THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Information Flow for Web Security and Privacy

ALEXANDER SJÖSTEN

CHALMERS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Information Flow for Web Security and Privacy
ALEXANDER SJÖSTEN

ISBN 978-91-7905-348-2
Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 4815
ISSN 0346-718X

Technical report 189D
Department of Computer Science and Engineering
Information Security Division

CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Gothenburg, Sweden
Telephone +46 (0)31-772 10 00

Printed at Chalmers
Gothenburg, Sweden 2020

Information Flow for Web Security and Privacy

Alexander Sjösten
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

The use of libraries is prevalent in modern web development. But how to ensure sensitive data is not being leaked through these libraries? This is the first challenge this thesis aims to solve. We propose the use of information-flow control by developing a principled approach to allow information-flow tracking in libraries, even if the libraries are written in a language not supporting information-flow control. The approach allows library functions to have *unlabel* and *relabel* models that explain how values are unlabeled and relabeled when marshaled between the labeled program and the unlabeled library. The approach handles primitive values and lists, records, higher-order functions, and references through the use of *lazy marshaling*.

Web pages can combine benign properties of a user's browser to a *fingerprint*, which can identify the user. Fingerprinting can be intrusive and often happens without the user's consent. The second challenge this thesis aims to solve is to bridge the gap between the principled approach of handling libraries, to practical use in the information-flow aware JavaScript interpreter JSFlow. We extend JSFlow to handle libraries and be deployed in a browser, enabling information-flow tracking on web pages to detect fingerprinting.

Modern browsers allow for browser modifications through *browser extensions*. These extensions can be intrusive by, e.g., blocking content or modifying the DOM, and it can be in the interest of web pages to detect which extensions are installed in the browser. The third challenge this thesis aims to solve is finding which browser extensions are executing in a user's browser, and investigate how the installed browser extensions can be used to *decrease* the privacy of users. We do this by conducting several large-scale studies and show that due to added security by browser vendors, a web page may uniquely identify a user based on the installed browser extension alone.

It is popular to use filter lists to block unwanted content such as ads and tracking scripts on web pages. These filter lists are usually crowd-sourced and mainly focus on English speaking regions. Non-English speaking regions should use a supplementary filter list, but smaller linguistic regions may not have an up to date filter list. The fourth challenge this thesis aims to solve is how to automatically generate supplementary filter lists for regions which currently do not have an up to date filter list.

Keywords: information-flow control, side-effectful libraries, web security, browser fingerprinting, browser extensions, filter list generation

This thesis is based on the work contained in the following papers, each presented individually. All the papers aside from Paper III are published at peer-reviewed conferences, while Paper III is currently under submission. Paper I, Paper II, and Paper IV in the thesis are the full versions of the published conference papers.

I. A Principled Approach to Tracking Information Flow in the Presence of Libraries
Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld
In *Proceedings of the 6th International Conference of Principles of Security and Trust (POST)*. Springer, 2017.

II. Information Flow Tracking for Side-Effectful Libraries
Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld
In *Proceedings of the International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. Springer, 2018.

III. EssentialFP: Exposing the Essence of Browser Fingerprinting
Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld
Under submission.

IV. Discovering Browser Extensions via Web Accessible Resources
Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld
In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2017.

V. Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks
Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld
In *26th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2019.

VI. Filter List Generation for Underserved Regions
Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits
In *WWW '20: The Web Conference 2020*. ACM / IW3C2, 2020.

Acknowledgements

Many extraordinary individuals help guide you through the sometimes bumpy ride of receiving a PhD degree.

First and foremost, I want to thank my supervisor Andrei Sabelfeld for all support, both academically but also personally with advice to awesome coffee places, bars, lunch runs to help alleviate the mind from work, and overall advice on the academic life. Working with you have been awesome.

I also want to thank my co-supervisor Daniel Hedin for great collaborations, gaming nights, beers, and coffees. Whenever I had a problem, I could always turn to you for advice — something I will always appreciate.

Thank you to all my co-authors for the collaborations. Every research project has been an experience, and they taught me things both about myself, but also how to organize my research from you.

I also want to thank all my office mates, both past and present: Pablo B, Daniel S, Per, Jeff, Iulia, Ivan, and Mohammad, for making the office a truly fantastic working environment, filled with nice discussions and laughter.

To my other colleagues at Chalmers: my sincerest thanks for help making Chalmers such a nice place to work during these five years. Especially a big thank you to the following persons. Boel, for all the tea-drinking, chats and support. You always watched over me, trying your hardest to keep me sane. Max and Sandro, for the beers, coffees, chats about everything and nothing, and simply being terrific people to be around. Benjamin for always having a smile on your face and always being positive. Believe it or not — it rubs off. Irene and Claudia, for the support and chats when things were not optimal. Pablo P, for nice discussions and bouncing ideas. Wolfgang, for advice on music and teaching. Ana, for trying to give us PhD students the best suited courses to help make the teaching experience as good as possible. Anton and Daniel H, for all your support and advice, both at and outside of Chalmers.

To my friends outside of Chalmers: thank you for helping me disconnect from work and sticking around even though I am not always the best at keeping in touch. You were always just a message away from sharing a fika, having lunch, playing board games, watching ice hockey, role-playing sessions, or simply hanging out. Special thanks to Jakob and Emelia for the support and discussions.

To my family, for always being there just one phone call away. Your support over these five years has been great, and knowing I can always take a trip back home to relax has helped a lot.

I save the best for last. This thesis would not exist had it not been for Pauline. I doubt I am the easiest person to live with, but you are always there ready to pick me up when I fall, supporting me every step I take. Words cannot describe how much that means. You are truly awesome!

Contents

Information Flow for Web Security and Privacy

Businesses today are completely reliant on Information Technology, and our daily lives are moving online at a fast pace. To give a few examples, we use streaming services to watch movies and listen to music, we visit web pages to read the news, buy merchandise and make bank transfers, and we use social networks to maintain social contacts with friends and families and schedule events. With more online interaction, the need to protect user data and privacy from attackers is increasing. Unfortunately, it is difficult to keep private information secured, even for domain experts. Recent years have seen hundreds of millions of users having sensitive information stolen [70]. This includes passwords [45, 34, 60, 2], phone numbers [45], and social security numbers [59], leading to financial losses. In some cases, such as a web page for having affairs [56], being identified by the stolen data can lead to loss of lives [39]. However, not every data leak comes from malicious intent.

When developing a web page, the code is usually divided into two groups. There is *first-party* code, which is code written by the web page developer, and there is *third-party* code, which provides a service the web page uses but does not control. The first-party code is trusted, but it can be difficult to isolate the first-party code from the third-party code and once the third-party code has been loaded, it is treated as first-party code by the browser. It is common for web pages to use third-party code to enhance user experience. As an example, to understand how a user interacts with a web page and collect statistics about the user's location and browser characteristics to help improve the user experience, web pages can employ *analytic scripts*. Unfortunately, this can lead to unintended leaks of sensitive data [64]. In a similar vein, to help yield revenue, a web page can use advertisements. The ads are usually served through an *ad network*, which will try to target ads based on information about the user. Loading ads through ad networks is usually done through third-party code, and there have been cases where malware has been included in the served ad, a process known as *malvertising*. Malvertising has been known to happen even in larger ad networks [62], and has hit popular services such as Spotify [55], news outlets such as The New York Times and the BBC [54], and the London Stock Exchange [7].

Both analytic scripts and advertisement scripts give good examples of what can be troublesome with third-party scripts. Information about the user is being sent to the third-party, who can use this information for monetization. Indeed, to increase the probability of a user clicking on an advertisement, the ad network will try and target ads specific to users. This means that the more web pages that use the same third-party, the more data the third-party can gather, leading to seemingly free services being paid for with

data instead of money. Put bluntly: users can be tracked across the web by third-parties. Although this used to be invisible to a user, the last couple of years have seen regulations such as GDPR [5] try to increase user privacy online. One method third-parties can use to identify different users is to perform *browser fingerprinting*, where seemingly benign data from a user's browser is compounded into a fingerprint. The more web pages that use the same third-party script, the more information about a potential user is given to the third-party offering the script.

Fortunately, privacy awareness has increased, both from web pages, browser vendors, and users. Techniques such as the Same-Origin Policy (SOP) [11], Content Security Policy (CSP) [4], and sandboxing [6] have emerged to help web pages better control third-party code. Browser vendors are proposing ways they will combat fingerprinting, and users can shape their browsing experience through the use of *browser extensions*, which can help, e.g., block third-party tracking scripts and advertisement.

The goal of this thesis is to help increase security and privacy online and will do so in four ways by:

1. defining a principled approach for tracking information flow in third-party libraries, allowing for a trusted application to use untrusted third-party code while ensuring no sensitive information is leaked (Papers I–II).

2. implementing the principled approach presented in Paper II to analyze how third-party browser fingerprinting scripts behave, compared to, e.g., analytic scripts (Paper III).

3. presenting how browser extensions can *decrease* privacy by allowing web pages to detect if a user has a specific browser extension installed (Papers IV–V).

4. increasing privacy for smaller linguistic regions by presenting an automated way for generating filter lists for ad blocking (Paper VI).

Section 1 introduces *Information-Flow Control (IFC)*, which is the main security mechanism used in this thesis. Section 2 introduces *browser fingerprinting*, before Section 3 gives a brief background of *browser extensions* and how they work. Section 4 introduces how *third-party content blocking* mainly is achieved before Section 5 presents the contributions made in this thesis.

1 Information-Flow Control

In modern software development, a common way of checking an application's correctness is through extensive testing and code reviews. This can

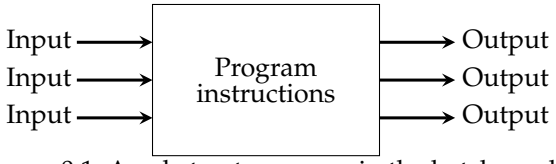

Figure 0.1: An abstract program in the batch model

find some security vulnerabilities, but severe ones are still missed (see e.g. Heartbleed [16] and Shellshock [14]).

Language-based security is a means to express security policies and enforcement mechanisms using programming language techniques [65]. This thesis focus on an area of language-based security called *Information-Flow Control (IFC)*. When modeling programs, they can be seen as a black box and is treated as a function from inputs to outputs. Inputs of a program are called *sources* and outputs are called *sinks*. This modeling approach is known as a *batch model* and is depicted in Figure 0.1. For all useful programs, the outputs of the program are dependent on the inputs, and the dependencies from the sources to the sinks are defined by the program source code.

Within IFC, we are interested in tracking the information flow from sources to sinks. This means we are interested in *how* the sources influence the sinks, and is done by tracking two types of flows: *explicit* and *implicit* flows. Explicit flows, which corresponds to *data flows* [46] in traditional program analysis, is when one or more values are combined into a new value. As an example, the assignment `x = y` introduces an explicit flow from `y` to `x`. Implicit flows, which corresponds to *control flows* [46] in traditional program analysis, is when a value indirectly influence another through the control flow of the program. To illustrate, the following program contains an implicit flow from `x` to `y`, as the value of `x` dictates which branch is taken and by that, which assignment that is made to `y`.

```
1 if (x) {
2   y := true;
3 } else {
4   y := false;
5 }
```

To allow for tracking the flow from sources to sinks, IFC is normally deployed in a *multi-level system* [40]. The information in a multi-level system is classified into different *levels*, based on a *lattice*. For intuition, consider the levels *unclassified* $\sqsubseteq$ *classified* $\sqsubseteq$ *secret* $\sqsubseteq$ *top secret*, where $\sqsubseteq$ is a relation over the partial order of the lattice, defining how the information is allowed to flow. In this example, *unclassified* information is allowed to flow anywhere in the program, but information that is classified *secret* is only allowed to

flow to sinks that are either *secret* or *top secret*. When using IFC, the aim is to enforce the information flow respect the relation $\sqsubseteq$. A multi-level system can be encoded as a two-level lattice: $L \sqsubseteq H$, where L is *public* (or *low*) data and H is *secret* (or *high*) data. The aim in this simplified setting is to enforce a security property called noninterference [49], which dictates secret sources do not influence public sinks.

1.1 Noninterference

Noninterference is achieved when all runs of a program, where the only difference between the runs is the high inputs, do not differ in the low outputs. Looking at Figure 0.2, to achieve noninterference, the crossed out dashed red line must not exist in any run of the program.

The work in this thesis only considers a noninterference policy called *termination-insensitive noninterference (TINI)*, with the implication that information leakage through *termination channels* is not in scope. Intuitively, if `high_val` is an integer labeled H, and `print` is a function that will output on a public channel, the following program is secure by state-of-the-art IFC tools using TINI.

```
1 for current in range(0, Number.MAX_VALUE) {
2     print(current);
3     if (current == high_val) then loop_forever
4 }
```

The `for`-loop does not depend on a secret, which means `current` will be labeled L. This makes the output on the public channel through `print(current)` valid. However, once `current == high_val`, the program will execute `loop_forever`, which represents an infinite loop. This ensures the last printed public value will be the same as the secret value `high_val`, which indicate there exists an implicit flow through a termination channel [35]. As TINI does not provide any guarantees for non-terminating runs, it would be unable to classify the program as insecure.

1.2 Enforcing Information-Flow Control

There are three main branches of IFC enforcement: *static*, *dynamic*, and *hybrid*. However, as the work in this thesis regarding IFC is focused on dynamic languages such as JavaScript, only dynamic IFC is considered. The reader is referred to [65, 52, 43, 44, 51] for more reading about other flavors of IFC.

A dynamic enforcement is executed *at runtime*, with the use of modified semantics of the language that allows for security checking. Dynamic IFC is often more permissive when working in a dynamic language such as

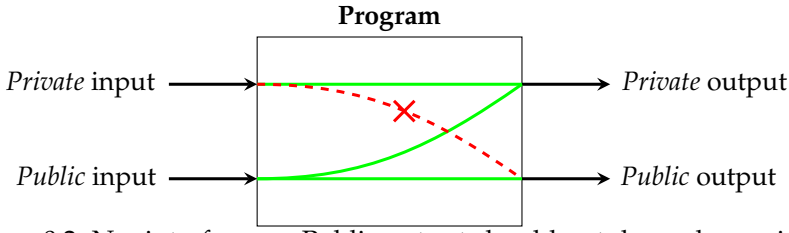

Figure 0.2: Noninterference. Public output should not depend on private input

JavaScript, as it has full access to the runtime environment and the runtime values. Runtime values are augmented with a representation of security labels, which are copied and joined to reflect the computations of the program. To track implicit flows, a *program counter (pc)* label is used to keep track of the current execution level, which is known as the *security context*. When secret data is used to compute e.g. the condition of an if-statement, the pc is updated to reflect the label of the condition, and the body of the if-statement is executed under secret control, which restricts the allowed side-effects. Indeed, while under secret control, no public side-effects are allowed to occur, as that indicates an implicit flow. However, it is not only values that must be protected — implicit flows can also occur in the security labels. As an example, consider the following code from [37], where `l` and `t` are initially labeled L, and `h` is initially labeled H.

```
1 l := true;
2 t := true;
3 if (h == true) then
4       t := false;
5 if (t == true) then
6       l := false;
```

If implicit flows are allowed into labels, the security labels of the variables `t` and `l` are upgraded if the assignments on Line 4 and Line 6 occur respectively. The result of executing the program (which can be seen in Table 0.1) leads to the value of `l` to be the same as the value of `h`, but retain the low security label.

To prevent this issue and avoid the implicit flows into labels, the enforcement can be based on *no sensitive-upgrades (NSU)*, which disallows upgrading labels of low values when branching on secret data [36, 71]. With NSU, the assignment on Line 4 is not allowed, as there would be a low upgrade under secret control which would cause the program to terminate before the leak of information occurs.

Table 0.1: Trace execution, showing why side effects into labels are dangerous, and can be used to leak secret information.

Executed code	`h := true`^{H}	`h := false`^{H}
`l := true;`	`l := true`^{L}	`l := true`^{L}
`t := true;`	`t := true`^{L}	`t := true`^{L}
`if (h == true) then`	branch taken, `pc` = H	branch not taken
`  t := false;`	`t` becomes `false`^{H}	`t` remains `true`^{L}
`if (t == true) then`	branch not taken	branch taken, `pc` = L
`  l := false;`	`l` remains `true`^{L}	`l` becomes `false`^{L}
	`l = true`^{L}	`l = false`^{L}

Papers I–II explores how IFC can be lifted to libraries written in a language that does not support IFC. The mechanism presented in Papers I–II follow both TINI and NSU.

Observable Tracking NSU can sometimes be too restrictive and mark seemingly valid programs as invalid. As an example, the following program can be argued to be secure, since `low_val` is never written to a public output.

```
1 low_val := true;
2 if (high_val == false) then low_val := false;
```

Similarly, if the value of a low value remains the same, a program can be deemed secure as an attacker would not be able to gain knowledge about the secret value `high_val`.

```
1 low_val := false;
2 if (high_val == false) then low_val := false;
```

To allow the latter example, one must employ value sensitivity [41], a topic that is not covered in this thesis. In Papers I–II both examples would be deemed insecure if `high_val` is `false`. However, Paper III employs *observable tracking* [38, 67], which is more permissive than NSU. As the name suggests, observable tracking tracks the observable implicit flows, as well as explicit flows. Observable tracking is more permissive, as it would allow implicit flows *as long as the implicit flow is not observable by an attacker*. Although the value of `low_val` is modified depending on `high_val` in the first example, it would be deemed secure with observable tracking since that implicit flow is not observed by an attacker.

To capture the essence of browser fingerprinting, where many different data sources are combined, a program must not halt as soon as NSU is triggered. This makes observable tracking a good fit for Paper III, as it presents an approach to detect fingerprinting.

2 Browser Fingerprinting

When browsing a web page, many properties of the web browser and underlying system are accessible by the web page, such as the screen width [13] and height [12], the user agent [9], and the language [10]. Although the properties that are accessible by the web page are benign in isolation, combining them may uniquely identify a user [47]. There are web pages, such as Panopticlick [27] and AmIUnique [19], where users can test their fingerprintability. Similarly, there are libraries such as FingerprintJS [23] that web pages can use to aid them in fingerprinting the users. As browser fingerprinting can help identify a user, this can be used by third-party code to track users during their browsing session.

What makes browser fingerprint troublesome is twofold: 1) it decreases the privacy for users, as they can be tracked easier, and 2) the act of fingerprinting is often completely invisible for the users. For users today, there are many different approaches how to defend themselves, ranging from using a browser that attempts to make all look the same [31, 30], to adding random noise to sensitive API calls known to be used when fingerprinting [32], using privacy budgets [28], to using filter lists to block the fingerprinting script to be loaded [24, 26]. All of these approaches indicate there is no uniformed way of detecting fingerprinting, something Paper III attempts to find.

As browser fingerprinting follows the distinct pattern of 1) a script accessing several different properties and 2) combining the properties into one value, searching for this pattern can help distinguish fingerprinting scripts from other types of scripts. Paper III tackles this problem by using observable IFC.

3 Browser Extensions

If users want to improve the web browsing experience, they can increase web browser functionality by installing browser extensions. More privacy aware users may install browser extensions to block advertisement on web pages, block tracking scripts executed on web pages, or a password manager to help make it easier to have a unique password for every service. But this comes at a cost: extensions are given permissions which are greater than those of a web page. As an example, an ad blocker must read the network requests made by the web page to determine if a resource should be blocked or not. But extensions can also inject arbitrary code [3], with some malicious extensions injecting their own tracking scripts, allowing the extension developer to track the users on every web page they visit [50]. Even worse, if an extension has a vulnerability, web pages may be allowed

to execute arbitrary code with the elevated privilege of the extension [33, 1]. As it stands, the current extension model allows for web pages to exploit browser extensions to gain access to sensitive data and bypass SOP, which poses a threat to user privacy [66].

But there is another side to the story as well. It can be in the interest of a web page to know which extensions a user has installed, as the presence of an extension can lead to, e.g., financial losses due to less advertisement revenue, or to prevent arbitrary code being injected when paying with a credit card online or accessing an internet bank. Web pages can detect extensions through *behavioral analysis*, where a web page detects extensions by looking for effects created by the extensions. An example would be to check if an element is present or absent on the web page, or analyzing specific changes to the web page which can be attributed to a specific extension [69, 68]. It may be difficult to determine the exact extension using behavioral analysis — there are, e.g., several different ad blockers which may have the same behavior. It can also be costly, as it requires time and effort to analyze keep up-to-date with extension updates.

Instead, one can exploit the fact that browser extensions must declare which resources they want to inject onto a web page. These resources, which are called *web accessible resources (WARs)*, are then accessible from the web page, and can be used to help detect installed extensions. In Chrome, the resources have a specific URL pattern, which allows web pages to enumerate known resources and request them. If the resource is accessible, the web page knows the extension is installed. Naturally, this is not necessarily good, which prompted Firefox to try and mitigate the enumeration by randomizing part of the resource URL. Unfortunately, this randomization is not done often enough, which means an extension that injects a resource will give the web page a token which can be used for tracking, uniquely identifying a user.

These are the topics for Papers IV–V, with Paper IV exploring how many browser extensions that can be trivially detected using WARs, and Paper V exploring how the use of randomized extension IDs actually can *decrease* the privacy for users.

4 Content blocking with filter lists

Filter lists can be used to block undesired content, with hundreds of millions of web users using filter lists. Simply put, a filter list is a collection of rules, dictating what resources to block, usually based on the URL. Some browsers have implemented the use of popular filter lists to help block ads [17] and tracking scripts [22, 24, 26], and users can also install browser extensions to increase protection, such as AdBlock [18], Privacy Badger [29], Ghostery [25],

and Disconnect [20]. This means filter lists can be used to maintain a secure, private, performant, and appealing web for users. Prior work show filter lists can help reduce data use [61], protect users from malware [57], and improve browser performance [48, 63].

Filter lists are usually crowd-sourced, where a group of users manually label resources to keep the filter lists up to date. Unfortunately, the popular filter lists focus on English web pages, and non-English regions should use a supplementary list to block regional resources not popular enough to be blocked by the global lists. Although there are a plethora of supplementary filter lists [21], if the regions for the supplementary list have smaller groups of people maintaining the filter list, e.g., due to being smaller linguistic regions, the supplementary list may be outdated or even non-existent, making the protection of users in these regions poorer. Paper VI presents an approach to automatically generate filter lists, focusing on three regions that have outdated supplementary filter lists.

5 Contributions

This thesis consists of six papers. Five of the papers (Papers I–II and Papers IV–VI) have been published in peer-reviewed conferences, and Paper III is currently under submission. This section outlines the contributions of each paper. In broad terms, the papers fall into four different categories, all aimed to increase web security and privacy by:

1. defining a theoretical framework for allowing IFC in the presence of libraries. This would allow deploying IFC tools, such as JSFlow, in settings where the libraries are written in a language which does not support IFC, by allowing marshaling between the labeled program and the unlabeled library. The approach enforces TINI and is presented in Papers I–II.

2. bridging the gap between the theoretical framework to handle libraries in an IFC setting by implementing library handling in JSFlow, while also deploying JSFlow in a browser to detect browser fingerprinting. This is presented in Paper III, where the theoretical framework of Paper II is implemented. The resulting implementation uses the IFC approach observable tracking.

3. looking at how the use of browser extensions can *decrease* privacy since web pages can detect and identify users based on the installed extension(s). This is based on the browser extension's WARs, and is presented in Papers IV–V.

4. increasing security and privacy for smaller linguistic regions where the supplementary filter lists are outdated by automatically generate filter lists rules. This will allow for smaller regions to have better supplementary filter lists and is presented in Paper VI.

The rest of this section summarizes the papers in this thesis.

5.1 A Principled Approach to Tracking Information Flow in the Presence of Libraries

Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld

In Paper I, a principled approach to tracking information flow in a program which use libraries was developed. There has been encouraging progress on IFC for programs in increasingly complex programming languages. However, as programs are typically deployed in an environment with rich APIs and powerful libraries, the need for tracking the propagation of information in these libraries arises. These APIs and libraries are usually unavailable or written in a different language that does not support IFC. The setting in this paper is the program is assumed to be written in an information-flow aware language, but the library is not. The development of the approach initially starts with a small core language with the notion of *split semantics* and *stateful marshaling*, before being extended with lists and higher-order functions. This paper aims to strike a balance between security and precision to find a middle ground between "shallow" signature-based modeling of libraries and "deep", stateful approaches where library models need to be supplied manually. The general idea for striking this balance is based on *unlabel* and *relabel* models, which define how labels are removed when marshaling to the library, and how they are added when marshaling back to the program. A key aspect of this paper is *lazy marshaling*, which increases the precision of the tracking since only used parts of lists and higher-order functions will affect the label when marshaling from the library to the program. Although not implemented in Paper I, the notion of lazy marshaling presented extends naturally to all types of structured data, including records and objects. Soundness is proved with respect to noninterference.

The paper presented in this thesis is the extended version of the published paper.

Statement of contribution This paper was co-authored with Daniel Hedin, Frank Piessens, and Andrei Sabelfeld. Alexander's contributions were to define syntax and semantics, implement prototypes for testing the ideas, and prove soundness of the different systems.

Appeared in: *Principles of Security and Trust (POST), Uppsala, Sweden, April 2017*

5.2 Information Flow Tracking for Side-effectful Libraries

Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld

Paper II is a continuation of Paper I, where the major contribution is the addition of side-effects through references. As Paper I passed the model state as an implicit parameter when marshaling between the program and the library, handling side-effects would be difficult since every marshaled value would have their own model state, with no obvious way of propagating modifications made by one function to another. Instead, Paper II makes a complete overhaul of the core system and introduces a *model heap* which is part of a shared execution environment. When marshaling, instead of passing the entire model state, we now pass the current stack of *heap pointers*, ensuring side-effects from one function is propagated to all functions which have the same pointers.

The introduced structured data in Paper I was modified to accommodate the model heap, and records, references, and side-effects were added. Lazy marshaling remained and was extended to include the records. To allow for modeling of side-effects, the model language was extended with *side-effect constraints*, which models how the side-effects can manipulate data. The theoretical work in this paper is formalized in Coq [15], showing the system is sound with respect to noninterference.

Papers I–II provides a theoretical core for how to track information flow in stateful libraries with structured data and higher-order functions.

The paper presented in this thesis is the extended version of the published paper.

Statement of contribution This paper was co-authored with Daniel Hedin and Andrei Sabelfeld. Alexander's contributions were to define the syntax and semantics, conduct the case study on a file system library, creating the examples and implementing the prototype.
Appeared in: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Madrid, Spain, June 2018*

5.3 EssentialFP: Exposing the Essence of Browser Fingerprinting

Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld

Paper III ties the knot between the theory presented in Paper II and practical use. In the setting of Paper III, "libraries" corresponds to the DOM API in the browser. It presents EssentialFP, a principled approach to detecting browser fingerprinting on the web. EssentialFP employs observable IFC to detect the pattern of 1) gathering information from a wide browser API surface, and 2)

communicating the information to the network, which captures the essence of fingerprinting.

The implementation of EssentialFP leverages, extends, and deploys JSFlow [53] in a browser, showing it is possible to spot fingerprinting on the web by evaluating it on several different categories of web pages. The evaluated categories are analytics, authentication, bot detection, and fingerprinting-enhanced Alexa top pages, and we can see a clear distinction between, e.g., analytics and fingerprinting-enhanced web pages.

As Paper III demonstrates how IFC tracking is possible within the DOM API, it would be possible to extend this in the future to also include browser extensions to see if the attacks presented in Papers IV–V can be detected using IFC as well.

Statement of contribution This paper was co-authored with Daniel Hedin and Andrei Sabelfeld. Alexander's contributions were to help with the implementation of the library handling presented in Paper II, create the crawler, conduct the empirical study, and analyze the results.
Under submission

5.4 Discovering Browser Extensions via Web Accessible Resources

Alexander Sjösten, Steven Van Acker, and Andrei Sabelfeld

Web pages can perform browser fingerprinting by combining seemingly benign properties in the browser and specific configurations of the hardware [47, 58, 42, 27]. Similarly, web pages can detect browser extensions based on their behavior [69, 68]. Paper IV shows how some extensions can be detected by web pages without analyzing the behavior and explores what knowledge can be gained by a web page about a user's installed extensions. It uses the fact that browser extensions must declare resources they want to inject as *web accessible resources (WARs)*, which becomes public resources [8] and can easily be fetched by any web page.

This work includes a large-scale empirical study, consisting of downloading all free extensions for Chrome and Firefox, as well as crawling the Alexa top 100,000 pages to determine if WARs are used to detect extensions in the wild. It also includes a discussion of potential measures to avoid this kind of extension detection.

It is worth to point out that the empirical study for Firefox mainly focuses on extensions based on the old extension model, and not the current WebExtensions.

The paper presented in this thesis is the extended version of the published paper.

Statement of contribution This paper was co-authored with Steven Van Acker and Andrei Sabelfeld. Alexander's contributions were the extensions experiment (all but the Alexa part), as well as defining the measures and develop the prototype for detecting extensions.
Appeared in: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, March 2017*

5.5 Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks

Alexander Sjösten, Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld

To help combat the probing of browser extensions used in Paper IV, Firefox randomized the ID, which is part of the URL to a WAR, of a browser extension for their extension model WebExtensions. Unfortunately, the randomized ID is rarely re-generated, which exacerbates the extension detection problem by allowing attackers to use the randomized ID as a reliable fingerprint. Paper V presents *revelation* attacks, where extensions reveal themselves by injecting content, and with this their random extension ID, on web pages. Once the random extension ID and the injected resource is in the hand of the web page, it can start to probe for other known resources to try and identify the extension. Paper V demonstrates how a combination of revelation and probing can uniquely identify 90% of all extensions injecting content, despite a randomization scheme, and presents a series of large-scale studies to estimate the possible implications of both probing and revelation attacks.

Lastly, the paper presents Latex Gloves: a browser-based mechanism that enables control over which extensions are loaded on which web pages, implemented as a proof of concept which blocks both classes of attacks.

Statement of contribution This paper was co-authored with Steven Van Acker, Pablo Picazo-Sanchez, and Andrei Sabelfeld. Alexander's contributions were developing the attacks, conducting empirical studies to decide which browser extensions are vulnerable, and designing the defence against both classes of attacks.
Appeared in: *Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, February 2019*

5.6 Filter List Generation for Underserved Regions

Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits

Filter lists play a crucial and growing role in protecting and assisting web users. The vast majority of popular filter lists are often crowd-sourced, where a large number of people manually label resources related to undesirable web resources, such as ads and trackers. Unfortunately, crowd-sourcing in regions of the web serving languages with (relatively) few speakers can perform poorly. Paper VI addresses this problem with a deep browser instrumentation called PageGraph, which allows for accurately generate *request chains*, which is a chain of requests which ended with a resource being loaded, and an ad classifier which combines perceptual and page-context features to remain accurate across multiple languages.

With the request chains, the aim is to find as high a point as possible to block an ad, without breaking the web page. This is applied to three regions of the web which had poorly maintained filter lists: Sri Lanka, Hungary, and Albania, generating several new filter list rules and increased the overall blocking by 30.1% across the regions.

This paper was the result of an internship at Brave Software during the summer of 2019.

Statement of contribution This paper was co-authored with Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Alexander's contributions were to help developing the browser instrumentation PageGraph, the full implementation of the hybrid classifier (aside from the perceptual classifier), conducting all the experiments, the inclusion chain creation, and the filter list rule generation.
Appeared in: *Proceedings of the Web Conference (WWW), Taipei, Taiwan, April 2020*

6 Bibliography

[1] Adobe: Adobe Acrobat Force-Installed Vulnerable Chrome Extension. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1088`. accessed: June 2020.

[2] Canva Security Incident – May 24 FAQs. `https://support.canva.com/contact/customer-support/may-24-security-incident-faqs/`. accessed: June 2020.

[3] Content Scripts. https://developer.chrome.com/extensions/content_scripts. accessed: June 2020.

[4] Content Security Policy (CSP). https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP. accessed: June 2020.

[5] General Data Protection Regulation. https://gdpr-info.eu/. accessed: June 2020.

[6] HTML Living Standard. https://html.spec.whatwg.org/#sandboxing. accessed: June 2020.

[7] London Stock Exchange site shows malicious adverts. https://www.bbc.com/news/technology-12597819. accessed: June 2020.

[8] Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources. accessed: June 2020.

[9] Navigatorid.useragent. https://developer.mozilla.org/en-US/docs/Web/API/NavigatorID/userAgent. accessed: June 2020.

[10] Navigatorlanguage.language. https://developer.mozilla.org/en-US/docs/Web/API/NavigatorLanguage/language. accessed: June 2020.

[11] Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy. accessed: June 2020.

[12] Screen.height. https://developer.mozilla.org/en-US/docs/Web/API/Screen/height. accessed: June 2020.

[13] Screen.width. https://developer.mozilla.org/en-US/docs/Web/API/Screen/width. accessed: June 2020.

[14] Shellshock: All you need to know about the Bash Bug vulnerability. https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=5ee60f4e-030f-4691-b5b4-dc3c9e3701d4&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments. accessed: June 2020.

[15] The Coq Proof Assistant. https://coq.inria.fr/. accessed: June 2020.

[16] The Heartbleed Bug. https://heartbleed.com. accessed: June 2020.

[17] What is "Shields"? https://support.brave.com/hc/en-us/articles/360022973471-What-is-Shields-. accessed: June 2020.

[18] AdBlock. https://getadblock.com/, accessed: June 2020.

[19] AmIUnique. https://amiunique.org/, accessed: June 2020.

[20] Disconnect. https://disconnect.me/, accessed: June 2020.

[21] FilterLists. https://filterlists.com/, accessed: June 2020.

[22] Fingerprinting Protections. https://github.com/brave/brave-browser/wiki/Fingerprinting-Protections, accessed: June 2020.

[23] FingerprintJS. https://fingerprintjs.com/open-source/, accessed: June 2020.

[24] Firefox 72 blocks third-party fingerprinting resources. https://blog.mozilla.org/security/2020/01/07/firefox-72-fingerprinting/, accessed: June 2020.

[25] Ghostery. https://www.ghostery.com/, accessed: June 2020.

[26] Learn about tracking prevention in Microsoft Edge. https://support.microsoft.com/en-us/help/4533959/microsoft-edge-learn-about-tracking-prevention, accessed: June 2020.

[27] Panopticlick. https://panopticlick.eff.org, accessed: June 2020.

[28] Potential uses for the Privacy Sandbox. https://blog.chromium.org/2019/08/potential-uses-for-privacy-sandbox.html, accessed: June 2020.

[29] Privacy Badger. https://privacybadger.org/, accessed: June 2020.

[30] Safari Privacy Overview. https://www.apple.com/safari/docs/Safari_White_Paper_Nov_2019.pdf, accessed: June 2020.

[31] Tor. https://www.torproject.org/, accessed: June 2020.

[32] What's Brave Done For My Privacy Lately? Episode #3: Fingerprint Randomization. https://brave.com/whats-brave-done-for-my-privacy-lately-episode3/, accessed: June 2020.

[33] C. S. Advisory. Cisco WebEx Browser Extension Remote Code Execution Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170717-webex. accessed: June 2020.

[34] C. Aiello. Under Armour says data breach affected about 150 million MyFitnessPal accounts. https://www.cnbc.com/2018/03/29/under-armour-stock-falls-after-company-admits-data-breach.html. accessed: June 2020.

[35] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS*, 2008.

[36] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.

[37] T. H. Austin and C. Flanagan. Permissive Dynamic Information Flow Analysis. In *PLAS*, 2010.

[38] M. Balliu, D. Schoepe, and A. Sabelfeld. We Are Family: Relating Information-Flow Trackers. In *ESORICS*, 2017.

[39] C. Baraniuk. Ashley Madison: 'Suicides' over website hack. `http://www.bbc.com/news/technology-34044506`. accessed: June 2020.

[40] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, 1973.

[41] L. Bello, D. Hedin, and A. Sabelfeld. Value Sensitivity and Observable Abstract Values for Information Flow Control. In *LPAR*, 2015.

[42] Y. Cao, S. Li, and E. Wijmans. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *NDSS*, 2017.

[43] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *ACSAC*, 2007.

[44] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *PLDI*, 2009.

[45] C. Cimpanu. Hacker selling data of 538 million Weibo users. `https://www.zdnet.com/article/hacker-selling-data-of-538-million-weibo-users/`. accessed: June 2020.

[46] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 1977.

[47] P. Eckersley. How Unique Is Your Web Browser? In *PETS*, 2010.

[48] K. Garimella, O. Kostakis, and M. Mathioudakis. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *WebSci*, 2017.

[49] J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P*, 1982.

[50] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In *DIMVA*, 2015.

[51] D. Hedin, L. Bello, and A. Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *CSF*, 2015.

[52] D. Hedin and A. Sabelfeld. A Perspective on Information-Flow Control. In *Software Safety and Security*. 2012.

[53] D. Hedin and A. Sabelfeld. Information-Flow Security for a Core of JavaScript. In *CSF*, 2012.

[54] A. Hern. Major sites including New York Times and BBC hit by 'ransomware' malvertising. https://www.theguardian.com/technology/2016/mar/16/major-sites-new-york-times-bbc-ransomware-malvertising. accessed: June 2020.

[55] A. Hern. Spotify hit by 'malvertising' in app. https://www.theguardian.com/technology/2016/oct/06/spotify-hit-by-malvertising-in-app. accessed: June 2020.

[56] B. Krebs. Online Cheating Site AshleyMadison Hacked. https://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/. accessed: June 2020.

[57] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *CCS*, 2012.

[58] K. Mowery and H. Shacham. Pixel Perfect: Fingerprinting Canvas in HTML5. In *Web 2.0 Security and Privacy (W2SP)*, 2012.

[59] A. Ng and S. Musil. Equifax data breach may affect nearly half the US population. https://www.cnet.com/news/equifax-data-leak-hits-nearly-half-of-the-us-population/. accessed: June 2020.

[60] J. Pagliery. Hackers selling 117 million LinkedIn passwords. https://money.cnn.com/2016/05/19/technology/linkedin-hack/index.html. accessed: June 2020.

[61] A. Parmar, M. Toms, C. Dedegikas, and C. Dickert. Adblock Plus Efficacy Study. http://www.sfu.ca/content/dam/sfu/snfchs/pdfs/Adblock.Plus.Study.pdf. accessed: June 2020.

[62] D. Pauli. Malware menaces poison ads as Google, Yahoo! look away. https://www.theregister.com/2015/08/27/malvertising_feature. accessed: June 2020.

[63] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed Users: Ads and Ad-Block Usage in the Wild. In *IMC*, 2015.

[64] O. Räisänen. Trackers leaking bank account data. http://www.windytan.com/2015/04/trackers-and-bank-accounts.html. accessed: June 2020.

[65] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[66] D. F. Somé. 7empoweb: Empowering web applications with browser extensions.

[67] C. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld. An Empirical Study of Information Flows in Real-World JavaScript. In *PLAS*, 2019.

[68] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis. Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *WWW*, 2019.

[69] O. Starov and N. Nikiforakis. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P*, 2017.

[70] D. Swinhoe. The 15 biggest data breaches of the 21st century. https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html. accessed: June 2020.

[71] S. A. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.