c_2 \xrightarrow{\alpha}{}^{\natural} c_2}
\qquad
[\texttt{Seq}_2]\ \frac{c_1 \xrightarrow{\alpha}{}^{\natural} c_1'}{c_1; c_2 \xrightarrow{\alpha}{}^{\natural} c_1'; c_2}
$$

$$
[\texttt{Com}]\ \frac{}{c_1 \| \ldots \| r\ :\ a?x\ ;\ c_i \| \ldots \| t\ :\ a?e\ ;\ c_j \| \ldots \| c_n \xrightarrow{r,t}{}^{\natural} c_1 \| \ldots \| c_i \| \ldots \| c_j \| \ldots \| c_n}
$$

$$
[\texttt{Par}]\ \frac{c_i \xrightarrow{\alpha}{}^{\natural} c_i'}{c_1 \| \ldots \| c_i \| \ldots \| c_n \xrightarrow{\alpha}{}^{\natural} c_1 \| \ldots \| c_i' \| \ldots \| c_n}\ c_i \neq r\ : a!e, c_i \neq r\ :\ a?x
$$

Figure 2.8: Abstract interpretation semantics [DFST02]

Based on these rules, a control flow graph is generated for a given program to affirm whether it is secure or otherwise. The various states are checked and noted for whether these

states are reachable.

The abstract semantics defined in rules of Figure 2.8 also have a concrete semantics involving more factors such as memory handling for the program, which were approximated to the abstract semantics.

In terms of application of abstract interpretation to non-interference, the seminal work of Mastroeni [Mas05] in formulating abstract non-interference is noteworthy. The model keeps track of the variables' states at the beginning and end of the branch across various conditionals and performs an evaluation to discover the information flow. The approach keep track of the implicit flows and maps the variable changes between different branches. If a given variable is assigned a constant value (value which is not from a variable or return of a function call), the states at the beginning and the end of the branch would be the same. Hence, the evaluation would not be able to see any dependencies which is a touted advantage of this approach.

Both abstract interpretation and secure type systems have similar effectiveness in handling static IFC analysis. Based on individual adaptations of the rules in these approaches, they can be used to describe and prove if adequate security has been achieved.

Since static analysis happens before the execution, it generally has no impact on performance during the execution. There are some special cases, such as in JavaScript [CMJL09], where the analysis takes place on a Just-In-Time (JIT) basis, keeping up with the dynamic compilation process. In these cases, the code is analyzed just before it is loaded for execution. Hence, there is no performance advantage gained under these specific cases.

Moreover, analyzing programs prior to execution is considerably more secure. This is because any possible leak can be caught by the analysis before execution thereby preventing insecure programs from being executed. Such an approach analyzes all possible traces of the program to find information flow across security levels in both the explicit and implicit information flow contexts.

However, there is possible loss of precision when making the semantics decidable. This is because in general, by the inference of Rice's theorem [Ric53], there is a compromise to be made between the precision of the analysis and its decidability or complexity. Further, in a language such as JavaScript the variables are all called by reference making it very complex and tedious to make a viable static analysis implementation for the language. This is one of the reasons that there is no known purely static analysis IFC implementation to handle JavaScript.

2.3.3.2 Dynamic Analysis

Dynamic analysis [GDNP12, DG09, Aus13, HS12b, CF07, BS10] is performed during the execution process. This approach is advantageous to analyze the current execution path and

finding if the execution may proceed further or not. It does not have any pre-execution overload since the entire process happens during execution. Dynamic analyses actually run the program and hence may actually end up executing malicious code if not properly detected. It incurs also significant run-time overhead. There are several categories of dynamic analysis namely, taint analysis [HS12b, Aus13], multi-execution [GDNP12] and multi-path execution [AF09, Aus13].

Taint analysis In taint analysis, the various variables are provided labels and these labels transmit meta-data between each other when information flow occurs. Hence the information flow is kept in check through the propagation of labels in tandem with the data for every program statement. This technique is used by different approaches such as the traditional tainting model applied in the JavaScript context by Hedin and Sabelfeld [HS12b] and the no-sensitive upgrade model proposed by Austin and Flannagan [Aus13].

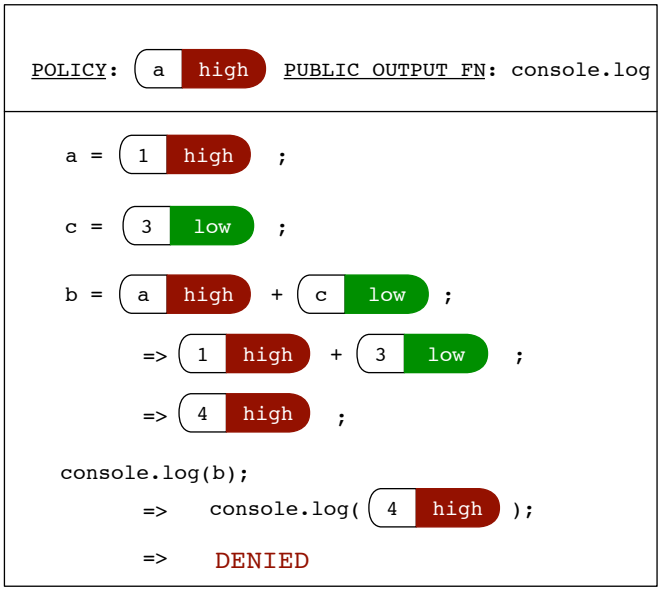


Figure 2.9: Label based taint marking

An example of tainting model is shown in Figure 2.9. In this example, there are two possible levels in the lattice, `high` and `low`. The policy stated in the diagram, a high, represents that the variable `a` is a high value. The public output function (represented in the figure as public output fn), represents the output function at the low level of the lattice. In this case, the function `console.log` is the public output function. When the program is executed the labels are also propagated along with the values. At the end of the execution, `b` gets a `high` label since it contains information about `a`. The execution of `console.log` is not permitted to

accept `high` values. Hence, the dynamic analysis considers such an execution as insecure.

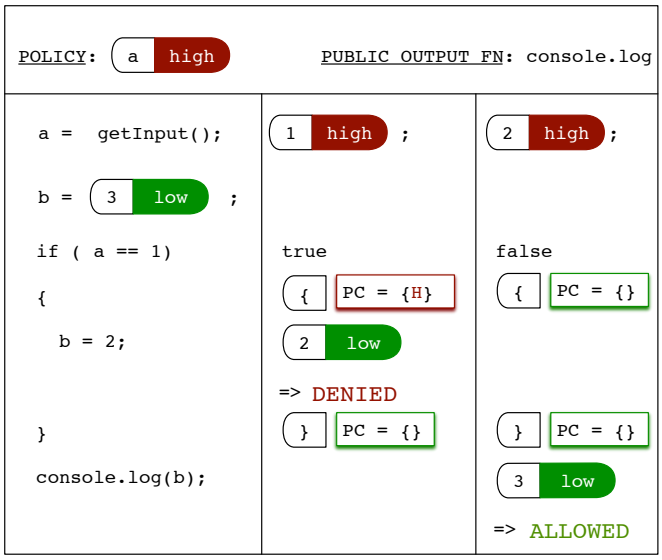


Figure 2.10: Indirect flows in no-sensitive upgrade

An example of no-sensitive upgrade model [Aus13] is shown in Figure 2.10. In this example, the preliminaries regarding the policy and the public output function are the same as the example in Figure 2.9. In this case, we show the working of the function in case the conditional was `true` as well as when it was `false`. Since the conditional is based on a high value from variable `a`, the program counter `pc` maintains this dependency till the end of the execution of the this `if` block, just like in static analysis. It can be seen that in the case of `false`, there is no value propagation, thereby, `b` remains a public value that can be given as a public output. However, in the case of `true`, the value of `b` becomes dependent on `a` due to implicit flows. In this scenario, the no-sensitive upgrade performs an over-approximation to stop further execution. Other models such as the one proposed by Hedin and Sabelfeld [HS12b], request an explicit upgrade instruction to handle this scenario. This explicit upgrade instruction allows to modify the label of the variable `b` thereby allowing the implicit flow from the secret variable `a` to `b`.

The main issue with the models implementing taint analysis is that they have failed to account for the same amount of rigor in analyzing all possible execution paths when compared to the static analysis techniques. This directly results in the failure to handle the context of implicit flows properly. The models only analyze the current execution path and do not take the other execution paths into account. Hence, when the label propagation happens because of an implicit flow, it cannot be propagated without explicit upgrade/declassification instructions.

Secure Multi-Execution The model of Secure Multi-Execution (SME) was proposed by Devriese and Piessens [DP10]. In SME the information flow across labels is segregated at the process level by providing a separate process for each level of sensitivity. Let us consider a system with two levels namely high and low. Such a scenario would imply that there is a dedicated process for high level computations and a dedicated process for low computations. The low level process is the only one that can influence public output and it can only receive public input. A high level process is given access to input from both the low and high levels but is denied access to the public output functions. A representation of a simple two level SME is shown in Figure 2.11. Devriese and Piessens proved that a termination-sensitive non-interference (TTSNI) can be achieved using such approach.

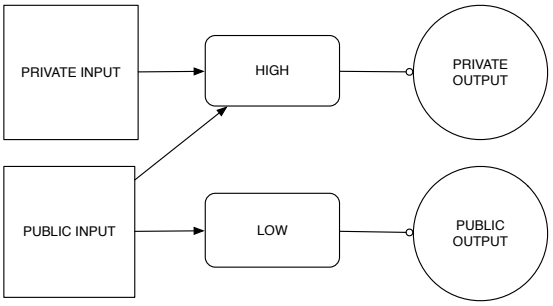


Figure 2.11: Secure Multi-Execution

The number of processes necessary can be directly inferred from the security lattice. Let us consider the lattice shown in Figure 2.1, page 32. In this case, the program would be run by twelve different processes such that each process infers to a particular lattice level. The input at each level can only comprise of information at that level or at a lower level. The output of each level would only be observable by those with authorization at that level or higher.

For example, a process at `{Secret,{Crypto}}` can have inputs from variables at `{Secret,{Crypto}}`, `{Secret,{}}`, `{,{Crypto}}` and `{,{}}` levels. Its output can be accessed by variables at `{Secret,{Crypto}}`, `{TopSecret,{Crypto}}`, `{Secret,{nuclear,crypto}}` and `{TopSecret,{nuclear,crypto}}` levels.

Let us consider the example of Figure 2.12. In this example, the variable `a` is given a high input for a lattice containing only two levels (high and low). This program is executed twice, once by a low process and once by the high process. A high input can only be provided to the high process. The public output function can only be executed in the low context. Hence, the function `console.log` will always print the value 3, regardless of the value of `a`, in this example. It can be clearly seen that since the low process can never access a high value, the `pc` is never updated in the low process. The implicit flow in the high process depends on the high value from the variable `a` and the `pc` keeps track of this dependency till the end of the scope of

the `if` block.

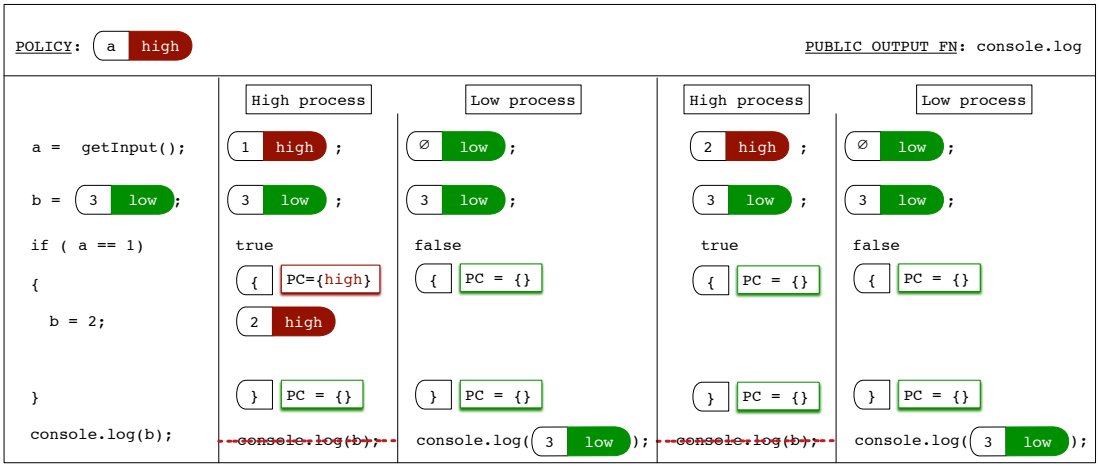


Figure 2.12: Indirect flows in SME

The use of SME automatically increases the complexity for the system. This is because any given program needs to be executed multiple number of times for various levels. Since private input cannot be read in a low process and public output can only be performed by the low process, they are mutually exclusive, thereby satisfying non-interference. Even if the high process results in an exception, a crash or an infinite loop, it would not influence the low process in any way. Hence, the time to obtain the public output would not be influenced by the value of the private input either. This is why SME provides the strong security guarantee of TTSNI.

Faceted approach The faceted approach that has been proposed by Austin et al. [AF09, Aus13] is the chief proponent of the multi-path execution approach. The authors attempt to mimic the functionality of SME with the use of a single process. The faceted approach attains termination-insensitive non-interference since the use of a single process cannot account for timing-sensitivity. This approach works on containing multiple copies of each variable to mimic the values of this variable in different processes in case of SME. A faceted value is represented as $\langle P ? a_{pr} : a_{pu} \rangle$. In this notation, P is the principal which can be equated to a given lattice level. In the faceted value representation, a_{pr} contains the secret value corresponding to the principal and a_{pu} is the value to be used if the output function that does not satisfy the principal when it tries to access the value of a.

For example, in a simple scenario where there are only two levels (high and low) in the security lattice, we can use two principals to represent these levels. If an output functions is tied to the low principal, it will only print the value a_{pu} of $\langle high ? a_{pr} : a_{pu} \rangle$.

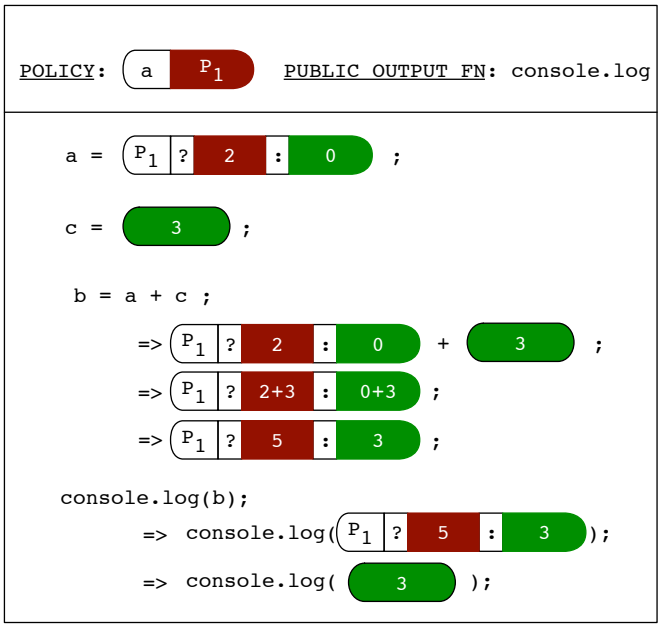


Figure 2.13: Faceted Approach

Whenever there is an operation performed on a faceted variable, the operation is repeated for every facet of the variable. This can be seen in the example illustrated by the Figure 2.13. Here, faceted approach behaves in a similar manner as SME by performing twice the operation `b = a + c`, once for each value of `a`. Of course, in SME, these two operations would have been performed as part of two different processes instead of using a single process as in the faceted approach. Further, computations not involving faceted values would only be done once in faceted approach while SME would evaluate them at every execution.

It must be noted that the faceted approach performs nested computations to keep up with more complex lattice structures involving multiple principals. This number of objects would be 2^n the growth of principals. The positive effect of this phenomenon is that, only the correct copy of the object is used when it is invoked by the public output function. For instance, if an object `c` were created by using two other objects `a` and `b`, each with its own principal, there would be four possible values for this object as illustrated in Figure 2.14.

In case of implicit flows, faceted approach evaluates the entire conditional block for each value of the faceted variable. This can be observed in the Figure 2.15. In case `a == 1` is `false`, no action is performed and `b` has only one value which is not a secret. The `pc` used in the faceted approach is similar as in all the prior static and dynamic analysis approaches. It keeps track of the principals that are used in the execution due to the conditional of the implicit flow. Hence for every value of any secret variable used in the conditional, the `pc` will keep

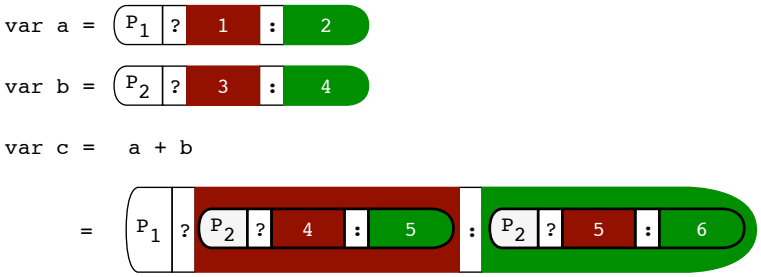


Figure 2.14: Faceted evaluation

track of the principals that need to be satisfied corresponding to the value used. Hence, in the example in the Figure 2.15, there are two possible values for the variable `a` which correspond to two executions. The `pc` keeps track of the principal P_1 which needs to be satisfied to access the secret value. When accessing the public value, there is no principal that needs to be satisfied, which is reflected in the `pc`.

In case the result is `true` however, `b` becomes a secret variable with a private value `2` and public value `3`. However, even in this case, the public output function `console.log` can only print the value `3`. Hence, faceted approach prevents secrets from being processed by public output functions.

Faceted approach generally requires less computational time than SME. On the other hand, it gives lesser guarantees than SME. The idea of attaching different values to variables is interesting but there is still a significant cost to this model due to nesting. Our approach is similar to the faceted approach in the context of having multiple copies for each variable. However, the number of copies held does not exceed two. We also do not evaluate multiple branches exhaustively preferring to execute only the required execution paths like the tainting models.

2.3.3.3 Hybrid Analysis

Hybrid analysis models tend to be a combination of static and dynamic analysis techniques and hence inherit the advantages and disadvantages of both models based on their implementation. The current path refers to the conditional branch that is taken over the course of a normal execution of the program. An alternative path refers to the branches in the program that are not taken (such as the other cases of a switch statement or an else part of an if-then-else statement) because of the conditional. In a hybrid analysis, the commonly used method is to analyze the alternative paths statically and the current path dynamically.

The hybrid analysis techniques [LGJ07, CF07, BBJ13] overwhelmingly use tainting models for their dynamic component while varying their static components. Using the two in tandem allows the approach to increase the precision of the taint propagation and to overcome

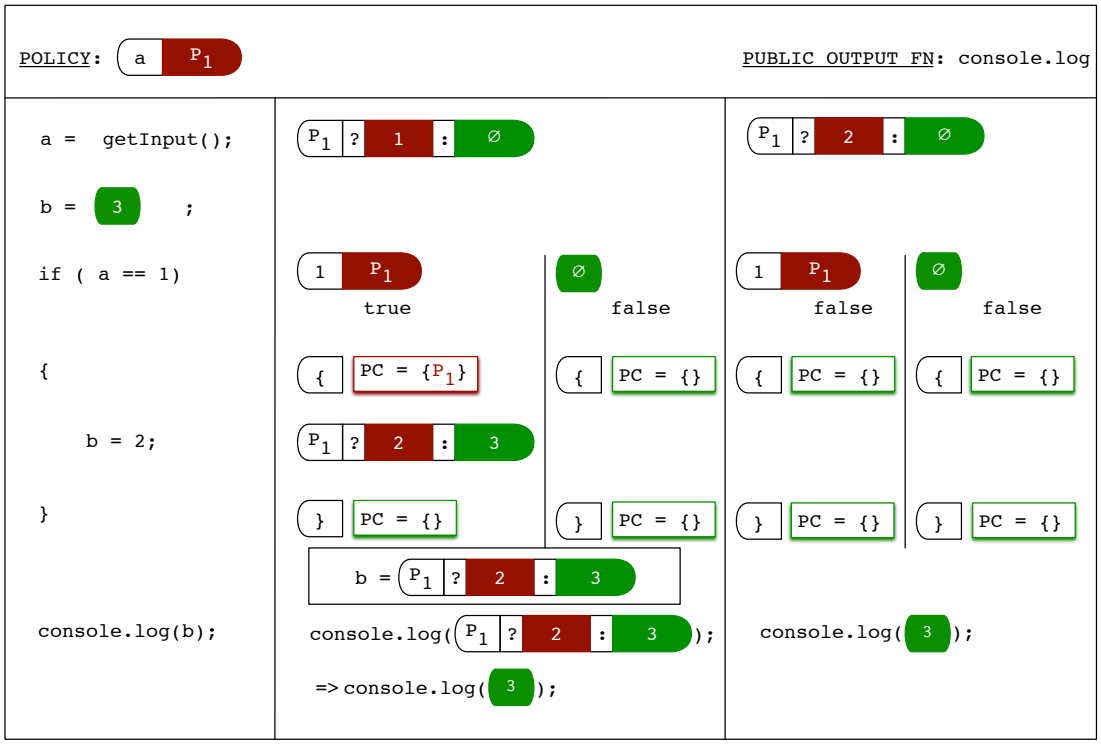


Figure 2.15: Faceted Approach - implicit flow

their weakness with regard to implicit flows. During the execution, the labels flow along with the information flow. Hence, for all explicit information flows, the analysis is very similar to the approach proposed by label based dynamic analysis approaches (see Figure 2.9, page 47). In the case of implicit flows, the dynamic analysis executes the current path. The program uses the various information flow labels from the current execution using dynamic analysis. This is coupled with static analysis results from all other branches. All these results for the control flow block are evaluated before continuing to evaluate the statement following this block.

The label of any variable whose value changed during the implicit information flow is computed. The computed label for any given variable refers to the lowest level in the lattice that can access the value assigned to that variable in the current path as well as the values assigned in the alternative paths. For a simple high-low lattice, if any of the paths contain a high value for a given variable, that variable will have a high label. If the conditional of the implicit flow is based on a secret, the pc is updated with the label of the secret variable.

Hybrid analysis can be divided between whether they check if same values are assigned to a variable between the current and alternative paths. We refer to this as the assignment rule. They can also be divided based on whether they compute if the result of a conditional is a constant. We refer to this as the conditional rule.

A noteworthy hybrid analysis approach in the context of JavaScript was proposed with formal semantics by Besson et al. [BBJ13]. This approach keeps up with the usual approach, which implies that the current path is explored by the dynamic analysis and the alternative paths are explored by static analysis. This approach has been formalized in a small imperative language that only considers native feature variables such as name of the browser which cannot be modified.

Based on the static analysis, the approach by Besson et al. does not change the label of the variables if they are assigned the same constant value at the end of the current as well as alternative paths. As an illustration, let us consider the example of Figure 2.16. The policy stated in the diagram, 

, represents that the variable `a` is a high value and `console.log` is the public output function. In this example, the variable's value because implicit flow is computed for the current path as well as the alternative path. The representation 

implies that the implicit information flow into the variable was from a constant value in the current path which is executed by the dynamic analysis. This implies that till the information flow from alternate paths can be determined, this variable neither be classified as a high value nor can it be confirmed as a low value. Hence, we represent it with the color orange. However, after the static analysis is completed, the determination can be made. Here, the variable `b` becomes classified as a high value while the variable `c` remains a low value.

Similarly, the approach by Besson et al. does not execute alternative paths if the conditional is a constant. Figure 2.17 illustrates this. Here, it can be seen that the variable `b` has been set to a constant value `3`. The analysis proposed in this case is theoretically able to determine that the else part of the if-block is unreachable. After such a determination, the static analysis is not performed for the else block. This directly prevents the wrongful classification of the variable `b`. However, this is only possible for the specific case that a conditional is determined a constant, which is not a common usecase. If the conditional cannot be determined to be a constant, which is possible due to the dynamic nature of JavaScript, the analysis would not be able to do such reductions.

It must be noted that the variables are not labelled only if the assignment is from a constant value and would be labelled if it is from a user input. The value of the variable obtained from the dynamic analysis is the final value of the variable and static analysis is only used to assist with the computation of the label that needs to be assigned to this variable. These two choices form the key factors in making the approach more precise in its analysis. This is because, the number of variables classified is reduced as a direct result of this policy while not affecting non-interference.

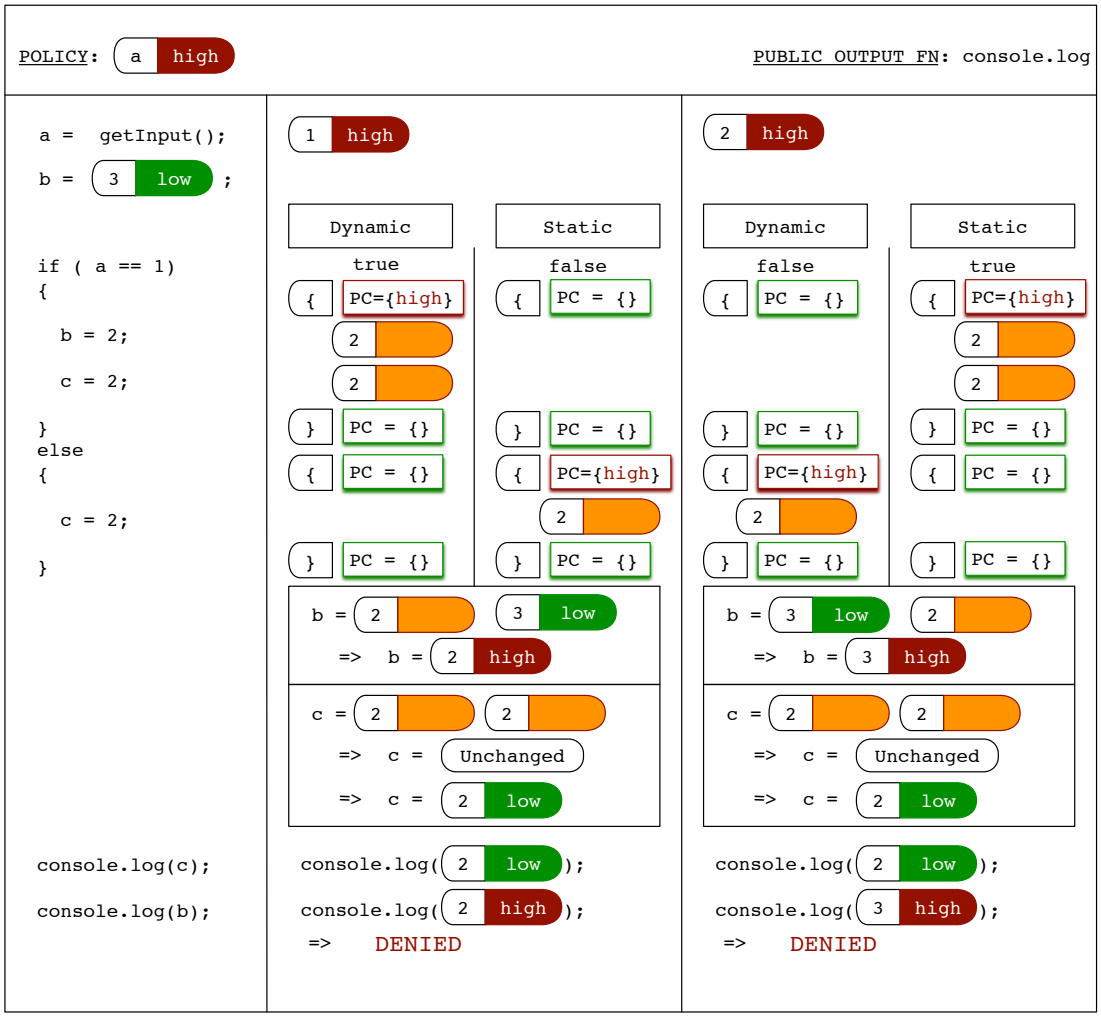


Figure 2.16: Hybrid Information Flow Control with the assignment rule

An earlier approach by Le Guernic et al. [LGBJS07], also outlined the necessity of the conditional rule. In this approach, the static analysis is not executed for any conditional that is not tied to a secret variable. Hence, the Figure 2.17 also holds true for this approach. However, Le Guernic et al. do not consider the assignment rule.

Earlier approaches by Chandra and Franz [CF07] as well as Le Guernic and Jensen [LGJ07], neither followed the conditional rule nor did they follow the assignment rule. In the absence of the conditional rule, the variables would be classified even if the conditional branch cannot be reached over the course of execution. Figure 2.18 illustrates this using the same example as shown earlier in Figure 2.17.

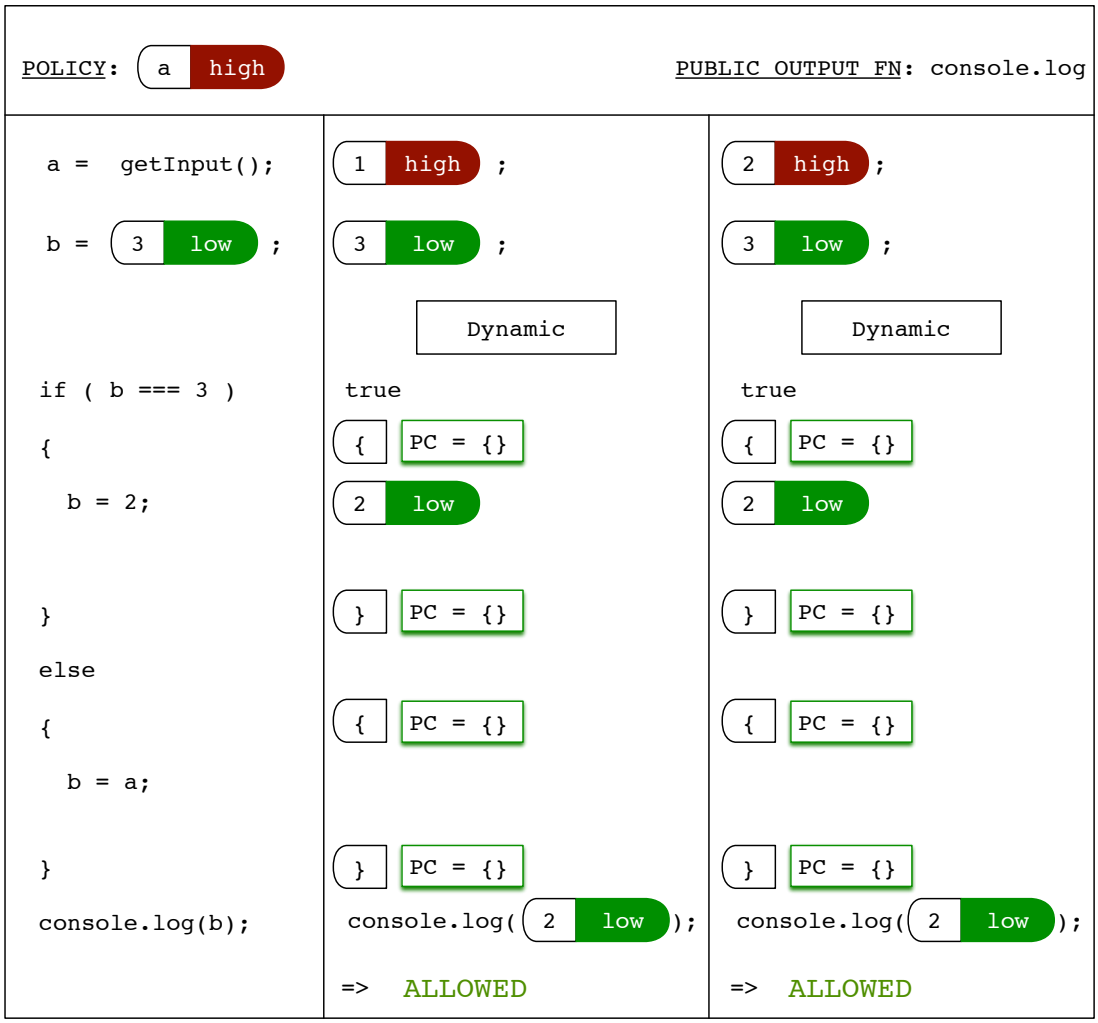


Figure 2.17: Hybrid Information Flow Control with the conditional rule

A model is said to be more precise if it classifies the least amount of variables while maintaining non-interference. Thus, the approach by Besson et al. is more precise than Le Guernic et al. which in turn is more precise than Chandra and Franz, and Le Guernic and Jensen. Of these various models, it is noteworthy that Chandra and Franz implemented their approach in a JVM and cemented the feasibility of the approach for a traditional strictly typed programming language. However, this implementation was in a more strictly typed language, i.e. Java. The approach by Besson et al. while more precise, remains largely theoretical with huge complexities involved in making a feasible implementation.

While hybrid analysis do offer significant benefits, they still rely on static analysis to a great extent. JavaScript's dynamic nature hence causes more complexities to this approach. The use

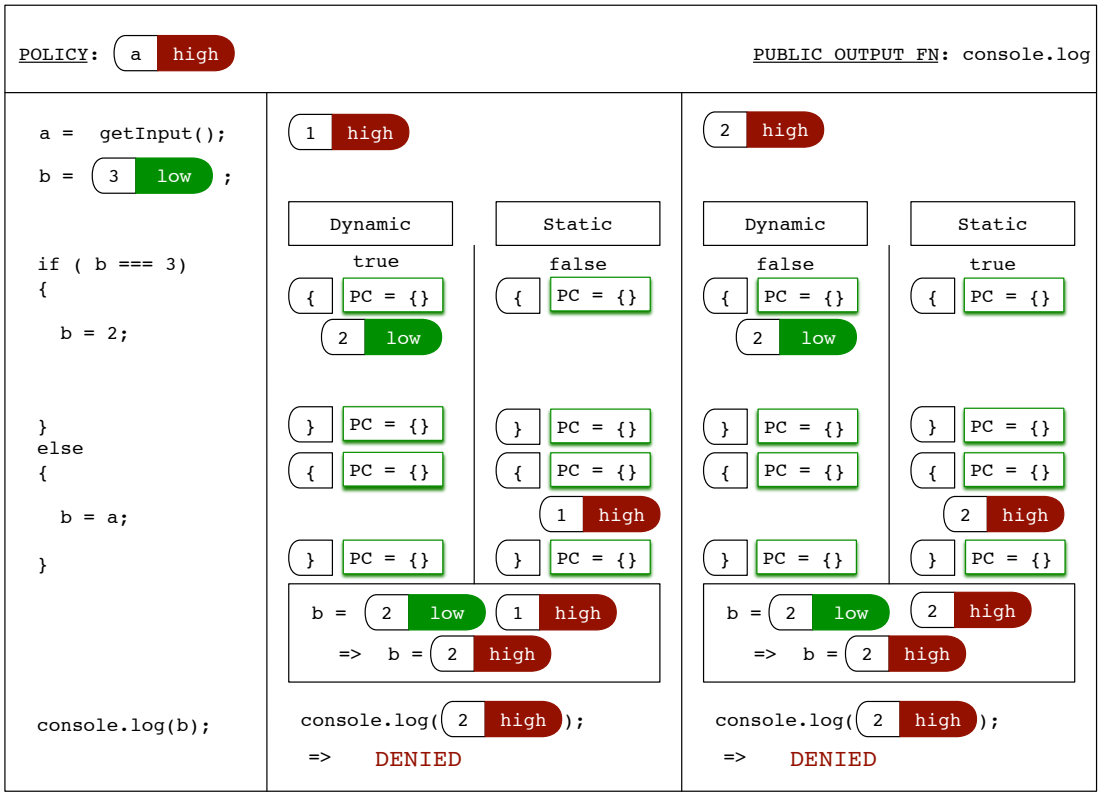


Figure 2.18: Hybrid Information flow without the conditional rule

of eval functions is one of the reasons. JavaScript variables are also accessed by reference rather than by value. This is disadvantageous to static analysis. These have been the key contentious argument towards deciding on a purely dynamic analysis for JavaScript, and the lack of an implemented hybrid approach in JavaScript only helps in cementing this argument.

2.4 Possibilistic web browser security models using IFC

There are several IFC models that have been designed for the web browser taking into account the nature of JavaScript. It is noteworthy that to the best of our knowledge, all these models have used a possibilistic approach. In this section we introduce the various traditional tainting models in the Subsection 2.4.1. We then describe FlowFox, an implementation of SME, and the implementation of the faceted approach in the Subsection 2.4.2.

2.4.1 Traditional tainting models

The most common approach that has been used in IFC has been the traditional label-based tainting approach. The work by Hedin and Sabelfeld [HS12b] distinguishes itself by providing a view of the approach described by Sabelfeld and Myers [SM03] in the context of JavaScript. The authors make an interesting case for the problem of the information flow being flow sensitive in JavaScript. This is because the data types of variables and fields are allowed to vary during the execution. Further, the objects in JavaScript are represented by reference. Hence, there may be cases in JavaScript such that two different variables refer to the same object. This increases the need to keep track of changing labels throughout the execution which becomes tedious with pure static approaches.

```
1 x = {};
2 x.f = 0;
3 y = x;
4 y.f = secret;
```

Figure 2.19: Example of references to same object in JavaScript

Consider the program of figure 2.19. In this example, the variable `x` points to an object and `y` becomes another variable pointing to the same object. In this case, the secret information `secret` flows into the property `y.f`. In static analysis, there is a need to assign the appropriate labels to `x.f` as well. This involves keeping track of all aliases to an object. However, dynamic analysis allows associating the tag directly to the allocated object. Accessing `x.f` would also reflect a labeled secret value.

The difference is that Hedin and Sabelfeld allow some upgrade instructions before the behest of the implicit information flow. The authors provide valid arguments for the implicit control flows generated by JavaScript especially by the *eval* function. For example, a print statement will never be able to print a secret value in the no-sensitive-upgrade scenario. However if an explicit upgrade instruction is given as part of the code before the print statement, it would be allowed to print in the case of Hedin and Sabelfeld's approach. It should also be noted that this approach is formally proved by the authors to fulfill the guarantees of termination-insensitive non-interference.

This work is relevant to our understanding of how the general language based IFC can be correctly adopted to JavaScript. However, requiring upgrade instructions for every possible implicit flow is tedious and without such statements, traditional tainting models will stop further evaluation. Our model does not require such modifications to the code and continues evaluation in the case of implicit flows. We follow a similar strategy as SME and faceted approaches of

having multiple copies of the variable to accomplish a continued execution in case of possible information leakage.

2.4.2 SME and Faceted approach

The SME and faceted approaches are dynamic IFC that execute the alternative branches for the various possible values of variable due to information flow. The approach of SME and faceted approach are able some of the problems related to public output such as clickjacking, cookie-hijacking and block any transmission of sensitive data to remote servers. Just like other IFC approaches, they do not prevent XSS but protect sensitive data from being transmitted to other domains.

FlowFox is a concrete implementation of SME on the Firefox web browser by De Groef et al. [GDNP12]. Since SME adheres to the property of termination-sensitive non-interference, it provides a high level of security guarantees providing a key differentiator for FlowFox. The authors have also taken into account some of the events that may cause information flows such as key-press, mouse move and page load.

De Groef et al. believe in the need to change the JavaScript interpreter of a full fledged browser thereby realizing a more significant result due to the ability to gauge the various factors such as performance (time and space complexity), model verification (the validity of the model in solving the intended problem) and implementation results (to check if the model is suitable in the real world scenario). The implementation has been made in a old version of FireFox and is not currently portable to a later version of the browser. The browser implements a simple security lattice comprising of a high and a low level.

The ZaphodFacets[2] is an implementation of the faceted approach as a plug-in in Firefox. It use the Narcissus JavaScript engine[3]. The Narcissus engine is a JavaScript interpreter written in JavaScript. It was developed to test some experimental features in FireFox and uses undocumented APIs to implement the JavaScript interpreter. The engine runs from an extension to FireFox called Zaphod which can be installed from the Mozilla webstore. There is less documentation on Narcissus due to the experimental nature of the engine. However, the implementation is capable of handling multiple principals and is hence not restricted to a simple high-low lattice.

It must be noted that while SME and faceted approach provide good formal guarantees, executing the various paths that are not part of the current execution can have unintended consequences. This is because, while the server sits on the output of the system, it is also expected to receive secret data. Hence, a function such as XMLHttpRequest would be executed

[2] `https://github.com/taustin/ZaphodFacets`

[3] `http://en.wikipedia.org/wiki/Narcissus_(JavaScript_engine)`

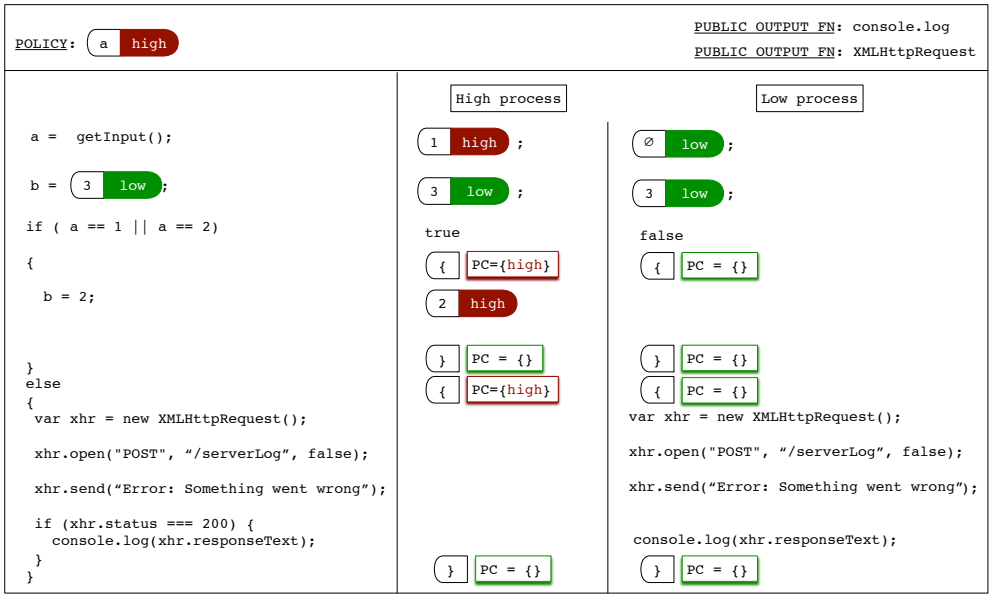


Figure 2.20: XMLHttpRequest in SME

.

and would pass the server false data at every execution. In the example shown in Figure 2.20, a is expected to get an input of 1 or 2. Any other input implies an error that needs to be notified to the server. In this example, it can be seen that the server would log an error regardless of the input for the secret variable a. This is because only the low level process can send such an output. While this action would adhere with non-interference, it would cause an immense amount of difficulty for the server.

This is one of the main reasons that we felt the need to avoid execution of alternate paths was an important consideration for our dynamic analysis approach. Our model adheres to this principle of not forcefully executing these branches for these practical considerations.

Another disadvantage of these approaches is that they are not able to protect sensitive data from being modified if there is a malicious XSS in the web page. This is a very specific problem to JavaScript since other programming languages have constructs such as private classes in Java to prevent unrestricted access to sensitive variables. Considering the example in Figure 2.13, it can be seen that an addition operation modifies both parts of the variable. Our model also considers the possibility of such malicious functions and tries to protect sensitive variables from unintended modifications.

2.5 Conclusion

The IFC models aim to address the granularity at the variables level. In this way, these models are suitable candidates for a holistic approach towards web browser security. Indeed, the information flow control allows to guarantee that the variables can only be accessed by authorized functions, and that no unauthorized modifications are done to the variables. In this regard, it allows to protect web browser against illegitimate scripts, and thus significantly reducing the effects of XSS and the risk of CSRF.

In this thesis, we propose a new IFC-based approach designed for the web browsers and JavaScript. Our approach falls into the category of dynamics approaches because we believe that the JavaScript's dynamic nature causes complexities to static analysis, in particular with respect to the `eval` function.

Our approach is similar to the faceted approach in the context of having multiple copies for each variable. However, the similarities stop there. In our approach we keep one true copy and one junk copy for a given variable. Thus, the growth in the number of copies held does not exceed two.

In our approach, we do not evaluate multiple branches exhaustively preferring the cautionary least required execution of the tainting models than the exploratory executions of the faceted approach and SME. SME and faceted approach perform such an analysis to enforce non-interference security guarantees. From this point of view, our model suffers by not providing the same guarantees as the faceted approach and SME.

Chapter 3

Address Split Design

Our approach focuses on typical web browser which generally consists of the JavaScript engine and a rendering engine. The JavaScript engine is composed of the interpreter and the interface to the rendering engine. We believe that an effective IFC could be achieved by directly modifying the interpreter. This is consistent with related research [Aus13, GDNP12].

Our model is termed the Address split design (ASD). The core of the model is the address-split. An address-split occurs when a variable is classified as a secret. This involves creating a dummy variable address space referring to the secret variable. Hence, for every secret variable, this process of splitting the variable is instantiated when the scope at which the variable exists is entered.

We represent a secret variable in the form of $\lceil \text{public}^p \parallel \text{private}^s \rfloor$. The split variable consists of a public value and a private value which are stored in different memory locations. A general outline of the model and its working is described in Section 3.1. This is followed by a detailed description of ASD with formal semantics in Section 3.2. This is followed by some IFC examples in Section 3.3. We provide a comparison of the working of our model with other models in Section 3.4.

3.1 General working of Address Split Design

Figure 3.1 represents the general working of the model. The various elements with a white background represent the existing mechanisms in JavaScript. The elements represented in blue are the new components introduced by our approach. In the figure, it can be seen that there are two different memory addresses for the variables `A`, `B` and `C`. This represents that these variables contain secret values. Our model would influence the JavaScript engine (JS Engine) to choose between the dummy public value and the private value when a lookup for the variable

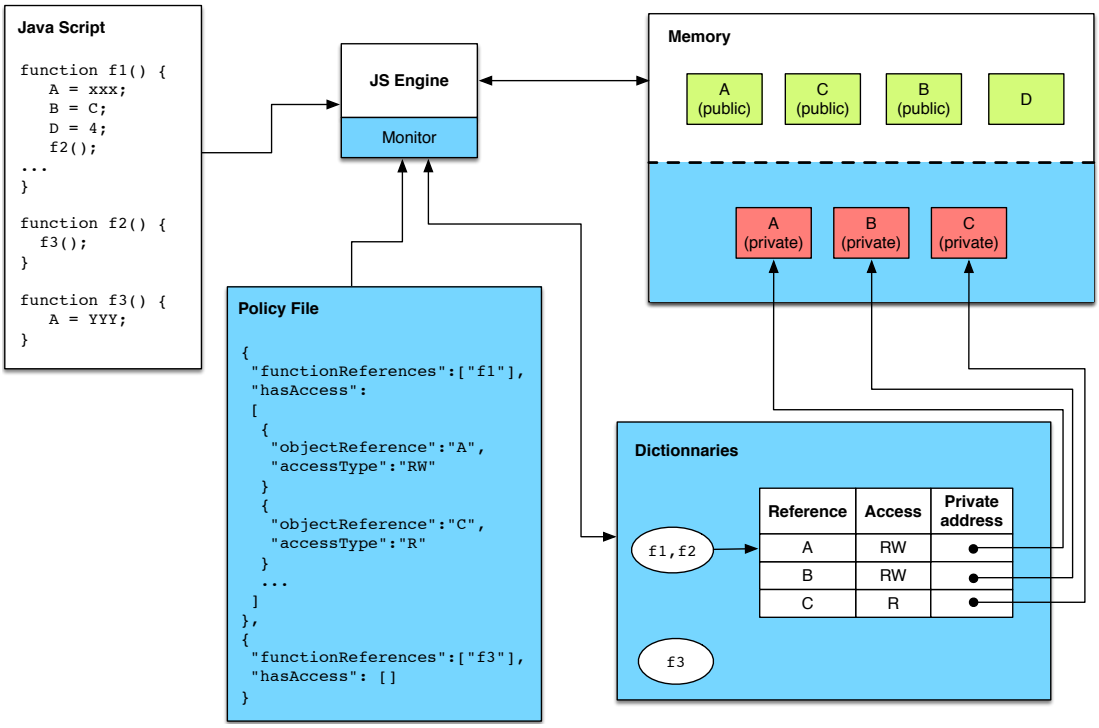


Figure 3.1: The address split design

is needed. To implement our model, we use a monitor which hooks on to the JavaScript engine. The monitor overloads the symbol table and determines which value would be used when a variable is used by the JavaScript function at runtime. The public values will be the default values of the variables and the memory handling for the private values is implemented by our approach. The various initial secret variables are given in the policies to the webpage. The monitor keeps track of the functions' privilege to access the variable using data-structures called dictionaries which it uses to overload the symbol table.

3.1.1 Policy specification

The policies specified for the webpage are passed to the monitor which is a component added to the JS Engine. The monitor is also responsible for keeping track of the information's level throughout the program's execution and subsequently propagating the privileges based on the execution steps. Hence, it performs both classification and declassification. Classification is when the variable is split and a secret value flows into it while declassification is when a previously unauthorized function is allowed to access the secret value.

The policy specification for the JavaScript program is passed concurrently with the program itself. However, due to the nature of JavaScript, it must be noted that the entire code segment

would be visible to any script accessing it. It is hence unsafe to pass any policies in the form of 'pragma marks' or comments or any other form that could have a textual representation in the code section. In our experience, the safest place to pass any such value would be the HTTP headers.

The policies are loaded before any JavaScript on the page is executed. This is an important consideration since the policies are not bound to variable references but rather to the objects that lie under these references. This means that as the variable reference evolves to reference different objects, the various rights given to the references will have changed. This is how the information flow evolves the various access rights.

A simple Backus–Naur form (BNF) grammar showing the syntax of our policy specification is shown in Figure 3.2. The list of policies specified are a collection function policies. Each function policy specifies the functions' references, the optional checksum of the function, the option parameter of providing the entire function string, and a list of privileges that are provided to the function initially. Every privilege attributed to a function will contain the variable-name as well as the type of privilege the function can obtain. The variable name is represented as a combination of the variable container as well as the name of the variable reference. For example, a global variable `a`, would be represented by the container `global` and variable reference `a`. A valid name is simply the acceptable variable name as restrained by the JavaScript language. A string is any arbitary string which is a collection of letters, digits and numbers.

3.1.2 Privileges

In JavaScript, variables are essentially pointers to objects. Given this consideration, we take a unique stand that it is vital to protect the variable references along with the actual variable object. Hence, we use privileges to assist the IFC mechanism in handling the protection of both the variable reference and the object pointed to by the variable. In our approach, we consider two privileges, read access and write access. The read access is used to protect the object. A function can gain or loose read access based on the object that is pointed to by the variable.

For example, let us consider that function `f1()` has access to variable `a` and `f2()` have access to both variable `a` and `b`. If function `f2()` includes the instruction `a = b;` then the execution of function `f2` would propagate the value of variable `b` into `a`. In this case, the function `f1` would loose access to `a` since it is not privileged to read the information from variable `b`.

The write access implies that the function can change the variable's value. In our model, a function cannot loose write access to a variable reference regardless of the information flow. However the read access evolves with the information flow. If secret information that a func-

```
<policy-list>  ::= <empty> | <function-policy> |
                   <function-policy> <policy-list>
<function-policy> ::= <function-references-list>  <opt-function-checksum>
                      <opt-function-string> <privilege-list>
<function-references-list> ::= <function-name> |
                               <function-name> <function-references>
<opt-function-checksum> ::= <empty> | <string>
<opt-function-string> ::= <empty> | <string>
<privilege-list> ::= <privilege> | <privilege> <privilege-list>
<privilege> ::= <variable-name> <privilege-type>
<variable-name> ::= <variable-container> "." <variable-reference-name>
<variable-container> ::= <valid-name> | <valid-name> "." <valid-name>
<variable-reference-name> ::= <valid-name>
<function-name> ::= <valid-name>
<privilege-type> ::= "R" | "RW" | "W" | ""
<valid-name> ::= <letter> | <letter> <name-string>
<string> ::= <letter> | <number> | <symbols> | <letter> <string> |
             <number> <string> | <symbols> <string>
<name-string> ::= <number> | <name-symbols> |
                  <letter> <name-string> | <number> <name-string> |
                  <name-symbols> <name-string>
<letter> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" |
               "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" |
               "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" |
               "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" |
               "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" |
               "t" | "u" | "v" | "w" | "x" | "y" | "z"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<symbol> ::= "|" | " " | "-" | "!" | "#" | "$" | "%" | "&" | "(" | ")" |
               "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" | "=" |
               ">" | "?" | "@" | "[" | "\" | "]" | "^" | "_" | "`" | "{" |
               "|" | "}" | "~" | "\n" | "\t"
<name-symbols> ::= "_"
```

Figure 3.2: Policy BNF grammar

tion cannot access flows into the variable, the function will loose read access to that variable. The monitor keeps track of the privileges of various functions using data structures called dictionaries.

3.1.3 Dictionaries

Dictionaries provide the actual address spaces of the various secret values. When a dictionary is loaded, it overloads the symbol table and provides the actual destination for the variable. It hence resembles the symbol table in its data representation. However, being a custom component, it is extensible and also contains additional meta-data that is useful to our IFC model. The dictionary is a component that is specific to a function.

Each dictionary contains the variable reference as the key and a memory location as a value. The dictionaries provide data on the variable based on the various privileges. Based on this information, the monitor will return the private or the public part of the variable.

In our model, the monitor will refer to the dictionaries associated to the function for each time a variable is read from or written to during the execution of the function. The monitor infers the address to be used based on the privilege of the function and returns the private address or the public address accordingly. Such an inference is repeated for everytime the variable is read from or written to since the dictionary is an evolving data-structure that depends on the information flow.

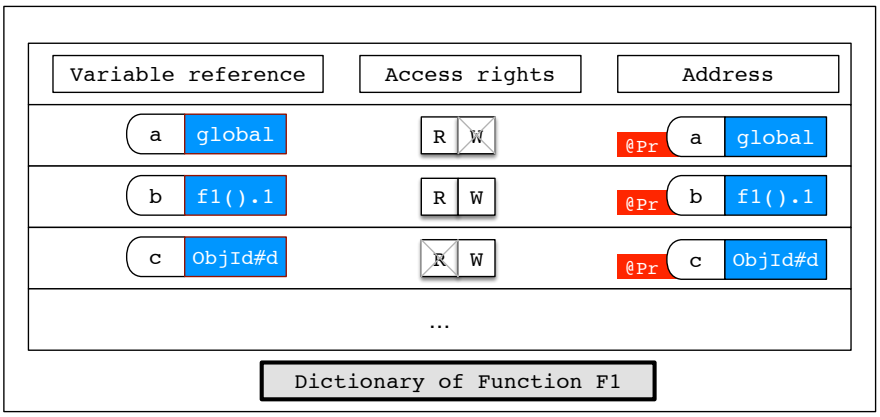


Figure 3.3: Dictionary example

An example of a simple dictionary is shown in Figure 3.3. In this example, it can be seen that the dictionary maintains data about a variable name, the privileges to that variable and the address of the variable itself. The variable name consists of the base object in which the name exists in as well as the name of the variable. In this example, `global` represents the global scope object. Similarly `f1().1` refers to a function scoped object. Here, `f1()` signifies that the variable is a local variable belonging to the function `f1` and `1` is an identifier to differentiate the various executions of the function for example in case of recursion. In case of an object property such as `d.c`, the representation will be scoped in `ObjId#d` where `ObjId` is the unique identifier of the allocated object `d` in the heap.

3.1.4 Function privileges

Functions are the components which can receive the privilege to access the secret variables. We propose that functions be segregated into varying types based on their required functionalities so as to simplify the policy specification.

The biggest difference in our model to that of the other approaches is in the way we provide privileges. In other models, functions which perform public output are added to a blacklist and if secret information flows into these functions, it will trigger the unauthorized information handler (see Section 2.3). There is no policy defining the privileges of other functions. These function in these models can access the secret values. In our model, functions are directly associated to their respective lattice levels. Assigning privileges directly to functions allows the policy specification to cater to the needs of individual functions.

In the ideal case, all functions are given necessary privileges in the policy specification. Since the privileges in the policy specification are reflected in the dictionaries, the functions are associated to these dictionaries. If a function is loaded, the associated dictionaries are also loaded and hence the various secrets are accessible. For the sake of clarity, we term all functions with defined privileges as Self-Sufficient Function (SSF). SSF are directly mentioned in the policy specification. In the Figure 3.1, `f1`, `f2` and `f3` are SSF.

The SSF provide the fundamental components where the information flow control is defined. However, in a typical JavaScript scenario, functions are not always independent. The use of libraries to do various actions is very common. For example, the function may use the jQuery[1] library to perform various actions such as post requests or regular expression operations. It might also use native functions such as XMLHttpRequests, Console.log, alert, etc. It would not be appropriate to give explicit permissions to these libraries under normal circumstances. This is because the libraries themselves can be used by both authorized as well as unauthorized functions in the same webpage. It is very important to ensure that the libraries hence must have no privileges to access secret variables when called by unauthorized functions. However, to ensure functionality, the libraries must have privileges corresponding to the authorized function that call them. These functions are hence classified as Utility Functions (UF).

An UF is a constant declassification mechanism that is part of our model. It is considered as a modular piece of code that has been made into a function for easier maintenance and reuse. Considering this, the UF does not have any privileges of its own. Every instance of an UF adheres to the privileges of its caller. This behavior is transitive over the function call. When laid in terms of the PER model [SS98], the UF perform the declassification 'when' they are called by a SSF. By default, a function is considered to be an UF if it does not feature in the policy specification. These functions bring a lot of functionality to life since they help perform a huge majority of actions as long as the caller SSF has the privileges. This is explained in the Figure 3.4. This Figure 3.4(a) shows that in case of Self Sufficient Functions `SSF1`, `SSF2` and `SSF3`, each function uses its own dictionary during execution. However in case of

[1] `https://jquery.com/`

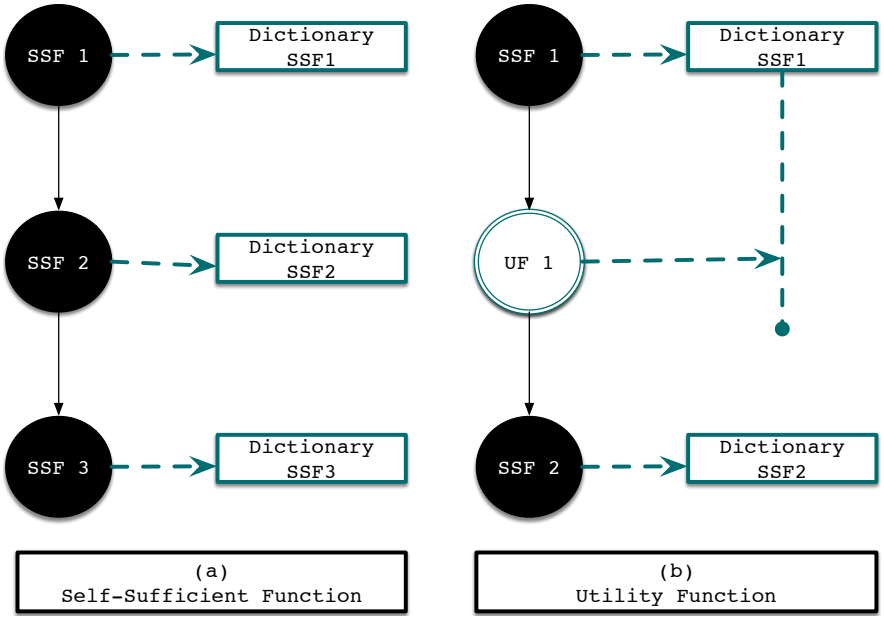


Figure 3.4: Working of utility functions

Utility Functions, the function `UF1` uses the dictionary of its calling function `SSF1` as shown in Figure 3.4(b).

It must be noted that an UF is not envisioned with an intention to perform tasks such as public output, but is meant to apply to cases such as performing a square-root, calculating the interest given a number and other use-cases which are provided to do common functionality to the program.

The list of UF can be restricted in the policy to include only native functions and functions tied to a specific `URL`. In this case, any other function would by default be considered a SSF with no privileges. Since the function itself is a normal JavaScript variable, it can also be split. In case of a split function, the public part of the variable would become a SSF with no privileges. However, the private part of the variable would either become a SSF if there is a policy associated to it or it would become an UF. This is done to protect the reference of the variable. If a native function is split, only the private part of the variable becomes an UF. For example, if the `console.log` which performs a print operation is split, the functions with access to `console.log` in their policy can print secret variables to which they have read access else no secret variable can be printed.

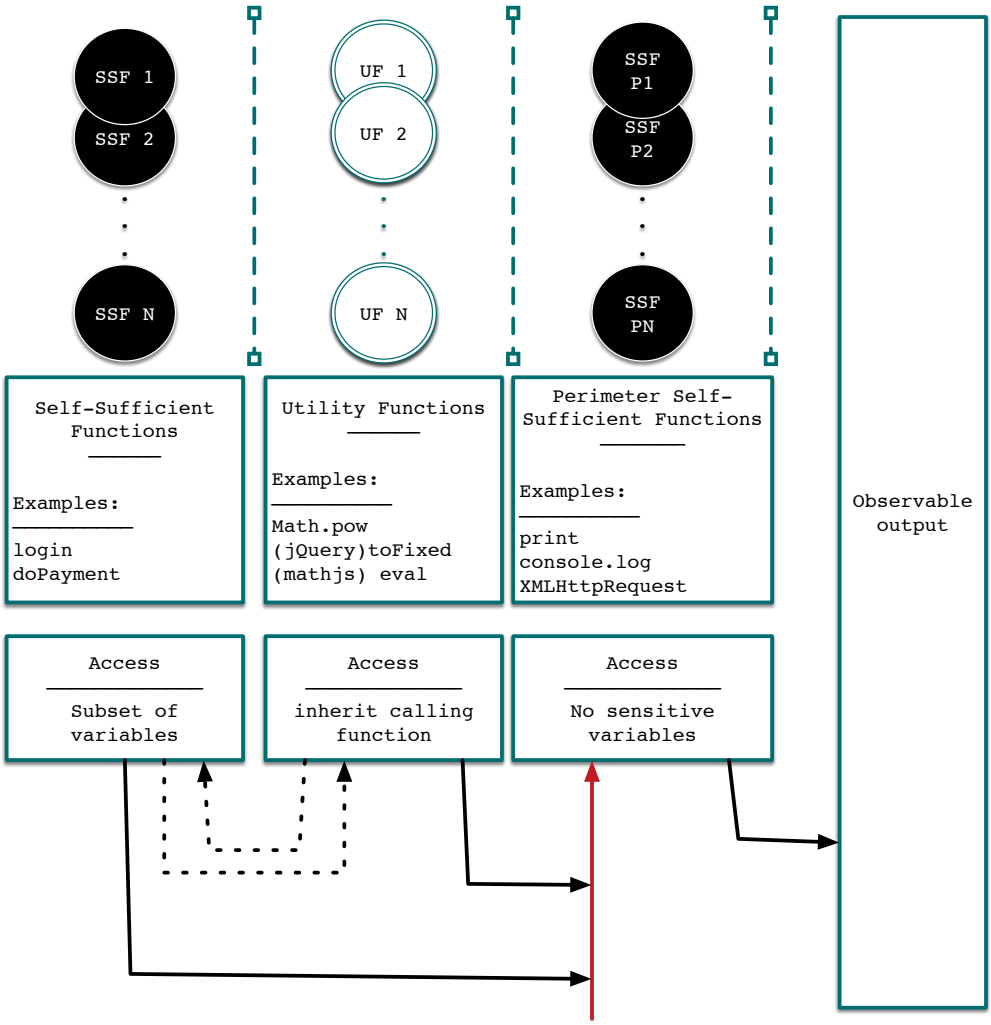


Figure 3.5: Defining the Perimeter using Self-Sufficient Functions

It must be noted that our model is able to perform similar operations as the label based approaches by defining perimeter functions which are responsible for public output. These perimeter functions are simply SSF without any privileges. The Figure 3.5 shows how the perimeter functions are able to prevent any sensitive variables from appearing in the public output.

Regardless of the privileges of the functions that call them, the perimeter functions would not have any privileges to print the secret because they contain empty dictionaries. This is hence synonymous with public output functions in traditional approaches. For example, functions such as the native console.log and alert could be considered public output functions with no privileges. In this case, no secret data would be accidentally printed or sent as an alert dialog

during execution. An even more fine case could be achieved by splitting XMLHttpRequest. In this case, the public part of XMLHttpRequest would become a perimeter function (a SSF with no privileges) and the private part would become a utility function. This would allow authorized functions to send secrets using XMLHttpRequest while XMLHttpRequest would act as a perimeter function for unauthorized functions.

It must also be noted that since all functions need access rights to access the variables, even the intermediary functions cannot change or corrupt the values of the variables if they do not have the suitable access rights. Traditional models have such restrictions only on the public output functions and all other functions can modify all the available parts of the variables. This would imply that for a language such as JavaScript where scripts and functions can be arbitrarily added at runtime, the traditional approaches cannot prevent corruption of data held in the variable. ASD can however protect crutial secrets from being corrupted by unauthorized functions even if these functions are not output/perimeter functions.

Let us consider the example illustrated in the Figure 3.6. In the figure, `Pr` represents the private value of a secret variable and is indicated by the color red and a public value is indicated by the color green. There is a function h(), and a global secret `x`. Since, `x` is a global object, changing the variable reference will have an impact across the rest of the program. The representation `read(x)->true` implies that the monitor has inferred from the dictionary of `h()` that the function has read access to the variable `x`. `write(x)->true` implies that the monitor has inferred from the dictionary of `h()` that the function has write access to the variable `x`.

Let us consider that $x = \lceil 2|1 \rfloor$. There are four cases in this figure namely `case (RW)`, `case (R)`, `case (W)` and `case (∅)`. In the `case (RW)`, the function has both read as well as write access to the secret variable. This implies that the function `h()` will be able to read the private part of the variable and change the object pointed to at the private part. In our approach, a function must always try to use the private part of the variable whenever possible. `case (W)` exemplifies this characteristic of our approach. In this case, the value is a public value and hence there is no restriction on using it to write into the private part of the secret variable. It can also be observed that in the `case (R)`, the result contains a secret value. However, `x` has already been split and `h()` cannot change `x`'s private value. Hence, in `case (R)` the result is discarded. This is represented by crossing out the operation `split x`. This mechanism adds a lot of malleability to our model since it induces a lot of fine-tuned control when handling information flows.

3.1.5 Dependency tracker

In JavaScript, when a variable is assigned a value, it implies that the variable is a pointer reference to that particular value. Hence, when a new secret value is assigned to the variable,

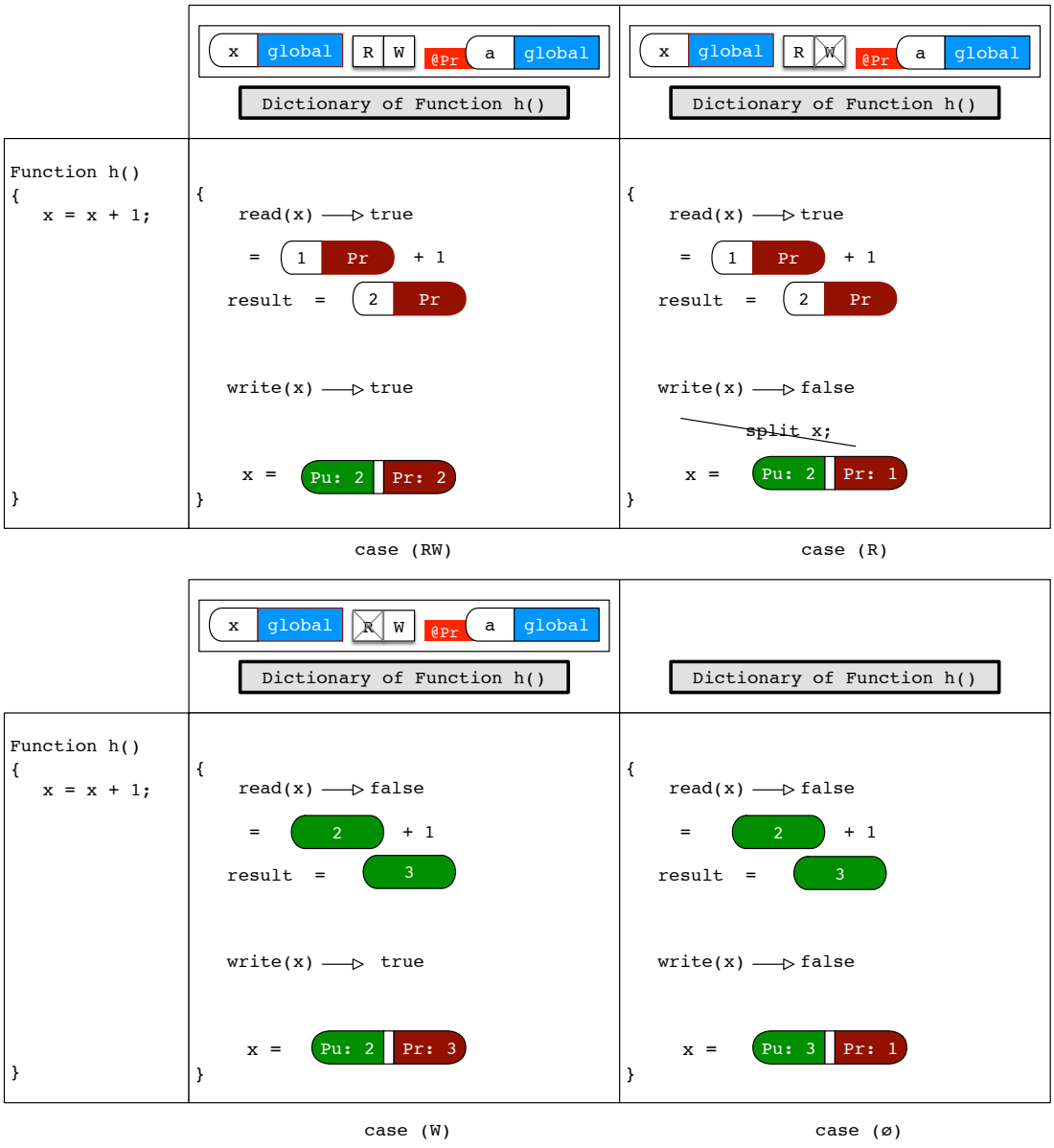


Figure 3.6: The various access rights

the address to the variable needs to be updated in the respective dictionaries as well. It is therefore important to keep track of various read/write operations to variables.

To keep track of the information flow of various read/write operations, our model uses a Dependency Tracker (`DT`). It keeps track of two pieces of information: the secret variable read from and the type of information flows associated to these variables (direct or indirect).

The `DT` keeps track of the various secret variables that are being used in the execution of a statement. When a private value is being read, the corresponding variable is added to the dependency tracker. This dependency tracker is then used to re-evaluate the affected dictionaries when a write is performed. The dependency tracker is attached to the function and is re-evaluated at every statement. A read operation is performed when the value of the variable is used in a statement including computation of expressions, as a function call parameter or as a return parameter. A write operation is performed due to an assignment operation.

Let us consider the case (RW) in the Figure 3.6. In this case, the DT is empty when the function starts executing. When the private part of the variable `x` is read, the variable `x` is added to the `DT`. In this statement, the resulting value is written back to variable `x`. Since the `DT` is non-empty, the value can only be written to the private part of `x`.

This concludes the general introduction to our model. In the following section, we give a more detailed description of our approach along with the formalisms related to this model.

3.2 ASD description and semantics

In this section, we describe the formalism along with the concepts involved in our approach. The semantics used by this paper is inspired by the while language [Ald06, CHM07]. While language is relatively simple when compared to a real language such as JavaScript and its usage is beneficial to explain the working of the model.

3.2.1 Metavariables and environment

We first define the list of metavariables and the environment in the declaration 1. These list of metavariables define the variables, values, mapping between variables and values, functions, and constants used in the rest of the model. Constants include λ which represents the null value, and boolean constants (`true` and `false`). There are special variables namely the current function (cf) and the variable that is returned (*returnVar*) which are used by the model for propagating some information needed to access the secret variables based on the execution. The cf is maintained by the environment to keep track of the procedure that is currently being executed and the *returnVar* is used to maintained the returned value at the end of a function call. The `getVal` provides a mapping between the variables and values respectively.

Declaration 1: [METAVARIABLES AND ENVIRONMENT]

Let, Variables	$x, y \in Var$
Values	$v \in Value$
Constants	$c \in Const$
Null	$\lambda \subset Const$
Boolean constants	$\texttt{true} \subset Const$
	$\texttt{false} \subset Const$
Functions	$f \in F$
Current Function	$cf \in F$
Return Value	$returnVar \in Var$
getVal	$Var \rightarrow Value$
State	$s \in \{s^p, s^s\}$
Privilege	$Priv \in \{read, write, read + write\}$
getVar	$(Var \times s) \rightarrow Var$
isSplit	$Var \rightarrow boolean$
Dictionary	$\mathbb{D} : (F \times Var \times Priv) \rightarrow boolean$
Flow Type	$FL \in \{FL_e, FL_i\}$
Dependency Tracker	$\texttt{dt}_\texttt{F} \subset Var \times FL$
Element	$E \in \{\texttt{dt}_\texttt{F}, \lambda\}$
Dependency Tracker Stack	$\texttt{DTS} : (Stk, push, pop, nthElem, top)$
	$Stk : E^*$
	$push : (Stk \times E) \rightarrow Stk$
	$pop : Stk \rightarrow (Stk \times E)$
	$nthElem : (Stk \times N) \rightarrow E$
	$top : Stk \rightarrow E$
	$prevDTS : Stk \rightarrow E$
Environment	$\eta \in Env$
	$\eta : \left\{ \begin{array}{c} getVal \\ getVar \\ isSplit \\ \texttt{DTS} \\ \mathbb{D} \end{array} \right\}$
	$where, \eta(x) \mapsto \eta(getVal(x))$

We also define some metavariables that are specially created for our model. The states, s, are represented as public, s^p, or secret, s^s, for every split variable. To access the variables, the privilege given to a function can be between `read`, `write` or both. To maintain such privileges, our model used the dictionary data-structure, which is represented by $\mathbb{D}$. The dictionary contains a list of variables, and the privilege to access the variable for a given function. The environment also maintains a mapping, `isSplit`, to keep track of whether or not the current variables have been split. The Dependency Tracker (`dt`) is used to keep track of the secret variables whose private values are influencing the current statement of the running function. When there are multiple functions called by one another, the stack of dependency trackers `DTS` is used. A new `dt` is pushed every time a function is called and is popped when the function returns. A Flow Type (FL) is used to keep track of whether the current information flow is an explicit flow or an implicit flow.

The environment itself is represented by η. It contains the mappings from variable to value (`getVal`), a mapping to get the variable based on the state (`getVar`), a mapping to check if the current variable has been split (`isSplit`), a mapping to the dependency tracker (`DTS`) and a mapping maintained by the dictionary $\mathbb{D}$. The shorthand $\eta[x]$ is used instead of $\eta[\texttt{getVal}(x)]$.

3.2.2 Syntax

We define the syntax used by the semantic rules in the declaration 2. In these rules, the general syntax of the while language has been modified to better define our model.

The arithmetic expressions are represented by a while the conditional operations are represented by b. The special cases of the arithmetic operation containing a variable x is represented by a^x and a conditional operation containing the variable x is represented by b^x. Hence, a^x would imply that the computation of arithmetic expression requires the value of x and the same implies for b^x.

The various statements used in the semantics are represented by S. In these, the statements of $split$, $callFn$, $loadPROC$ and $removeFLe$ are custom internal events of ASD. These custom events are not supposed to be present in the original program written by the developer but are added by the interpreter. These events are triggered by the monitor and are used to influence the model specific environment. The $split$ event triggers a variable split if the variable has not been split before. This event is triggered both when interpreting the policy as well as when a variable is being dynamically split due to information flows. The $removeFLe$ is an event that is used when the DT needs to purge all explicit flow dependencies. The $callFn$ and $loadPROC$ are function specific custom events. A function in our semantics is a set of statements followed by a return statement. For the sake of simplicity, we do not consider local variables and other features in a function in our rules. We however define function stacks which keep track of the

Declaration 2: `[SYNTAX]`

`Arithmetic`	$a \in A_{exp}$
	$a := c \mid x \mid a_1 \bigstar a_2$
	where, $\bigstar :=$ `binary operation`
`Arithmetic with variable x`	$a^x \in A_{exp}$
	$a^x := x \mid a_1^x \bigstar a_2 \mid a_1 \bigstar a_2^x$
`Conditional operations`	$b \in Conditional$
	$b := true \mid false \mid a_1 * a_2 \mid (b_1 \& b_2) \mid (b_1 \mid b_2)$
	where, $* := == \mid > \mid < \mid >= \mid <=$
`Conditional operations with x`	$b^x \in Conditional$
	$b^x := a_1^x * a_2 \mid a_1 * a_2^x \mid (b_1 \& b_2^x) \mid (b_1^x \& b_2) \mid (b_1^x \mid b_2) \mid (b_1 \mid b_2^x)$
`Statements`	$S \in Stm$
	$S := x = a \mid Skip \mid S_1; S_2$
	$\mid$ if b then S_1 else $S_2 \mid$ while b do S
	$\mid split(x) \mid$ `return` x
	$\mid$ `call` f `with`$(y_1, ..., y_n) \mid loadPROC(f)$
	$\mid x = callFn(f, \{y_1, .., y_n\}) \mid removeFLe$
`Procedures`	$PROC : F \rightarrow S$;`return x;`
`Function Stack`	`FunStk` $: (FStk, pushF, popF, nthElemF, topF)$
	$FStk : F^*$
	$pushF : (FStk \times F) \rightarrow FStk$
	$popF : FStk \rightarrow (FStk \times F)$
	$nthElemF : (FStk \times N) \rightarrow F$
	$topF : FStk \rightarrow F$
	$prevF : FStk \rightarrow F$

current function as well as the list of function calls to the current function.

3.2.3 Splitting model

The semantic using these syntax pertaining to our model are defined in the Rules 1 to 25. The first and simplest action that forms the core of the model is to split the variable into two address spaces. We first define how a variable is split in the Rule 1. The variable is split either at the stage of policy specification or when it is upgraded due to information flows. In both cases, the split event is triggered. In this rule, two other variables, $getVar(x, s^p)$ and $getVar(x, s^s)$, are created. They are used based on whether the public or private values need to be used respectively.

Rule 1: [SPLIT]

$$\frac{\eta(isSplit(x)) = \texttt{false}}{(\eta, split(x)) \;\mapsto\; \eta[getVar(x,s^s) \mapsto \lambda, getVar(x,s^p) \mapsto getVal(x), isSplit(x) \mapsto \texttt{true}]}$$

The public variable is used under when there is no privilege to access the secret. The public variable is hence associated to the previous value of x before the split event was triggered. We should notice that the variable x itself is split but the resolved public variable $getVar(x, s^p)$ and the resolved private variable $getVar(x, s^s)$ are not split by default.

Rule 2: [RUNTIME: RESOLVE VARIABLE VALUES]

$$\frac{\eta(x) = v \;\; \eta(isSplit(x)) = \texttt{false}}{\eta \vdash x \downarrow v}$$

It must be noted that the value of a split variable needs to have been substituted with either the public or the private variable before any of the other operations such as assignment can be made. This is defined in the Rule 2. If we consider an unsplit variable, the Rule 2 applies directly and the value is resolved. If the variable is split however, such an application is impossible, the variable would fail to be resolved into a value. For all the rules pertaining to evaluation of arithmetic or boolean expressions, a variable needs to be resolved into a value before such an operation can be carried forward. Once all the split variable have been substituted, the expression can be evaluated using standard semantics for arithmetic and boolean expressions. If the variable has already been split, the variable would need to be substituted. The semantics for such substitutions are explained in the forthcoming Section 3.2.4.

The standard semantics for a sequence of statements as well as a sequence containing the skip statement are defined in Rule 3 and Rule 4.

Rule 3: [RUNTIME: STATEMENT SEQUENCE]

$$\frac{(\eta, S_1) \mapsto (\eta', S_1')}{(\eta, S_1; S_2) \mapsto (\eta', S_1'; S_2)}$$

Rule 4: [RUNTIME: SKIP SEQUENCE]

$$\frac{}{(\eta, Skip; S_2) \mapsto (\eta, S_2)}$$

3.2.4 Assignment and substitution

The Rules 5-11 describe the various actions to be taken when an assignment operation is carried out. When the variable is not split, and the DT is empty, the value flows into the variable without any other changes to the environment as shown in the Rule 5.

Rule 5: [ASSIGNMENT: UNSPLIT VARIABLE]

$$\frac{\eta(isSplit(x)) = \texttt{false} \;\; \eta(top(\texttt{DTS})) = \lambda \vdash a \downarrow v}{(\eta, x = a) \;\mapsto\; \eta\,[x \mapsto v]}$$

The Rules 6 to 9 state the actions that need to be taken when there is an information flow into a split variable. In the Rule 6, the dictionaries that have permission to read the current value will have to satisfy the DT to retain their access rights.

Rule 6: [ASSIGNMENT: SPLIT VARIABLE]

$$\frac{\eta[isSplit(x) \mapsto \texttt{true}, top(\texttt{DTS}) \mapsto \lambda, cf \mapsto f, \mathbb{D}(f, x, \texttt{write}) \mapsto \texttt{true}] \vdash a \downarrow v}{(\eta, x = a) \;\mapsto\; \eta\,[getVar(x, s^s) \mapsto v]}$$

Rule 7 states that if the value does not contain secrets and the DT is empty, the value of the public part of the variable is updated. Rule 8 shows that no real update is performed to any part of the variable if a secret value tries to flow into the public part of the variable. This

is consistent with the example shown in Figure 3.6. For all these rules where the DT is non-empty, the event *removeFLe* is triggered to remove all the explicit dependencies at the end of the statement.

Rule 7: [ASSIGNMENT: SPLIT VARIABLE 2]

$$\frac{\eta[isSplit(x) \mapsto \texttt{true}, top(\texttt{DTS}) \mapsto \lambda, cf \mapsto f, \mathbb{D}(f,x,\texttt{write}) \mapsto \texttt{false}] \vdash a \downarrow v}{(\eta, x = a) \;\mapsto\; \eta\,[getVar(x, s^p) \mapsto v]}$$

Rule 8: [ASSIGNMENT: SPLIT VARIABLE 3]

$$\frac{\eta[isSplit(x) \mapsto \texttt{true}, top(\texttt{DTS}) \neq \lambda, cf \mapsto f, \mathbb{D}(f,x,\texttt{write}) \mapsto \texttt{false}] \vdash a \downarrow v}{(\eta, x = a) \;\mapsto\; \eta \vdash removeFLe(cf)}$$

In the Rule 9, the DT is non-empty and hence the rights of various functions change at the end of the information flow. In this case, only the functions having the rights to all the elements in the DT would continue to have access to the variable.

There are a lot of variables that would become containers for secret values over the course of the information flow. These variables are hence upgraded at runtime. Any upgrade involves the splitting of the variable, adding the variable to the appropriate dictionaries and then assigning the various access control rights for the variable before running the statement. It can be noticed that all the dictionaries get the write access to this dynamically split variable. This is because, the variable was initially public and upgraded only for the information it holds. The container itself is hence not protected though the data inside is secret. Hence, while read access is withheld from other functions, they are still permitted to write into the variable. This is the information recited by the rules 10.

The Rule 11 is a specific rule which defines the event of removing the various explicit flows from the DT at the end of the assignment. This rule is only triggered if the DT is non-empty.

Rule 11: [RUNTIME: REMOVE EXPLICIT FLOW]

$$\frac{\eta(top(\texttt{DTS})) \neq \lambda \quad \eta(cf) = f}{(\eta, removeFLe(f)) \;\mapsto\; \eta\,[\forall (x, FL = e) \in top(\texttt{DTS})\{top(\texttt{DTS}) \mapsto top(\texttt{DTS}) - \{(x, FL = e)\}\}]}$$

The Rules 12 to 15 provide the rules for substitution of the public or private variables for a given split variables. In the Rules 12 and 13, the simple case of reading a variable x that has been split is shown. In this case, there is substitution of the variable x with its secret variable

Rule 9: [ASSIGNMENT: SPLIT VARIABLE 4]

$$\frac{\eta[isSplit(x) \mapsto \texttt{true}, top(\texttt{DTS}) \neq \lambda, cf \mapsto f, \mathbb{D}(f,x,\texttt{write}) \mapsto \texttt{true}] \vdash a \downarrow v;}{(\eta, x = a) \;\mapsto\; \eta \left[\begin{array}{l} getVar(x,s^s) \mapsto v, \\ (\forall f_1 \in F_{ss}) \left\{ \begin{array}{l} if\ \mathbb{D}(f_1,x,\texttt{read})\ then \\ \quad \left\{ \begin{array}{r} if\ (\forall (y,FL) \in top(\texttt{DTS}))\{\mathbb{D}(f,y,\texttt{read})\} \\ then\ \mathbb{D}(f,x,\texttt{read}) \mapsto \texttt{true} \\ else\ \mathbb{D}(f,x,\texttt{read}) \mapsto \texttt{false} \end{array} \right\} \end{array} \right\} \end{array} \right] \vdash removeFLe(cf)}$$

Rule 10: [ASSIGNMENT: VARIABLE UPGRADE]

$$\frac{\eta[isSplit(x) \mapsto \texttt{false}, top(\texttt{DTS}) \neq \lambda] \vdash a \downarrow v;}{(\eta, x = a) \mapsto \eta\,[(\forall f_1 \in F_{ss})\,\{\mathbb{D}(f_1,x,\texttt{write}) \mapsto \texttt{true},\}]\,; split(x); x = a;}$$

obtained using $getVar(x,s^s)$ and the public value using $getVar(x,s^p)$. Similarly, substitutions can also be done to the conditional statements b as shown in the Rule 14 and Rule 15.

Rule 12: [RUNTIME: FUNCTION READ ACCESS]

$$\frac{\eta(isSplit(x)) = \texttt{true}\;\; \eta(cf) = f\;\; \eta(\mathbb{D}(f,x,\texttt{read})) = \texttt{true}}{(\eta, a^x) \;\mapsto\; \eta[top(\texttt{DTS}) \mapsto (top(\texttt{DTS}) \bigcup \{(x,FL_e)\})] \vdash a^{getVar(x,s^S)}}$$

Rule 13: [RUNTIME: FUNCTION READ ACCESS DENIED]

$$\frac{\eta(isSplit(x)) = \texttt{true}\;\; \eta(cf) = f\;\; \eta(\mathbb{D}(f,x,\texttt{read})) = \texttt{false}}{(\eta, a^x) \;\mapsto\; (\eta \vdash a^{getVar(x,s^P)})}$$

Rule 14: [RUNTIME: FUNCTION READ ACCESS : CONDITIONAL]

$$\frac{\eta(isSplit(x)) = \texttt{true}\;\; \eta(cf) = f\;\; \eta(\mathbb{D}(f,x,\texttt{read})) = \texttt{true}}{(\eta, = b^x) \;\mapsto\; \eta[top(\texttt{DTS}) \mapsto (top(\texttt{DTS}) \bigcup \{x,FL_i\})] \vdash b^{getVar(x,s^S)}}$$

Rule 15: [RUNTIME: FUNCTION READ ACCESS DENIED : CONDITIONAL]

$$\frac{\eta(isSplit(x)) = \texttt{true} \;\; \eta(cf) = f \;\; \eta(\mathbb{D}(f, x, \texttt{read})) = \texttt{false}}{(\eta, b^x) \;\mapsto\; \eta \vdash b^{getVar(x, s^{\mathrm{P}})}}$$

3.2.5 Functions

The Rules 16 to 25 represent the various rules that are used to quantify the actions performed in the context of a function. In this formalism, we consider all the functions present to be self sufficient in nature. The Rules 16 to 19 are necessary to enumerate the rules for the actions to be followed at the end of a function's execution.

The Rule 16 is used to quantify the actions to be taken at the end of the execution of a function when there is no split variable and there are no dependencies in the DT. In this case, the function is popped out of the function stack and the environment's *returnVar* is initialized with the return value of the function. The DT that corresponds to the function is also popped out of the DTS.

Rule 16: [SELF-SUFFICIENT FUNCTION: EXECUTION ENDED]

$$\frac{\eta(isSplit(x)) = \texttt{false} \;\; \eta(top(\texttt{DTS})) = \lambda}{(\eta, \texttt{return}\;\; x) \mapsto \eta \left[\begin{array}{r} popF(\texttt{FunStk}), \texttt{returnVar} \mapsto getVal(x), \\ , pop(\texttt{DTS}), cf = topF(\texttt{FunStk}) \end{array} \right]}$$

In our semantics, a function must always return a value (thought this might also be λ). When a function ends the execution, the current function is changed in the environment and the DT corresponding to the function is removed. The value to be returned (*returnVar*) is set to the secret value if the calling function also has access to the elements in the DT. Else a value of λ is passed. These are shown in Rules 17 and 18.

Rule 17: [SELF-SUFFICIENT FUNCTION: EXECUTION ENDED : ANY VARIABLE]

$$\frac{\eta(top(\texttt{DTS})) \neq \lambda \;\; \exists (y, FL_1) \in top(\texttt{DTS}), \eta(\mathbb{D}(prevF(\texttt{FunStk}), y, \texttt{read})) = \texttt{false}}{(\eta, \texttt{return}\;\; x) \mapsto \eta \left[\begin{array}{r} popF(\texttt{FunStk}), \texttt{returnVar} \mapsto \lambda, \\ , pop(\texttt{DTS}), cf = topF(\texttt{FunStk}) \end{array} \right]}$$

Once this value is returned, it is used in the assignment operation of the calling function as shown in the Rule 19. In this case, the environment variable *returnVar* is checked for non null

Rule 18: [SELF-SUFFICIENT FUNCTION: EXECUTION ENDED : SECRET VALUE]

$$\frac{\eta(top(\texttt{DTS})) \neq \lambda \;\; \forall(y,FL_1) \in top(\texttt{DTS}), \eta(\mathbb{D}(prevF(\texttt{FunStk}),y,\texttt{read})) = \texttt{true}}{(\eta,\texttt{return}\;\; x) \mapsto \eta \left[\begin{array}{l} popF(\texttt{FunStk}),\texttt{returnVar} \mapsto v, \\ \quad \forall(y',FL) \in top(\texttt{DTS})\,\{prevDTS(top(\texttt{DTS}) \bigcup \{y',FL_e\}\}, \\ \qquad\qquad\qquad\qquad pop(\texttt{DTS}), cf = topF(\texttt{FunStk}) \end{array}\right]}$$

values. The assignment of the variable is made to the value held in *returnVar* and subsequently *returnVar* is reinitialized to $\varnothing$.

Rule 19: [SELF-SUFFICIENT FUNCTION: RETURN VALUE]

$$\frac{\eta[returnVar \neq \varnothing] \vdash x \downarrow \texttt{call}\; f\; \texttt{with}(y_1,...,y_n)}{(\eta, x = \texttt{call}\; f\; \texttt{with}(y_1,...,y_n)) \mapsto \eta[x \mapsto \texttt{returnVar}, \texttt{returnVar} \mapsto \varnothing]}$$

The Rules 20 to 25 describe the rules for calling a function. When there is a function call, we perform a transformation and execute the function before the actual call as shown in Rule 20. A *callFn* event is introduced into the execution. When evaluating *callFn*, the DT is checked before loading the various statements of the function. These rules are enumerated in Rules 21, 22, 23, and 24. If the function does not have read access to any of the variables that are in the the DT due to an *implicit* flow, there is no procedure that is loaded as shown in Rule 24. The return value is then used to evalutate the arithmetic expression when the function execution has ended as shown in the Rule 19. Such an evaluation ensures that the function calls are handled correctly.

Rule 20: [SELF-SUFFICIENT TRANSFORM]

$$\frac{\eta[returnVar \mapsto \varnothing] \vdash x \downarrow \texttt{call}\; f\; \texttt{with}(y_1,...,y_n)}{(\eta, x = \texttt{call}\; f\; \texttt{with}(y_1,...,y_n)) \mapsto \eta \vdash callFn(f,\{y_1,...,y_n\}); x = \texttt{call}\; f\; \texttt{with}(y_1,...,y_n);}$$

3.2.6 Example in While language

Let us consider an example shown in the Figure 3.7. In this case, let us consider the policy that the variable x is a secret and that the function f has read and write access to x.

When the policy is loaded (before the code itself is loaded), variable x is split according to the Rule 1. The $isSplit(x)$ is hence set to `true` and there is now a public and a private variable corresponding to x.

Rule 21: [SELF-SUFFICIENT BEFORE FN CALL]

$$\frac{\eta\left[\begin{array}{l} cf \mapsto f_1, \\ \quad (\forall(y, priv_1 = FL_i) \in top(\texttt{DTS}))\mathbb{D}(f, y, \texttt{read}) \mapsto \texttt{true} \\ \qquad\qquad (\forall(x) \in \{y_1, ..., y_n\})(!isSplit(x)) \end{array}\right]}{(\eta, callFn(f, \{y_1, ..., y_n\})) \;\mapsto\; \eta[\texttt{DTS} \mapsto push(\texttt{DTS}, f), cf \mapsto f] \vdash loadPROC(f);}$$

Rule 22: [SELF-SUFFICIENT BEFORE FN CALL]

$$\frac{\eta\left[\begin{array}{l} cf \mapsto f_1, \\ \quad (\forall(y, priv_1 = FL_i) \in top(\texttt{DTS}))\mathbb{D}(f, y, \texttt{read}) \mapsto \texttt{true} \\ \qquad (\exists(x) \in \{y_1, ..., y_n\})((isSplit(x))\ \mathbb{D}(f, x, \texttt{read}) \mapsto \texttt{false} \end{array}\right]}{(\eta, callFn(f, \{y_1, ..., y_n\})) \;\mapsto\; \eta \vdash callFn\left(f, \left\{\begin{array}{l} \forall y_0 \in \{y_1, , ..., y_n\} \\ \quad if\ x = y_0\ then\ getVar(x, s^p)\ else\ y_0 \end{array}\right\}\right)}$$

Rule 23: [SELF-SUFFICIENT BEFORE FN CALL]

$$\frac{\eta\left[\begin{array}{l} cf \mapsto f_1, \\ \quad (\forall(y, priv_1 = FL_i) \in top(\texttt{DTS}))\mathbb{D}(f, y, \texttt{read}) \mapsto \texttt{true} \\ \qquad (\exists(x) \in \{y_1, ..., y_n\})((isSplit(x))\ \mathbb{D}(f, x, \texttt{read}) \mapsto \texttt{true} \end{array}\right]}{(\eta, callFn(f, \{y_1, ..., y_n\})) \;\mapsto\; \eta \vdash callFn\left(f, \left\{\begin{array}{l} \forall y_0 \in \{y_1, , ..., y_n\} \\ \quad if\ x = y_0\ then\ getVar(x, s^s)\ else\ y_0 \end{array}\right\}\right)}$$

Rule 24: [SELF-SUFFICIENT BEFORE FN CALL]

$$\frac{\eta\left[\begin{array}{l} cf \mapsto f_1, \\ \quad (\exists(y, priv_1 = FL_i) \in top(\texttt{DTS}))\mathbb{D}(f, y, \texttt{read}) \mapsto \texttt{false} \end{array}\right]}{(\eta, callFn(f, \{y_1, ..., y_n\})) \;\mapsto\; \eta[\texttt{returnVar} \mapsto \lambda] \vdash skip();}$$

Rule 25: [SELF-SUFFICIENT CALLED]

$$\frac{\eta(PROC(f)) = S_1; S_2; ...; S_n; \texttt{return}\ x}{(\eta, loadPROC(f)) \mapsto \;\; \eta \vdash S_1; S_2; ...; S_n; \texttt{return}\ x;}$$

```
1 PROC: f  with () →
2   {
3     x = 2;
4     z = x;
5     return z;
6   };
7 x = 1;
8 y = call f with ();
```

Figure 3.7: Simple Information flow example

In line 7 the global scope does not have access to the variable x. Therefore, the Rule 7 applies to evaluate the assignment. Hence, the value is added to the public variable corresponding to variable x.

The function `f` is called in line 8. Before this function is called, the code is transformed to insert the *callFn* event using the Rule 20. Subsequently, due to Rule 21, the *loadPROC* event is triggered and the function is loaded by the Rule 25. The variable assignment of line 2 modifies the private variable corresponding to `x` as per the Rule 6. An unsplit variable `z` is assigned a value in line 4, and this assignment corresponds to the Rule 5. The function then returns the value corresponding to `z` which is stored in the environment variable *returnVar* as shown in Rule 16. This value is subsequently used in the assignment operation of the variable y in line 8 by first resolving the value using Rule 19 and then assigning it using the Rule 5.

There are however limitations to our proposed approach due to some of the design choices made for our model. For instance, we choose to pop the DT at the end of the execution of a SSF without transferring the implicit dependencies to the calling function. We deliberately do this to keep this approach more practical. By doing this, we are unable to provide any formal guarantees such as TINI because it is possible to leak information regarding the state of the variable. The main reason for this is because in case of an implicit flow, dynamic splitting of variables would not occur if that branch was not taken. Our over-approximation ensures that this would not affect any of the operations in the function's execution. However, by popping the DT at the end of the execution, we choose to end the over-approximation. In this case, at the end of the function's execution, the global variables may or may not be split based on the branch taken. Generally, our model would continue using either of the values and the rules of our model tend to be less intrusive. However, there is one intrusive rule, namely the Rule 24 which prevents the execution of a function based on the implicit flow. Therefore if a dynamically split global variable is used in an implicit flow and this prevents the execution of the function, the fact that the variable has been split can be determined. To explain this further we provide an example in Figure 3.8. In this example, the variable `V1` is a secret variable. Here

we consider that the functions `f4` and `f6` have read access to `V1`. The function `f5` does not have access to `V1`.

```
1  //V1 = secret true/ false
2  V10 = true;
3  V11 = true;
4  PROC: f4 with ()  →
5    {
6      if V1
7        then V10 = false
8        else skip;
9      if V10
10       then V11 = false
11       else skip;
12     return V11;
13   };
14 PROC: f5 with (x) →
15   {
16     y = x + 1;
17     call print with (y);
18     return 0;
19   };
20 PROC: f6 with () →
21   {
22     if V10
23       then call f5 with (V10)
24       else call f5 with (0);
25     return 0;
26   };
27 call f4 with ();
28 call f6 with ();
```

Figure 3.8: ASD Limitation Example

During the execution, the function `f4` is first executed. In the line 6, there is a conditional `if` statement. This triggers the Rule 14 and the variable `V1` is added to the DT. If `V1` = true^s, in line 7, the public variable `V10` becomes a split variable due to the variable upgrade Rule 10. In this case, the functions `f6` and `f5` would get access to the variable `V10` using the Rule 9. If `V1` = false^s, in line 10, the public variable `V11` becomes a split variable due to the variable upgrade Rule 10. This is because the DT still contains the variable `V1` due to the conditional statement on line 6. This is the over-approximation in our approach. However, at the end of the execution of the function `f4`, the DT is popped.

When the function `f6` is called, it would execute `f5` in case of `V1` = false^s, since `V10` is not split. However in the case that `V1` = true^s, the function `f5` would not be executed due to the

Rule 24 because it does not have read access to the split variable `V10`. Hence, our model could leak whether a variable has been split. However, the actual value contained in the variable is not leaked.

It must be noted that had the DT not been popped, the over-approximation would remain and the function `f5` would never be called, thereby, making it possible to provide security guarantees. However, for issues of practicality, we choose to drop the over-approximation at the end of a function's execution.

3.2.7 Applying the model to JavaScript

In the previous sections, we have formally defined our model using the while language to present the core concepts of our approach. This formal model does not take into account many of the mechanisms of JavaScript and is rudimentary. However, it is sufficient to explain the working of our model and its intended behavior with regard to JavaScript.

The limitations of the model described above include the following. The functions in the formal model have been restricted to always return a value. This value can only be assigned to a variable and cannot be used directly inside an expression. However, this is done merely to simplify the explanation of the core of ASD and is not a limitation of our approach. Similarly, the formal model of the while language only consists of the `if-then-else` conditional branching statement and `while-do` conditional looping statement. The language does not include the more complex `for`, `for-in`, `for-of` and `switch-case` branches which are part of the JavaScript logic. However, we assume that explanations for the branching statements that we have provided suffice in providing an understanding of how ASD handles these scenarios without being complete in the context of JavaScript.

Further, we only describe general variables and do not delve into the other data-types of JavaScript such as objects, arrays and properties of objects. We assume that the splitting process for the variables is extensible to the objects , properties of the objects and arrays. We also do not handle exceptions as part of the model. This is true for both the formal model and for ASD in general.

While language consists of assignment rules, handles statements and conditionals which are all part of a programming language. JavaScript does not provide direct access to any of the concurrently models such as threading. Further, all the notations used in while language form a valid subset of JavaScript. Hence, we believe that explaining ASD for while language would be beneficial towards understanding its workings.

3.3 Examples on JavaScript

The following example illustrates the different concepts described in the previous sections. We use one example application to explain the different notions of our model. However, we provide incremental code and policy modifications on a need to know basis. We will first present the basic functionalities of our model in section 3.3.1. The section 3.3.2 illustrates right propagations.

3.3.1 Basic functionalities: variable splitting and policy interpretation

Let us consider the code in Figure 3.9. In this example, there are variables `a`, `b` and `c` and there are functions `init`, `compute`, `printc` and `unauth`. `printc` is a function which performs a public output on the variable `c`. The function `unauth` is an unauthorized random function that has been added to this context.

```
1  var a = 1;
2  var b = 2;
3  var c = 10;
4  var init =  function()
5      {
6        a = 4;
7        b = 10;
8      };
9  var compute = function()
10     {
11       c =  b - a;
12     }
13 var printc =  function()
14     {
15       console.log(c);
16     };
17 init();
18 compute();
19 printc();
20
21 var unauth =  function()
22     {
23       c =  b -  a;
24       printc();
25     };
26 unauth();
```

Figure 3.9: Example for flow

We will interpret these above steps using different policies to show how they would in-

terpret with each change in policy. The first policy is shown in Figure 3.10. This policy is quite simple. It outlines that the functions `init` and `compute` have been assigned rights to the variable `a`. The function `printc` is a SSF that does not have access to any secret variable. If this function were to attempt to print any value with a secret, such an operation would be suppressed.

```
[
  {
    "functionReferences":["init","compute"],
    "hasAccess":
    [
      {
        "objectReference":"a",
        "accessType":"RW"
      }
    ]
  },
  {
    "functionReferences":["printc"],
    "hasAccess":
    [

    ]
  }
]
```

Figure 3.10: A detailed Policy Specification

The policy specification is loaded as directed in the Section 3.1.1. This implies that the policies are loaded prior to the loading of the JavaScript. This is an important consideration since it will influence variable splitting. Indeed, each variable that is mentioned in a policy specification of a given function will be split before execution.

The function `init` is the initialization function that provides the starting secret value for the variable `a`. Hence, at the end of the execution of the function `init`, the variable `a` is $\lceil 1|4 \rfloor$ and variable `b` is 10.

When the function `compute` is executed, the variable `c` is split because the secret value from variable `a` flows into the variable `c`. It must be noted that the variable `b` has not been split during this entire period. Hence, at the end of execution of the function `compute`, the value of the variable c is $\lceil 10|4 \rfloor$. At this point the function `printc` would not have access to the variable `c` and would hence only be able to print the public value of 10.

Now let us consider a scenario with the presence of an unauthorized piece of code. In this case there is an unauthorized function `unauth` which is an UF. In this case, the function can

only impact the public value of the variable c and print only this value.

3.3.2 Dictionary evolution and rights propagation

Let us consider two more functions `compute2` and `staticFunction`, shown in Figure 3.11. This example illustrates the propagation of the various variable accesses across the dictionaries in greater detail.

```
1  var d = 0; //Global scoped public variables
2  var compute2 =
3    function()
4    {
5      var localA = a - 1;
6      d = c - localA;
7    };
8  var staticFunction =
9    function()
10   {
11     ...
12   };
13 compute2();
```

Figure 3.11: Example for flow

In addition to the first policy specification, detailed in Figure 3.10 for the functions `init`, `compute` and `printc`, we specify policies for functions `compute2` and `staticFunction` in Figure 3.12. We assume that the function `compute2` is executed after the function `printc` in line number 19 of Figure 3.10.

The local variables that are added into the dictionary have the parameter describing their scope. For variables with the function scope, the corresponding scope is defined in the dictionary. The functions are assigned unique scope identifiers when they are run and this is used as a reference to identify the corresponding variable in the dictionary. We use the terminology `function().instanceId` to denote that the function's current run's scope identifier is used.

Since the function `compute2` has as both read and write access to the variables `a` and `d`, at the end of the execution of line 5 in Figure 3.11, the dictionaries would change as follows. In the various dictionaries shown in figures 3.13, 3.14, 3.15 and 3.16, we highlight the last added row in gray.

```
[{
  "functionReferences":["compute2"],
  "hasAccess":
  [
  {
    "objectReference":"a",
    "accessType":"RW"
  },
  {
    "objectReference":"d",
    "accessType":"RW"
  }
  ]
},
{
  "functionReferences":["staticFunction"],
  "hasAccess":
  [
  {
    "objectReference":"a",
    "accessType":"R"
  },
  {
    "objectReference":"d",
    "accessType":"W"
  }
  ]
},
]
```

Figure 3.12: A detailed Policy Specification

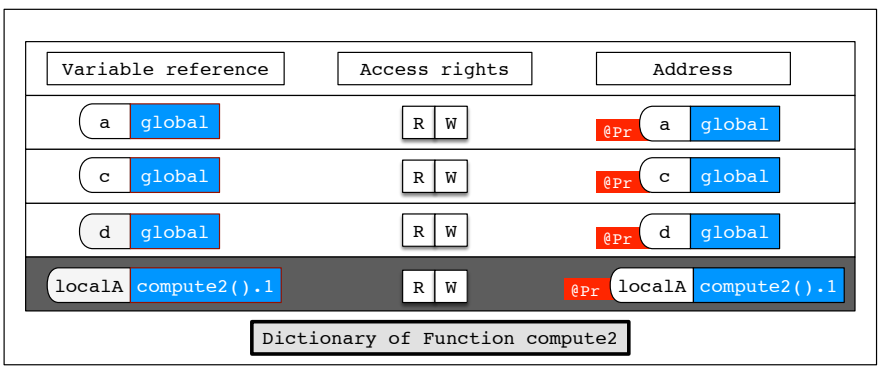


Figure 3.13: Dictionary of function `compute2`

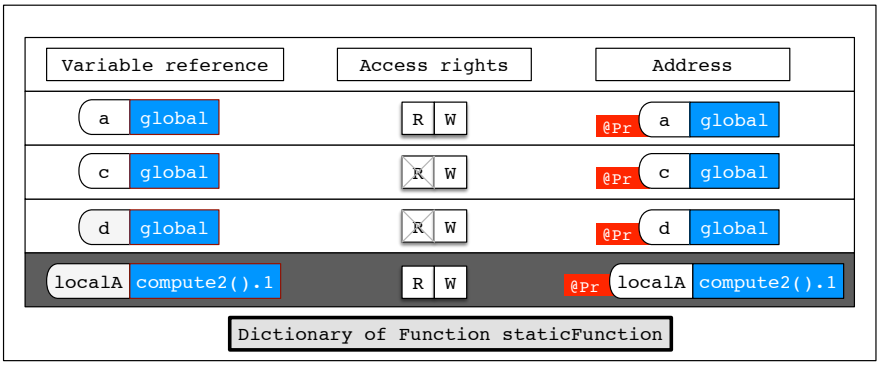


Figure 3.14: Dictionary of function `staticFunction`

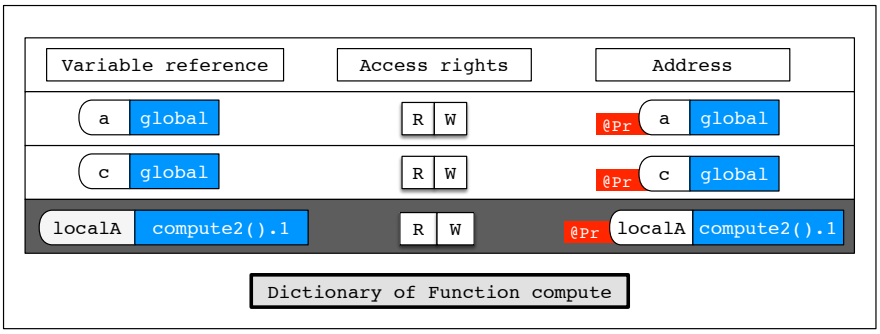


Figure 3.15: Dictionary of function `compute`

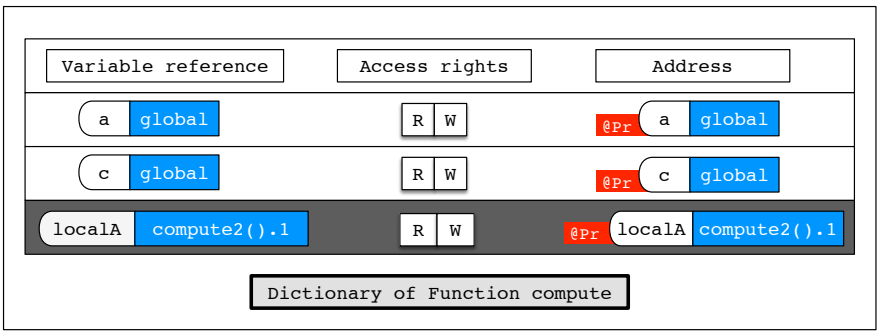


Figure 3.16: Dictionary of function `init`

It can be seen that the various local variable `localA` of the function `compute2` are being added to all the dictionaries. This is a necessary step since this information is needed to compute the various objects that the function has access to. This is an important consideration in IFC since the flow of information needs to be reflected by flow of privileges. The various new objects created, which are referenced by local variables, could be used to affect global vari-

ables or might be used as a returned parameter. Therefore, we add them to the dictionaries of all functions that have sufficient privileges to access these objects.

When the function completes execution, the final dictionaries would have changed to reflect that the function scoped variables no longer exist. Figures 3.17 and 3.18 give the final dictionnaries of functions `compute2` and `staticFunction`. While the dictionaries of the function `compute` and `init` also have some minor updates, we do not show them since the changes are not significant.

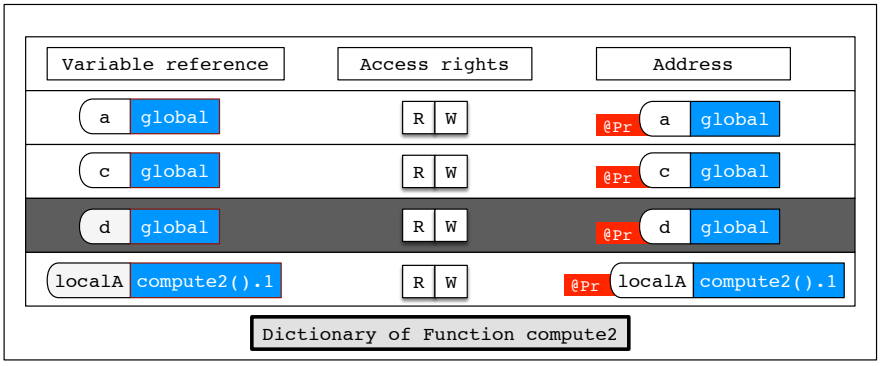


Figure 3.17: Dictionary of function `compute2`

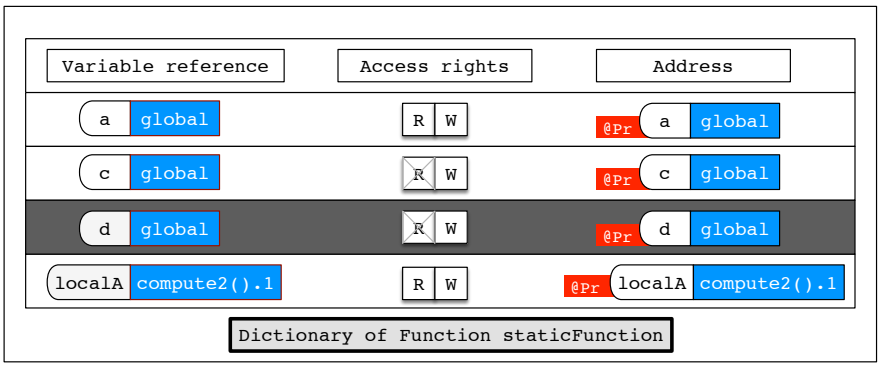


Figure 3.18: Dictionary of function `staticFunction`

The variable `d` gets data from variable `a` and this is reflected in the information flow. It is interesting to see that the function `staticFunction` is now having both `read` and `write` access to `d` though it initially only had a `write` access to the variable `d`. While this may seem like a privilege escalation, the write permission is there to protect the reference to the object rather than the object itself. Hence we do not propagate the write permission. At the end of the execution of the function, all the variables that are associated with that instance of the function are to be deleted from every dictionary.

3.4 Comparison of the approaches

As seen from the formalism, our model is capable of handling both implicit and explicit flows. However, it uses over-approximation to handle implicit flows rather than evaluating all alternative branches. This over-approximation is used by our model to help in deciding the classification of a variable. We show this in the comparison of the various approaches in this section.

The Figure 3.19 and Figure 3.20 compares our approach with the different approaches explained in the literature review using a simple example. The various approaches that have been compared use different representations. The approaches of No Sensitive Upgrade (NSU) (Section 2.3.3.2) and the approach by Hedin & Sabelfeld (Section 2.4.1) use labels and program counter (pc). The faceted approach uses principals and program counter (pc). Our model uses functions and `DT`. The set of functions that have access to a particular variable can be comparable to the principal for that variable.

In this example, we use a secret variable `x` which is labeled as high (`h`). This variable is represented as part of a principal P_1 in the faceted approach. The principal represents all the functions which are allowed to access the secret values at the high level or at the respective principal. Finally, our approach keeps track of secret variables directly and this is represented by adding the variable reference `x` to the `DT`. Since only functions with a dictionary entry for the variable can access the variable, the function `h(x)` can access the variable `x` which is similar to how the principal has access to the variable `x` in the faceted approach.

The Figure 3.19 and Figure 3.20 refer to the same function with the variable `x` being false and true respectively. These two cases are explained in greater detail below.

Figure 3.19 - [`x = false`]: In this case, only our approach performs a significantly different analysis when compared to the others. Both the naive methods of NSU and the sophisticated algorithms of the faceted approach and SME categorize the variable `z` as public and proceed accordingly. However, this is not true in our approach. In our case, the `DT` is augmented with the variables' label. Our approach classifies `z` at step 6 as secret in this case because of the over-approximation we used.

The key difference however lies in the fact that the `DT` determines the return value. If the calling function in the stack has access to the variables in the `DT` (`x` in our example), our mechanism will permit the value to be passed. The notation DT ↓?false : undefined; means that only if the calling function has read access to the dependencies, the value `false` is returned. If not, the value returned is `undefined`. The `DT` of the calling function, represented as DT_{prev} is augmented with the current list of dependencies before the final return is completed.

In this case, the value of `z` remains true for the public part in our approach which is a

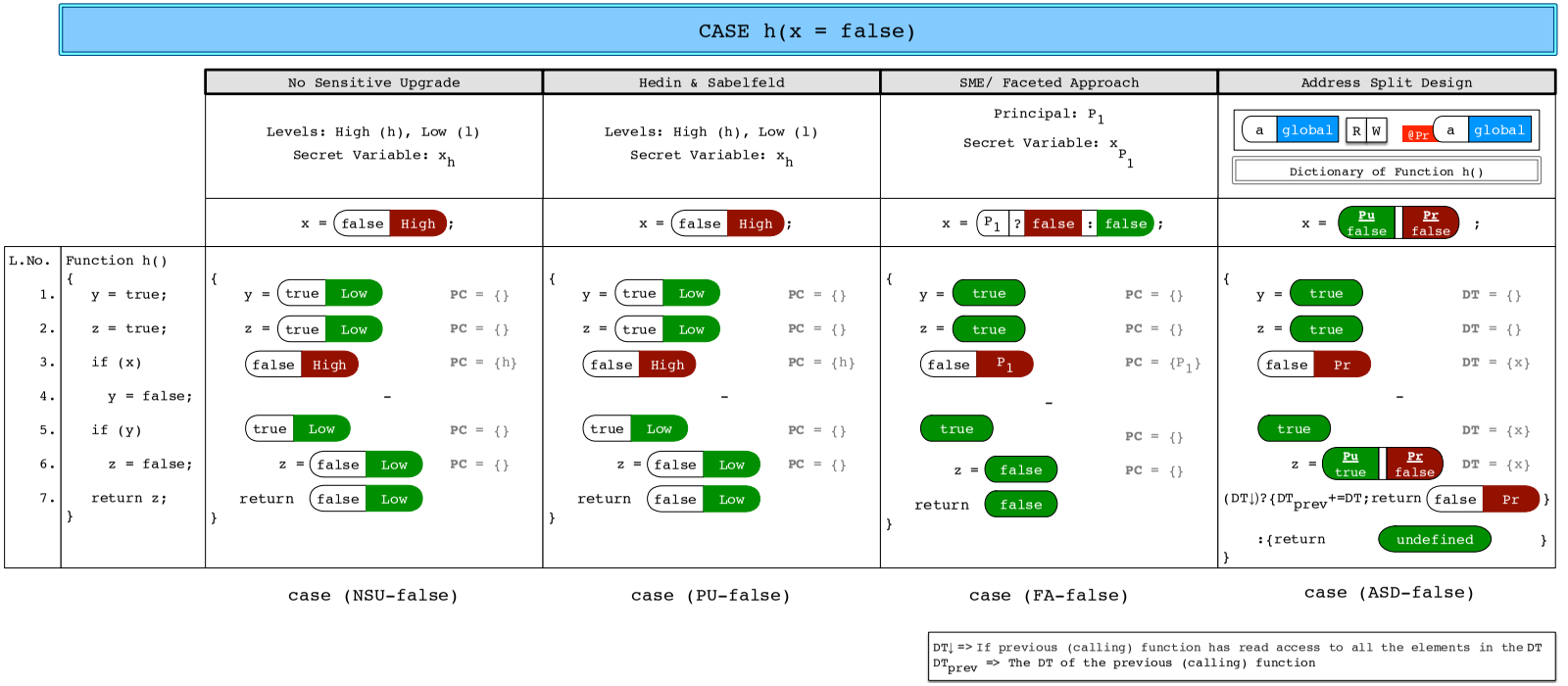


Figure 3.19: Comparison between various approaches - Case h(x=false)

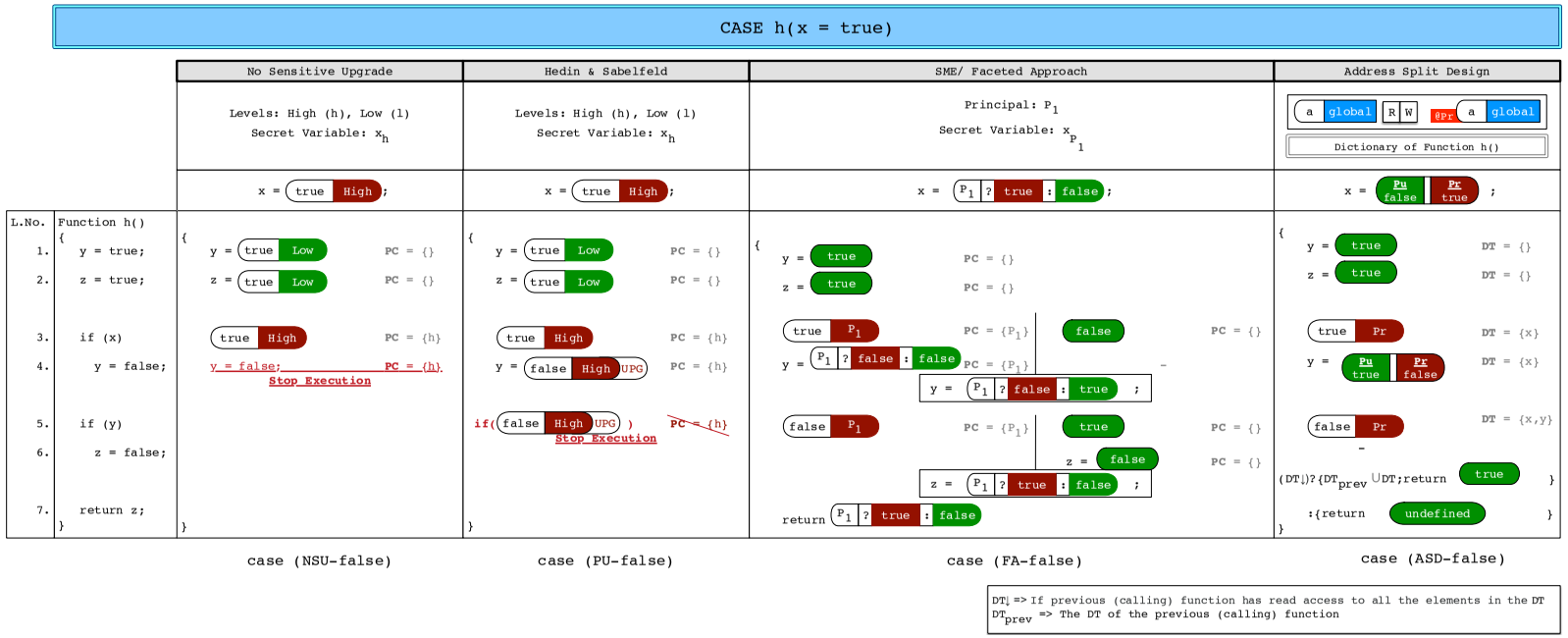


Figure 3.20: Comparison between various approaches - Case h(x=true)

significant difference with the other approaches. Our approach is also rightly classifies `z` as one which contains secret information. While this is a result of over-approximation, it must be noted that other approaches are unable to do such a classification.

Figure 3.20 - Case [`x = true`] : In our model, the dependency on `x` still remains. This is augmented with the dependency on the variable `y`. Hence, the assignment at step 6 is done to the private value of the variable `z`. The approach of NSU (No-Sensitive Upgrade) results in a stop in all execution since the upgrade is not permitted for such information flows. In the case of the approach proposed by Hedin and Sabelfeld [HS12b], an upgrade instruction is necessary for the variable `y` in line 4 before the flow can continue in a safe manner. Such limitations are not necessary in our approach, or in the case of SME and Faceted approaches. The clear difference between these SME/Faceted approaches and our model is that, in the step 5, in the case of the statement `if(y)`, the evaluation of this statement is different because `y` contains multiple values. This statement is evaluated only once in the case of our model for the value of `y = false;`. However, both the faceted approach and SME evaluate the other branch for the public value of `y = true;`. This is significant because our approach predominantly handles the various cases using over-approaximation. There is a significant overhead associated to the other approaches and this increases exponentially with the number of facets. Further, in a dynamic language such as JavaScript, there can be un-intended side-affects from executing un-necessary statements which is avoided in our approach.

The final difference is that while the public output observed at the faceted approach is always `false`, it is always `true` in the case of our approach. Further, there is no impact on the performance in our approach as there is no execution of alternative branches.

The overall advantage of our model lies in the fact that the function does not allow the secret to flow into the variable `y` and `z` in both cases without the need for executing multiple branches. The approaches of NSU, and Hedin & Sabelfeld require declassification to complete execution. The approaches of Faceted/SME require to be executed for both values of y. Our approach provides the balance of not executing additional statement while also not requiring declassification to complete the execution of the function for both values of `x`.

The disadvantage of our approach is that policies have to be precise with regard to the functions' privileges. In other approaches, only the public output functions are restricted. However, in our model, functions can only access either the public or the private value of the secret variable based on the policies. For example, when public output functions are provided with blank policies, they would have no privileges and would never be able to use the secret value. Similarly, functions need only be provided access to particular variables.

We wish to re-iterate that a utility function is not envisioned with an intention to perform

tasks such as public output, but is meant to apply to cases such as performing a square-root, calculating the interest given a number and other use-cases which are provided to do common functionality to the program. In these cases, the utility function takes on the privileges of the calling function for that execution, which is also a different perspective to other approaches brought about due to function-level policies. We consider that such fine-tuning is an advantage of function-level policies that our model has over other approaches compared in this section.

This is especially useful in JavaScript where malicious functions can be injected due to vulnerabilities like XSS. While all other traditional approaches allow such scripts to modify the secret variables and corrupt the data, ASD provides mechanisms to protect against such modifications. This is mainly due to access rights that are assigned to variables to protect them from un-intended modifications and is unique to our approach.

3.5 Conclusion

ASD is a novel dynamic IFC model that is designed to function on the modern internet and takes into consideration the nature of JavaScript. We show that it is configurable with function level fine-tuning and separate read/write privileges for functions to access secret variables. Further, the model allows for constant selective declassification using its mechanism of utility functions. Utility functions only gain access to the secrets when called by self-sufficient functions. ASD provides mechanisms to control information at every level and not just at the perimeter like traditional approaches. Using ASD, even intermediary functions would not gain access to the actual secret data if they are not privy to such information.

Chapter 4

Implementation and evaluation

In this chapter, we discuss about the practical implementation of our model on a web browser and the results of our evaluation. To confirm our model we first implemented it in Chromium and subsequently verified the security provided to data from access against unauthorized functions.

4.1 Implementation details

We implemented our model on the Chromium V8 JavaScript engine. This JavaScript engine is open source and is well documented. We also considered the use of the Narcissus engine, which is a JavaScript interpreter written in JavaScript for the Mozilla Firefox browser. This was because Narcissus was used in the implementation of the faceted approach, which we hold in high regard. However, due to the experimental nature of the engine and inexistent support to the project, we chose the V8 engine as our primary choice. We were further motivated in using a realistic JavaScript engine to evaluate the effectiveness of our approach and measure its impact on performance. Hence, while it is more complex to implement our approach on a full-fledged engine such as V8, considering the maintenance, and documentation of the code base, we choose to use the V8 engine.

We did however face some hurdles because of the size of the codebase and the need to do very intrusive changes to the compilation process. It must be noted that the initial codebase of Chromium consists of several gigabyte of data. The architecture of V8 is quite complex and relies on three just in time (JIT) compilers, namely full codegen, crankshaft and TurboFan. V8 dynamically change the compiler to optimise JS code that is often executed. We decided to modify only the primary compiler (full codegen) and disable the other compilers. Indeed, the size of the codebase made it harder to identify all the points in the code blocks to insert hooks

and this became increasingly difficult in case of more efficient compilers such as crankshaft and TurboFan. We hence decided to focus on a single compiler and choose full codegen which was easier to hook into than others though there were a lot of modifications to be made on this compiler as well. We disable the other compilers, which degrades the performance of V8 for JS code that are frequently executed.

During the course of our modifications we found several issues that were subsequently solved with unique tailored solutions. The foremost of these was the inline caching mechanism. This mechanism is used to cache previously used variables for faster subsequent access. We disabled this mechanism and modified the code for the caching to force a lookup at every variable access. Similarly, we had to modify the variable creation module. We had huge difficulties in this module since there were a lot of complex memory checks performed for code blocks and it was not feasible to change the variable block to contain two variables (to reflect a split variable). Such modifications would have required several core files as well as the behavior of the garbage collector to be modified. Hence, we modified the object properties of the global object and intercepted the object creation module to be triggered twice for a split variable. Such variables are mapped by ASD's internal modules and maintained. These modules also needed to be created in such a way that they were not accidentally garbage collected.

Another issue that needed to be handled was the way optimizations were done to function parameter variables. In chromium's compilation modules, the lookup for the function parameter variable would be skipped and optimized completely if this parameter was only read from and there was no observable write. If there is a write, chromium would de-optimize this mechanism at runtime. We modified the code of the function at compile time to be in a dynamic lookup context to ensure that the code lookup occurs for every variable call. ASD hooks were inserted into the variable lookup process to change the variable that needs to be used at runtime.

Our modifications to Chromium hence primarily involves the need to change the compilation process and to add various hooks at the variable lookup level so as to make decisions on using the public or the private version of the variable. It must be noted that our modifications have been limited to the V8 JavaScript engine and our proof-of-concept implementation covers only the global variable names and function local variable names. Further, we have not implemented any hooks on the DOM variables from the `blink` rendering engine. While variables related to the DOM such as `innerHtml`, `class`, `name` and `data-elements` are still parsed through the V8 engine, the DOM rendering remains untouched in our prototype.

It must be noted that there is a high degree of optimization that takes place in the code based on the number of function calls and variables used. We have disabled several of these optimization mechanisms in the V8 engine to facilitate easier implementation of our mechanisms. We have also added various hooks into V8. Every variable read and write operation

is monitored. We have done this by modifying the inline caching mechanism and the scope getter/setter mechanisms.

Despite such hooks, our prototype implementation is crude to a great degree. We have added a source code wrapper which changes the source code of the loaded JavaScript before compilation. This wrapper helps to facilitate the loading/unloading of dictionaries, allowing/-denying the function call, checking the function parameters for secret values, and adding the function scoped variables to the monitor. These are made through inserting custom defined JavaScript functions pointing to our C++ libraries. The wrapper uses a series of regular expressions to modify the code. The wrapper has been added to the core compilation module of V8 and is capable of handling eval functions. There is hence an overhead introduced by our model which we believe can be improved. One proposal is to add the functionality directly to the compiler rather than adding wrappers to modify the code in a just-in-time manner and hooks to third party implementations as is the case currently.

Over the course of our work we added a core ASD C++ code base of 3509 lines. These do not include the other third party libraries we used or the code we added into specific parts of V8 to add hooks to the engine. We added approximately 40 functional hooks at varying locations. There were several more hooks that we used but later discarded due to finding better alternatives or due to other complications. The code added is actually miniscule when compared to the size of V8. The core of V8 alone is about 174.8 MB in size and our code contributed towards around 700 kilobytes of data without including third party libraries such as boost which we used in our process. We also ported our code to Chromium replacing the default V8 compiler with the version of the ASD prototype. There were also some JavaScript code added to the engine but these JavaScript code acted as native functions communicating with the ASD libraries. Adding all our routines to the compilation manifest has been a tedious process to interweave our approach into the core library with its immense number of integrity checks.

4.2 Performance evaluation

In thi section, we further describe the various tests that have been performed to validate the characteristics of the ASD implemented on a web browser. In the Subsection 4.2.1, we evaluate the performance of performing read and write operations in case of a split variable on the V8 engine implementing ASD. The Subsection 4.2.2 provides some experiments to compare ASD with the performance degradation of SME and faceted approach.

4.2.1 Performance estimation based on number of dictionaries

First, we intended to measure the performance of read/write operations due to the implementation of our model with regard to the number of dictionaries. This is done by measuring the execution time on the system. We reassert that when a secret variable is modified, every function that has write access to it needs to re-evaluate its dictionaries for continued access. Such a modification would hence have a significant overhead associated to it. This is not the case for read access. Only the dictionary of the current function that is being executed is used to infer the value to be used.

The graph in Figure 4.1 shows the performance of V8 when a function which contains read or write instructions to a secret variable is executed. This is quantified by measuring the execution time of the read and write statements. The results were obtained by inserting code to measure time before and after the statement and computing the difference. Each of these results shown in these graphs is the average of the values obtained over 1000 tries. The Table 4.1 provides the maximum and minimum values obtained during the course of these trials and the standard deviation obtained. We measure this time taken for execution for different number of self-sufficient functions with their own dictionaries. In our test case, this number of dictionaries ranges from 0 to 10000. The results computed are based on the fact that all functions have access to the secret variable and hence their dictionaries would need to be modified when this secret variable is modified.

In our setup we use a generic V8 engine which was modified to use only one of its three compilers (codegen). The V8 with ASD also use only one compiler. However, it is also modified to have all the hooks related to ASD plus the setup phase which modifies the JavaScript code to be executed, since some part of our monitor are directly inlined in the original source code to be executed.

In Figure 4.1, the subfigure 4.1a shows the results obtained when the function only performs one read. It can be seen that the impact is quite limited for increasing number of dictionaries. This is because the read operation uses only the current function's dictionary. The execution time is increasing in this case since the number of dictionaries created and maintained by the model is increasing. Hence, to load the dictionary corresponding to a given function takes marginally greater amount of time. We estimate that using much more efficient data-structures could reduce the time taken.

The overhead between the de-optimized V8 engine and the V8 engine running ASD depends to a great extend on the setup phase which inlines the modification to the JavaScript. This setup phase is only executed once, when the JavaScript is loaded and is constant with regard to the number of dictionaries. This is also highlighted by the subfigure 4.1c representing 100 reads performed by the JavaScript function. In that case the relative overhead between

ASD-V8 and V8 is less important since the overhead due to the setup phase does not depends on the number of reads.

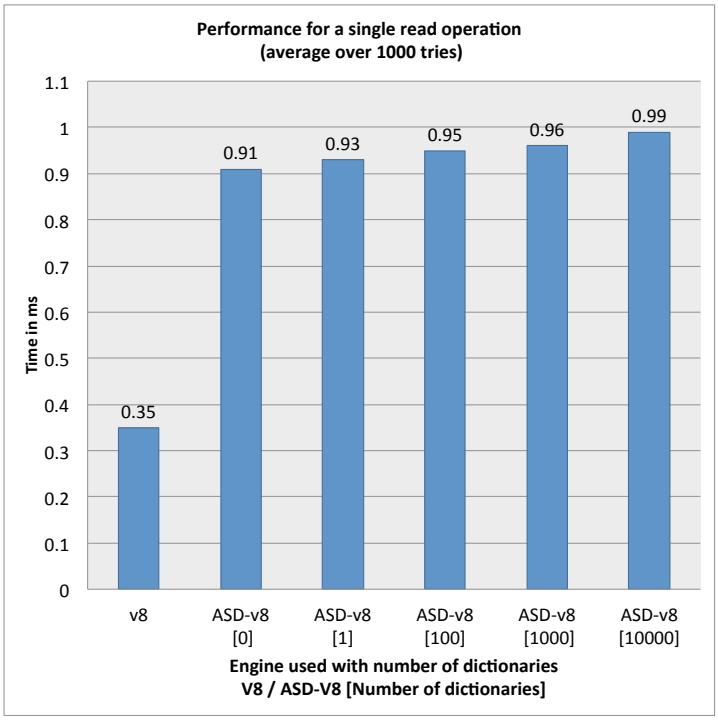
When looking at write operations, the dictionaries of all functions need to be re-evaluated when a secret variable is changed. This causes a significant performance degradation on write operations as can be observed in subfigure 4.1b in comparison to the read operation in subfigure 4.1a. This difference is observable at 10000 dictionaries. Similarly, the subfigure 4.1d which represents 100 writes amplifies this issue.

Despite the overhead presented in the graphs, we consider such a scenario of 10000 self-sufficient functions to be an abnormal usecase. To verify the number of functions in a standard website, we used a list of websites from Alexa top 125[1]. This is a standard list of websites that is maintained by the Alexa compagny to rank the various websites across the world based on their traffic. We iterated through Alexa top 125 websites and computed the list of functions present in each of them. Our tests on the these websites resulted in a maximum of 8500 named functions and a minimum of 141 named functions with an average of around 2780 functions. These figures include all libraries, advertisements and cross-domain scripts that run on the webpage.

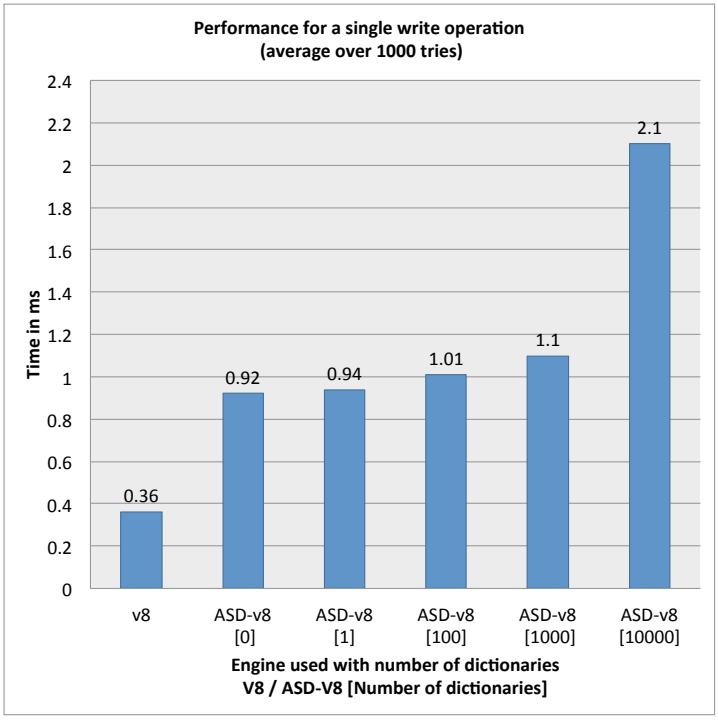
Our fundamental design choices have been made with the assumption that the number of self-sufficient functions would be low, i.e. most functions would not need to access secret variables.

In a scenario where every function is self-sufficient, other optimizations such as a group of functions sharing the same dictionary or a single dictionary based on script origin URL can be used to reduce the overhead. In this case, all the access rights of all the functions that fall in this group would be the same and there would need to be only one dictionary. This would be comparable with a label based approach having one label for all the functions in that group. It is possible to group functions based on their origin URL as well. In this case, any functions that are obtained from a particular URL could be grouped together into a single category having the same privileges. For example, all functions in a file named `login.js` could be given access to a variable `username`. This would provide for a possible optimization to reduce the number of dictionaries maintained by the compiler.

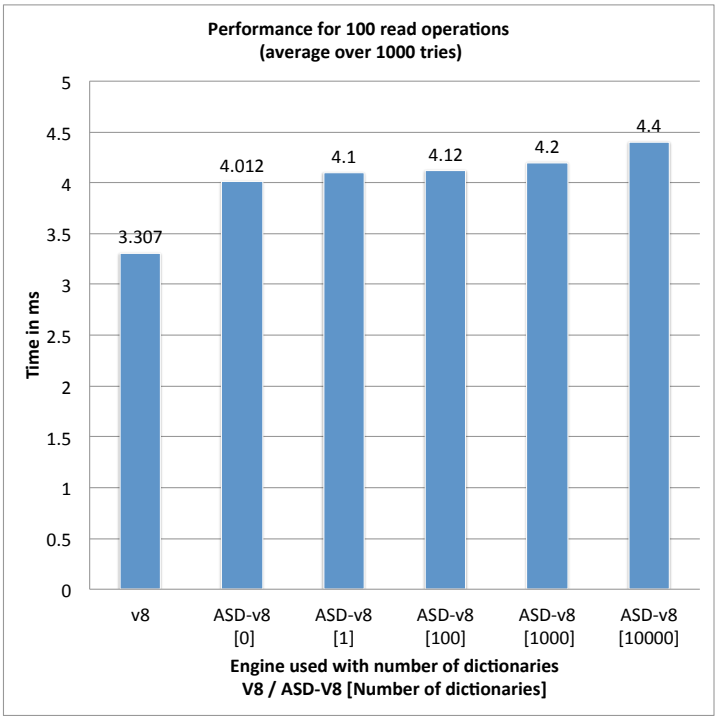
[1] `https://www.alexa.com/topsites`



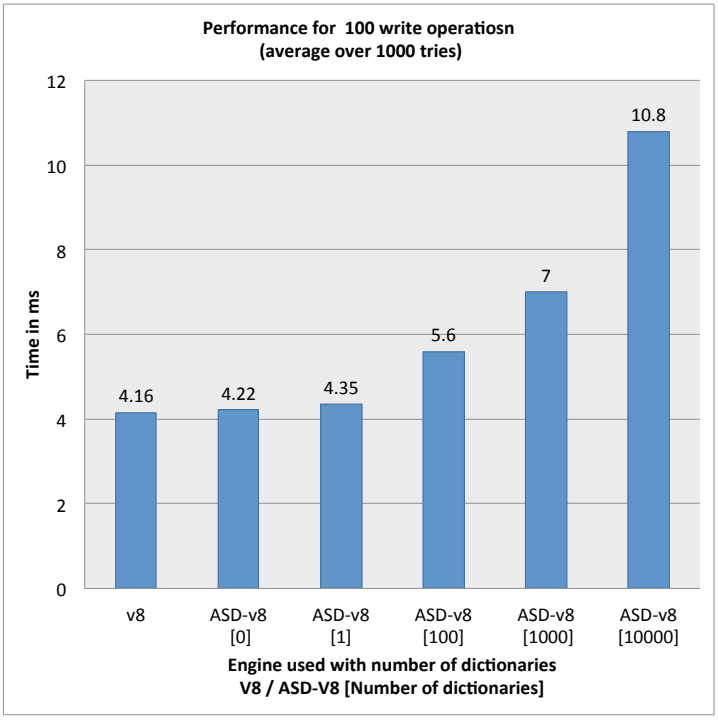
(a) Performance for 1 read



(b) Performance for 1 write



(c) Performance for 100 reads



(d) Performance for 100 writes

Figure 4.1: Performance tests of V8 vs ASD.V8

Table 4.1: Standard deviation for the performance test

Operation	Number of operations	Engine/No. of Dictionaries	Max	Min	SD
Read	1	V8	0.36	0.34	0.0058
Read	1	ASD-V8/0	0.95	0.87	0.0234
Read	1	ASD-V8/10	0.98	0.88	0.028
Read	1	ASD-V8/100	1	0.9	0.0286
Read	1	ASD-V8/1000	1.02	0.901	0.0351
Read	1	ASD-V8/10000	1.04	0.94	0.0284
Write	1	V8	0.42	0.3	0.0346
Write	1	ASD-V8/0	0.96	0.88	0.0230
Write	1	ASD-V8/10	0.97	0.91	0.0173
Write	1	ASD-V8/100	1.04	0.98	0.0170
Write	1	ASD-V8/1000	1.14	1.06	0.0230
Write	1	ASD-V8/10000	2.169	2.03	0.0391
Read	100	V8	3.347	3.267	0.0228
Read	100	ASD-V8/0	4.062	3.962	0.0286
Read	100	ASD-V8/10	4.14	4.06	0.0228
Read	100	ASD-V8/100	4.15	4.09	0.0170
Read	100	ASD-V8/1000	4.22	4.18	0.0116
Read	100	ASD-V8/10000	4.45	4.35	0.0287
Write	100	V8	4.23	4.09	0.0407
Write	100	ASD-V8/0	4.24	4.2	0.0114
Write	100	ASD-V8/10	4.4	4.3	0.0294
Write	100	ASD-V8/100	5.659	5.54	0.0352
Write	100	ASD-V8/1000	7.04	6.96	0.0227
Write	100	ASD-V8/10000	10.83	10.77	0.0167

4.2.2 Comparison of performance with SME and faceted approach

In this section we present an experiment that is used to measure the performance deterioration of the various approaches with increasing number of dictionaries. Our approach aims to have a reduced performance consumption and the results that we obtained are consistent with our expectations.

The Table 4.2 shows the time difference between the evaluation of SME, faceted approach and ASD for the program used in the Figure 3.20 in Section 3.4. The tests have been performed for different number of principals in SME and faceted approach, and number of dictionaries for ASD. The implementation of SME and faceted approach use the Narcissus JavaScript engine [Wike].[2] It must be noted that the time taken for executing this program on a standard

[2]Obtained by email from Dr. Thomas Austin, the developer of this code base. `http://www.sjsu.edu/people/thomas.austin/`

Mechanism:	SME	Faceted Approach		ASD	
		Best Case	Worst Case	Best Case	Worst Case
Number of principals or dictionaries	*Time in ms*				
0	3	5	5	2	2
1	4	5	5	3	3
2	10	6	6	3	3
3	21	6	7	3	3
4	58	6	13	3	3
5	112	6	25	3	3
6	222	7	30	3	3
7	470	7	42	3	3
8	1026	8	75	3	3

Table 4.2: Comparison between SME, Faceted Approach and ASD

Narcissus engine was 3ms. Similarly the time taken to run this program on a standard V8 engine was 2ms.

Regardless of the number of principals to be satisfied to have access to the secret, SME executes the program 2^n times, where n is the total number of principals. Faceted approach has a best case and a worst case. The number of times it executes the implicit flow depends on the principals that need to be satisfied for access to the secret. The best case implies that there is only one principal that needs to be satisfied for access to the secret though there are n principals in the program. The worst case implies that there are n principals and all of them need to be satisfied to get access to the secret. In any case, ASD performance are quite similar to the best case for the faceted approach as shown in the table. The best case for ASD is when only one dictionary has read access to the variable, hence only one dictionary needs to be modified. The worst case for ASD is when all dictionaries have read access to the variable thereby requiring more number of updates when the value of the variable changes. There is no observable difference in ASD results for the best and the worst case scenario for the number of dictionaries used in the program. As shown in the read, write tests above, ASD is able to scale better than SME and faceted approach. It only shows significant deterioration in performance above 10000 dictionaries which is a reasonable limit.

4.2.3 Impact of ASD on real websites

In this section we measured the impact of using a browser with an active `asd` implementation on the Alexa top 125. These tests are intended to measure the performance degradation due

to the additional computation performed due to the presence of ASD as well as measure the differences in the page loads between a browser with `asd` and one without `asd`.

For all these tests we have used three versions of the Chromium browser. The first is the unmodified original Chromium browser that was compiled in the system. We refer to this as "original Chromium". The second is the Chromium with inline caching disabled and using only the full-codegen compiler of V8. We call this the "de-optimized Chromium". These de-optimizations have also been used in the version that implements the ASD model. We call this "ASD Chromium ".

For the first test we intent to find if there is any noticeable difference between a webpage loaded in the original Chromium and a page loaded by ASD Chromium. This is done to verify if adding ASD adversely affects the actual working of a browser. However, in the modern web, the webpages are no longer static and differ in their content by user, time and many other factors. The first step it to have a base vector to compare with. The original chromium browser is allowed to load a webpage and this webpage is recorded. Then it is used to load the page a second time and the similarities between the two page loads are noted. Finally, ASD Chromium loads the same webpage and the similarity between ASD Chromium and the first run of the original chromium browser are made. This is done by obtaining the rendered document tree after the page load is completed, as a string representation and then computing the difference between these two strings.

The Table 4.3 shows a portion of these results. It can be seen from this table that ASD's integration has not adversely affected the page load of the browser. In this table, pages that have no dynamic content such as Tco are not supposed to have any difference between two consecutive loads on two different browsers. Hence, the percentahe similarity is at 100%. However, for pages such as Youtube, there is notable difference between different page loads. This is because even if a few video recommendations change or the order of the recommendations change, it significantly affects the similarity score. In blogspot while the percentage similarity was 100% on the original Chromium between multiple loads, ASD Chromium did not have the same similarity with the original browser. This was because one of the elements loaded by the server had a different dynamic HTML5 "data" attribute string associated to it. The result disparity in Stackoverflow is because the list of "Top Questions" on the page is refreshed very often. This behavior is consistent across all browsers. The % Similarity for ASD in the case of Stackoverflow is high because majority of the questions were the same at the time both browsers loaded the page due to coincidence.

In almost all other cases, the results were quite similar as the expected results as per the page's dynamic behavior. This can be seen from a the Figure 4.2 which estimates the difference between the similarities observed by the original Chromium and ASD Chromium. In this figure

it can be seen that in more than 72% of the websites, both browsers have a difference of less than 1% and only 2.8% of websites have a difference greater than 10%. Our observations indicate that these differences were because of the dynamic content in the webpage being very different or arranged in very different orders. There was no observable error that was unique to either browser.

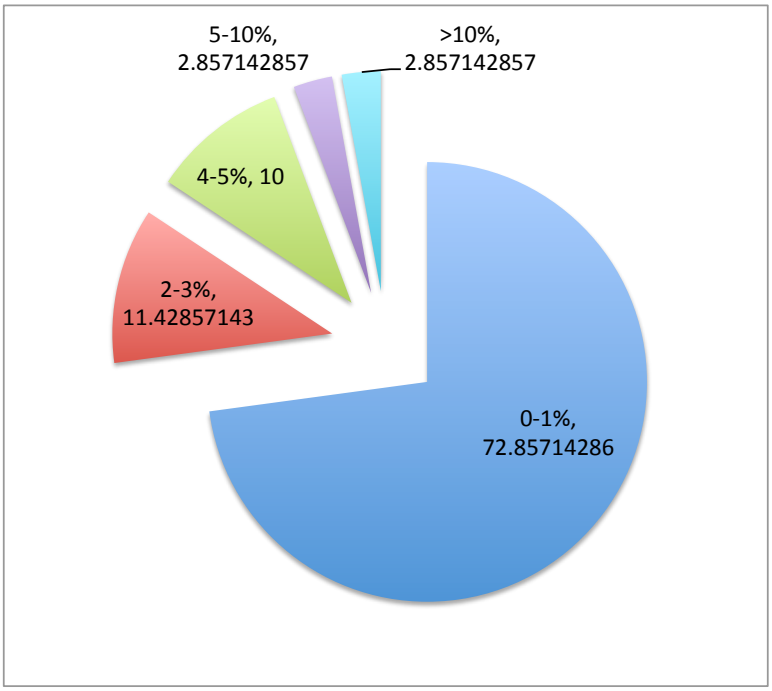


Figure 4.2: Percentage difference between the similarities by original Chromium and ASD Chromium

NAME	SITE	% SIMILARITY	% SIMILARITY ASD
Youtube	`http://youtube.com`	77.0942	82.4149
Facebook	`http://facebook.com`	77.7414	77.0141
Baidu	`http://baidu.com`	99.5651	99.6272
Amazon	`http://amazon.com`	98.2958	98.1195
Wikipedia	`http://wikipedia.org`	100	100
GoogleIn	`http://google.co.in`	99.6008	99.7876
Twitter	`http://twitter.com`	98.9562	98.5978
Live	`http://live.com`	97.3093	96.4687

GoogleJp	http://google.co.jp	99.6781	99.6638
Bing	http://bing.com	98.432	96.7512
YahooJp	http://yahoo.co.jp	88.4197	85.1356
Linkedin	http://linkedin.com	99.7962	99.6703
Vk	http://vk.com	99.9041	99.8374
YandexRu	http://yandex.ru	95.0448	94.527
GoogleDe	http://google.de	93.2195	93.2175
GoogleRu	http://google.ru	99.7186	99.5478
AmazonJp	http://amazon.co.jp	97.835	97.9298
GoogleUK	http://google.co.uk	99.5479	99.4161
360cn	http://360.cn	99.7773	99.8856
Tmail	http://tmall.com	97.9849	95.4378
GoogleBr	http://google.com.br	99.6878	99.3843
Tco	http://t.co	100	100
MailRu	http://mail.ru	99.0407	99.1003
Microsoft	http://microsoft.com	97.2932	98.3188
Paypal	http://paypal.com	98.7449	98.8882
Wordpress	http://wordpress.com	99.756	99.7543
Onclickads	http://onclickads.net	99.3622	99.7267
GoogleEs	http://google.es	99.6989	99.4232
Blogger	http://blogspot.com	100	99.911
Tumblr	http://tumblr.com	96.5471	94.9632
Apple	http://apple.com	99.9947	99.4453
Imgur	http://imgur.com	96.4429	99.7809
Stackoverflow	http://stackoverflow.com	71.0377	96.5764
Gmw	http://gmw.com	98.5964	98.3954
Aliexpress	http://aliexpress.com	89.9766	91.8698
GoogleMx	http://google.com.mx	99.6573	99.5413
Imdb	http://imdb.com	93.4776	93.0096
Fc2	http://fc2.com	99.7609	99.7788
GoogleHk	http://google.com.hk	99.6541	99.5967
Chinadaily	http://chinadaily.com	99.4052	100
OkRu	http://ok.ru	88.0338	99.8105
Naver	http://naver.com	89.096	90.6036

AmazonDe	http://amazon.de	97.9261	97.9848
Github	http://github.com	98.3017	98.1234
Ask	http://ask.com	99.7212	99.7064
Diply	http://diply.com	97.9284	98.3483
Rakuten	http://rakuten.co.jp	85.1809	86.5742
GoogleId	http://google.co.id	99.4116	99.2816
Office	http://office.com	99.8268	99.9263
GoogleTr	http://google.com.tr	99.4653	99.7198
Tianya	http://tianya.cn	97.297	99.3753
Alibaba	http://alibaba.com	98.2012	99.8124
Craigslist	http://craigslist.com	99.9867	100
Pixnet	http://pixnet.net	84.5572	84.7843
Jd	http://jd.com	91.6158	95.1255
Niconico	http://nicovideo.jp	94.4795	97.5141
AmazonIn	http://amazon.in	98.1498	98.2191
AmazonUK	http://amazon.co.uk	97.8823	97.9771
GoogleKr	http://google.co.kr	99.8599	99.5936
Cctv	http://cntv.cn	100	100
GooglePl	http://google.pl	99.5443	99.5497
Whatsapp	http://whatsapp.com	100	100
GoogleAu	http://google.com.au	99.6765	99.6493
Outbrain	http://outbrain.com	97.2997	99.5575
Dropbox	http://dropbox.com	81.7632	82.1363
Coccoc	http://coccoc.com	98.3057	99.7119
Adobe	http://adobe.com	96.1428	96.8437
Sogou	http://sogou.com	92.186	99.6805
Microsoftonline	http://microsoftonline.com	93.9214	91.9238
China	http://china.com	98.8469	99.6702

Table 4.3: Percentage Similarity of pages across multiple page loads

We also verified the errors (if any) that were displayed on the page. In any case, if there were any errors in the original browser, they were also reflected in ASD. We have not encountered any error that was unique to either browser during the course of our experimentation.

We continued further experimentation on the Alexa 125 to check the performance degra-

dation due to the presence of ASD. To do this we implemented a plugin and forcefully inserted policies at page load to ensure that the ASD mechanism has some active dictionaries at the run-time of these functions. The plugin also computed and aggregated the page load times for the various websites. The plugin starts loading the website and after the load is completed, the default Chromium APIs are used to obtain the load time for the page. The results are shown in Table 4.4.

The page load times of the de-optimized Chromium and ASD Chromium have been then compared to check the degradation caused by ASD. There are some cases where ASD performs better than the de-optimized Chromium. However, these are specific cases due to network lag, or because there have been limited JavaScript in the page (such as Tco). Many Chinese domain name sites such as Weibo experience a significant and inconsistent lag when loading the page. This is the reason we could determine for the negative percentage difference. We also believe that there are some other network caching mechanisms provided by the internet provider which influence load time of resources. AmazonDe which experienced significant performance degradation did not have any network issues and the performance degradation can be directly correlated to the JavaScript on the page being executed with ASD hooks. However, this is the maximum % difference observed by us over all our samples. The second highest was on ebay at 180.22% and the third was on AmazonJp at 150.82%.

To understand more on this, we made an analysis as shown in Figure 4.3. In this analysis, the pages where ASD consumed more load time as measured. In this case for 41% of the webpages, ASD consumed less than 10% overhead to load the page. Only for 6% of the pages did ASD consume more than 50% more of overhead.

Name	Page Load Time	ASD Page Load Time	% Difference
Youtube	1698	2224	30.98
Facebook	1749	2841	62.44
Amazon	1498	1570	4.81
Wikipedia	409	624	52.57
GoogleIn	113	134	18.58
Twitter	1016	717	-29.43
Taobao	9302	11372	22.25
Live	247	292	18.22
Sina	27151	36042	32.75
Bing	219	319	45.66
Msn	523	1064	103.44
YahooJp	6563	11538	75.8

Weibo	6881	4861	-29.36
LinkedIn	647	737	13.91
Vk	896	876	-2.23
YandexRu	1144	1230	7.52
Instagram	1249	1222	-2.16
Ebay	2654	7437	180.22
GoogleRu	671	106	-84.2
AmazonJp	1523	3820	150.82
Reddit	4643	4851	4.48
GoogleUK	705	814	15.46
360cn	10125	13207	30.44
Tmail	10254	17060	66.37
Pinterest	1237	1467	18.59
Tco	187	117	-37.43
MailRu	2847	3655	28.38
Microsoft	1173	1018	-13.21
GoogleIt	740	782	5.68
Paypal	505	945	87.13
Wordpress	1046	1165	11.38
Onclickads	813	1165	43.3
GoogleEs	633	797	25.91
Blogger	781	811	3.84
Tumblr	5834	6608	13.27
Apple	1038	1098	5.78
Imgur	2096	1878	-10.4
Stackoverflow	563	682	21.14
Gmw	19221	26603	38.41
Aliexpress	4372	3088	-29.37
GoogleMx	479	492	2.71
Fc2	768	1059	37.89
GoogleHk	138	153	10.87
Chinadaily	21198	22431	5.82
OkRu	1748	1898	8.58
GoogleCa	87	107	22.99

Naver	11115	12929	16.32
AmazonDe	1796	6194	244.88
Github	1492	1481	-0.74
Ask	972	539	-44.55
Diply	1846	2095	13.49
Rakuten	27448	36662	33.57
GoogleId	652	739	13.34
Office	1094	1265	15.63
GoogleTr	131	116	-11.45
Tianya	10871	20310	86.83
Alibaba	10874	18740	72.34
Sogou	12820	15557	21.35
Craigslist	1368	1341	-1.97
Pixnet	8214	10840	31.97
Jd	18591	19767	6.33

Table 4.4: Load times of the original and ASD integrated Chromium browsers

Our implementation can hence be achieved with a low loss in performance to modern web pages and is not disruptive to the current pages on the internet. It is our belief that it is hence practical for a modern web browser.

4.2.4 Standard benchmark tests

In this section we list the results obtained when ASD is run on standard benchmarks and tested for both conformance as well as performance degradation due to the presence of ASD.

To check if implementing ASD adhered to the conformance of web standards we performed the `acid3 tests` [Wika] on both the original Chromium browser and the Chromium browser which is implementing the ASD model.

This is a standard test suite developed by the `Web Standards Project` to the check the browser's compliance with the various web standards. Both the original Chromium browser and the ASD Chromium browser obtained a score of 100/100 which means that addition of the ASD module did not break any of the standard web functionalities of the web browser with regard to CSS and JavaScript.

Other than the conformance test, we also performed tests on six standard benchmarks. These results are shown in the Figure 4.4. The first of these is SunSpider [Webc]. This bench-

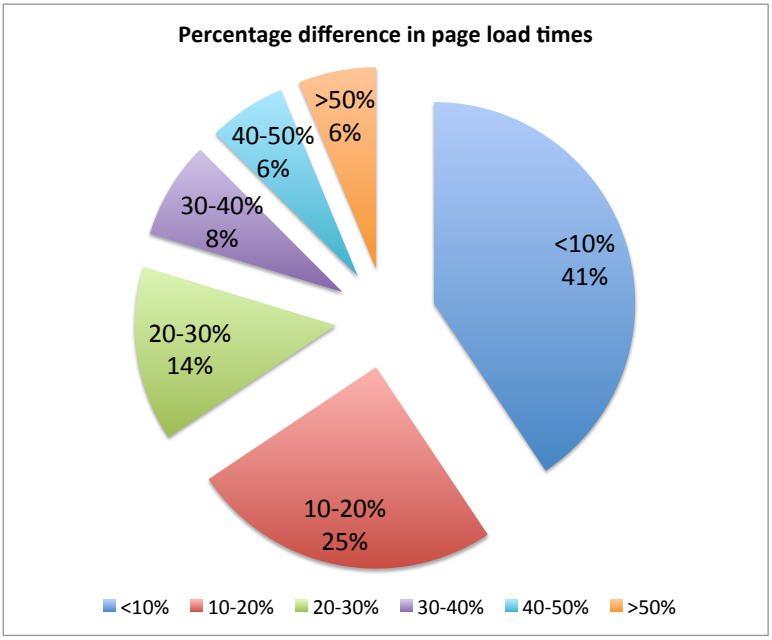


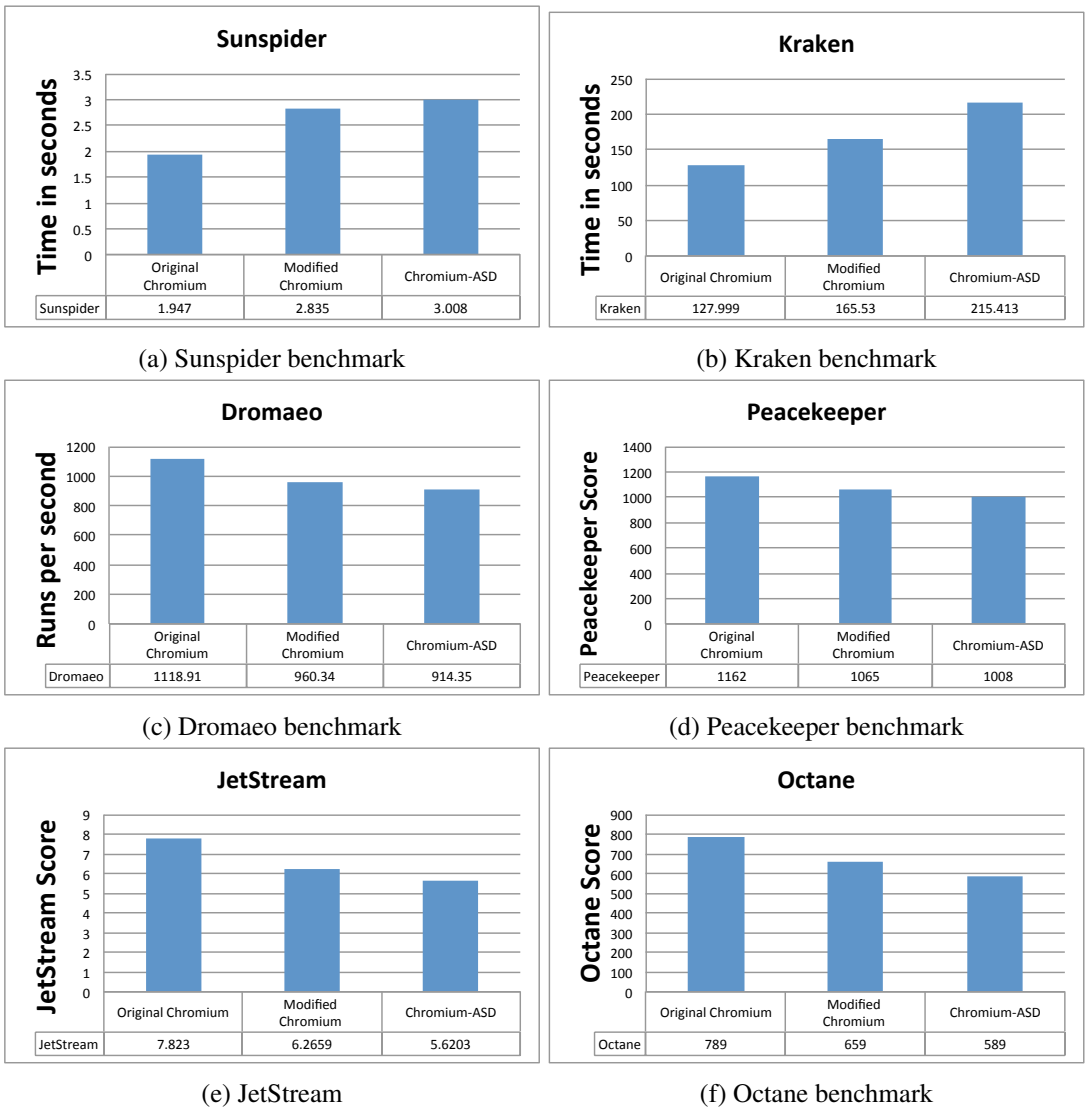
Figure 4.3: Percentage difference between the page load times of de-optimized Chromium and ASD Chromium

mark only tests the core JavaScript implementation and does not check the DOM or CSS. However, it is a test that focuses on use-case based analysis such as code compression, encryption and text manipulation. The results of this test are shown in the subfigure 4.4a. It must be noted that a lower result is better in this analysis. The modified de-optimized Chromium only performed marginally better than ASD Chromium in this case. This result is within expectations since the benchmark focuses on use-case based analysis rather than performing more intense tests.

Another benchmark, named Kraken [Mozc], was developed along the same line of thought of SunSpider by Mozilla. This benchmark performs some additional tests such as an A-star search algorithm and some cryptographic routines. These results are shown in the subfigure 4.4b. Following SunSpider, the results for this benchmark are also better if they are lower. However, Kraken is significantly more intensive than SunSpider and the tests reveal that the ASD Chromium browser takes about 30% more time for the same set of JavaScript tests. This is an overhead due to our mechanism.

Dromaeo [Mozb] is a benchmark suite that performs both JavaScript and DOM test. It is maintained by Mozilla. The results for this test are shown in the subfigure 4.4c. Unlike other tests prior to this, it computes number of runs per second. Hence, a higher result is better. ASD performs significantly well in this test by only causing a 4.8% overhead over the

modified de-optimized Chromium. This is because the code is re-run multiple number of times and the wrapper function that is used prior to compilation need not be reused many times in this use-case.



(a) Sunspider benchmark

(b) Kraken benchmark

(c) Dromaeo benchmark

(d) Peacekeeper benchmark

(e) JetStream

(f) Octane benchmark

Figure 4.4: Standard benchmark comparisons between Original Chromium, Modified Chromium and Chromium-ASD

Peacekeeper [Fut] is a benchmark to measure JavaScript performance using test cases derived from pages such as youtube. It tests many new HTML5 technologies such as WebGL and video. The result obtained is a peacekeeper score which is computed at the end of the execution and is shown in subfigure 4.4d. A higher score is better in this benchmark. It must

be noted that peacekeeper is a realistic test case, hence does not exclusively test the rendering, DOM manipulation or JavaScript but is a combination of all three. ASD chromium performs significantly well in this test by only causing a 5.35% overhead over the modified de-optimized Chromium. This is because this test contains many components including the rendering of HTML5 video on the page. These features are not directly impeded by our implementation and the tests involving the rendering engine are not affected due to our model. However, the test also requires JavaScript which is processed by the V8 engine and these are affected due to ASD. Further many of these tests are related to parts of the code where the hooks by ASD are done purely in C++ and hence the better efficiency.

Finally, we performed tests using Octane [Goo] and JetStream [Webb] benchmarks. These tests also verify the time taken to compile and then run the same code multiple times. The results are shown in subfigure 4.4e and subfigure 4.4f. In both these tests the higher score is better. In this case, due to having no optimizations especially without inline caching, the modified de-optimized Chromium and the ASD Chromium have a significant performance degradation in performance.

The benchmarks and performance analysis show that ASD is practical in the modern web. The overhead for 66% of the Alexa top 125 pages is less than 20%. Only 6% of the webpages in the Alexa 125 require more than 50% overhead to load the page. There is less than 5% difference in page load similarity for obout 95% of the webpages in the Alexa 125 as well. This assures that implementing ASD does not break the core of the browser and also assures that the overhead is within acceptable limits.

4.3 Security considerations: handling vulnerabilities

In this section, we intend to evaluate how our approach is useful in remediating the problems caused by vulnerabilities present in modern websites. It must be noted that ASD is not capable of directly eliminating the vulnerability, it is there to protect key information from being affected despite the presence of these vulnerabilities. However, information flow control in general, and especially our approach, provides solutions to the various problems mentioned in Section 1.2. In this section we list how ASD can handle these problems.

In all the examples listed below, we assume that there is legitimate code running on the system and that malicious code was inserted into the system. Such malicious code could be triggered by the presence of vulnerabilities such as cross-site scripting, malicious advertisements, malicious browser extensions and unverified libraries hosted by the server. Once any JavaScript is loaded on the DOM, it will be executed. Often, a tiny malicious script is first loaded using any of the mechanisms mentioned above and this script creates more `<script>`

DOM elements therefore extending its capabilities.

4.3.1 Protecting the Cross-Site Request Forgery Token

In the example listed in the code 4.1, a CSRF token is used to prevent CSRF. However, it is necessary to keep this string a secret from unnecessary functions in the page. This variable is then used in every subsequent request. The function XMLHttpRequest would only be able to use the secret value of `csrfToken` when the function `setUserFirstName` calls it.

It must be noted that Cross-Site Request Forgery is a vulnerability where the server is not able to distinguish between a legitimate and an illegitimate request, as explained earlier in Section 1.2.2. For example, if a malicious request was sent to make a bank transfer and the server processes the request, it could be devastating. In this example, this malicious action is done in the function `maliciousSetParameter`, that is supposed to be injected. This function is able to use the variable `csrfToken` to send another legitimate request to the server.

```
1  var csrfToken = 0;
2  function initToken()
3  {
4    csrfToken = 'jkhahhahsjhasdkkdhkcbh2e23u';
5  };
6  function setUserFirstName()
7  {
8    if(confirm('Set first name
      to:'+document.getElementById('EditFN').value+' ?'))
9    {
10     XMLHttpRequest('/setusername?fname='
11     + urlencode(document.getElementById('EditFN').value)
12     + '&csrftoken=' + csrfToken);
13   }
14 };
15 initToken();
16 setUserFirstName();
17 function maliciousSetParameter()
18 {
19    var xhr3 = new XMLHttpRequest();
20   xhr3.open("GET", '/setusername?fname=' +
      urlencode('badString') + '&csrftoken=' + csrfToken, false);
21   xhr3.send(null);
22   if (xhr3.status === 200) {
23       responseMessage = (xhr3.responseText);
```

```
24      }
25  };
```

Code 4.1: CSRS example

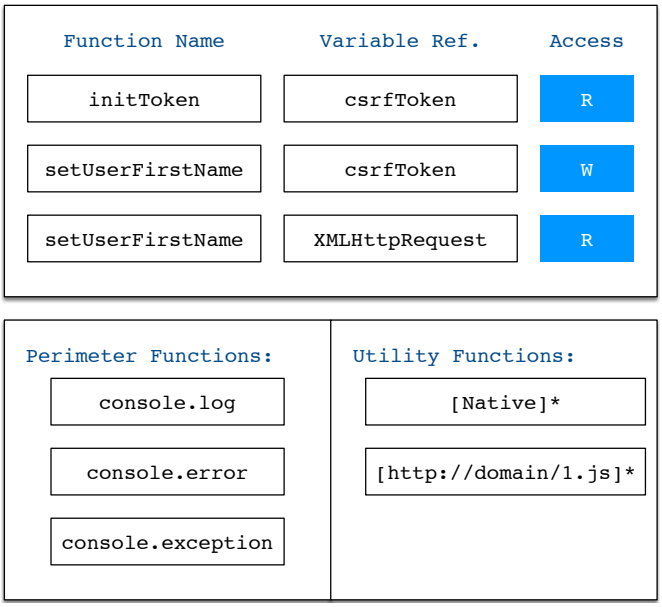


Figure 4.5: Policy for CSRF

In this example, we present a solution using ASD as shown in Figure 4.5, the functions `initToken` and `setUserFirstName` are the authorized functions to access the `csrfToken`. The function `initToken` is used to get the initiate the `csrftoken` using a hardcoded string. This string is set to different values by the server when the page is loaded. Hence, it is given write access to the variable. This token is used in the function `setUserFirstName`. Here, we consider the perimeter functions of console.log to be a public output with no privileges while XMLHttpRequest is used as a utility function. Since it is a utility function, XMLHttpRequest would take on the privileges of calling function at runtime. It must be noted that the XMLHttpRequest used here is also a split variable with the function `setUserFirstName` having read access to it. Since no policies for the function has been specified, the private part of the variable becomes a utility function. However, by convention of split functions, the public part of the variable becomes a self-sufficient function with no privileges. Hence, private part of XMLHttpRequest inherits the rights of the calling function a.k.a. `setUserFirstName`. However the public part cannot have any rights to inherit when accessed from the function `maliciousSetParameter`.

Hence, only specific privileges need to be given to specific functions and this would ensure

that only permitted actions would be performed. ASD is capable of handling this task. In this example, even if the malicious script tried to modify XMLHttpRequest, it would only affect the public part of XMLHttpRequest, making it futile to affect even the reference of the variable.

4.3.2 WebRTC

WebRTC is a modern technology with a specific recommendation that only trusted scripts should be allowed in the website since the various variables required for secure communication remain accessible to all scripts in the page. It is hence necessary to focus on protecting these variables from insecure scripts.

Let us consider the attack in Code 4.2. It can be seen that this piece of code first tries to overload the `onmessage` function of the `MessageChannel` which is referenced by the variable `mc`. This is done in the function `maliciousHook`. In this function, the event `onmessage` is changed to infer to malicious function. This function triggers `changeAllIdentities` which then changes the `sdp` object therefore effectively hijacking the connection.

```
1  // rpc is a RTCPeerConnection object
2  // mc is a MessageChannel object
3  function maliciousHook()
4  {
5    mc.port1.onmessage = function(e) {
6      newOffer.sdp = changeAllIdentities(e.data,mc.offer.sdp);
7      rpc.trueCallback(newOffer);
8    };
9  };
10 maliciousHook();
11 function changeAllIdentities(newIdentity,sdp){
12   identityExtraction = base64(newIdentity);
13   return sdp.replace(/identity:[A-Z0-9]*\n/g,
       'identity:'+identityExtraction);
14 };
15 function passMessage(message,mc)
16 {
17  ...
18 };
```

Code 4.2: WebRTC

Consider the policy as specified in the Figure 4.6. In this case, the `sdp` objects need to be secure and accessible only to authorized functions. ASD is capable of performing such an action on the web browser.

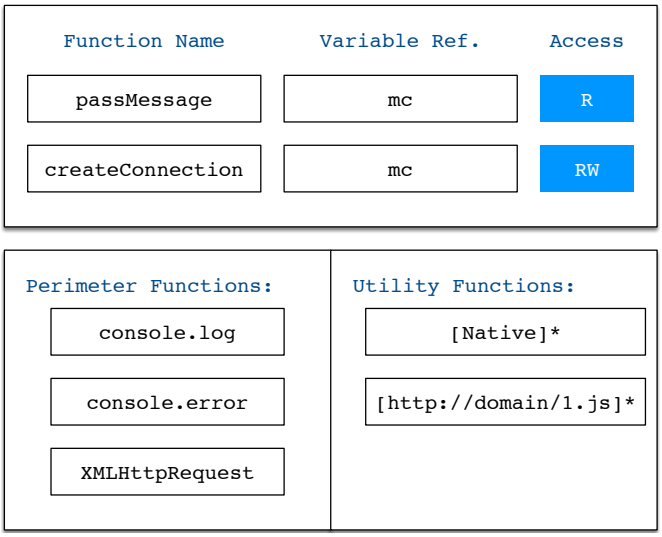


Figure 4.6: Policy for WebRTC

The use of our model allows protection of WebRTC objects from untrusted scripts in the page thereby being an effective remediation to this issue. In a similar manner we are also able to keep the connection objects of the Websockets safe as shown in the next subsection.

4.3.3 Websockets

WebSockets [FM11] is a modern HTML5 standard which makes communication between client and server a lot more simpler than ever. Modern technologies have made it possible to sandbox the browser on a whole new scale thereby making it possible for riskier modes of access to be made. Hence a lot of newer technologies have been introduced including WebSockets. WebSockets is slightly different than the standard TCP or UDP socket implementation. In this case, the protocol is still HTTPS and this makes it more versatile than opening a TCP connection on another port since WebSockets would have no problems in working with HTTP proxies. The protocol is itself symbolized by ws[s]:// and it keeps the established connection open to continuously send or receive messages.

Once a connection is established, it is kept open for further transmission of data. It must be noted that authentication is completed when the connection is established and hence a secure channel is available to the web page for further communication.

In the example shown in Code 4.3, the function `maliciousJS` uses an established connection `websocketConnection` to pass a message posing as the user. There are four functions in this program. Three of these are legitimate functions and there is a malicious function represented by `maliciousJS`. The connection is first created when the `initiateConnection` is

called to start the connection to the host. The functions `onGotMessage` and `onWSOpen` are registered to the events of the connections. The maliciousJS here, tries to send a message using the sensitive channel.

```
1  var websocketConnection;
2  function onGotMessage(evt)
3  { console.log("Received from server: " + evt.data); };
4  function onWSOpen() {
5      console.log("Connected to " + malwsUri);
6      websocketConnection.send('hello');
7  };
8  function initiateConnection()
9  {
10 var wsUri = "wss://" + document.location.host + "/wsendpoint";
11 websocketConnection = new
       WebSocket(wsUri+'?csrftoken='+csrftoken);
12 websocketConnection.onmessage = onGotMessage;
13 websocketConnection.onopen = onWSOpen;
14 };
15 initiateConnection();
16 function maliciousJS()
17 {
18  websocketConnection.send('Try this app at
       http://malicious.com/app');
19 };
20 maliciousJS();
```

Code 4.3: jsonp.js

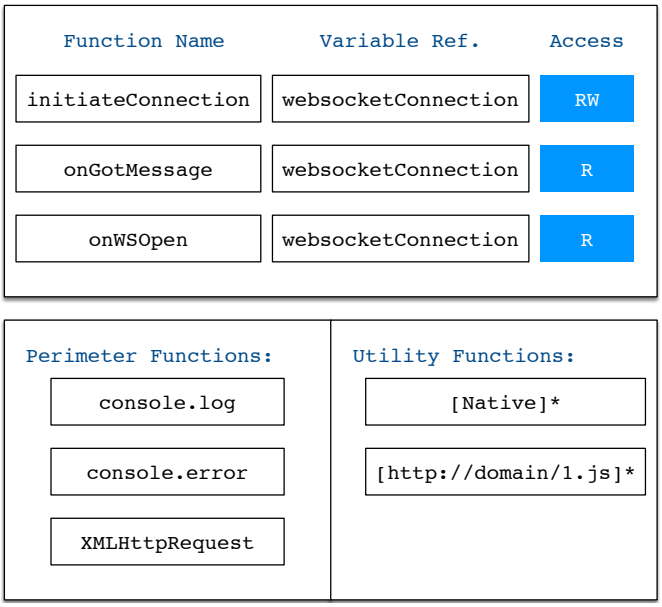


Figure 4.7: Policy for WebSockets

The policy shown in the Figure 4.7 provides a possible remediation using ASD. The policy intends to make the `websocketConnection` a secret variable thereby avoiding this misuse by the function `maliciousJS`. Once the variable `websocketConnection` is made a secret, the function `maliciousJS` would not have access to it.

Hence, is can be stated that the ASD model is able to protect variables effectively from access from malicious scripts in the webpage. We prove this in multiple critical scenarios on the modern web by protecting the csrf token, the secure link between two clients in WebRTC and then the communication channel to a server in case of web sockets. Further, ASD is also able to protect core functions from modifications. We conclude that ASD can be an efficient and effective solution to handle the common problems in the web.

Conclusion

This chapter concludes our thesis.We first sum-up the results we have achieved and then give some perspective and future work.

Results

The objective of our work has been to provide a methodology that provides information flow control for the web browser. The main characteristic we intended for our approach is to continue further execution without blocking and to have a limited impact on performance. Another important consideration was to provide a mechanism to prevent unauthorized functions from modifying secret variables.

Over the course of our work, we have proposed the Address Split Design (ASD) which focuses on splitting the memory into two different locations to maintain the two possible states for each secret variable. Based on the context of the operation, the approach switches between the public and the secret states. We choose to have a model that focuses on more fine-tuned function level control instead of traditional approaches. In this case, we consider that each function needs to be given access rights and not just the public output functions as is the case in traditional approaches. Since each function needs to be provided specific access rights to the split variables, it allows a policy where functions can be given access to only a subset of variables to which they are expected to have read access.

Since our approach provides access rights, functions can only modify secret variables if they have such privileges. This is effective against malicious scripts inserted via vulnerabilities such as cross-site scripting. This mechanism is unique to ASD and is only feasible due to the function level control.

We have also made a formal description of our approach and implemented this model on the chromium V8 engine, a full-fledged JavaScript engine and subsequently ported these changes to the Chromium web browser as well. Following the implementation, performance and conformance testing were done on our implementation. The measured performance drop

is significantly smaller than other comparative approaches. We further showed that implementation of our approach does not affect the general working of existing websites by performing such a test over the top websites of the internet. Further, we have also been able to verify that our model can be used to protect variables in several scenarios that would have otherwise caused disclosure of secret information.

Open questions and future work

While we have accomplished the objectives of our work, questions still remain on how to improve the model and make it more feasible to be adapted by the larger audience.

Over the course of the work, we use a data structure called the dependency tracker to keep track of the secrets that could influence the information flow. The dependency tracker makes some over-approximations on implicit information flow to forsake additional computation. To make the approach more practical, we discard the over-approximations in the dependency tracker at the end of the execution of the function. By choosing to do so, we forgo providing any strong security guarantees for the model. It is possible to have had a strong security guarantee if we had maintained the over-approximations throughout the execution but such a choice would have resulted in a largely impractical approach as the over-approximations would augment over time.

Therefore the open question remains on how to remain practical as well as improve the security guarantees provided by our model. The solution to the problem, with regard to ASD is to make all variables attain the same state (split/unsplit) regardless of the path taken. One of the solutions would be to use machine learning to learn about the variables that are dynamically split over different executions of the function thereby analyzing the different possible execution paths. We believe that using such a model, eventually, all paths could be analyzed. In this approach, the possible leaks in ASD for a given function would decrease as the function is executed multiple number of times eventually tending towards zero. Such a mechanism could slowly improve the guarantees provided over time.

Further, we also think that the information flow mechanism we are using, which is possibilistic in nature, could be extended to assimilate probabilistic information flows. One solution is to prevent information flow from splitting variables if the information leakage is less than a particular entropy.

Finally, we also believe that the performance can be enhanced by sharing dictionaries between several functions and implementing the model in other more efficient compilers.

Acronyms

ASD Address split design. 61, 69, 84, 95, 98–100, 103–107, 109, 110, 112, 113, 115, 117, 118, 121, 123, 124, 142

BNF Backus–Naur form. 63

CORS Cross-Origin Resource Sharing. 25

CSRF Cross-Site Request Forgery. 6, 16, 19, 22–24, 27, 28, 116

DOM Document Object Model. 13, 18, 19, 26, 28, 98, 115, 116

DTLS Datagram Transport Layer Security. 14

HTTP Hyper-Text Transport Protocol. 10, 11, 23, 25, 119

IdP Identity Provider. 14, 15, 20

IFC Information Flow Control. 8, 28, 29, 31, 32, 34–36, 38, 40, 44, 55–57, 59, 61, 63, 64, 89, 95

MAC Mandatory Access Control. 30, 31

OSI Open Systems Interconnection. 14

SDP Session Description Protocol. 15, 21, 141

SME Secure Multi-Execution. 47–50, 55–59, 91, 94, 104, 105, 142

SOP Same-Origin Policy. 25, 26

SRTP Secure Real-time Transport Protocol. 14, 15

SSF Self-Sufficient Function. 66

TCP Transmission Control Protocol. 10, 14, 119

UDP User Datagram Protocol. 14

WebRTC Web Real-Time Communication. 9, 13–16, 20–22, 119, 141

XSS Cross-Site Scripting. 6, 16–20, 22–24, 26, 27, 95, 141

Bibliography

[AAP10] Mário S. Alvim, Miguel E. Andrés, and Catus Palamidessi. Probabilistic Information Flow. In 25th Annual IEEE Symposium on Logic in Computer Science, pages 314–321. IEEE, July 2010. doi:10.1109/LICS.2010.53.

[ACPS12] Mário S. Alvim, Kostas Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring Information Leakage Using Generalized Gain Functions. In 2012 IEEE 25th Computer Security Foundations Symposium, volume 0, pages 265–279. IEEE, June 2012. doi:10.1109/CSF.2012.26.

[ADO04] ADOC. Une nouvelle méthode d'organisation de pot. Revue Française pour une soutenance de thèse bien arro... réussie, page 1, 2004.

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. ACM SIGPLAN Notices, 44(8):20, December 2009. doi:10.1145/1667209.1667223.

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Proceedings of the 13th European Symposium on Research in Computer Security, pages 333 – 348. Springer-Verlag Berlin, Heidelberg, 2008. doi:10.1007/978-3-540-88313-5_22.

[Ald06] Jonathan Aldrich. Semantics of WHILE, 17-654/17-754: Analysis of Software Artifacts, 2006. URL: http://www.cs.cmu.edu/{~}aldrich/courses/654-sp06/notes/2-semantics-notes.pdf.

[Alv14] H. Alvestrand. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft draft-ietf-rtcweb-overview-13, Internet Engineering Task Force, November 2014. Work in progress. URL: http://tools.ietf.org/html/draft-ietf-rtcweb-overview-13.

[AN16] M. Assaf and D. A. Naumann. Calculational design of information flow monitors. In 2016 IEEE 29th Computer Security Foundations Symposium (CSF), pages 210–224, June 2016. doi:10.1109/CSF.2016.22.

[Ang] AngularJS. AngularJS. URL: https://angularjs.org/.

[AR80] Gregory R. Andrews and Richard P. Reitman. An axiomatic approach to information flow in programs. ACM Trans. Program. Lang. Syst., 2(1):56–76, jan 1980. URL: http://doi.acm.org/10.1145/357084.357088, doi:10.1145/357084.357088.

[Aus] Thomas H. Austin. Original ZaphodFacets. URL: https://github.com/taustin/ZaphodFacets.

[Aus13] TH Austin. Dynamic information flow analysis for Javascript in a web browser. PhD thesis, University of California, Santa Cruz, 2013.

[Bac05] Michael Backes. Quantifying Probabilistic Information Flow in Computational Reactive Systems. In Proceedings of 10th European Symposium on Research in Computer Security (ESORICS), pages 336–354, Milan, Italy, 2005. Springer. URL: http://link.springer.com/chapter/10.1007%2F11555827_20, doi:10.1007/11555827_20.

[BBC14] V. Beltran, E. Bertin, and N. Crespi. User identity for webrtc services: A matter of trust. Internet Computing, IEEE, 18(6):18–25, Nov 2014. doi:10.1109/MIC.2014.128.

[BBJ13] Frédéric Besson, Nataliia Bielova, and Thomas Jensen. Hybrid Information Flow Monitoring Against Web Tracking. Computer Security Foundations Symposium (CSF), 2013. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6595832.

[BBJN15] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-Time Communication Between Browsers. W3C Editor's Draft, 2015.

[BDD+14] Bert Bos, Elwyn Davies, Lieven Desmet, Stephen Farrell, Martin Johns, and Rigo Wenning. D1.2 Case Study: Security Assessment of WebRTC. Technical report, STREWS project consortium, 2014. https://www.strews.eu/images/webrtc.pdf. URL: https://www.strews.eu/images/webrtc.pdf.

[BDHK07] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. International Journal of Information Security, 7(2):155–169, October 2007. URL: http://link.springer.com/10.1007/s10207-007-0046-6, doi:10.1007/s10207-007-0046-6.

[Bib75] K. J. Biba. Integrity Considerations for Secure Computer Systems. Technical report, The Mitre Corporation, 1975.

[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, 2013. doi:10.1016/j.jlap.2013.05.001.

[Bit] The problems and some security implications of websockets - Cross-site WebSockets Scripting (XSWS). http://subudeepak.bitbucket.org/#!/reading/0. [Online; accessed November 01, 2015].

[BL73] DE Bell and LJ LaPadula. Secure Computer Systems : Mathematical Foundations. Technical report, dtic.mil, 1973. URL: http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0770768.

[BMN+04a] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol. RFC 3711, Mar 2004. URL: https://tools.ietf.org/html/rfc3711.

[BMN+04b] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004. Updated by RFCs 5506, 6904. URL: http://www.ietf.org/rfc/rfc3711.txt.

[BP04] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. International Journal of Information Security, 3(1):42–60, July 2004. URL: http://link.springer.com/10.1007/s10207-004-0039-7, doi:10.1007/s10207-004-0039-7.

[BS06] N. Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In Lecture Notes in Computer Science, pages 180–196, 2006.

[BS10] Niklas Broberg and David Sands. Paralocks: Role-based information flow control and beyond. In Proceedings of the 37th Annual ACM SIGPLAN-SIGACT

Symposium on Principles of Programming Languages, POPL '10, pages 431–444, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1706299.1706349, doi:10.1145/1706299.1706349.

[BT14] R.L. Barnes and M. Thomson. Browser-to-browser security assurances for webrtc. Internet Computing, IEEE, 18(6):11–17, Nov 2014. doi:10.1109/MIC.2014.106.

[CBR11] Eric Y Chen, Jason Bau, and Charles Reis. App isolation: get the security of multiple browsers with just one. In Proceedings of the 18th ACM conference on Computer and communications security, pages 227–238. ACM, 2011. doi:10.1145/2046707.2046734.

[CC77] Patrick Cousot and Radhia Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by constuction or approzimation of fixpoints. In ACM Symposium on Principles of Programming Languages, pages 238–252, 1977.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. Journal of Logic and Computation, 2(4):511–547, 1992. URL: http://logcom.oxfordjournals.org/content/2/4/511.abstract, arXiv:http://logcom.oxfordjournals.org/content/2/4/511.full.pdf+html, doi:10.1093/logcom/2.4.511.

[Ced14] Jorgen Cederlof. Authentication in quantum key growing. PhD thesis, Linkopings Universitet, 2014. URL: http://www.lysator.liu.se/~jc/mthesis/mthesis.pdf.

[CF07] Deepak Chandra and Michael Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pages 463–475. IEEE, December 2007. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4413012, doi:10.1109/ACSAC.2007.37.

[CHM07] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security, 15(3):321–371, 2007.

[CMJL09] Ravi Chugh, Jeffrey a. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In Proceedings of the 2009 ACM SIGPLAN conference on

Programming language design and implementation, volume 44, pages 50–62, May 2009. doi:10.1145/1543135.1542483.

[Cou] Patrick Cousot. Abstract interpretation in a nutshell. URL: http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html.

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

[Den75] Dorothy Elizabeth Robling Denning. Secure Information Flow in Computer Systems. PhD thesis, Purdue University, West Lafayette, IN, USA, 1975. AAI7600514.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. Commun. ACM, 19(5):236–243, May 1976. URL: http://doi.acm.org/10.1145/360051.360056, doi:10.1145/360051.360056.

[DFST02] Nicoletta De Francesco, Antonella Santone, and Luca Tesei. Abstract interpretation and model checking for checking secure information flow in concurrent systems. Fundam. Inf., 54(2-3):195–211, June 2002. URL: http://dl.acm.org/citation.cfm?id=873906.873913.

[DG09] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In Annual Computer Security Applications Conference 2009, pages 382–391, 2009. doi:10.1109/ACSAC.2009.43.

[DGSJ+16] Willem De Groef, Deepak Subramanian, Martin Johns, Frank Piessens, and Lieven Desmet. Ensuring endpoint authenticity in webrtc peer-to-peer communication. In Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, pages 2103–2110, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2851613.2851804, doi:10.1145/2851613.2851804.

[DJ14] Lieven Desmet and Martin Johns. Real-time communications security on the web. Internet Computing, IEEE, 18(6):8–10, Nov 2014. doi:10.1109/MIC.2014.117.

[DP10] Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. 2010 IEEE Symposium on Security and Privacy, pages 109–124, 2010. doi:10.1109/SP.2010.15.

[ECM15a] ECMA. Draft ECMAScript 2015 Language Specification (RC4). [online], http://people.mozilla.org/~jorendorff/es6-draft.html, April 2015.

[Ecm15b] Ecma International. ECMAScript 2015 Language Specification, 2015. URL: http://www.ecma-international.org/ecma-262/6.0/.

[Emb] EmberJS. EmberJS. URL: http://emberjs.com/.

[Epi] Epic Privacy Browser. URL: https://www.epicbrowser.com/.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Jun 1999. URL: https://tools.ietf.org/html/rfc2616#section-1.3.

[FM11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, Dec 2011. URL: https://tools.ietf.org/html/rfc6455.

[For] Chromium Forum. Bug tracker on Chromium: Basic Authentication. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=82250#c7.

[Fut] Peacekeeper Benchmark. URL: http://peacekeeper.futuremark.com/faq.action.

[GDNP12] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. FlowFox: a web browser with flexible and precise information flow control. In Proceedings of the 2012 ACM conference on Computer and communications security, pages 748—-759, Raleigh, North Carolina, USA, 2012. ACM. doi:10.1145/2382196.2382275.

[GM82] J. A. Goguen and J. Mesajuer. Security policies and Security Models. In IEEE Symposium on Security and Privacy, pages 11–20, 1982.

[Goo] Octane Benchmark. URL: https://developers.google.com/octane/.

[HCS+10] Steve Hanna, Eui Chul, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperor's new apis: On the (in) secure usage of new client-side primitives. In Web 2.0 Security and Privacy (W2SP 2010), 2010. URL: http://www.eecs.berkeley.edu/~sch/w2sp2010ena.pdf.

[Hic15a] Ian Hickson. HTML5 Web Messaging. W3C Recommendation, 2015. http://www.w3.org/TR/webmessaging/.

[Hic15b] Ian Hickson. The Cross-Document Messaging Standard, May 2015. URL: https://www.w3.org/TR/webmessaging/.

[HMM+12] T. S. Hoang, A. K. McIver, L. Meinicke, C. C. Morgan, A. Sloane, and E. Susatyo. Abstractions of non-interference security: probabilistic versus possibilistic. Formal Aspects of Computing, 26(1):169–194, June 2012. URL: http://link.springer.com/10.1007/s00165-012-0237-4, doi:10.1007/s00165-012-0237-4.

[HR98] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM. URL: http://doi.acm.org/10.1145/268946.268976, doi:10.1145/268946.268976.

[HRU76] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. Commun. ACM, 19(8):461–471, aug 1976. URL: http://doi.acm.org/10.1145/360303.360333, doi:10.1145/360303.360333.

[HS12a] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Benedikt Hauptmann Tobias Nipkow, Orna Grumberg, editor, NATO Science for Peace and Security Series - D: Information and Communication Security, volume 33: Software Safety and Security, pages 319 – 347. IOS Press, 2012. doi:10.3233/978-1-61499-028-4-319.

[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, June 2012. doi:10.1109/CSF.2012.19.

[HSY+15] Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In Conference on Principles of Security and Trust (POST). Springer, April 2015.

[HTML07] Guillaume Hiet, Triem Tong, Benjamin Morin, and M Ludovic. Monitoring both OS and Program Level Information Flows to Detect Intrusions against Network Servers. In IEEE Workshop on "Monitoring, Attack Detection and Mitigation", 2007.

[JDS+03] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) Basic Call Flow Examples. RFC 3665 (Best Current Practice), December 2003. URL: http://www.ietf.org/rfc/rfc3665.txt.

[khr] WebGL Security. URL: http://www.khronos.org/webgl/security/.

[Kno] KnockoutJS. KnockoutJS. URL: http://knockoutjs.com/.

[KWH11] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In 2011 IEEE Symposium on Security and Privacy, pages 413–428. IEEE, May 2011. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5958043, doi:10.1109/SP.2011.19.

[LCQC14] Li Li, Wu Chou, Zhihong Qiu, and Tao Cai. Who is calling which page on the web? Internet Computing, IEEE, 18(6):26–33, Nov 2014. doi:10.1109/MIC.2014.105.

[LGBJS07] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues, ASIAN'06, pages 75–89, Berlin, Heidelberg, 2007. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1782734.1782741.

[LGJ07] Gurvan Le Guernic and Thomas Jensen. Monitoring Information Flow. In Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSFS20), pages 19–30. IEEE Computer Society, jul 2007.

[Mas94] James L Maseey. Guessing and Entropy. In Proceedings of International Symposium on Information Theory, page 204. IEEE, 1994.

[Mas05] Isabella Mastroeni. Abstract Non-Interference - An Abstract Interpretation-based approach to Secure Inform
PhD thesis, Universit‘ a degli Studi di Verona, 2005.

[Mit] CWE - CWE-200: Information Exposure.

[ML97] Andrew C Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. In Proceedings of the 16th ACM Symposium on Operating System Principles, October 1997.

[MMR10] R. Mahy, P. Matthews, and J. Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 5766 (Proposed Standard), April 2010. URL: http://www.ietf.org/rfc/rfc5766.txt.

[Moza] HTML5 - Web developer guide. URL: https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5.

[Mozb] Dromaeo Benchmark. URL: https://wiki.mozilla.org/Dromaeo.

[Mozc] Kracken Benchmark. URL: https://wiki.mozilla.org/Kraken.

[Mye99] Andrew C Myers. JFlow : Practical Mostly-Static Information Flow Control. In Symposium on Principles of Programming Languages, pages 228–241, January 1999.

[Naf] Ahamed Nafeez. JS Suicide: Using Javascript security features to kill itself. Presented at BlackHat Asia 2014. URL: https://www.blackhat.com/asia-14/briefings.html#Nafeez.

[NCC+04] Efstratios Nikolaidis, Sophie Chen, Harley Cudney, Raphael Haftka, and Raluca Rosca. Comparison of probability and possibility for design against catastrophic failure under uncertainty. 126, 05 2004.

[Opea] Category:OWASP Top Ten Project - OWASP. URL: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[Opeb] Clickjacking - OWASP. URL: https://www.owasp.org/index.php/Clickjacking.

[Opec] Cross-Site Request Forgery (CSRF) - OWASP. URL: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[Oped] Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - OWASP. URL: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet.

[Opee] Cross Site Scripting Flaw - OWASP. URL: https://www.owasp.org/index.php/Cross_Site_Scripting_Flaw.

[Opef] DOM Based XSS. URL: https://www.owasp.org/index.php/DOM_Based_XSSl.

[PF06] Stefano Di Paola and Giorgio Fedon. 23 rd ccc conference subverting ajax 1, 2006.

[Pli00] John O Pliam. On the Incomparability of Entropy and Marginal Guesswork in Brute-Force Attacks. In Progress in Cryptology - INDOCRYPT 2000, First International Conference in Cryptology in India, pages 67–79, 2000. doi:10.1007/3-540-44495-5_7.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ml. ACM Trans. Program. Lang. Syst., 25(1):117–158, January 2003. URL: http://doi.acm.org/10.1145/596980.596983, doi:10.1145/596980.596983.

[Ren60] Alfred Renyi. ON MEASURES OF ENTROPY AND INFORMATION. In Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics, pages 547–561, 1960.

[Res15a] E. Rescorla. Security Considerations for WebRTC. Internet-Draft draft-ietf-rtcweb-security-08, Internet Engineering Task Force, February 2015. Work in progress. URL: http://tools.ietf.org/html/draft-ietf-rtcweb-security-08.

[Res15b] E. Rescorla. WebRTC Security Architecture. Internet-Draft draft-ietf-rtcweb-security-arch-11, Internet Engineering Task Force, March 2015. Work in progress. URL: http://tools.ietf.org/html/draft-ietf-rtcweb-security-arch-11.

[Ric53] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society, 74(2):358–366, 1953. URL: http://www.jstor.org/stable/1990888.

[RM12] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012. URL: http://www.ietf.org/rfc/rfc6347.txt.

[RMMW08] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389 (Proposed Standard), October 2008. Updated by RFC 7350. URL: http://www.ietf.org/rfc/rfc5389.txt.

[Ros10] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. RFC 5245 (Proposed Standard), April 2010. Updated by RFC 6336. URL: http://www.ietf.org/rfc/rfc5245.txt.

[RS02] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157. URL: http://www.ietf.org/rfc/rfc3264.txt.

[Ré60] Alfred Rényi. ON MEASURES OF ENTROPY AND INFORMATION. In Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics, pages 547–561, 1960.

[Ré61] Alfréd Rényi. On measures of entropy and information. In Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, volume 1, pages 547–561, 1961.

[SA11] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011. URL: `http://www.ietf.org/rfc/rfc6120.txt`.

[Sey15] Ryan Seys. Mozilla Tin Can, 2015. URL: `https://addons.mozilla.org/en-us/firefox/addon/tin-can-auth`.

[Sha48] CE Shannon. A Mathematical theory of communication. Bell System Technical Journal, 27:379–423,625–656, 1948.

[SHB] Deepak Subramanian, Guillaume Hiet, and Christophe Bidan. Inflow code base. URL: `https://gforge.inria.fr/git/inflow/inflow.git`.

[SHB16] Deepak Subramanian, Guillaume Hiet, and Christophe Bidan. Preventive information flow control through a mechanism of split addresses. In ACM 9th International Conference on Security of Information and Networks 2016. ACM, july 2016. `doi:10.1145/2947626.2947645`.

[SHB17] Deepak Subramanian, Guillaume Hiet, and Christophe Bidan. A Self-correcting Information Flow Control Model for the Web-Browser, pages 285–301. Springer International Publishing, Cham, 2017. URL: `http://dx.doi.org/10.1007/978-3-319-51966-1_19`, `doi:10.1007/978-3-319-51966-1_19`.

[SM03] Andrei Sabelfeld and A.C. Myers. Language-based information-flow security. IEEE Journal on Selected Areas in Communications, 21(1):5–19, January 2003. `doi:10.1109/JSAC.2002.806121`.

[SS98] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. Higher-order and symbolic computation, 14:40–58, 1998. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.2737`.

[SS13] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In Network and Distributed System Security Symposium (NDSS'13), 2013.

[Ste07] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007. Updated by RFCs 6096, 6335, 7053. URL: `http://www.ietf.org/rfc/rfc4960.txt`.

[STK09] Deepak Subramanian, Ha Thanh, and Kok Keong. Assuring Quality in Vulnerability Reports for Security Risk Analysis. International Journal on Advances in Security, 2(2 & 3):226–241, 2009.

[Sub] Deepak Subramanian. Modified Zaphod-Facets — Bitbucket. URL: `https://bitbucket.org/subudeepak/zaphod-facets`.

[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 Symposium on Operating Systems Design and Implementation (OSDI). USENIX, October 2014.

[Tec11] Technical Committee : ISO/IEC JTC 1/SC 22. ISO/IEC 16262:2011 - Information technology – Programming languages, their environments and system software interfaces – ECMAScript language specification, 2011. URL: `https://www.iso.org/standard/55755.html`.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. J. Comput. Secur., 4(2-3):167–187, January 1996. URL: `http://dl.acm.org/citation.cfm?id=353629.353648`.

[vK14] Anne van Kesteren. Cross-origin resource sharing. W3c technical reports, W3C, jan 2014. URL: `https://www.w3.org/TR/cors/`.

[VS97a] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In Proceedings of the 10th IEEE Workshop on Computer Security Foundations, CSFW '97, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society. URL: `http://dl.acm.org/citation.cfm?id=794197.795081`.

[VS97b] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In Proceedings of the 7th International Joint Conference CAAP/FASE

on Theory and Practice of Software Development, TAPSOFT '97, pages 607–621, London, UK, UK, 1997. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=646620.697712.

[W3C] Confinement with Origin Web Labels - W3C working draft. URL: https://www.w3.org/TR/COWL.

[WBV15] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Working Draft, W3C, jul 2015. Work in progress. http://www.w3.org/TR/2015/CR-CSP2-20150721/.

[Weba] DOM Based XSS - report (Web Application Security Consortium). URL: http://www.webappsec.org/projects/articles/071105.shtml.

[Webb] JetStream Benchmark. URL: https://webkit.org/perf/sunspider/sunspider.html.

[Webc] SunSpider Benchmark. URL: https://webkit.org/perf/sunspider/sunspider.html.

[WFHJ07] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07, pages 1–16, New York, New York, USA, 2007. ACM Press. doi:10.1145/1294261.1294263.

[Whi] WhiteHat Aviator. URL: https://www.whitehatsec.com/aviator/.

[Wika] Acid3. URL: https://en.wikipedia.org/wiki/Acid3.

[Wikb] Bank card number. URL: http://en.wikipedia.org/wiki/Bank_card_number.

[Wikc] JSONP. http://en.wikipedia.org/wiki/JSONP. [Online; accessed November 01, 2013].

[Wikd] Mashup (web application hybrid). URL: https://en.wikipedia.org/wiki/Mashup_%28web_application_hybrid%29.

[Wike] Narcissus (JavaScript engine). URL: http://en.wikipedia.org/wiki/Narcissus_(JavaScript_engine).

[Wikf] Oracle machine. URL: http://en.wikipedia.org/wiki/Oracle_machine.

[Wikg] PAN Truncation. URL: http://en.wikipedia.org/wiki/PAN_truncation.

[Wikh] Same-Origin Policy. URL: http://en.wikipedia.org/wiki/Same-origin_policy.

[Wiki] Site-specific browser. URL: http://en.wikipedia.org/wiki/Site-specific_browser.

[XSS] Excess-XSS. URL: https://excess-xss.com/#xss-attacks.

[Zan12] Matteo Zanioli. Information Flow Analysis by Abstract Interpretation. PhD thesis, Universit‘ a Ca’ Foscari di Venezia Via, 2012.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In in Proc. IEEE Computer Security Foundations Workshop, pages 15–23. IEEE Computer Society Press, 2001.

List of Figures

1.1 Uniform Resource Locator . . . 12
1.2 The process of loading a page . . . 13
1.3 Simple eval function example . . . 14
1.4 A typical webpage . . . 15
1.5 Simple architectural view of WebRTC . . . 16
1.6 WebRTC architecture based on [Res15b] . . . 17
1.7 WebRTC integration of the Identity Provider. . . . 18
1.8 OM based XSS example [Opef] . . . 21
1.9 Example attack showing how to compromise the certificate fingerprint by replacing the SDP offer with an attacker-controlled version. . . . 23
1.10 Example attack showing how to modify the identity string to a fake identity. . . . 24

2.1 Partial order on information . . . 32
2.2 Explicit/Implicit Flow . . . 34
2.3 Information Flow Working . . . 35
2.4 Information flow code example . . . 36
2.5 Termination-insensitive non-interference . . . 39
2.6 Timing- and Termination- sensitive non-interference . . . 40
2.7 Secure-type system [SM03, VIS96] . . . 43
2.8 Abstract interpretation semantics [DFST02] . . . 45
2.9 Label based taint marking . . . 47
2.10 Indirect flows in no-sensitive upgrade . . . 48
2.11 Secure Multi-Execution . . . 49
2.12 Indirect flows in SME . . . 50
2.13 Faceted Approach . . . 51
2.14 Faceted evaluation . . . 52
2.15 Faceted Approach - implicit flow . . . 53

2.16 Hybrid Information Flow Control with the assignment rule 55
2.17 Hybrid Information Flow Control with the conditional rule 56
2.18 Hybrid Information flow without the conditional rule 57
2.19 Example of references to same object in JavaScript 58
2.20 XMLHttpRequest in SME 60

3.1 The address split design 64
3.2 Policy BNF grammar 66
3.3 Dictionary example 67
3.4 Working of utility functions 69
3.5 Defining the Perimeter using Self-Sufficient Functions 70
3.6 The various access rights 72
3.7 Simple Information flow example 84
3.8 ASD Limitation Example 85
3.9 Example for flow 87
3.10 A detailed Policy Specification 88
3.11 Example for flow 89
3.12 A detailed Policy Specification 90
3.13 Dictionary of function `compute2` 90
3.14 Dictionary of function `staticFunction` 91
3.15 Dictionary of function `compute` 91
3.16 Dictionary of function `init` 91
3.17 Dictionary of function `compute2` 92
3.18 Dictionary of function `staticFunction` 92
3.19 Comparison between various approaches - Case h(x=false) 94
3.20 Comparison between various approaches - Case h(x=true) 95

4.1 Performance tests of V8 vs ASD.V8 105
4.2 Percentage difference between the similarities by original Chromium and ASD Chromium 109
4.3 Percentage difference between the page load times of de-optimized Chromium and ASD Chromium 115
4.4 Standard benchmark comparisons between Original Chromium, Modified Chromium and Chromium-ASD 116
4.5 Policy for CSRF 119
4.6 Policy for WebRTC 121
4.7 Policy for WebSockets 123

Résumé

Le monde moderne a évolué au point où de nombreux services tels que la banque et le shopping sont fournis grâce aux applications web. Ces applications Web dépendent de logiciels reposant sur le modèle client-serveur. Parce que ces applications Web fournissent aux utilisateurs des services sensibles tels que la banque et le shopping, leur sécurité est d'une importance cruciale. Du côté serveur, la gamme des menaces de sécurité comprend des attaques telles que le déni de service, la mauvaise configuration de sécurité et l'injection de code malveillant par exemple, l'injection SQL). Du côté client, la majeure partie des problèmes de sécurité relève du navigateur Web qui est l'interface entre les utilisateurs et l'application côté serveur: comme n'importe quel logiciel, il peut être sujet à des attaques telles que des dépassements de tampon.

Cependant, il n'est pas suffisant d'empêcher indépendamment les menaces de sécurité côté client et côté serveur, car certains problèmes de sécurité des applications Web sont intrinsèques aux applications Web elles-mêmes. Par exemple, dans l'Internet moderne, une page Web se compose de plusieurs pages Web aggrégées. Cette aggrégation de pages Web permet de construire une application Web qui utilise le contenu de plusieurs sources pour créer un seul nouveau service accessible via une interface graphique unique. Plus généralement, la difficulté de la sécurité des applications web réside dans le fait que l'exploitation d'une vulnérabilité côté serveur peut avoir un impact côté client, et inversement. Il est à noter que de nombreuses vulnérabilités côté serveur telles que Cross-Site Scripting (XSS) et Cross-Site Request Forgery (CSRF) ont un impact direct sur le navigateur Web.

Dans cette thèse, nous nous concentrons sur la sécurité côté client, c'est-à-dire des navigateurs web, et nous nous limitons au contexte de Javascript. Nous ne considérons pas la résolution des vulnérabilités elles-mêmes, mais fournissons un mécanisme dans lequel les informations sensibles de l'utilisateur sont protégées de la divulgation (confidentialité) ainsi que des modifications non autorisées (intégrité) malgré la vulnérabilité exploitée. À cet effet, nous affirmons que les vulnérabilités basées sur des scripts malveillants sont caractérisées par des flux d'informations illégaux. Par conséquent, nous proposons une approche basée sur le contrôle du flux d'information (IFC - Information Flow Control). En effet, les approches basées sur IFC sont plus globales dans leur portée pour résoudre les problèmes et fournissent également des solutions plus simples pour gérer la sécurité de l'information dans son intégralité. Notre approche est basée sur un modèle IFC concret, appelé Address Split Design (ASD), qui consiste à séparer toute variable contenant des données sensibles et à maintenir la table de symboles pour protéger les accès à la partie secrète de telles variables. Nous avons implémenté notre modèle dans le moteur V8 chrome, un moteur JavaScript à part entière. Après la mise en œuvre, des tests de performance et de conformité ont été effectués sur notre implémentation. La baisse de

performance mesurée est significativement plus faible que d'autres approches comparatives. Nous avons également démontré que la mise en œuvre de notre approche n'affecte pas le fonctionnement général des sites Web existants en effectuant un test sur les principaux sites Web d'Internet. De plus, nous avons également pu vérifier que notre modèle peut être utilisé pour protéger des variables dans plusieurs scénarios qui auraient autrement provoqué la divulgation d'informations secrètes.

Abstract

The modern world has evolved to the point where many services such as banking and shopping are provided thanks to web applications. These Web applications depend on server-side as well as client-side software. Because these web applications provide to users sensitive services such as banking and shopping, their security is of pivotal importance. From the server side, the range of the security threats includes attacks such as denial of service, security misconfiguration and injection of malicious code (i.e. SQL injection). From the client side, major part of the security issues come with the web browser that is the interface between the users and server side application: as any software, it can be subject to attacks such as buffer overflows.

However, it is not sufficient to independently prevent security threats from each side, because some security issues of web applications are intrinsic to the web applications themselves. For instance, the modern internet consists of several webpages which are mashup webpages. A mashup, in web development, is a web page, or web application, that uses content from more than one source to create a single new service displayed in a single graphical interface. More generally, the difficulty of web application security lies in the fact that exploiting a server-side vulnerability can have a client-side impact, and vice versa. It must be noted that many vulnerabilities on the server side such as Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF) have a direct impact on the web browser.

In this thesis, we focus on the client side security of the web browsers, and limit ourselves to the context of Javascript. We do not consider solving the vulnerabilities themselves but providing a mechanism where user's sensitive information is protected from disclosure (confidentiality) as well as unauthorized modifications (integrity) despite the vulnerability being exploited. For that purpose, we affirm that the vulnerabilities based on malicious script are characterized by illegal information flows. Hence, we propose an approach based on Information Flow Control (IFC). Indeed, IFC-based approaches are more encompassing in their scope to solve problems and also provide more streamlined solutions to handling the information security in its entirety. Our approach is based on a practical IFC model, called Address Split Design (ASD), that consists in splitting any variable that contains sensitive data and maintaining

the symbol table to protect accesses to the secret part of these variables. We have implemented our model on the chromium V8 engine, a full-fledged JavaScript engine. Following the implementation, performance and conformance testing have been done on our implementation. The measured performance drop is significantly smaller than other comparative approaches. We further showed that implementation of our approach does not affect the general working of existing websites by performing such a test over the top websites of the internet. Further, we have also been able to verify that our model can be used to protect variables in several scenarios that would have otherwise caused disclosure of secret information.

List of Publications

The various publications made during the course of our work are listed below

International Peer-Reviewed Publications

1. **Ensuring endpoint authenticity in webrtc peer-to-peer communication**, Willem De Groef, Deepak Subramanian, Martin Johns, Frank Piessens, and Lieven Desmet, In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 2103–2110, ACM, July 2016, doi:10.1145/2851613.2851804.

2. **Preventive information flow control through a mechanism of split addresses**, Deepak Subramanian, Guillaume Hiet, and Christophe Bidan, Pages 1-8. In: *ACM 9th International Conference on Security of Information and Networks* 2016. New York, doi:10.1145/2947626.2947645.

3. **Self-correcting Information Flow Control Model for the Web-Browser**, Deepak Subramanian, Guillaume Hiet, and Christophe Bidan. In: Cuppens F., Wang L., Cuppens-Boulahia N., Tawbi N., Garcia-Alfaro J. (eds) *Foundations and Practice of Security* 2016, *Lecture Notes in Computer Science, vol 10128*, pages 285–301. Springer, Cham, 2016, doi:10.1007/978-3-319-51966-1_19.

National Peer-Reviewed Publications

1. **Preventive information flow control through a mechanism of split addresses**, Deepak Subramanian, Guillaume Hiet, and Christophe Bidan. In: *9ème Conférence sur la Sécurité des Architectures Réseaux et des Systèmes d'Information*, Saint-Germain-Au-Mont-d'Or, France, May 2014. URL: `https://hal.inria.fr/hal-01344563`.