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 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

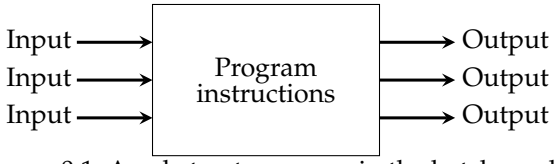


Figure 0.1: An abstract program in the batch model

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

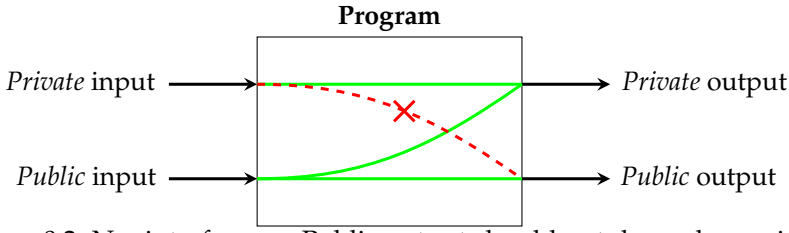


Figure 0.2: Noninterference. Public output should not depend on private input

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 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.

Paper I

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

Daniel Hedin, Alexander Sjösten, Frank Piessens, Andrei Sabelfeld

Principles of Security and Trust (POST), Uppsala, Sweden, April 2017

Abstract

There has been encouraging progress on information flow control for programs in increasingly complex programming languages, tracking the propagation of information from input sources to output sinks. Yet, programs are typically deployed in an environment with rich APIs and powerful libraries, posing challenges for information flow control when the code for these APIs and libraries is either unavailable or written in a different language.

This paper presents a principled approach to tracking information flow in the presence of libraries. With the goal to strike the balance between security and precision, we present a framework that explores the middle ground between the "shallow", signature-based modeling of libraries and the "deep", stateful approach, where library models need to be supplied manually. We formalize our approach for a core language, extend it with lists and higher-order functions, and establish soundness results with respect to the security condition of noninterference.

1 Introduction

The prevalent way to extend a language with functionality, e.g., to interact with its execution environment, is via libraries. As an example, consider a library that provides a collection of functions to provide the language with network capabilities. Since the language functionality in such cases is fundamentally extended, these libraries cannot be written in the language itself, but must be provided by some other means such as a *foreign function interface* (e.g. [26] in Java, [33] in Haskell and [29] in node.js) or via the execution environment.

Recently, there has been a growing interest in retrofitting libraries with *dynamic* execution monitors to provide additional runtime checks. One prominent example of this is *monitors for secure information flow* [15, 1, 18, 17, 3]. The interest in information flow control lies in the realization that access control is often not enough in cases when it is important what a program does with the information it has access to [30]. As an example, when a user enters credit card information into an application to perform a purchase, information flow control can guarantee that the credit card information is only used for the purpose of enabling the purchase (i.e., by passing the information to the payment provider) and is not being sent or gathered for illicit purposes.

Dynamic monitoring is similar to dynamic type checking, and works by augmenting the semantics of the language, with additional runtime information that provides an abstract view of the execution and enables enforcement of the desired properties. In the case of dynamic types, the

additional information is a runtime representation of the types of values, and in the case of information flow control it is the security level.

In the presence of libraries written in another language, dynamic monitors face two important challenges: (i) the library is not able to work with values in the augmented semantics, and, more fundamentally, (ii) is not able to maintain the abstract view of the execution. With respect to the first challenge, some kind of marshaling must take place — this already occurs for the values of the language, but must be extended to first remove any additional runtime information. With respect to the second challenge, it is important that the removed runtime information is kept, in order to be able to reestablish the augmentation, once the library returns.

Thus, the challenges above translate to these pivotal questions:

(i) how should the runtime augmentation be removed when entities are passed from the monitored program into the unmonitored library, and

(ii) how should the runtime augmentation be reinstated when entities are passed from the unmonitored library to the monitored program.

On the surface, those questions may seem fairly straightforward, but prove surprisingly involved in the presence of common programming language features, such as structured data and higher-order functions.

In the work targeting secure information flow, one can identify two extremes with respect to library models [15, 6, 1, 19, 18, 27, 17, 3]. On one hand are the *shallow models*, essentially corresponding to providing static boundary types, and on the other hand are the *deep models*, where the information flow inside the library is modeled in detail, frequently requiring a reimplementation of the library in the monitored semantics.

In JavaScript, already the standard API introduces information flow challenges. Consider, for instance, the following example, that makes use of the standard JavaScript function `Array.every` which, given a predicate, returns `true` if every element in the array on which `every` is called, is in the extension of the predicate.

```
[1,2,3,0,4,5].every(function(elem) { return elem > 0; })
```

In both JSFlow [17, 16] and FlowFox [13, 14], accurate modeling of many library functions, such as `Array.every`, requires hand-written, deep models. This is both labor-intensive and hard to maintain, not scaling to models for a rich set of libraries, as would be needed in a rich execution environment such as a browser or node.js [24, 25, 23]. For this reason, JSFlow attempts at providing a way of automatically wrapping libraries. However, JSFlow's approach is somewhat ad hoc and lacks formal underpinning. While for simple cases correctness is evident, it is unclear if this approach scales to

more complex interactions with libraries such as for promises [21], e.g., when functions are passed to and from the library.

Contribution We investigate how to provide concise library models, in the setting of dynamic information flow control, for a small functional language. We present the development in a gradual way and investigate different programming language constructs in isolation, as extensions of a common core language. The modeling is such, that the results combine with relative ease. For space reasons, we limit ourselves to the treatment of structured data and higher-order functions. The main contributions of this paper are:

- a *split semantics* with *stateful marshaling* for a simple core;
- a split semantics with stateful marshaling for structured data in the form of lists and the concept of *lazy* marshaling;
- a split semantics for higher-order functions that introduces the concept of *abstract names*, enabling the connection between callbacks and *label models*.

The focus of this paper is on the stateful marshaling, leaving the label models relatively simple. The presented model does, however, allow for more advanced label models including (value) dependent models that harness the power coming from the knowledge of runtime values. We discuss possible extensions beyond the limitations of the provided label model language.

Outline The rest of the paper is laid out as follows. Section 2 introduces the core language and the notion of split semantics with stateful marshaling. Section 3 investigates lists in terms of an extension to the core language and introduces the notion of lazy marshaling. Section 4 investigates higher-order functions in terms of an extension to the core language and introduces the notion of abstract names. Finally, Section 5 discusses related work, and Section 6 discusses future work and concludes.

2 Core language $\mathcal{C}$

We present syntax and split semantics with stateful marshaling for a small core language. The notion of split semantics entails that a program is built up by two distinct parts: 1) the monitored program executing a labeled information flow aware semantics, and 2) the unmonitored library, executing an unlabeled standard semantics. For simplicity, the two parts of the program

share syntax and semantics — the labeled semantics is an extension of the unlabeled. This is to keep the exposition small and the value-level marshaling to a minimum and is not a fundamental limitation of the approach.

2.1 Syntax

The syntax of the core language is defined as follows.

$$e ::= n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid f\ \boldsymbol{e} \mid f_{lib}\ \boldsymbol{e} \mid e_1 \oplus e_2$$

Let $\boldsymbol{x}$ denote a list of x, where $[\,]$ is the empty list and $\cdot$ is the cons operator. The top-level definitions, $d ::= f\ \boldsymbol{x} = e$, are restricted to function definitions, and *function models*, $m ::= f :: \boldsymbol{\varphi} \rightarrow \gamma$. A function model defines how labeled values are marshaled to the unlabeled function, $\boldsymbol{\varphi}$, and how the unlabeled return value is marshaled back into the labeled world, γ, see below. All unlabeled functions called from the labeled world must have a corresponding function model.

A *program* is a triple, $(\boldsymbol{d}, \boldsymbol{d}, \boldsymbol{m})$, where the first component corresponds to the monitored program, the second component corresponds to the unmonitored library, and the third component is the *library model* consisting of function models. Execution starts in the *main* function of the monitored program. In the following, we refer to the monitored part of the program as the program, and the unmonitored library as the library.

The bodies of functions are made up of expressions, consisting of integers n, identifiers x and f (denoting functions), conditional branches, let bindings, function calls, library calls and binary operators $\oplus$. Library calls are not allowed in the library part of the program.

2.2 Semantics

As indicated above, $\mathcal{C}$ has two semantics, one *labeled* and one *unlabeled*. To distinguish between the two, without unnecessary notational burden, we use $\hat{X}$ to denote an entity in the labeled semantics corresponding to X in the unlabeled semantics.

Values The *labeled values*, $\hat{v}$, and *unlabeled values*, v, are defined as labeled and unlabeled integers respectively. The labels, ℓ, are taken from a two-point upper semi-lattice $L \sqsubseteq H$, where L denotes *low* ("public" when modeling confidentiality or "trusted" when modeling integrity) and H denotes *high* ("secret" when modeling confidentiality or "untrusted" when modeling integrity). While we focus on confidentiality throughout the paper, information flow integrity can be modeled dually [5].

$$\hat{v} ::= n^{\ell} \qquad v ::= n$$

For labels let $\ell_1 \sqcup \ell_2$ denote the least upper bound of ℓ_1 and ℓ_2, and let $\hat{v}^{\ell_2} = v^{\ell_1 \sqcup \ell_2}$ for $\hat{v} = v^{\ell_1}$.

Stateful marshaling A function model defines how to marshal values between the program and the library in terms of the parameters and the return value, i.e., how to *unlabel* the parameters and *label* the result. Since the result is dependent on the parameters, it follows that the label of the result must be dependent on the labels of the parameters. For this reason, the removed labels must be stored for the duration of the library call in such a way that they can be used when relabeling the result. To achieve this, the unlabel process creates a *model state*[1], $\xi : \alpha \to \ell$, based on identifiers α, given by the unlabel model, φ. This model state is used in the labeling process in the interpretation of the label model, γ. The unlabel and label models follow the structure of the values, and are defined as follows for the core language

$$\varphi ::= \alpha \qquad \gamma ::= \kappa$$

where $\kappa ::= \alpha \mid \kappa_1 \sqcup \kappa_2 \mid \ell$ and the interpretation of κ in a model state ξ is given by

$$\begin{array}{lcl} [\![\alpha]\!]_\xi & = & \begin{cases} L & \text{if } \xi[\alpha] \text{ is undefined} \\ \xi[\alpha] & \text{otherwise} \end{cases} \\ [\![\ell]\!]_\xi & = & \ell \\ [\![\kappa_1 \sqcup \kappa_2]\!]_\xi & = & [\![\kappa_1]\!]_\xi \sqcup [\![\kappa_2]\!]_\xi \end{array}$$

From this, we define an unlabel operation, $v^\ell \downarrow \alpha$, and a label operation, $v \uparrow_\xi \kappa$, as follows

$$v^\ell \downarrow \alpha = (v, [\alpha \mapsto \ell]) \qquad v \uparrow_\xi \kappa = v^{[\![\kappa]\!]_\xi}$$

The label operation takes an unlabeled value, v, a label model $\gamma = \kappa$ and a model state, ξ and labels the value in accordance with the interpretation of the label model in the model state. The unlabel operation takes a labeled value, $\hat{v}$, and an unlabel model, $\varphi = \alpha$, and returns an unlabeled value and a model state, ξ. The unlabel operation is lifted to sequences of values by chaining, in the following way, where $\amalg$ denotes disjoint union.

$$\begin{array}{lcl} [\,] \downarrow [\,] & = & ([\,], [\,]) \\ \hat{v} \cdot \hat{\boldsymbol{v}} \downarrow \varphi \cdot \boldsymbol{\varphi} & = & (v \cdot \boldsymbol{v}, \xi_1 \amalg \xi_2) \text{ where } \hat{v} \downarrow \varphi = (v, \xi_1) \text{ and } \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_2) \end{array}$$

[1]Note that here, and in the following, for simplicity, we identify sets with the meta variables ranging over them.

$$\text{int}\frac{}{\delta \models n \rightsquigarrow n} \qquad \text{var}\frac{\delta[x] = v}{\delta \models x \rightsquigarrow v}$$

$$\text{op}\frac{\delta \models e_1 \rightsquigarrow v_1 \qquad \delta \models e_2 \rightsquigarrow v_2}{\delta \models e_1 \oplus e_2 \rightsquigarrow v_1 \oplus v_2}$$

$$\text{if}_1\frac{\delta \models e_1 \rightsquigarrow v \qquad v \neq 0 \qquad \delta \models e_2 \rightsquigarrow v}{\delta \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow v}$$

$$\text{if}_2\frac{\delta \models e_1 \rightsquigarrow v \qquad v = 0 \qquad \delta \models e_3 \rightsquigarrow v}{\delta \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \rightsquigarrow v}$$

$$\text{let}\frac{\delta \models e_1 \rightsquigarrow v_1 \quad \delta[x \mapsto v_1] \models e_2 \rightsquigarrow v_2}{\delta \models \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \rightsquigarrow v_2} \qquad \text{app}\frac{\Delta[f] = (\boldsymbol{x}, e_f) \qquad \delta \models \boldsymbol{e} \rightsquigarrow \boldsymbol{v} \quad [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v}{\delta \models f\ \boldsymbol{e} \rightsquigarrow v}$$

Figure 1.1: Unlabeled semantics

Unlabeled semantics Let the unlabeled variable environments, $\delta : x \to v$, be maps from identifiers to values, and let $\Delta : f \to (\boldsymbol{x}, e)$ be a map from identifiers to function definitions representing the unmonitored library. For simplicity we leave Δ implicit, since it is unmodified by the execution. Update of δ is defined recursively as

$$\frac{\delta_2 = \delta_1[x \mapsto v] \qquad \delta_3 = \delta_2[\boldsymbol{x}_r \mapsto \boldsymbol{v}_r]}{\delta_1[x \cdot \boldsymbol{x}_r \mapsto v \cdot \boldsymbol{v}_r] \to \delta_3} \qquad \frac{}{\delta[[\,] \mapsto \boldsymbol{v}] \to \hat{\delta}}$$

The unlabeled semantics, defined in Figure 1.1, is of the form $\delta \models e \rightsquigarrow v$, read, expression e evaluates to v in the unlabeled variable environment δ. For space reasons, since the unlabeled semantics is entirely standard, it is not explained further.

Labeled semantics Let the labeled variable environments, $\hat{\delta} : x \to \hat{v}$, be maps from identifiers to labeled values, let $\hat{\Delta} : f \to (\boldsymbol{x}, e)$ be a map from identifiers to function definitions representing the monitored program, and let $\Lambda : f \to (\boldsymbol{\varphi}, \gamma)$ represent the library model. The labeled semantics, defined in Figure 1.2, is of the form $\hat{\delta} \models e \to \hat{v}$, read, expression e evaluates to $\hat{v}$ in the labeled variable environment $\hat{\delta}$. Similarly to the unlabeled semantics we leave Δ, $\hat{\Delta}$, and Λ implicit. Also, as for the unlabeled semantics, updating $\hat{\delta}$ is defined recursively as

$$\frac{\hat{\delta}_2 = \hat{\delta}_1[x \mapsto \hat{v}] \qquad \hat{\delta}_3 = \hat{\delta}_2[\boldsymbol{x}_r \mapsto \hat{\boldsymbol{v}}_r]}{\hat{\delta}_1[x \cdot \boldsymbol{x}_r \mapsto \hat{v} \cdot \hat{\boldsymbol{v}}_r] \to \hat{\delta}_3} \qquad \frac{}{\hat{\delta}[[\,] \mapsto \hat{\boldsymbol{v}}] \to \hat{\delta}}$$

$$\text{int}\ \frac{}{\hat{\delta} \models n \to n^L} \qquad \text{var}\ \frac{\hat{\delta}[x] = \hat{v}}{\hat{\delta} \models x \to \hat{v}}$$

$$\text{op}\ \frac{\hat{\delta} \models e_1 \to v_1^{\ell_1} \qquad \hat{\delta} \models e_2 \to v_2^{\ell_2}}{\hat{\delta} \models e_1 \oplus e_2 \to (v_1 \oplus v_2)^{\ell_1 \sqcup \ell_2}}$$

$$\text{if}_1\ \frac{\hat{\delta} \models e_1 \to v^{\ell} \qquad v \neq 0 \qquad \hat{\delta} \models e_2 \to \hat{v}}{\hat{\delta} \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \to \hat{v}^{\ell}}$$

$$\text{if}_2\ \frac{\hat{\delta} \models e_1 \to v^{\ell} \qquad v = 0 \qquad \hat{\delta} \models e_3 \to \hat{v}}{\hat{\delta} \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \to \hat{v}^{\ell}}$$

$$\text{let}\ \frac{\hat{\delta} \models e_1 \to \hat{v}_1 \qquad \hat{\delta}[x \mapsto \hat{v}_1] \models e_2 \to \hat{v}_2}{\hat{\delta} \models \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \to \hat{v}_2}$$

$$\text{app}\ \frac{\begin{array}{c}\hat{\Delta}[f] = (\boldsymbol{x}, e_f) \\ \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \\ [\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e_f \to \hat{v}\end{array}}{\hat{\delta} \models f\ \boldsymbol{e} \to \hat{v}} \qquad \text{lib}\ \frac{\begin{array}{cc}\Delta[f] = (\boldsymbol{x}, e_f) & \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \\ \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} & \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \\ [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v & v \uparrow_{\xi} \gamma = \hat{v}\end{array}}{\hat{\delta} \models f_{lib}\ \boldsymbol{e} \to \hat{v}}$$

Figure 1.2: Labeled semantics

Of the rules for the core language, `lib` is the only non-standard. It corresponds to the situation, where an unmonitored library function is called from the monitored semantics. Execution proceeds as follows. First, the function definition, $(\boldsymbol{x}, e_f)$, and the function model, $(\boldsymbol{\varphi}, \gamma)$, are found, then the parameters, $\boldsymbol{e}$, are evaluated to labeled values, $\hat{\boldsymbol{v}}$. Before being passed to the library, the labeled values are first unlabeled in accordance with the function model, resulting in unlabeled values, $\boldsymbol{v}$, and a model state, ξ. The body of the library function is evaluated in an environment $[\boldsymbol{x} \mapsto \boldsymbol{v}]$, where the formal parameters of the function maps to the corresponding arguments, and the result, v, is labeled in accordance with the function model, interpreted in the model state, ξ, produced by the previous unlabeling.

2.3 Correctness

We prove correctness under the assumption that the library model correctly models the library, i.e., that every modeled function in the library respects its function model. Semantically, we express this in terms of the execution of the library, the unlabeling of the parameters and the labeling of the result.

Definition 1 (Correctness of the library models)**.** *A library model correctly models a library if every function, f, in the library, $\Delta[f] = (\boldsymbol{x}, e)$, respects the associated function model, $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$, if present.*

$$\begin{aligned}\forall f \,.\, \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \Delta[f] = (\boldsymbol{x}, e) \\ \wedge\ \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi) \\ \wedge\ [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e \rightsquigarrow v \wedge [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models e \rightsquigarrow v' \Rightarrow v \uparrow_\xi \gamma \simeq v' \uparrow_\xi \gamma\end{aligned}$$

As is standard, we prove noninterference as the preservation of a low-equivalence relation under execution, defined as follows for values and labeled variable environments.

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{n_1^H \simeq n_2^H} \qquad \frac{dom(\hat{\delta}) = dom(\hat{\delta}') \quad \forall x \in dom(\hat{\delta}) \,.\, \hat{\delta}[x] \simeq \hat{\delta}'[x]}{\hat{\delta} \simeq \hat{\delta}'}$$

Under the assumption that Definition 1 holds, we can prove noninterference for labeled execution.

Theorem 1 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \rightarrow \hat{v} \wedge \hat{\delta}' \models e \rightarrow \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

Proof. By induction on the height of the derivation tree $\hat{\delta} \models e \rightarrow \hat{v}$. The proof of this and the other theorems are reported in Appendix A, Appendix B and Appendix C. □

2.4 Examples

To illustrate how $\mathcal{C}$ can be used, we give two examples. The first example is the identity function.

```
id :: α → α
id x = x
```

The function model for `id` expresses that the label of the result should be the label of the parameter. This is computed by storing the label under the name α in the model state, when `id` is called, and then interpreting the α in the resulting model state, when the function returns.

The second example is the `min` function, which illustrates how more than one label can be stored into the model state.

```
min :: α1 α2 → α1 ⊔ α2
min x y = if x < y then x else y
```

Since the result of the `min` function is dependent on both parameters, the result should be the least upper bound of the labels of the parameters. To achieve this, both labels are stored in the model state on the call; the first label as α_1 and the second as α_2. The function model uses the label expression $\alpha_1 \sqcup \alpha_2$, which, when interpreted in the model state results in the least upper bound of the labels.

2.5 A note on the policy language

While we, in this work, strive to keep the model language simple, to enable us to study the processes of labeling and unlabeling vis-à-vis different language constructs, it is worthwhile to mention a few possible avenues for extensions. First, consider the following example, where the library function `f` calls the library function `min`. Instead of forcing the model of `f` to repeat the model of `min` it would be possible to add some form of *model application*, where the model of `min` is instantiated with the labels from `f`.

```
f :: α1 α2 → min α1 α2
f x y = min x y
```

This allows for a systematic construction of more complex models (nothing prevents us from introducing models that don't correspond to library functions).

Further, since the models are evaluated at runtime, they could be extended to have access to the *values* of the parameters in addition to the labels. This would allow for *dependent models*, where different labels are computed depending on the value of the parameters. Consider, for instance, the following library function.

```
f :: α1 α2 → x?α1 ⊔ α2 : α1
f x y = if x then y else 0
```

In this example the model uses the value of the parameter (stored in the model state under the parameter name) in order to select between two labels. In a language more complex than $\mathcal{C}$, those additions provide important expressiveness to the model language.

3 Lists $\mathcal{L}$

Structured data pose interesting challenges in relation to marshaling between the monitored and unmonitored semantics. While the unlabel and label processes must follow the structure of the values passed, structured data offer more freedom in the design of the unlabel and label models. In addition, fundamental questions pertaining to the time and extent of labeling and unlabeling arise. When passing a labeled list to the library, should the list

be marshaled in a strict or a lazy fashion? For library functions that only use parts of the passed data, strict marshaling can be both expensive and potentially imprecise, in particular when large object graphs are passed to or from the library (cf., getting an object from the DOM, where strict marshaling would be prohibitively expensive).

For this reason, we explore the notion of lazy marshaling. The idea is to marshal only when the opposite program part actually makes use of the data that has been passed. Unlabeling (or labeling in the dual setting) occurs only when the library (dually, program) actually uses the data, and only the part of the data that was used is unlabeled. This requires us to be able to pass data in such a manner that we can trap any interaction and unlabel or relabel on the fly. To this end, we opt for a solution that is inspired by the Proxy objects of JavaScript [22] but cast in terms of lists, and use a representation of lists that allow for proxying. The approach is general in the sense that it scales well to other types of structural data and that it can be implemented in different ways, e.g., proxies and accessor methods, both available in a range of languages, including JavaScript, Python and Objective C. One limitation of the approach is that some form of programming language support, that allows for trapping the read and write interaction of the library with given objects, is needed. If such support is not available, one can always resort to strict marshaling, which corresponds to a relatively immediate lifting of the label and unlabel functions of the core language to structured data. Most of the ideas presented in this paper should carry over to strict marshaling with little effort at the cost of efficiency and precision of the marshaling.

3.1 Syntax

From a syntactic standpoint the extension of $\mathcal{C}$ to support lists is small; the empty list, [ ], the cons operation, :, and operations for getting the head, $head$, and tail, $tail$, of lists are added.

$$\begin{aligned} e ::= \quad & n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid f\ \boldsymbol{e} \mid f_{lib}\ \boldsymbol{e} \mid e_1 \oplus e_2 \mid \\ & [\,] \mid e : e \mid \mathit{head}\ e \mid \mathit{tail}\ e \end{aligned}$$

3.2 Semantics

In JavaScript, a Proxy is an object that forwards all interactions to a set of user defined functions, provided at the creation time of the Proxy. Once the Proxy object has been created, it can be interacted with like a normal object. Thus, e.g., by defining a function corresponding to *get*, all property reads of the proxy object can be trapped and modified — the return value of the function will be the result of the read. The fundamental property that

makes Proxies suitable for lazy marshaling is that they allow the functions to modify all possible interactions with the object.

Unlike the strict marshaling of the core language, where the model state is computed before entering the library, the introduction of lazy marshaling requires the model state to be updated during the execution of the library function (in case the function interacts with the passed data). In a practical setting, the monitored program and the unmonitored library would share memory (they are different parts of the same program). This means that it is easy to maintain the model state in the presence of lazy marshaling. In an operational semantics, mutable state is modeled by threading the state through the evaluation.

Values We model proxyable lists as pairs of functions $(\hat{H}, \hat{T})$ and (H, T) respectively.

$$\hat{v} ::= n^{\ell} \mid (\hat{H}, \hat{T})^{\ell} \mid [\,]^{\ell} \qquad v ::= n \mid (H, T) \mid [\,]$$

The idea is that $\hat{H}$ and H return the head of the list, and $\hat{T}$ and T return the tail (which can be the empty list). This representation allows for an elegant lazy marshaling of lists, when they are passed between the program and the library, by wrapping the head and tail functions. The actual marshaling takes place only when the function is called, i.e., when the respective value is read.

Stateful marshaling In order to support unlabeling and labeling of lists we must extend the unlabel and label models. Since we are mainly interested in the stateful marshaling, we use a simple extension that differentiates between the labels of the values and the label of the structure of the lists [18]. See Section 3.5 for a discussion on possible extensions.

$$\varphi ::= \alpha \mid [\,\varphi\,]_{\alpha} \qquad \gamma ::= \kappa \mid [\,\gamma\,]_{\kappa}$$

The intuition for unlabel models is that, whenever a value is read from the list, the model state is updated accordingly. This means that the model state can be changed during the execution of the library, which must be reflected in the unlabeled semantics. The same is not true for the labeled semantics; any value passed from the unlabeled world will be labeled with respect to the model state at the time of return, even if the labeling is lazy. This leads to a seeming asymmetry in the semantics reflected by the definition of the head and tail functions for lists.

$$\begin{array}{llll} \hat{H} & : () \to \hat{v} & H & : \xi \to (\xi, v) \\ \hat{T} & : () \to \hat{v} & T & : \xi \to (\xi, v) \end{array}$$

The way to interpret this asymmetry is not that the unlabeled semantics has to be changed to enable marshaling — as described above, mutable state is modeled by threading the state through the computation. Rather, the asymmetry arises from the fact that the model state is only important for the evaluation of library functions called from the monitored semantics.

With respect to the unlabel and label operations, they must be updated to handle the extended unlabel and label models.

$$\begin{array}{lcl} [\,]^{\ell} \downarrow [\,\varphi\,]_{\alpha} & = & ([\,], [\alpha \mapsto \ell]) \\ (\hat{H}, \hat{T})^{\ell} \downarrow [\,\varphi\,]_{\alpha} & = & ((\mathsf{unlabel}(\hat{H}, \varphi), \mathsf{unlabel}(\hat{T}, [\,\varphi\,]_{\alpha})), [\alpha \mapsto \ell]) \end{array}$$

The unlabeling of lists updates the structure label and wraps the head and tail of the list (if present) with unlabeling wrappers, that unlabel with respect to the unlabel model. On access the wrapper receives the model state (of the current call to the library), after which it uses $\hat{H}$ to get the labeled value, and φ to unlabel. The unlabeled value is returned together with an updated model state, where $\xi \sqcup \xi'$ is defined as the union of ξ and ξ' under least upper bound of shared mappings. The wrapper for the tail of the list works analogously, but with respect to the full unlabel model of the list $[\,\varphi\,]_{\alpha}$.

$$\mathsf{unlabel}(\hat{H}, \varphi) = \lambda\xi\,.\,(\xi \sqcup \xi', v), \quad \text{where } \hat{H}() = \hat{v} \text{ and } \hat{v} \downarrow \varphi = (v, \xi')$$

$$\mathsf{unlabel}(\hat{T}, [\,\varphi\,]_{\alpha}) = \lambda\xi\,.\,(\xi \sqcup \xi', v), \quad \text{where } \hat{T}() = \hat{v} \text{ and } \hat{v} \downarrow [\,\varphi\,]_{\alpha} = (v, \xi')$$

The labeling of lists is similar, with the difference that the labeling is done with respect to the final model state. Once evaluation has returned, nothing can change the model state corresponding to the call.

$$\begin{array}{lcl} [\,] \uparrow_{\xi} [\,\gamma\,]_{\kappa} & = & [\,]^{[\![\kappa]\!]_{\xi}} \\ (H, T) \uparrow_{\xi} [\,\gamma\,]_{\kappa} & = & (\mathsf{label}(H, \xi, \gamma), \mathsf{label}(T, \xi, [\,\gamma\,]_{\kappa}))^{[\![\kappa]\!]_{\xi}} \end{array}$$

The wrappers are given the model state, ξ, and the label model, γ. On access the wrapper uses H to get the unlabeled value, v. Notice, how this may actually extend the model state to ξ' (it could be the case that H is an unlabel wrapper) and that ξ' is used together with γ to compute a label for v. This new model state does not have to be propagated, though. If the value was used by the unlabeled world in the creation of the tail of the list its label is already included in ξ.

The relabeling of the tail of the list works analogously, but with respect to the label model of the list $[\,\gamma\,]_{\kappa}$. Any extension of the model state is passed to the wrapping of the tail.

$$\mathsf{label}(H, \xi, \gamma) = \lambda()\,.\,\hat{v}, \quad \text{where } H(\xi) = (\xi', v) \text{ and } v \uparrow_{\xi'} \gamma = \hat{v}$$

$$\mathsf{label}(T, \xi, [\,\gamma\,]_{\kappa}) = \lambda()\,.\,\hat{v}, \quad \text{where } T(\xi) = (\xi', v) \text{ and } v \uparrow_{\xi'} [\,\gamma\,]_{\kappa} = \hat{v}$$

$$\text{empty}\ \frac{}{\hat{\delta} \models [\,] \to [\,]^L} \qquad \text{cons}\ \frac{\hat{\delta} \models e_1 \to \hat{v}_1 \qquad \hat{\delta} \models e_2 \to \hat{v}_2}{\hat{\delta} \models e_1 : e_2 \to \text{lcons}(\hat{v}_1, \hat{v}_2)^L}$$

$$\text{head}\ \frac{\hat{\delta} \models e \to (\hat{H}, \hat{T}) \qquad \hat{H}() = \hat{v}}{\hat{\delta} \models \mathit{head}\ e \to \hat{v}} \qquad \text{tail}\ \frac{\hat{\delta} \models e \to (\hat{H}, \hat{T}) \qquad \hat{T}() = \hat{v}}{\hat{\delta} \models \mathit{tail}\ e \to \hat{v}}$$

$$\text{lib}\ \frac{\Delta[f] = (\boldsymbol{x}, e_f) \qquad \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \qquad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \qquad \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \qquad [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi, e_f \rangle \rightsquigarrow \langle \xi', v \rangle \qquad v \uparrow_{\xi'} \gamma = \hat{v}}{\hat{\delta} \models f_{lib}\ \boldsymbol{e} \to \hat{v}}$$

Figure 1.3: Labeled semantics of lists

Unlabeled and labeled semantics The additions to the labeled semantics, found in Figure 1.3, are straightforward given the above modeling. Let $\text{lcons}(\hat{v}_1, \hat{v}_2) = (\lambda().\ \hat{v}_1, \lambda().\ \hat{v}_2)$ be the creation of labeled cons cells[2], used in the evaluation of the : operator (`cons`). The evaluation of head and tail (`head`, and `tail`) uses the head and the tail function respectively to get the value. Notice, how the model state may be modified during the execution of the library, and how the return value is labeled in the modified state (`lib`).

With respect to the unlabeled semantic, the entire semantics must be lifted to thread the model state, $\delta \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, v \rangle$. This modification is straightforward and can be found, along with the additions to the unlabeled semantics, in Figure 1.4. Let $\text{ucons}(v_1, v_2) = (\lambda\xi\ .\ (\xi, v_1), \lambda\xi\ .\ (\xi, v_2))$ be the creation of unlabeled cons cells, used in the evaluation of the : operator (`cons`). The evaluation of head and tail (`head`, and `tail`) uses the head and tail function respectively to get the value. Notice that the model state is threaded in this case — this is what allows for the lazy unlabeling. In case the head or tail function is an unlabel wrapper, the state will be updated.

3.3 Correctness

Definition 2 (Correctness of the library models)**.** *A library model correctly models a library if every function, f, in the library, $\Delta[f] = (\boldsymbol{x}, e)$, respects the associated function model, $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$, if present. Notice that, even though the final model states may differ (due to different interactions with marshaled labeled values in the two runs), a correct library model must ensure that the label is independent on the differences and that the values are low-equivalent with respect to the labeling.*

[2]The term originates from Lisp. In addition, cons is used as the name for the list-forming operator in many functional languages.

$$\text{int}\frac{}{\delta \models \langle \xi, n\rangle \rightsquigarrow \langle \xi, n\rangle} \qquad \text{var}\frac{\delta[x] = v}{\delta \models \langle \xi, x\rangle \rightsquigarrow \langle \xi, v\rangle}$$

$$\text{op}\frac{\delta \models \langle \xi_1, e_1\rangle \rightsquigarrow \langle \xi_2, v_1\rangle \qquad \delta \models \langle \xi_2, e_2\rangle \rightsquigarrow \langle \xi_3, v_2\rangle}{\delta \models \langle \xi_1, e_1 \oplus e_2\rangle \rightsquigarrow \langle \xi_3, v_1 \oplus v_2\rangle}$$

$$\text{if}_1\frac{\delta \models \langle \xi_1, e_1\rangle \rightsquigarrow \langle \xi_2, v\rangle \qquad v \neq 0 \qquad \delta \models \langle \xi_2, e_2\rangle \rightsquigarrow \langle \xi_3, v\rangle}{\delta \models \langle \xi_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\rangle \rightsquigarrow \langle \xi_3, v\rangle}$$

$$\text{if}_2\frac{\delta \models \langle \xi_1, e_1\rangle \rightsquigarrow \langle \xi_2, v\rangle \qquad v = 0 \qquad \delta \models \langle \xi_2, e_3\rangle \rightsquigarrow \langle \xi_3, v\rangle}{\delta \models \langle \xi_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3\rangle \rightsquigarrow \langle \xi_3, v\rangle}$$

$$\text{let}\frac{\begin{array}{c}\delta \models \langle \xi_1, e_1\rangle \rightsquigarrow \langle \xi_2, v_1\rangle \\ \delta[x \mapsto v_1] \models \langle \xi_2, e_2\rangle \rightsquigarrow \langle \xi_3, v_2\rangle\end{array}}{\delta \models \langle \xi_1, \mathit{let}\ x = e_1\ \mathit{in}\ e_2\rangle \rightsquigarrow \langle \xi_3, v_2\rangle}$$

$$\text{app}\frac{\begin{array}{c}\Delta[f] = (\boldsymbol{x}, e_f) \qquad \delta \models \langle \xi_1, \boldsymbol{e}\rangle \rightsquigarrow \langle \xi_2, \boldsymbol{v}\rangle \\ [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_2, e_f\rangle \rightsquigarrow \langle \xi_3, v\rangle\end{array}}{\delta \models \langle \xi_1, f\ \boldsymbol{e}\rangle \rightsquigarrow \langle \xi_3, v\rangle}$$

$$\text{empty}\frac{}{\delta \models \langle \xi, [\,]\rangle \rightsquigarrow \langle \xi, [\,]\rangle}$$

$$\text{cons}\frac{\begin{array}{c}\delta \models \langle \xi_1, e_1\rangle \rightsquigarrow \langle \xi_2, v_1\rangle \\ \delta \models \langle \xi_2, e_2\rangle \rightsquigarrow \langle \xi_3, v_2\rangle\end{array}}{\delta \models \langle \xi_1, e_1 : e_2\rangle \rightsquigarrow \langle \xi_3, \mathsf{ucons}(v_1, v_2)\rangle}$$

$$\text{head}\frac{\begin{array}{c}\delta \models \langle \xi_1, e\rangle \rightsquigarrow \langle \xi_2, (H, T)\rangle \\ H(\xi_2) = (\xi_3, v)\end{array}}{\delta \models \langle \xi_1, \mathit{head}\ e\rangle \rightsquigarrow \langle \xi_3, v\rangle} \qquad \text{tail}\frac{\begin{array}{c}\delta \models \langle \xi_1, e\rangle \rightsquigarrow \langle \xi_2, (H, T)\rangle \\ T(\xi_2) = (\xi_3, v)\end{array}}{\delta \models \langle \xi_1, \mathit{tail}\ e\rangle \rightsquigarrow \langle \xi_3, v\rangle}$$

Figure 1.4: Unlabeled semantics of lists

$$\begin{array}{r}\forall f\ .\ \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \Delta[f] = (\boldsymbol{x}, e) \\ \wedge\ \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1) \\ \wedge\ [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e\rangle \rightsquigarrow \langle \xi_2, v\rangle \wedge [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models \langle \xi_1, e\rangle \rightsquigarrow \langle \xi_2', v'\rangle \Rightarrow \\ v \uparrow_{\xi_2} \gamma \simeq v' \uparrow_{\xi_2'} \gamma\end{array}$$

As is standard we prove noninterference as the preservation of a low-equivalence relation under execution, extended from Section 2.3 with lists as follows.

$$\frac{}{[\,]^L \simeq [\,]^L} \qquad \frac{}{v_1^H \simeq v_2^H} \qquad \frac{\hat{H}() \simeq \hat{H}'() \quad \hat{T}() \simeq \hat{T}'()}{(\hat{H}, \hat{T})^L \simeq (\hat{H}', \hat{T}')^L}$$

Under the assumption that Definition 2 holds, we can prove noninterference for labeled execution.

Theorem 2 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

3.4 Examples

We present a selection of examples to illustrate different aspects of our models. Consider first the `length` function, that recursively computes the length of the given list.

```
length :: [ α1 ]α2  →  α2
length l = if l == [] then 0 else 1 + length (tail l)
```

The function traverses the list until the empty list is found without looking at the elements. During this traversal, the security labels corresponding to the cons cells are accumulated into the label variable α_2, which is used to label the result. This corresponds precisely to the structure security label of lists in [18]. It is, thus, possible to have functions that are dependent on the structure of a list, but not the content.

The other way, however, is not possible. Getting an element from a list always reveals information about the structure of the list. Thus, the `sum` function, which sums the element of the list must also take the labels of the cons cells into account.

```
sum :: [ α1 ]α2  →  α1 ⊔ α2
sum l = if l == [] then 0 else head l + sum (tail l)
```

Consider the function `replicate`, that creates a list by replicating a given element, x, n times. The length of the list is given by the label of n and the label of the elements by the label of x. Notice the limitation in the current label models. By giving the second argument the unlabel model α_2, we force `replicate` to take integers — lists cannot be unlabeled by α_2. In such cases, *polymorphic models* are needed, see below in Section 3.5.

```
replicate :: α1 α2  →  [ α2 ]α1
replicate n x = if n == 0 then []
                else x : replicate (n - 1) x
```

Related to both `sum` and `replicate` consider the function `take`, that takes an integer, n, and a list, l, and returns the n first elements of l. Clearly, the length of the list is dependent on both the label of n, α_1, and the structure of the list α_3. Notice, that the label of the structure of the list is accumulated into α_3 as the function traverses the list. This means that, given a list, where the first k cons cells are public, followed by some number of secret cons cells,

`take` will yield lists with public structure, as long as no more than k elements are taken. Once more than k elements are taken, however, the labels of all cons cells will be secret. Unfortunately, this is the same for the labels of the values, which are all joined into α_2, see Section 3.5.

```
take :: α1 [ α2 ]α3 → [ α2 ]α1⊔α3
take n l = if l == [] || n == 0 then []
           else head l : take (n - 1) (tail l)
```

Finally, consider the function `takeUntilZero`, that takes an unknown number of elements from the list. In this function, the length of the list is dependent on the labels of the values of the list, as well as the labels of the traversed cons cells. As before, only the labels of the cons cells that actually take part in the computation are part of the accumulated label for α_2.

```
takeUntilZero :: [ α1 ]α2 → [ α1 ]α1⊔α2
takeUntilZero l = if l == [] || head l == 0 then []
                  else head l : takeUntilZero (tail l)
```

3.5 A note on the policy language

With respect to the policy language, there are a number of possible paths to explore. First, consider a form of polymorphic models, where we add variables, x, to the policy language. Unlike α, the intention is that x can map to structured labels (potentially in combination with the values, see Section 2.5). This would enable the following.

```
replicate :: α x → [ x ]α
replicate n x = if n == 0 then []
                else x : replicate (n - 1) x
```

where x would allow any type of value to be repeated. It is also possible to envision other operations on such variables, such as $@x$, the computation of the least upper bound of the labels reachable from x.

Additionally, it is natural to extend the model language with some form of pattern matching on lists, as follows.

```
f :: (α1 : α2 : [ α3 ]α4) → α3 ⊔ α4
f ls = sum (drop 2 ls)
```

In this case, the first two elements are dropped before the remainder is summed together. An interesting avenue of research is to explore this in combination with dependent models and richer models for building structured data.

4 Higher-order functions $\mathcal{F}$

After having investigated how to pass structured and unstructured data between the program and the library, we turn the attention to the passing of computations, in terms of higher-order functions. The passing of functions between programs and libraries is commonplace, used in the presence of, e.g., asynchronous operations. Examples of this are *callbacks*, where functions are passed to the library, allowing it to inform the program of certain events, and promises [21], that rely on the ability to pass functions in both directions.

4.1 Syntax

To investigate higher-order functions, we extend the core language with a function expression, $\mathit{fun}\ \boldsymbol{x} \Rightarrow e$ and change function calls to a computed call target. The introduction of higher-order functions subsumes top-level function definitions. Instead, we allow for top-level *let* declarations, $\mathit{let}\ x = e$, and corresponding model declarations, $x :: \gamma$.

$$\begin{array}{rl} e ::= & n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid f\ \boldsymbol{e} \mid f_{\mathit{lib}}\ \boldsymbol{e} \mid e_1 \oplus e_2 \mid \\ & e\ \boldsymbol{e} \mid \mathit{fun}\ \boldsymbol{x} \Rightarrow e \\ \\ d ::= & \mathit{let}\ x = e \\ m ::= & x :: \gamma \end{array}$$

4.2 Semantics

Fundamentally, we use the same approach as with lists and represent closures as functions instead of structured values. This allows us to marshal functions from the labeled world to the unlabeled world and back without the need to distinguish between the origin of the values in the respective semantics. Intuitively, this corresponds to using functions as the calling convention and mimics what is actually in a practical implementation[3].

Following the development of Section 3, we add functional closures to the values as follows.

$$\hat{v} ::= n^{\ell} \mid \hat{F}^{\ell} \qquad v ::= n \mid F$$

where labeled closures, $\hat{F}$, take sequences of labeled values to labeled values and unlabeled closures, F, also thread a model state

$$\hat{F} : \hat{\boldsymbol{v}} \to \hat{v} \qquad F : (\xi, \boldsymbol{v}) \to (\xi, v)$$

[3] In a practical implementation, the program and the library would use the calling convention of the computer — regardless of the implementation language of the two.

With respect to the asymmetry of the semantics, the intuition is the same as before: the model state resides in shared memory, but, since the labeled semantics never modifies the model state we do not need to thread the model state through the labeled semantics.

Stateful marshaling Conceptually, any function defined in the library that can be called from the monitored program, whether passed as a closure or called, must be given a label model, that defines how to label the closure as a value, how to unlabel the parameters and label the result (c.f., the function models in Section 2). The question is, how to unlabel a closure, when passing it from the monitored program to the library. Intuitively, the unlabel model should be the dual of the label model, i.e., unlabel the closure as a value, label the parameters and unlabel the result. The problem is, that both unlabeling and labeling is performed in relation to a model state, which cannot be assumed to be the same as when the closure was passed as a parameter (it could be an extension — the passed closure could be called from an inner function). For this reason, we cannot tie an unlabel model to the closure at the point of unlabeling; it must be provided at the point of call. To be able to connect closures to calls, closures are tagged with a provided abstract identifier, π, when unlabeled. This abstract identifier is used in the label models for library functions to connect called closures with *call models* that express how to label the parameters and unlabel the result in the model state of the caller.

$$\varphi ::= \alpha \mid \pi^{\alpha} \qquad \gamma ::= \kappa \mid (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} \qquad \zeta ::= \pi\, \boldsymbol{\gamma} \to \varphi$$

Unlabel models for labeled closures, π^{α}, provide both abstract identifiers, π, and label variables, α, while the label models of unlabeled closures, $(\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa}$, contain how to label the closure as a value, κ, how to unlabel the parameters, $\boldsymbol{\varphi}$, how to label the result, γ, and how to label calls to callbacks, $\boldsymbol{\zeta}$. These call models, ζ, tie abstract identifiers, π, to call models, i.e., how to label the parameters, $\boldsymbol{\gamma}$, and how to unlabel the result, φ. Linked by the abstract identifier, the unlabel model for labeled closures together with the call models can be seen as duals to the label models for unlabeled closures.

Unlabeling of labeled closures is similar to unlabeling of values and lists, and places an unlabel wrapper around the labeled closure. The unlabel wrapper is, additionally, given the abstract identifier, π, used to tie future calls to the corresponding call models.

$$v^{\ell} \downarrow \alpha = (v, \xi[\alpha \mapsto \ell]) \qquad \hat{F}^{\ell} \downarrow \pi^{\alpha} = (\text{unlabel}(\hat{F}^{\ell}, \pi), [\alpha \mapsto \ell])$$

The unlabel wrapper becomes an unlabeled closure, that takes a model state, ξ, and a sequence of unlabeled values, $\boldsymbol{v}$, and finds the call model $\boldsymbol{\gamma} \to \varphi$

$$\text{fun}\ \frac{v = \text{lclos}(\hat{\delta}, \boldsymbol{x}, e)}{\hat{\delta} \models \mathit{fun}\ \boldsymbol{x} \Rightarrow e \to v^L} \qquad \text{app}\ \frac{\hat{\delta} \models e \to \hat{F}^\ell \quad \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \quad \hat{F}(\hat{\boldsymbol{v}}) = \hat{v}}{\hat{\delta} \models e\ \boldsymbol{e} \to \hat{v}^\ell}$$

$$\text{lib}\ \frac{\delta_0[f] = F \quad \xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa \quad F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa = \hat{F}^\ell}{\hat{\delta} \models f_{lib} \to \hat{F}^\ell}$$

Figure 1.5: Labeled semantics for higher-order functions

corresponding to the abstract identifier, π. Thereafter, $\boldsymbol{\gamma}$ is used to label the values, which are passed to the labeled closure, $\hat{F}$, to get a labeled value, $\hat{v}$. The labeled value is unlabeled using φ, which produces an unlabeled value and an update to the model state, ξ'. The result of the call to the wrapper is an updated model state and the unlabeled value. Notice how the label of the closure ℓ is used to raise the returned value before the unlabeling.

$$\begin{aligned}&\text{unlabel}(\hat{F}^\ell, \pi) = \lambda(\xi, \boldsymbol{v})\ .\ (\xi \amalg \xi', v),\\ &\quad \text{where } \xi[\pi] = \boldsymbol{\gamma} \to \varphi \text{ and } \hat{F}(\boldsymbol{v} \uparrow_\xi \boldsymbol{\gamma}) = \hat{v} \text{ and } \hat{v}^\ell \downarrow \varphi = (v, \xi')\end{aligned}$$

Labeling of unlabeled closures places a label wrapper around the closure. The label wrapper is additionally given the model state, ξ, how to unlabel the parameters, $\boldsymbol{\varphi}$, how to label return value, γ, and the call models, $\boldsymbol{\zeta}$.

$$v \uparrow_\xi \kappa = v^{[\![\kappa]\!]_\xi} \qquad F \uparrow_\xi (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa = \text{label}(F, \xi, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_\xi}$$

The label wrapper becomes a labeled closure, that takes a sequence of labeled values, $\hat{\boldsymbol{v}}$, unlabels the value producing a sequence of values, $\boldsymbol{v}$, and an update to the model state, ξ'. The updated model state is extended with the call models of the function (replacing the previously defined), producing a new model state ξ_2 by threading

$$[\![\pi\ \boldsymbol{\kappa} \to \varphi]\!]_\xi = \xi[\pi \mapsto (\boldsymbol{\kappa} \to \varphi)]$$

through the sequence $\boldsymbol{\zeta}$. The produced model state is used in the execution of the unlabeled closure, F, together with the unlabeled values producing an unlabeled value, v, and the final model state, ξ_3. The result is the labeled value $\hat{v}$, created by labeling v with respect to γ and the final model state.

$$\begin{aligned}&\text{label}(F, \xi, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta}) = \lambda\hat{\boldsymbol{v}}\ .\ \hat{v},\\ &\quad \text{where } \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi') \text{ and } [\![\boldsymbol{\zeta}]\!]_{\xi \amalg \xi'} = \xi_2\\ &\quad \text{and } F(\xi_2, \boldsymbol{v}) = (\xi_3, v) \text{ and } v \uparrow_{\xi_3} \gamma = \hat{v}\end{aligned}$$

$$\text{fun}\ \frac{v = \text{uclos}(\boldsymbol{\delta}, \boldsymbol{x}, e)}{\boldsymbol{\delta} \models \langle \xi, \mathit{fun}\ \boldsymbol{x} \Rightarrow e \rangle \rightsquigarrow \langle \xi, v \rangle}$$

$$\text{app}\ \frac{\boldsymbol{\delta} \models \langle \xi_1, e \rangle \rightsquigarrow \langle \xi_2, F \rangle \quad \boldsymbol{\delta} \models \langle \xi_2, \boldsymbol{e} \rangle \rightsquigarrow \langle \xi_3, \boldsymbol{v} \rangle \quad F(\xi_3, \boldsymbol{v}) = \langle \xi_4, v \rangle}{\boldsymbol{\delta} \models \langle \xi_1, e\ \boldsymbol{e} \rangle \rightsquigarrow \langle \xi_4, v \rangle}$$

Figure 1.6: Unlabeled semantics for higher-order functions

.

Labeled semantics The labeled semantics is mostly unaffected by the extension, apart from the rule for higher-order functions (`fun`), the rule for function call (`app`) and the rule for library call (`lib`). The modified rules are found in Figure 1.5 and make use of closure creation, lclos, defined as follows.

$$\text{lclos}(\hat{\boldsymbol{\delta}}, \boldsymbol{x}, e) = \lambda \hat{\boldsymbol{v}}\ .\ \hat{v}, \text{where } \hat{\delta}[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e \rightarrow \hat{v}$$

In the semantics $\hat{\delta}_0$, and δ_0 are created by evaluating the top levels of the labeled and the unlabeled world, respectively. This creates all top level closures used in function and library calls. Similarly, ξ_0 is created from the model definitions of the library, and is used as the initial model state.

Function call (`app`) evaluates the function expression to a closure and the parameters to a sequence of labeled values, $\hat{\boldsymbol{v}}$. The closure is called by supplying the labeled values and the result is returned, but with the label raised to the label of the closure. The library call has been replaced with a rule that lifts an unlabeled closure to the labeled world (`lib`). This is done by looking up the unlabeled closure in the initial environment of the library δ_0, and the corresponding function model in the initial model state ξ_0. The labeled (wrapped) closure is then returned as the result. Thus, in line with the intuition of using functions as the calling convention, functions in the program and in the library are translated to functions that are called in the same manner in the function call rule.

Unlabeled semantics In the unlabeled semantics, a rule for higher-order functions (`fun`) has been added and the rule for function application (`app`) has been changed. The modified rules are found in Figure 1.6 and are analogous with the changes made to the labeled semantics, including the use of closure creation defined as follows.

$$\text{uclos}(\boldsymbol{\delta}, \boldsymbol{x}, e) = \lambda(\xi_1, \boldsymbol{v})\ .\ (\xi_2, v), \text{where } \delta[\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e \rangle \rightarrow \langle \xi_2, v \rangle$$

4.3 Correctness

We prove correctness under the assumption that the library model correctly models the library.

Definition 3 (Correctness of the library models)**.** *A library model correctly models a library if every closure, f, in the library, $\delta_0[f] = F$, respects the associated function model, $\xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa$, if present.*

$$\begin{aligned}\forall f\,.\,\xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^\kappa \wedge \delta_0[f] = F \\ \wedge\, \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1) \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1) \wedge [\![\boldsymbol{\zeta}]\!]_{\xi_1} = \xi_2 \wedge \\ F(\xi_2, \boldsymbol{v}) = (\xi_3, v) \wedge F(\xi_2, \boldsymbol{v}') = (\xi_3', v') \wedge \Rightarrow v \uparrow_{\xi_3} \gamma \simeq v' \uparrow_{\xi_3'} \gamma\end{aligned}$$

As is standard we prove noninterference as the preservation of a low-equivalence relation under execution, extended from Section 2.3 with higher-order functions as follows.

$$\frac{}{v_1^H \simeq v_2^H} \qquad \frac{\forall \hat{v}, \hat{v}'\,.\,\hat{v} \simeq \hat{v}' \Rightarrow \hat{F}(\hat{v}) \simeq \hat{F}'(\hat{v}')}{\hat{F}^L \simeq \hat{F}'^L}$$

Under Definition 3 holds, we can prove noninterference for labeled execution.

Theorem 3 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

4.4 Examples

To illustrate models for higher-order functions we consider three examples. In the examples, the library top-level contains a let with a higher-order function, which is paired with a function model. Before the program is run the top-level let bindings in the library and the unmonitored program (in that order) is evaluated to values. As illustrated in the second example, this means that execution no longer needs to start in a predefined function. Instead, computation can be started from any of the let bindings that do not produce closures.

The first example takes a callback and immediately calls it with a constant, and the associated function model expresses that the function takes a closure, which will be unlabeled as α_1 and associated with the abstract name x (nothing prevents us from using the same name as the parameter). Further, the closure is called with a public parameter, and the result will be unlabeled as α_2, which is also the label of the result of the function.

```
f :: (x^α1 -> α2, x L -> α2)^L
let f = fun x => x 42
```

When calling the closure, the call model will be looked up and used to label the parameters — in this case giving 42 labeled with L. The result of the call will be unlabeled as α_2, before being labeled by α_2 and returned by the function.

The second example illustrates why callbacks cannot be associated with an unlabel model on the point of unlabeling.

```
let cb = fun x => x + 1
let main = let g = f_lib cb in g 10

-- library part
f :: (x^α1 -> (α2 -> α3, x α2 -> α3)^L)^L
let f = fun x => fun y => x y
```

When the callback `cb` is passed to `f` it is not called, rather a closure is returned which takes another parameter that is unlabeled into α_2, which in turn is used as the parameter to the callback. Thus, in order to correctly label the value of the parameter to the callback, α_2 must be in the model state. This is true for the second call `g 10` but not for the first f_{lib} `cb` in the monitored program.

Finally, consider an example with a conditional callback.

```
f :: (x^α1 -> (α2 -> α2 ⊔ α3, x α2 -> α3)^L)^L
let f = fun x => fun y => if y then x 42 else 42
```

The example illustrates the situation, where the callback may or may not be called depending on other values inspired by the frequent use of *coercions* in JavaScript libraries. This means that in some executions the variable α_2 may not be set. To handle this kind of situations it suffices that $[\![\alpha]\!]_\xi = L$, when $\xi[\alpha]$ is undefined. In addition, this interpretation allows for a limited form of dependent models.

5 Related work

There has been a substantial body of work in the area of dynamic information flow control in the past decade, to a large extent motivated by the desire to provide security and privacy for JavaScript web applications. There are two big lines of work. First, execution monitors [15, 1, 18, 17, 3] attach additional metadata (for instance, a security level) and propagate that metadata during the execution of a program. Second, multi-execution based approaches [6, 19, 27] essentially execute a program multiple times, and make sure that the execution that performs outputs at a certain security level has only seen information less than or equal to that security level. The multiple-facets

approach [2] is an optimized implementation of multi-execution, but it is less transparent. Bielova and Rezk [4] give a detailed survey and comparison of all kinds of dynamic information flow mechanisms, and we refer the reader to that paper for a detailed discussion. Both lines of work on dynamic information flow control (execution monitoring and multi-execution) have been applied to JavaScript in the browser [13, 16], and both have dealt with the problem of interfacing with libraries in a relatively ad-hoc way — essentially by manual programming of models of the library functions, or by treating API calls as I/O operations [14]. Rajani et al. [28] propose detailed and rigorous formal models of the DOM and event-handling parts of the browser, and find several potential information leaks. The work in this paper is a first step to a more principled approach of interfacing with such libraries that avoids the labor-intensive manual construction of such models (at the cost of potentially losing some precision).

The problem of interfacing with libraries where no dynamic checking of information flow control is possible, is related to the problem of checking contracts at the boundary between statically type-checked code and dynamically type-checked code. The problem of checking such contracts has been studied extensively in higher-order programming languages. Findler and Felleisen pioneered this line of work and proposed higher-order contracts [11]. The main challenge addressed is that of function values passed over the boundary. Compliance of such function values with their specified contract is generally undecidable. But it can be handled by wrapping the function with a wrapper that will check the contract of the function value at the point where the function is called. This is similar to how we handle function values in this paper, and an interesting question for future work is whether we can avoid the use of abstract identifiers for closures by injecting the appropriate labeling/unlabeling functionality using proxies only guided by how this is done in higher-order contract checking [8]. One concern that has received extensive attention is the proper assignment of *blame* once a contract violation is detected [12, 7]. Assigning blame for information flow violations has been investigated by King et al. [20] in the setting of static information flow checking. Our work could be seen as an application of the idea of dynamic higher-order contract checking to information flow contracts, something that to the best of our knowledge has not yet been considered before. We do not consider the issue of assigning blame: if the library does not comply with the specified contract, this is not detected at run-time.

Gradual typing [31, 32] is an approach to support the evolution of dynamically typed code to statically typed code, and it shares with our work the challenge of interfacing soundly between the dynamically checked part

of the program and the statically checked part that no longer propagates all run-time type information. It has also been applied in the setting of security type systems [9, 10], but it fundamentally differs in objective from our work. With gradual typing, the idea is to start from a program that is checked dynamically, and to gradually grow the parts that are statically checked. Our objective is to support interfacing with parts of the program for which dynamic checking is infeasible, either because the part is written in another language like C, or because dynamic checking would be too expensive to start with.

6 Conclusion

In this paper we have explored a method, *stateful marshaling*, that enables an information flow monitored program to call unmonitored libraries. The approach relies on storing the labels in a *model state* in accordance with an *unlabel model* before calling the library, and labeling the returned result by interpreting a *label model* in that model state.

Additionally, we have investigated *lazy marshaling* of structured data in terms of lists. The idea is similar to the concept of proxies and works by semantically representing lists as pairs of functions, that can be wrapped without recursively marshaling the entire list. When interacted with, the wrappers unlabel one step and return unlabeled primitive values or new lazy wrappers.

Finally, using functions to represent closures, we have shown how higher-order functions can be allowed to be passed in both directions. The approach relies on the concept of *abstract identifiers* that tie labeled closures, passed from the monitored program to the library, to call models, which describe how to label the parameters and unlabel the result with respect to the model state of the caller.

Future work We have preliminary results that show that lazy marshaling in combination with abstract identifiers is able to successfully handle references and the challenging combination of references and higher-order functions. Further, as discussed above, we aim to explore richer model languages, including but not limited to dependent models and model polymorphism. Finally, experiments with integrating our approach into JSFlow are subject to our current and future work.

Acknowledgments This work was partly funded by the European Community under the ProSecuToR project and the Swedish research agency VR.

7 Bibliography

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

[2] T. H. Austin and C. Flanagan. Multiple Facets for Dynamic Information Flow. In *POPL*, 2012.

[3] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *POST*, 2014.

[4] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *POST*, 2016.

[5] A. Birgisson, A. Russo, and A. Sabelfeld. Unifying facets of information integrity. In *ICISS*, 2010.

[6] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *S&P*, 2010.

[7] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, 2011.

[8] C. Dimoulas, M. S. New, R. B. Findler, and M. Felleisen. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*, 2016.

[9] T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.

[10] L. Fennell and P. Thiemann. Gradual security typing with references. In *CSF*, 2013.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[12] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL*, 2010.

[13] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *CCS*, 2012.

[14] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 2014.

[15] G. L. Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.

[16] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 2015.

[17] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

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

[19] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *S&P*, 2011.

[20] D. King, T. Jaeger, S. Jha, and S. A. Seshia. Effective blame for information-flow violations. In *FSE*, 2008.

[21] B. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI*, 1988.

[22] Mozilla Developer Network. Proxy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. accessed: Oct 2016.

[23] Mozilla Developer Network. Web APIs. `https://developer.mozilla.org/en-US/docs/Web/API`. accessed: Oct 2016.

[24] Node.js v6.9.1 Documentation. `https://nodejs.org/dist/latest-v6.x/docs/api/`. accessed: Oct 2016.

[25] Node Package Manager. `https://www.npmjs.com/`. accessed: Oct 2016.

[26] Oracle. Java Native Interface. `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`. accessed: Oct 2016.

[27] W. Rafnsson and A. Sabelfeld. Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent. In *CSF*, 2013.

[28] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer. Information Flow Control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015.

[29] N. Rajlich. node-ffi. `https://www.npmjs.com/package/node-ffi`. accessed: Oct 2016.

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

[31] J. G. Siek and W. Taha. Gradual Typing for Functional Languages. In *SFP*, 2006.

[32] J. G. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP*, 2007.

[33] H. wiki. Foreign Function Interface. `https://wiki.haskell.org/Foreign_Function_Interface`. accessed: Oct 2016.

A Soundness for $\mathcal{C}$

Theorem 1 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

The $\simeq$ relation is defined to be

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{n_1^H \simeq n_2^H} \qquad \frac{dom(\hat{\delta}) = dom(\hat{\delta}') \quad \forall x \in dom(\hat{\delta}) \,.\, \hat{\delta}[x] \simeq \hat{\delta}'[x]}{\hat{\delta} \simeq \hat{\delta}'}$$

Proof. By induction on the height, h, of the derivation tree $\hat{\delta} \models e \to \hat{v}$ taking the form of a case analysis on the last rule applied.

- **case** n: Based on the rule `int`, we have to show:

  $$\hat{\delta} \simeq \hat{\delta}' \Rightarrow n^L \simeq n^L$$

  The result follows immediately by the definition of $\simeq$.

- **case** x: Based on the rule `var`, we have to show:

  $$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta}[x] = \hat{v} \wedge \hat{\delta}'[x] = \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

  The result follows immediately by the definition of $\hat{\delta} \simeq \hat{\delta}'$.

- **case** $if\ e_1\ then\ e_2\ else\ e_3$: Conditionals have two rules; `if-1` and `if-2`. We must show

  $$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models if\ e_1\ then\ e_2\ else\ e_3 \to \hat{v} \\ & \wedge & \hat{\delta}' \models if\ e_1\ then\ e_2\ else\ e_3 \to \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

  Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

  1) $\hat{\delta} \models if\ e_1\ then\ e_2\ else\ e_3 \to \hat{v}$,

2) $\hat{\delta}' \models \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \to \hat{v}'$.

Show $\hat{v} \simeq \hat{v}'$.

Have 4) $\hat{\delta} \models e_1 \to v^{\ell}$ from 1.

Have 5) $\hat{\delta}' \models e_1 \to v'^{\ell'}$ from 2.

Have 6) $v^{\ell} \simeq v'^{\ell'}$ from IH, 0, 1, 2.

Have, 7) $\ell = \ell'$ by 6.

We get three cases based on 6, 7

- **case** $\ell = \ell' = L$ and $v = v' = 1$.
  Have 8) $\hat{\delta} \models e_2 \to \hat{v}$
  Have 9) $\hat{\delta}' \models e_2 \to \hat{v}'$
  The result follows from IH, 0, 8, 9.
- **case** $\ell = \ell' = L$ and $v = v' = 0$.
  Have 8) $\hat{\delta} \models e_3 \to \hat{v}$
  Have 9) $\hat{\delta}' \models e_3 \to \hat{v}'$
  The result follows from IH, 0, 8, 9.
- **case** $\ell = \ell' = H$
  Have 8) $\hat{v} = \hat{v}^{H}$
  Have 9) $\hat{v}' = \hat{v}'^{H}$
  The result follows trivially by definition of $\simeq$.

- **case** $\mathit{let}\ x = e_1\ \mathit{in}\ e_2$: Based on the rule `let`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e_1 \to \hat{v}_1 \wedge \hat{\delta}[x \mapsto \hat{v}_1] \models e_2 \to \hat{v}_2 \\ & \wedge & \hat{\delta}' \models e_1 \to \hat{v}_1' \wedge \hat{\delta}'[x \mapsto \hat{v}_1'] \models e_2 \to \hat{v}_2' \\ & \Rightarrow & \hat{v}_2 \simeq \hat{v}_2' \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\hat{\delta} \models e_1 \to \hat{v}_1$,

2) $\hat{\delta}[x \mapsto \hat{v}_1] \models e_2 \to \hat{v}_2$,

3) $\hat{\delta}' \models e_1 \to \hat{v}_1'$,

4) $\hat{\delta}'[x \mapsto \hat{v}_1'] \models e_2 \to \hat{v}_2'$.

Show $\hat{v}_2 \simeq \hat{v}_2'$.

Have 5) $\hat{v}_1 \simeq \hat{v}_1'$ from IH, 0, 1, 3.

Have 6) $\hat{\delta}[x \mapsto \hat{v}_1] \simeq \hat{\delta}'[x \mapsto \hat{v}_1']$ from Lemma 1.

Result follows from IH, 6, 2, 4.

- **case** $f\ \boldsymbol{e}$: Based on the rule `app`, we have to show:

$$\begin{array}{lll} \hat{\delta} \simeq \hat{\delta}' \wedge \hat{\Delta}[f] = (\boldsymbol{x}, e_f) & \wedge & \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \wedge [\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e_f \to \hat{v} \\ & \wedge & \hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}' \wedge [\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}'] \models e_f \to \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\hat{\Delta}[f] = (\boldsymbol{x}, e_f)$,

2) $\hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}}$,

3) $[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e_f \to \hat{v}$,

4) $\hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}'$,

5) $[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}'] \models e_f \to \hat{v}'$.

Show $\hat{v} \simeq \hat{v}'$.

Have 6) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$ from consecutively IH, $0, 2, 4$.

Have 7) $[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \simeq [\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}']$ from Lemma 1 (having $\hat{\delta}_1 = [\,], \hat{\delta}_1' = [\,]$), 6.

Result follows from IH, $7, 3, 5$.

- **case** $f_{lib}\ \boldsymbol{e}$: Based on the rule `lib`, we have to show:

$$\begin{array}{lll} \hat{\delta} \simeq \hat{\delta}' & \wedge & \Delta[f] = (\boldsymbol{x}, e_f) \wedge \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \\ & \wedge & \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi) \\ & \wedge & [\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v \wedge v \uparrow_\xi \gamma = \hat{v} \\ & \wedge & \hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi) \\ & \wedge & [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models e_f \rightsquigarrow v' \wedge v' \uparrow_\xi \gamma = \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\hat{\Delta}[f] = (\boldsymbol{x}, e_f)$,

2) $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$,

3) $\hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}}$,

4) $\hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi)$,

5) $[\boldsymbol{x} \mapsto \boldsymbol{v}] \models e_f \rightsquigarrow v$,

6) $v \uparrow_\xi \gamma = \hat{v}$,

7) $\hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}'$,

8) $\hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi)$,

9) $[\boldsymbol{x} \mapsto \boldsymbol{v}'] \models e_f \rightsquigarrow v'$,

10) $v' \uparrow_\xi \gamma = \hat{v}'$.

Show $\hat{v} \simeq \hat{v}'$.

Have 11) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$ from IH, 0, 3, 7.

Result follows from Definition 1 together with 11.

- **case** $e_1 \oplus e_2$: Based on the rule `op`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e_1 \to v_1^{\ell_1} \wedge \hat{\delta} \models e_2 \to v_2^{\ell_2} \\ & \wedge & \hat{\delta}' \models e_1 \to v_1'^{\ell_1'} \wedge \hat{\delta}' \models e_2 \to v_2'^{\ell_2'} \\ & \Rightarrow & (v_1 \oplus v_2)^{\ell_1 \sqcup \ell_2} \simeq (v_1' \oplus v_2')^{\ell_1' \sqcup \ell_2'} \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\hat{\delta} \models e_1 \to v_1^{\ell_1}$,

2) $\hat{\delta} \models e_2 \to v_2^{\ell_2}$,

3) $\hat{\delta}' \models e_1 \to v_1'^{\ell_1'}$,

4) $\hat{\delta}' \models e_2 \to v_2'^{\ell_2'}$.

Show $(v_1 \oplus v_2)^{\ell_1 \sqcup \ell_2} \simeq (v_1' \oplus v_2')^{\ell_1' \sqcup \ell_2'}$.

Have 5) $v_1^{\ell_1} \simeq v_1'^{\ell_1'}$ from IH, 0, 1, 3.

Have 6) $v_2^{\ell_2} \simeq v_2'^{\ell_2'}$ from IH, 0, 2, 4.

Have 7) $\ell_1 = \ell_1'$ from definition of $\simeq$, 5.

Have 8) $\ell_2 = \ell_2'$ from definition of $\simeq$, 6.

Have 9) Let ℓ denote $\ell_1 \sqcup \ell_2 = \ell_1' \sqcup \ell_2'$ from 7, 8.

Proceed by case analysis on ℓ

case $\ell = L$

Have 10) $v_1 = v_1'$ from 5.

Have 11) $v_2 = v_2'$ from 6.

The result follows from 10, 11 and that $\oplus$ is a function.

case $\ell = H$

The result follows from the definition of $\simeq$.

□

B Soundness for $\mathcal{L}$

Theorem 2 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

The $\simeq$ relation is defined to be

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{n_1^H \simeq n_2^H} \qquad \frac{dom(\hat{\delta}) = dom(\hat{\delta}') \quad \forall x \in dom(\hat{\delta}) \,.\, \hat{\delta}[x] \simeq \hat{\delta}'[x]}{\hat{\delta} \simeq \hat{\delta}'}$$

$$\frac{}{[\,]^L \simeq [\,]^L} \qquad \frac{}{v_1^H \simeq v_2^H} \qquad \frac{\hat{H}() \simeq \hat{H}'() \quad \hat{T}() \simeq \hat{T}'()}{(\hat{H}, \hat{T})^L \simeq (\hat{H}', \hat{T}')^L}$$

Proof. By induction on the height, h, of the derivation tree $\hat{\delta} \models e \to \hat{v}$ taking the form of a case analysis on the last rule applied. Since the rules `int`, `var`, `op`, $\texttt{if}_1$, $\texttt{if}_2$, `let`, `app` are analogous to the proof for $\mathcal{C}$, we refrain from repeating them. Hence, we only need to show soundness for `empty`, `cons`, `head`, `tail` and `lib`, defined in Figure 1.3.

- **case** $[\,]$: Based on the rule `empty`, we have to show:

  $$\hat{\delta} \simeq \hat{\delta}' \Rightarrow [\,]^L \simeq [\,]^L$$

  The result follows immediately by the definition of $[\,]^L \simeq [\,]^L$.

- **case** $e_1 : e_2$: Based on the rule `cons`, we have to show:

  $$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e_1 \to \hat{v}_1 \wedge \hat{\delta} \models e_2 \to \hat{v}_2 \\ & \wedge & \hat{\delta}' \models e_1 \to \hat{v}_1' \wedge \hat{\delta} \models e_2 \to \hat{v}_2' \\ & \Rightarrow & \text{lcons}(\hat{v}_1, \hat{v}_2)^L \simeq \text{lcons}(\hat{v}_1', \hat{v}_2')^L \end{array}$$

  Have 0) $\hat{\delta} \simeq \hat{\delta}'$,
  1) $\hat{\delta} \models e_1 \to \hat{v}_1$,
  2) $\hat{\delta} \models e_2 \to \hat{v}_2$,
  3) $\hat{\delta}' \models e_1 \to \hat{v}_1'$,
  4) $\hat{\delta}' \models e_2 \to \hat{v}_2'$.
  Show $\text{lcons}(\hat{v}_1, \hat{v}_2)^L \simeq \text{lcons}(\hat{v}_1', \hat{v}_2')^L$.
  Have 5) $\hat{v}_1 \simeq \hat{v}_1'$ from IH, $0, 1, 3$.
  Have 6) $\hat{v}_2 \simeq \hat{v}_2'$ from IH, $0, 2, 4$.

Have 7) $\text{lcons}(\hat{v}_1, \hat{v}_2) = (\lambda().\ \hat{v}_1, \lambda().\ \hat{v}_2)$ from definition of lcons.
Have 8) $\text{lcons}(\hat{v}'_1, \hat{v}'_2) = (\lambda().\ \hat{v}'_1, \lambda().\ \hat{v}'_2)$ from definition of lcons.
Have 9) $(\lambda().\ \hat{v}_1, \lambda().\ \hat{v}_2) = (\hat{H}, \hat{T})$ from definition of $\hat{H}$ and $\hat{T}$.
Have 10) $(\lambda().\ \hat{v}'_1, \lambda().\ \hat{v}'_2) = (\hat{H}', \hat{T}')$ from definition of $\hat{H}$ and $\hat{T}$.
Have 11) $\hat{H}() \simeq \hat{H}'()$ from $5, 9, 10$.
Have 12) $\hat{T}() \simeq \hat{T}'()$ from $6, 9, 10$.
Result follows from definition of $(\hat{H}, \hat{T})^L \simeq (\hat{H}', \hat{T}')^L, 11, 12$.

- **case** *head* e: Based on the rule `head`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e \to (\hat{H}, \hat{T}) \wedge \hat{H}() = \hat{v} \\ & \wedge & \hat{\delta}' \models e \to (\hat{H}', \hat{T}') \wedge \hat{H}'() = \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,
1) $\hat{\delta} \models e \to (\hat{H}, \hat{T})$,
2) $\hat{H}() = \hat{v}$,
3) $\hat{\delta}' \models e \to (\hat{H}', \hat{T}')$,
4) $\hat{H}'() = \hat{v}'$.
Show $\hat{v} \simeq \hat{v}'$.
Have 5) $(\hat{H}, \hat{T}) \simeq (\hat{H}', \hat{T}')$ from IH, $0, 1, 3$.
Result follows from $5, 2, 4$.

- **case** *tail* e: Based on the rule `tail`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e \to (\hat{H}, \hat{T}) \wedge \hat{T}() = \hat{v} \\ & \wedge & \hat{\delta}' \models e \to (\hat{H}', \hat{T}') \wedge \hat{T}'() = \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,
1) $\hat{\delta} \models e \to (\hat{H}, \hat{T})$,
2) $\hat{T}() = \hat{v}$,
3) $\hat{\delta}' \models e \to (\hat{H}', \hat{T}')$,
4) $\hat{T}'() = \hat{v}'$.
Show $\hat{v} \simeq \hat{v}'$.

Have 5) $(\hat{H}, \hat{T}) \simeq (\hat{H}', \hat{T}')$ from IH, $0, 1, 3$.

Result follows from $5, 2, 4$.

- **case** $f_{lib}\ e$: Based on the rule `lib`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \Delta[f] = (\boldsymbol{x}, e_f) \\ & \wedge & \Lambda[f] = (\boldsymbol{\varphi}, \gamma) \wedge \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \wedge \hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1) \\ & \wedge & [\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e_f \rangle \rightsquigarrow \langle \xi_2, v \rangle \wedge v \uparrow_{\xi_2} \gamma = \hat{v} \\ & \wedge & \hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}' \wedge \hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1') \\ & \wedge & [\boldsymbol{x} \mapsto \boldsymbol{v}'] \models \langle \xi_1', e_f \rangle \rightsquigarrow \langle \xi_2', v' \rangle \wedge v' \uparrow_{\xi_2'} \gamma = \hat{v}' \\ & \Rightarrow & \hat{v} \simeq \hat{v}' \end{array}$$

Have $0)$ $\hat{\delta} \simeq \hat{\delta}'$,

$1)$ $\Delta[f] = (\boldsymbol{x}, e_f)$,

$2)$ $\Lambda[f] = (\boldsymbol{\varphi}, \gamma)$,

$3)$ $\hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}}$,

$4)$ $\hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi_1)$,

$5)$ $[\boldsymbol{x} \mapsto \boldsymbol{v}] \models \langle \xi_1, e_f \rangle \rightsquigarrow \langle \xi_2, v \rangle$,

$6)$ $v \uparrow_{\xi_2} \gamma = \hat{v}$,

$7)$ $\hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}'$,

$8)$ $\hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi_1')$,

$9)$ $[\boldsymbol{x} \mapsto \boldsymbol{v}'] \models \langle \xi_1', e_f \rangle \rightsquigarrow \langle \xi_2', v' \rangle$,

$10)$ $v' \uparrow_{\xi_2'} \gamma = \hat{v}'$.

Show $\hat{v} \simeq \hat{v}'$.

Have $11)$ $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$ from IH consecutively, $0, 3, 7$.

Have $12)$ $\xi_1 = \xi_1'$ from $11, 4, 8$.

Result follows from Definition 2 together with $1, 2, 11, 4, 8, 12, 5, 9$

□

C Soundness for $\mathcal{F}$

Theorem 3 (Noninterference for labeled execution)**.**

$$\hat{\delta} \simeq \hat{\delta}' \wedge \hat{\delta} \models e \to \hat{v} \wedge \hat{\delta}' \models e \to \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

The $\simeq$ relation is defined to be

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{n_1^H \simeq n_2^H} \qquad \frac{dom(\hat{\delta}) = dom(\hat{\delta}') \quad \forall x \in dom(\hat{\delta}) \,.\, \hat{\delta}[x] \simeq \hat{\delta}'[x]}{\hat{\delta} \simeq \hat{\delta}'}$$

$$\frac{}{v_1^H \simeq v_2^H} \qquad \frac{\forall \hat{\boldsymbol{v}}, \hat{\boldsymbol{v}}' \,.\, \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \Rightarrow \hat{F}(\hat{\boldsymbol{v}}) \simeq \hat{F}'(\hat{\boldsymbol{v}}')}{\hat{F}^L \simeq \hat{F}'^L}$$

$$\frac{\hat{\delta} \simeq \hat{\delta}'}{\text{lclos}(\hat{\delta}, \boldsymbol{x}, e)^L \simeq \text{lclos}(\hat{\delta}', \boldsymbol{x}, e)^L}$$

Proof. By induction on the height, h, of the derivation tree $\hat{\delta} \models e \to \hat{v}$ taking the form of a case analysis on the last rule applied. Since the rules `int`, `var`, `op`, $\texttt{if}_1$, $\texttt{if}_2$, `let` are analogous to the proof for $\mathcal{C}$, we refrain from repeating them. To show soundness of $\mathcal{F}$, it is enough to show soundness of the added and modified rules, which are `fun`, `app` and `lib`, which are found in Figure 1.5.

- **case** *fun* $\boldsymbol{x} \Rightarrow e$: Based on the rule `fun`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & v = \text{lclos}(\hat{\delta}, \boldsymbol{x}, e) \wedge v' = \text{lclos}(\hat{\delta}', \boldsymbol{x}, e) \\ & \Rightarrow & v^L \simeq v'^L \end{array}$$

  Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

  1) $v = \text{lclos}(\hat{\delta}, \boldsymbol{x}, e)$,

  2) $v' = \text{lclos}(\hat{\delta}', \boldsymbol{x}, e)$.

  Show $v^L \simeq v'^L$.

  Result follows immediately from definition of $\text{lclos}(\hat{\delta}, \boldsymbol{x}, e)^L \simeq \text{lclos}(\hat{\delta}', \boldsymbol{x}, e)^L, 1, 2$.

- **case** $e\ \boldsymbol{e}$: Based on the rule `app`, we have to show:

$$\begin{array}{rcl} \hat{\delta} \simeq \hat{\delta}' & \wedge & \hat{\delta} \models e \to \hat{F}^\ell \wedge \hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}} \wedge \hat{F}(\hat{\boldsymbol{v}}) = \hat{v} \\ & \wedge & \hat{\delta}' \models e \to \hat{F}'^{\ell'} \wedge \hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}' \wedge \hat{F}'(\hat{\boldsymbol{v}}') = \hat{v}' \\ & \Rightarrow & \hat{v}^\ell \simeq \hat{v}'^{\ell'} \end{array}$$

  Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\hat{\delta} \models e \to \hat{F}^{\ell}$,

2) $\hat{\delta} \models \boldsymbol{e} \to \hat{\boldsymbol{v}}$,

3) $\hat{F}(\hat{\boldsymbol{v}}) = \hat{v}$,

4) $\hat{\delta}' \models e \to \hat{F}'^{\ell'}$,

5) $\hat{\delta}' \models \boldsymbol{e} \to \hat{\boldsymbol{v}}'$,

6) $\hat{F}'(\hat{\boldsymbol{v}}') = \hat{v}'$.

Show $\hat{v}^{\ell} \simeq \hat{v}'^{\ell'}$.

Have 7) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$ from IH consecutively, $0, 2, 5$.

Have 8) $\hat{F}^{\ell} \simeq \hat{F}'^{\ell'}$ from IH, $0, 1, 4$.

We get two cases based on 8.

case $\hat{F} = \mathbf{lclos}(\hat{\delta}, \boldsymbol{x}, e)$ **and** $\hat{F}' = \mathbf{lclos}(\hat{\delta}', \boldsymbol{x}, e)$

Have 9) $\hat{\delta}[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \models e \to \hat{v}$ by expansion of lclose in 3.

Have 10) $\hat{\delta}'[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}'] \models e \to \hat{v}'$ by expansion of lclose in 6. The result follows from IH, $7, 9, 10$ and 1.

otherwise

Have 9) $\ell = \ell'$ from 8.

There are two cases; $\ell = \ell' = L$ and $\ell = \ell' = H$.

case $\ell = \ell' = H$ The result follows immediately from the definition of $v_1^H \simeq v_2^H$.

case $\ell = \ell' = L$ The result follows immediately from the definition of $\hat{F}^L \simeq \hat{F}'^L, 7, 3, 6$.

- **case** f_{lib}: Based on the rule `lib`, we have to show:

$$\begin{array}{lll} \hat{\delta} \simeq \hat{\delta}' \wedge \delta_0[f] = F & \wedge & \xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} \\ & \wedge & F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} = \hat{F}^{\ell} \\ & \wedge & F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} = \hat{F}'^{\ell'} \\ & \Rightarrow & \hat{F}^{\ell} \simeq \hat{F}'^{\ell'} \end{array}$$

Have 0) $\hat{\delta} \simeq \hat{\delta}'$,

1) $\delta_0[f] = F$,

2) $\xi_0[f] = (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa}$,

3) $F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} = \hat{F}^{\ell}$,

4) $F \uparrow_{\xi_0} (\boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{\kappa} = \hat{F}'^{\ell'}$.

Have 5) $\hat{F}^{\ell} = \text{label}(F, \xi_0, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_{\xi_0}}$ by expansion of $\uparrow$ in 3.

Have 6) $\hat{F}'^{\ell'} = \text{label}(F, \xi_0, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_{\xi_0}}$ by expansion of $\uparrow$ in 4.

Have $\hat{F}^{\ell} = \text{label}(F, \xi_0, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_{\xi_0}} = \hat{F}'^{\ell'}$ from 5, 6.

Hence, $\ell = \ell' = [\![\kappa]\!]_{\xi_0}$.

If $\ell = \ell' = H$, the result $\hat{F}^{H} \simeq \hat{F}'^{H}$ follows from definition of $\simeq$.

Have $\ell = \ell' = L$.

Show 7) $\hat{F}^{L} \simeq \hat{F}'^{L}$, i.e., $\forall \hat{\boldsymbol{v}}, \hat{\boldsymbol{v}}' \,.\, \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \Rightarrow \hat{F}(\hat{\boldsymbol{v}}) \simeq \hat{F}'(\hat{\boldsymbol{v}}')$

Assume 8) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$,

Show 9) $\hat{F}(\hat{\boldsymbol{v}}) \simeq \hat{F}'(\hat{\boldsymbol{v}}')$.

Have 10) $v \uparrow_{\xi_3} \gamma = \hat{F}(\hat{\boldsymbol{v}})$, where $\hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi')$, $[\![\boldsymbol{\zeta}]\!]_{\xi_0 \sqcup \xi'} = \xi_2$, $F(\xi_2, \boldsymbol{v}) = (\xi_3, v)$ from expansion of label in 5.

Have 11) $v' \uparrow_{\xi'_3} \gamma = \hat{F}'(\hat{\boldsymbol{v}}')$, where $\hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi')$, $[\![\boldsymbol{\zeta}]\!]_{\xi_0 \sqcup \xi'} = \xi_2$, $F(\xi_2, \boldsymbol{v}') = (\xi'_3, v)$ from expansion of label in 6.

The result $v \uparrow_{\xi_3} \gamma = \hat{F}(\hat{\boldsymbol{v}}) \simeq v' \uparrow_{\xi'_3} \gamma = \hat{F}'(\hat{\boldsymbol{v}}')$ follows from Definition 3 together with 10 and 11.

□

D Supporting lemmas

Lemma 1 (Soundness of environment updates)**.**

$$\hat{\delta}_1 \simeq \hat{\delta}'_1 \wedge \hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}' \wedge \hat{\delta}_2 = \hat{\delta}_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}] \wedge \hat{\delta}'_2 = \hat{\delta}'_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}'] \Rightarrow \hat{\delta}_2 \simeq \hat{\delta}'_2$$

Proof. By structural induction on $\boldsymbol{x}$. Since $\hat{\delta}_2 = \hat{\delta}_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}]$ and $\hat{\delta}'_2 = \hat{\delta}'_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}']$ we know that $\boldsymbol{x}$, $\hat{\boldsymbol{v}}$ and $\hat{\boldsymbol{v}}'$ share structure.

case 0) $\boldsymbol{x} = [\,]$

Have 1) $\hat{\delta}_1 \simeq \hat{\delta}'_1$,

2) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$,

3) $\hat{\delta}_2 = \hat{\delta}_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}]$,

4) $\hat{\delta}'_2 = \hat{\delta}'_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}']$.

Show $\hat{\delta}_2 \simeq \hat{\delta}'_2$.

Have 5) $\hat{\boldsymbol{v}} = \epsilon$ from 0.

Have 6) $\hat{\boldsymbol{v}}' = \epsilon$ from 0.

Have 7) $\hat{\delta}_2 = \hat{\delta}_1$ from 3, 5.

Have 8) $\hat{\delta}'_2 = \hat{\delta}'_1$ from 4, 6.

Result follows from 1, 7, 8.

case 1) $\boldsymbol{x} = x \cdot \boldsymbol{x}_r$

Have 1) $\hat{\delta}_1 \simeq \hat{\delta}'_1$,

2) $\hat{\boldsymbol{v}} \simeq \hat{\boldsymbol{v}}'$,

3) $\hat{\delta}_2 = \hat{\delta}_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}]$,

4) $\hat{\delta}'_2 = \hat{\delta}'_1[\boldsymbol{x} \mapsto \hat{\boldsymbol{v}}']$.

Show $\hat{\delta}_2 \simeq \hat{\delta}'_2$.

Have 5) $\hat{\boldsymbol{v}} = \hat{v} \cdot \hat{\boldsymbol{v}}_r$ from 0.

Have 6) $\hat{\boldsymbol{v}}' = \hat{v}' \cdot \hat{\boldsymbol{v}}'_r$ from 0.

Have 7) $\hat{v} \simeq \hat{v}'$ from $2, 5, 6$.

Have 8) $\hat{\boldsymbol{v}}_r \simeq \hat{\boldsymbol{v}}'_r$ from $2, 5, 6$.

Have 9) $\hat{\delta}_2 = \hat{\delta}_x[\boldsymbol{x}_r \mapsto \hat{\boldsymbol{v}}_r]$ and $\hat{\delta}_x = \hat{\delta}_1[x \mapsto \hat{v}]$ from definition of $\hat{\delta}$ updates

Have 10) $\hat{\delta}'_2 = \hat{\delta}'_x[\boldsymbol{x}_r \mapsto \hat{\boldsymbol{v}}'_r]$ and $\hat{\delta}'_x = \hat{\delta}'_1[x \mapsto \hat{v}']$ from definition of $\hat{\delta}$ updates

Have 11) $\hat{\delta}'_x \simeq \hat{\delta}'_x$ from $1, 7$

Result follows from IH, $11, 8$.

□

Paper II

Information Flow Tracking for Side-effectful Libraries

Alexander Sjösten, Daniel Hedin, Andrei Sabelfeld

International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE), Madrid, Spain, June 2018

Abstract

Dynamic information flow control is a promising technique for ensuring confidentiality and integrity of applications that manipulate sensitive information. While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to libraries and APIs. The state of the art is largely an all-or-nothing choice: either a *shallow* or *deep* library modeling approach. Seeking to break out of this restrictive choice, we formalize a general mechanism that tracks information flow for a language that includes higher-order functions, structured data types and references. A key feature of our approach is the *model heap*, a part of the memory, where security information is kept to enable the interaction between the labeled program and the unlabeled library. We provide a proof-of-concept implementation and report on experiments with a file system library. The system has been proved correct using Coq.

1 Introduction

While useful, access control is not enough: it is crucial what applications do with the data after access has been granted [26]. Information flow control tracks the propagation of data in programs, thus enforcing confidentiality and integrity policies. Due to the widespread use of highly dynamic languages, such as JavaScript, there has been a growing interest in *dynamic information flow control*. There are two basic kinds of flows to consider: *explicit* and *implicit* [5], related to the notions of *data flow* and *control flow*. Dynamic information flow is tracked at runtime by extending the data with *security labels*, which are propagated and checked against a *security policy* during execution. The detection of potential security violations cause program execution to halt.

While much progress has been made on increasingly powerful programming languages ranging from low-level machine languages to high-level languages for distributed systems, surprisingly little attention has been devoted to *libraries* and *APIs*[1]. The main challenge is when the library is not written in the language itself, and thus not compatible with the labeled semantics of the program. There are mainly two situations where this occurs: 1) when the library is part of the standard execution environment, and 2) when the library is brought into the language using some form of *foreign function interface* (FFI). In such cases, values passing between the program

[1]For elegance of expression, when we write library in this paper we refer to both libraries and APIs.

and the library must be translated. The process of translating values from one programming language to another is known as *marshaling*.

Marshaling of labeled values additionally entails that security labels must be removed from the values being passed from the program to the library, and reattached on the values returned from the library to the program. We refer to those steps as *unlabeling* and *relabeling* of the values, and the description of how it should be done as a *library model*. The main difference between standard marshaling and marshaling of labeled values is the latter removes information from the values passed to the library. To be able to correctly relabel values going from the library to the program, the labels removed during the unlabeling process must be used, since the returned value contains no security information. This means that the library models are inherently stateful — the removed labels are stored in a *model state* used when relabeling.

Library models can be split into two categories: *deep* and *shallow* models [14]. Deep models track information flow inside the library, requiring precise modeling of the execution of the library, while shallow models are limited to the security labels on the boundary of the library. Often, deep models necessitate reimplementation of parts of the library functionality within the model, making them difficult to create and maintain. Shallow models, on the other hand, are significantly more lightweight, but possibly too imprecise. In this work, we are interested in the boundary between deep and shallow models.

Current state of the art in dynamic information-flow tracking does not fit this classification entirely, in part due to ad-hoc handling of libraries. To the extent addition of new libraries is supported, the models used tend towards shallow models. This is true for, e.g., FlowFox [13], and experimental extensions of JSFlow [15]. On the other hand, JSFlow and FlowFox both use deep models to provide fine grained information-flow tracking for built in libraries. JSFlow, e.g., implements the full ECMA-262 version 5 standard using what is best considered a deep approach.

In recent work, Hedin et al. [18] initiate a framework for tracking information flow in libraries. The setting is a labeled *program* and an unlabeled *library* that share the same core semantics (*split semantics*) in order to limit the marshaling to security labels only. Their work targets a focused functional language with higher-order functions (which allows for both callbacks and promises to exist), and structured data in terms of lists. It does not, however, handle side effects, which means that many libraries cannot be modeled in a satisfactory way. As an example, it is unavoidable for a standard file system library to maintain state to keep track of open files, stream positions and buffers. The success of a function `read(path, success, fail)` is dependent

both on the file path and the state of the library which must be reflected by security models for the library.

The combination of state and higher-order functions significantly complicates the library models and the model state over the ones used by Hedin et al. If the state is first-class (i.e., it can be sent around as values, as in languages with mutable references, records or objects) the situation is further complicated. *This is the setting we are interested in handling, as it captures the essence of many of the problems found when modeling real libraries.*

To this end we introduce a model heap, allowing library values to be tied to a mutable model state, which allows for secure modeling of the interaction between first-class state and higher-order functions.

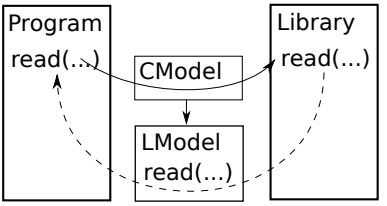


Figure 2.1: Model heap illustration

Consider the file system example, depicted in Figure 2.1. When the program calls the library function `read`, the library function is first lifted into the program using the corresponding function model defined by the library model, LModel. The lifting (illustrated by the dotted arrow in the figure) is done by means of wrapping and results in an unlabeled function that can be called by the program. When the wrapper is called with labeled arguments, a new call model state, CModel, is created and used to hold the labels of the arguments, since the underlying library function requires unlabeled values. As can be seen in the figure, the call model state is connected to the library model state and together they define the model state that the function model of `read` interacts with. Any other values, including higher-order functions and first-class state, defined in the library share the same library model state, which guarantees that they have the same view of the library state, even in the presence of mutability.

There are two main benefits of our work over ad-hoc modeling of libraries. First, it lowers the modeling effort significantly, and, second, given that the models properly describe the library, it guarantees noninterference. Both benefits stem from expressing the models in a simplified model language that controls the marshaling process, thus sidestepping the need to reimplement it repeatedly.

Considering the dimension of shallow and deep models, our work can be seen as exploring the boundary. Shallow models are expressed solely in terms of the boundary labels, while our work gains access to intermediate labels when models for lazy marshaling, higher-order functions and first-class state are triggered. In addition, it is relatively easy to extend our system

to allow models to use the runtime values allowing for *dependent models* [18]. Compared to fully deep models, our work is limited to the information passing between the program and the library at the point of passing. Thus, intermediate values and labels that do not participate in cross-boundary activity is without reach. While deep models in theory have access to more information and therefore have the potential to be more precise, it is unclear if the added precision is significant in practice, in particular in the light of the added implementation cost.

Contributions The main contributions of this paper are:

- We have created a language containing three cornerstones of library modeling: higher-order functions, first-class state, and structured values (the syntax and semantics are presented in Section 2 and Section 3, respectively, while Section 6 discusses correctness).
- We have implemented a prototype and used it to explore the interaction between the different features of the language (examples that illustrate our mechanism are reported in Section 4).
- We have conducted a case study on a file system library, inspired by the file system library in node.js [10], showing that our language is able to handle stateful libraries (the case study is reported in Section 5).
- We have formalized the language and its correctness proof in Coq [20].

The scope of the prototype is to experimentally verify applicability of models, not to assess performance in a full-scale implementation. The prototype serves as a complement to the formal proof to create a system that is both correct and useful. The full version of the paper, along with the formalization in Coq and the proof-of-concept prototype can be found at [28].

2 Syntax

The language we present is a small functional language with split semantics and lazy marshaling. The syntax of the language is defined as follows, where n denotes numbers and x denotes identifiers.

$$\begin{aligned} e \ ::= \ & n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{fun}\ x = e \mid e_1\ e_2 \mid \\ & x_{\mathit{lib}} \mid e_1 \oplus e_2 \mid \ominus e \mid \mathit{head}\ e \mid \mathit{tail}\ e \mid e_1 : e_2 \mid [\,] \mid (e_1, e_2) \mid (\,) \mid \\ & \mathit{ref}\ e \mid !e \mid \{\ \underline{x} : \underline{e}\ \} \mid e.x \mid e_1 := e_2 \mid e_1\ ;\ e_2 \mid \mathit{upg}\ e\ \ell \end{aligned}$$

The syntax of the language is entirely standard apart from the x_{lib} construction that lifts a *library value* to a *program value*, and $\mathit{upg}\ e\ \ell$ that gives the

result of the expression a given label, $\ell ::= L \mid H$. For simplicity, we identify sets with the meta variables ranging over them. Let $\underline{X}$ range over lists of X for any set X, where [ ] denotes the empty list and $\cdot$ denotes the cons operator. An application in the language is a triple $(\underline{d}_p, \underline{d}_l, \underline{m})$, where the first component is the labeled *program*, the second component is the unlabeled *library* and the third component is the *library model*. Throughout the rest of this paper, we use *program* when referring to the labeled part, and *library* when referring to the unlabeled part.

The top-level definitions, d, allow for named definitions of functions and values $d ::= \mathit{fun}\ f(x) = e \mid \mathit{let}\ x = e$. The top-level model definitions, m, allow for named definitions of models and labels $m ::= \mathit{mod}\ x :: \gamma \mid \mathit{lbl}\ x :: \kappa$, where γ denotes *relabel models* and κ denotes *label terms*. The label terms, $\kappa ::= \ell \mid \alpha \mid \kappa_1 \sqcup \kappa_2$ are terms that evaluate to labels in a given model state and consist of *labels*, ℓ, *label variables*, α, and the least upper bound of two label terms. The relabel models, γ, used to relabel library values, are defined as follows

$$\gamma ::= \ \kappa \mid (\gamma_1, \gamma_2)^\kappa \mid [\gamma]^\kappa \mid (\varphi \to \gamma, \underline{\zeta})^\kappa \mid \mathrm{ref}(\varphi, \gamma)^\kappa$$

where φ denotes *unlabel models*, used to unlabel program values, and ζ denotes *effect constraints* defined below. All values are given a label by a label term, and the relabeling of structured values follows the structure of the value. To relabel a function, we must know how to unlabel the argument, how to relabel the result, and how the function interacts with the model state. To relabel a reference we must know how to unlabel the values written and how to relabel the values read. The unlabel models, φ, are defined as follows.

$$\varphi ::= \alpha \mid \#\alpha^\alpha \mid (\varphi_1, \varphi_2)^\alpha \mid [\varphi]^\alpha$$

Unlabeling of values is performed by storing the label of the value in the corresponding label variable in the model state. As for relabeling, unlabeling of structured values follows the structure of the value. Unlabeling of functions and references introduces an *abstract name*, $\#\alpha$, used by library functions to tie any interaction to their model state in the effect constraints, ζ.

$$\zeta ::= \ !\#\alpha \to \varphi \mid \kappa \vdash \#\alpha \leftarrow \gamma \mid \kappa \vdash \#\alpha\ \gamma \to \varphi \mid \kappa \vdash \alpha \leftarrow \kappa$$

In the order of definition: a library function that reads a labeled reference defines how to unlabel the read value, a library function that writes to a labeled reference defines the security context in which the write occurs and how to relabel the value to be written, a library function that calls a labeled function defines the security context in which the call occurs, how to relabel the parameter and how to unlabel the result, and finally, a library function that modifies the library state defines the security context of the update and how the security model changes.

3 Semantics

We define the semantics step-wise in three parts. The first part defines the labeled values, and the execution environment. The second part defines the evaluation relation and how the function representations of the values are created and used in the semantics. Finally, the third part defines how values are marshaled between the program and the library. For space reasons, parts of the semantic definitions have been left out. We refer the reader to the appendix for the missing definitions.

3.1 Values

In order to differentiate between the labeled semantics and the unlabeled semantics, we use $\hat{X}$ to denote an entity in the labeled semantics corresponding to the entity X in the unlabeled semantics. We only give the labeled values. The unlabeled values are defined analogously. The values in the language, $\hat{v}$, are integers n, tuples, higher-order functions $\hat{F}$, lists $(\hat{H}, \hat{T})$, references $(\hat{R}, \hat{W})$, and records $\hat{O}$, where higher-order functions, lists, references and records are represented as (pairs of) functions in order to simplify the marshaling.

$$\hat{v} \ ::= n^{\ell} \mid (\hat{v}_1, \hat{v}_2)^{\ell} \mid ()^{\ell} \mid \hat{F}^{\ell} \mid (\hat{H}, \hat{T})^{\ell} \mid [\,]^{\ell} \mid (\hat{R}, \hat{W})^{\ell} \mid \hat{O}^{\ell}$$

The labels, ℓ, form a two-point upper semi-lattice $L \sqsubseteq H$, where L denotes *low* (public) and H denotes *high* (private). Let $\ell_1 \sqcup \ell_2$ denote the least upper bound of ℓ_1 and ℓ_2, and let $\hat{v}^{\ell_2} = v^{\ell_1 \sqcup \ell_2}$ for $\hat{v} = v^{\ell_1}$.

The execution environment is a triple $(\varsigma, \Gamma, \Sigma)$ of the security context, ς, the stack, and the heap. The security context ς ranges over labels ℓ. The stack Γ is a triple of stacks $(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}})$, containing pointers to the labeled frames, the unlabeled frames and the model frames, respectively. The heap Σ is a triple of heaps, $(\hat{\sigma}, \sigma, \ddot{\sigma})$, consisting of the labeled heap, the unlabeled heap and the model heap. The labeled and unlabeled heaps can contain values (for implementing references), and frames, whereas the model heap only contains frames. The labeled and unlabeled frames, $\hat{\omega}$ and ω, are maps from identifiers to values, and the model frames, $\ddot{\omega}$ are maps from identifiers to *model items*. Each frame represents a scope, and together with the corresponding stacks they form scope chains. The model items, $\ddot{\imath} ::= \ell \mid \gamma \mid \zeta$, consists of labels, relabel models and effect constraints.

3.2 Evaluation relations

The evaluation relation for program execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, \hat{v})$, read "expression e evaluates in the environment consisting of the

$$\texttt{int}\ \frac{}{\varsigma, \Gamma \models (\Sigma, n) \to (\Sigma, n^L)} \qquad \texttt{var}\ \frac{\text{lookupL}(\Gamma, \Sigma, x) = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x) \to (\Sigma, \hat{v})}$$

$$\texttt{if-true}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 \neq 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_2^\ell)}$$

$$\texttt{if-false}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 = 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_3) \to (\Sigma_3, \hat{v}_3)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_3^\ell)}$$

$$\texttt{fun}\ \frac{}{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma, \mathit{fun}\ x = e) \to (\Sigma, \text{lclos}(\hat{\underline{\rho}}, x, e)^L)}$$

$$\texttt{app}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{F}^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_1) \qquad \hat{F}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}_1) = (\Sigma_4, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ e_2) \to (\Sigma_4, \hat{v}_2^\ell)}$$

$$\texttt{ref}\ \frac{\varsigma, \Gamma \models (\Sigma, e) \to ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}) \qquad \hat{\rho}\ \text{fresh}}{\varsigma, \Gamma \models (\Sigma, \mathit{ref}\ e) \to ((\hat{\sigma}[\hat{\rho} \mapsto \hat{v}], \sigma, \ddot{\sigma}), (\text{lread}(\hat{\rho}), \text{lwrite}(\hat{\rho}))^L)}$$

$$\texttt{deref}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, !e) \to (\Sigma_3, \hat{v}^\ell)}$$

$$\texttt{assign}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}) \qquad \hat{W}(\varsigma \sqcup \ell, \Gamma, \Sigma_3 \hat{v}) = \Sigma_4}{\varsigma, \Gamma \models (\Sigma_1, e_1 := e_2) \to (\Sigma_4, \hat{v})}$$

$$\texttt{lib}\ \frac{\text{lookupU}(\Gamma, \Sigma, x) = v \qquad \text{lookupM}(\Gamma, \Sigma, x) = \gamma \qquad v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x_{lib}) \to (\Sigma, \hat{v})}$$

Figure 2.2: Selected labeled semantics

security context, ς, the stack, Γ, and the heap, Σ_1, resulting in the updated heap Σ_2 and value $\hat{v}$". Similarly, library execution is of the form $\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, v)$, where the unlabeled semantics is parameterized over the security context to model that the context is global and always available to the marshaling functions[2].

Figure 2.2 contains a selection of the semantic rules of the program semantics related to the marshaling of values. See Appendix B for a full set of rules.

[2]In an operational semantics global non-constant values must be passed around during execution, similar to in a pure functional language.

The rules of the core language are standard. Whenever an integer is created (`int`), it is always originally labeled L. Variables are retrieved from the labeled heap using lookupL in `var`. If-statements (`if-true` and `if-false`) evaluate the conditional expression and based on the result select which branch to take. The branch taken is evaluated in a security context of $\varsigma \sqcup \ell$ and the returned value is raised to ℓ, where ℓ is the label of the result of the conditional expression.

Function closures are represented as functions, $\hat{F} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \to (\Sigma_2, \hat{v})$, created by lclos (`fun`) in the following way.

$$\begin{aligned}&\text{lclos}(\underline{\hat{\rho}}', x, e) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v}_1)\ .\ (\Sigma, \hat{v}_2)\\ &\quad \text{where } \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \{x \mapsto \hat{v}_1^{\varsigma}\}], \hat{\rho} \text{ fresh}\\ &\quad \text{and } \varsigma, (\hat{\rho} \cdot \underline{\hat{\rho}}', \underline{\rho}, \underline{\ddot{\rho}}) \models ((\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1), e) \to (\Sigma, \hat{v}_2)\end{aligned}$$

The function closure will, when interacted with, create a new pointer to a labeled frame containing the mapping of the parameter name x and the actual value $\hat{v}_1$, which is raised to the current security context. The function expression e is then evaluated, using the newly created pointer along with the updated heap. When applying a function closure (`app`), the body of the function is executed in the program semantics, under the elevated context consisting of the current security context raised to the label of the function closure. Creation and application of library closures, $F : (\varsigma, \Gamma, \Sigma_1, v) \to (\Sigma_2, v)$, is analogous.

Safe implementation of marshaling of references requires the ability to trap and modify reads and writes in order to marshal the values passed by the interaction. For this reason, references are represented as pairs of functions, one function for reading the reference, $\hat{R} : (\varsigma, \Gamma, \Sigma_1) \to (\Sigma_2, \hat{v})$, and one function for updating the reference, $\hat{W} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \to \Sigma_2$. This allows us to marshal references by wrapping the read and the write functions in functions that perform the marshaling of the values at the time of interaction, similar to lazy marshaling of lists [18]. Most languages do not support the creation of functions that are triggered on interaction with values such as references or objects, which means they cannot support marshaling of first-class mutable state. A notable exception to this is JavaScript that allows methods to be tied to different aspects of object interaction via the use of Proxy objects [23].

Creation of references given a fresh pointer into the labeled heap is defined by lread and lwrite as follows.

$$\begin{aligned}&\text{lread}(\hat{\rho}) = \lambda(\varsigma, \Gamma, (\hat{\sigma}, \sigma, \ddot{\sigma}))\ .\ ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}), \text{ where } \hat{v} = \hat{\sigma}[\hat{\rho}]\\ &\text{lwrite}(\hat{\rho}) = \lambda(\varsigma, \Gamma, (\hat{\sigma}_1, \sigma_1, \ddot{\sigma}_1), \hat{v})\ .\ (\hat{\sigma}_2, \sigma_1, \ddot{\sigma}_1)\\ &\quad \text{where } v^{\ell} = \hat{\sigma}_1[\hat{\rho}], \varsigma \sqsubseteq \ell, \hat{\sigma}_2 = \hat{\sigma}_1[\hat{\rho} \mapsto \hat{v}^{\varsigma}]\end{aligned}$$

References (`ref`) are created by selecting a fresh heap location made to point to the value of the reference. The heap location is then used to create a pair of access functions. The created reference follows the same intuition as for all created values. All values are labeled L upon creation, which is why the pair of access functions are labeled L in `ref`. Note that the value that the reference is referring to may be labeled differently, due to the distinction between reference as a value and the value the reference is referring to. Dereferencing (`deref`) uses the read function of the reference to get the value to be read, while assignment (`assign`) uses the write function. Creation and use of library references, $R : (\varsigma, \Gamma, \Sigma_1) \to (\Sigma_2, v)$ and $W : (\varsigma, \Gamma, \Sigma_1, v) \to \Sigma_2$ is analogous.

It is worthwhile to point out the *no-sensitive upgrade* (NSU) check in lwrite, which demands that the context, which the label of the reference is a part of, is lower or equal to the label of the referenced value, $\varsigma \sqsubseteq \ell$. Allowing labels of values to change freely leads to an unsound system, due to the possibility of implicit flows into the labels themselves [1, 29].

Disregarding the encoding of functions and references into functions, up to this point, the labeled and unlabeled semantics are equivalent to their standard formulations. The essence of this paper is in the marshaling of values between the program and the library, performed by the unlabeling and relabeling functions, defined in the following section.

3.3 Marshaling

All interaction between the program and the library is initiated by lifting named library values into the program. This is done (`lib`) by looking up the value, and the corresponding relabel model used to relabel the value. Interaction with the relabeled value may cause further marshaling. Unlabeling of a value is done w.r.t. an unlabel model, φ, which defines how to store the removed label(s) in the model state. Relabeling of a value is done w.r.t. a relabel model, γ, which defines how to compute the label in terms of the model state. Formally, unlabeling is a function of the form $\hat{v} \downarrow_{\varsigma,\Gamma,\Sigma_1} \varphi = (\Sigma_2, v)$ taking a labeled value $\hat{v}$, an environment, $\varsigma, \Gamma, \Sigma_1$ and an unlabel model φ and returning an updated heap, Σ_2, and an unlabeled value v. Similarly, relabeling is a function of the form $v \uparrow_{\Gamma,\Sigma} \gamma = \hat{v}$, taking an unlabeled value, v, an environment, Γ, Σ, and a relabel model, γ, and returning a labeled value $\hat{v}$. The only modified part of the heap for both unlabeling and relabeling is the model heap.

There are six types of values: integers, tuples, lists, records, higher-order functions and references. In the rest of this section we describe how to evaluate label terms (used when relabeling) and how to marshal higher-

order functions and references. We refer the reader to the appendix for the treatment of the other constructs.

Label terms Evaluation of label terms is done w.r.t. a model state, where lookupM is used to traverse the model scope chain to find the first label corresponding to a given label variable.

$$[\![\alpha]\!]_{\Gamma,\Sigma} = \begin{cases} \ell, & \text{if lookupM}(\Gamma,\Sigma,\alpha) = \ell \\ L, & \text{otherwise} \end{cases}$$
$$[\![\ell]\!]_{\Gamma,\Sigma} = \ell$$
$$[\![\kappa_1 \sqcup \kappa_2]\!]_{\Gamma,\Sigma} = [\![\kappa_1]\!]_{\Gamma,\Sigma} \sqcup [\![\kappa_2]\!]_{\Gamma,\Sigma}$$

Higher-Order Functions Marshaling of higher-order functions involves both marshaling the functions as values as well as ensuring the parameter and return value are properly marshaled.

Unlabeling. Unlabeling a program closure removes and stores the label and returns a library closure created by wrapping the program closure. The library closure is tied to the abstract name, π, used by the wrapper to relabel the parameters before the call and unlabel the result after the call.

$$\hat{F}^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} \#\pi^{\alpha} = (\text{updateM}(\varsigma,\Gamma,\Sigma,\alpha,\ell), \text{u-lclos}(\hat{F},\ell,\#\pi))$$

The translation of a program closure, $\hat{F}$, into an library closure is performed by u-lclos, that takes the program closure, the label of the program closure and the abstract name.

$$\begin{aligned} \text{u-lclos}(\hat{F},\ell_1,\#\pi) &= \lambda(\varsigma,\Gamma,\Sigma_1,v_1)\,.\,(\Sigma_3,v_2) \\ \text{where}\quad & \kappa \vdash \gamma \rightarrow \varphi = \text{lookupM}(\Gamma,\Sigma_1,\pi) \\ & \ell_2 = [\![\kappa]\!]_{\Gamma,\Sigma_1} \\ & \hat{v}_1 = v_1 \uparrow_{\Gamma,\Sigma_1} \gamma \\ & (\Sigma_2,\hat{v}_2) = \hat{F}(\varsigma \sqcup \ell_1 \sqcup \ell_2,\Gamma,\Sigma_1,\hat{v}_1) \\ & (\Sigma_3,v_2) = \hat{v}_2 \downarrow_{\varsigma\sqcup\ell_1\sqcup\ell_2,\Gamma,\Sigma_2} \varphi \end{aligned}$$

When the library closure returned by u-lclos is applied the following occurs. First, the function call model bound to the abstract name is fetched using lookupM. The function call model contains a label term representing the security context of the application, how to relabel the parameter and how to unlabel the return value. Second, the relabel model, γ, is used to relabel the parameter, v_1. Third, the program closure is called in the security context of the call raised to the label of the closure and the evaluation of the context label term, κ. The result of the call is a labeled value, $\hat{v}_2$. Finally, $\hat{v}_2$ is unlabeled which gives the result, v_2, of the application of the unlabeled closure. Notice that all relabeling and unlabeling is done with respect to the model state of the caller.

Relabeling. Relabeling a library closure is done by labeling the program closure created by wrapping the library closure. The wrapper unlabels the arguments before the call and relabels the result of the call.

$$F \uparrow_{\Sigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}})} (\varphi \to \gamma, \underline{\zeta})^{\kappa} = \text{l-uclos}(F, \underline{\ddot{\rho}}, (\varphi \to \gamma, \underline{\zeta}))^{[\![\kappa]\!]_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma}}$$

The process is controlled by the function relabel model, $(\varphi \to \gamma, \underline{\zeta})^{\kappa}$, where the evaluation of κ gives the label of the wrapper closure.

The translation of the library closure, F, into a program closure is performed by l-uclos, which takes the library closure, the current model frame stack, the unlabel model for the parameters, φ, the relabel model for the return value, γ, and the effect constraints, $\underline{\zeta}$.

$$\begin{aligned}
&\text{l-uclos}(F, \underline{\ddot{\rho}}_2, (\varphi \to \gamma, \underline{\zeta})) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), (\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}_1).\ (\Sigma_4, \hat{v}_2) \\
&\quad \text{where} \quad \Sigma_1 = (\hat{\sigma}, \sigma, \ddot{\sigma}[\ddot{\rho} \mapsto \varnothing]),\ \ddot{\rho} \text{ fresh} \\
&\qquad\qquad (\Sigma_2, v_1) = \hat{v}_1 \downarrow_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_1} \varphi \\
&\qquad\qquad \Sigma_3 = \{|\underline{\zeta}|\}_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_2} \\
&\qquad\qquad (\Sigma_4, v_2) = F(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}_1), \Sigma_3, v_1) \\
&\qquad\qquad \hat{v}_2 = v_2 \uparrow_{(\underline{\hat{\rho}},\underline{\rho},\ddot{\rho}\cdot\underline{\ddot{\rho}}_2),\Sigma_4} \gamma
\end{aligned}$$

When called the program closure produces a fresh frame pointer, pointing to a new model frame in the model heap. The parameter to the library function is unlabeled based on the unlabel model, φ, and the effect constraints, $\underline{\zeta}$, are evaluated to update the model state accordingly. After that, the library function is called with the unlabeled parameter in the security context, ς, of the call. The result of the function call is relabeled with the relabel model, γ, and returned to the program. Note that all labeling and unlabeling is done w.r.t. the model frame stack of the unlabeled closure. Also note that the order is important; if the unlabeling of the parameter occurs after evaluating the effect constraints, the label of the parameter cannot be used when updating the model state with the side effects.

Effect constraints. Effect constraints define how a library function interacts with unlabeled program functions and references and how the library function changes the model state. Model state changes are effectuated on call to the library function whereas effect constraints that define interaction with unlabeled program functions and references are stored in the model state. When a library function or reference is interacted with, the abstract name will tie the interaction to the corresponding effect constraint in the model state of the interaction. The meaning of the effect constraints is defined as follows

$$
\begin{array}{rcl}
\{|!\#\alpha \to \varphi|\}_{\varsigma,\Gamma,\Sigma} & = & \text{defineM}(\Gamma, \Sigma, \alpha, \varphi) \\
\{|\kappa \vdash \#\alpha \leftarrow \gamma|\}_{\varsigma,\Gamma,\Sigma} & = & \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma) \\
\{|\kappa \vdash \#\alpha\ \gamma \to \varphi|\}_{\varsigma,\Gamma,\Sigma} & = & \text{defineM}(\Gamma, \Sigma, \alpha, \kappa \vdash \gamma \to \varphi) \\
\{|\kappa_1 \vdash \alpha \leftarrow \kappa_2|\}_{\varsigma,\Gamma,\Sigma} & = & \text{updateM}(\varsigma \sqcup [\![\kappa_1]\!]_{\Gamma,\Sigma}, \Gamma, \Sigma, \alpha, [\![\kappa_2]\!]_{\Sigma,\Gamma}) \\
 & & \quad \text{when } \varsigma \sqcup [\![\kappa_1]\!]_{\Gamma,\Sigma} \sqsubseteq \text{lookupM}(\Gamma, \Sigma, \alpha)
\end{array}
$$

where defineM binds the name α to its corresponding model value in the top model frame, if α is not defined in that model frame, updateM updates the label pointed to by α in the scope chain, or inserts it if it is not present, and lookupM returns the model value that is the first to match the name α in the scope chain.

References Marshaling of references shares some similarities with marshaling of higher-order functions. Calling a function passes the argument and the return value in opposite directions, similar to reading and writing to a reference.

Unlabeling. Unlabeling a program reference removes and stores the label, and the read and write functions are wrapped to create library counterparts.

$$
\begin{array}{l}
(\hat{R}, \hat{W})^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} \#\pi^{\alpha} = \\
\quad (\text{updateM}(\varsigma, \Gamma, \Sigma, \alpha, \ell), (\text{u-lread}(\hat{R}, \ell, \#\pi), \text{u-lwrite}(\hat{W}, \ell, \#\pi)))
\end{array}
$$

The read and the write functions are translated independently w.r.t. the abstract name $\#\pi$.

The program read function, $\hat{R}$ is translated by u-lread, which takes the read function, the label of the reference and the abstract name.

$$
\begin{array}{rl}
\text{u-lread}(\hat{R}, \ell, \#\pi) = \lambda(\varsigma, \Gamma, \Sigma_1)\ .\ (\Sigma_3, v) & \\
\text{where} & \varphi = \text{lookupM}(\Gamma, \Sigma_1, \pi) \\
 & (\Sigma_2, \hat{v}) = \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_1) \\
 & (\Sigma_3, v) = \hat{v} \downarrow_{\varsigma \sqcup \ell, \Gamma, \Sigma_2} \varphi
\end{array}
$$

When the resulting library read function is interacted with, the program read function is used to get the labeled value of the reference. This value must be unlabeled before being returned, which is done by looking up a program reference read model, φ, in the model state of the interaction. It is the model of the caller, i.e., a library function model that provides the read model for the references it reads.

The program write function, $\hat{W}$ is translated by u-lwrite, which takes the write function, the label of the reference and the abstract name.

$$\begin{aligned}\text{u-lwrite}(\hat{W}, \ell, \#\pi) &= \lambda(\varsigma, \Gamma, \Sigma_1, v)\ .\ \Sigma_2 \\ \text{where}\quad & \kappa \vdash \gamma = \text{lookupM}(\Gamma, \Sigma_1, \pi) \\ & \hat{v} = v \uparrow_{\Gamma,\Sigma_1} \gamma \\ & \Sigma_2 = \hat{W}(\varsigma \sqcup \ell \sqcup [\![\kappa]\!]_{\Gamma,\Sigma_1}, \Gamma, \Sigma, \hat{v})\end{aligned}$$

When the resulting library write function is used, the associated program reference write model, $\kappa \vdash \gamma$, is fetched in the current model state. This model defines both how to relabel the written unlabeled value, and the context in which the write occurs. Then the unlabeled value, v is relabeled before being written using the labeled write function in a context consisting of the current security context of the call raised to the reference label and the evaluation of the context label term, κ.

Relabeling. Relabeling a library reference is done by translating the read and write functions into program counterparts and relabeling the result.

$$\begin{aligned}&(R, W) \uparrow_{\Sigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}})} ref(\varphi, \gamma)^{\kappa} = \\ &\quad (\text{l-uread}(R, \underline{\ddot{\rho}}, \gamma), \text{l-uwrite}(W, \underline{\ddot{\rho}}, \gamma, \varphi))^{[\![\kappa]\!]_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma}}\end{aligned}$$

The read and the write functions are translated independently w.r.t. the relabel model, $ref(\varphi, \gamma)^{\kappa}$.

The library read function, R, is translated by l-uread, which takes the read function, the current model frame stack, and the relabel model, γ.

$$\begin{aligned}\text{l-uread}(R, \underline{\ddot{\rho}}_2, \gamma) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1)\ .\ (\Sigma_2, \hat{v}) \\ \text{where}\quad & (\Sigma_2, v) = R(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1) \\ & \hat{v} = v \uparrow_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_2} \gamma\end{aligned}$$

When the resulting program read function is interacted with, the unlabeled read function is used to fetch the unlabeled value of the reference. The result is relabeled using the relabel model in the model state of the reference and the result is returned.

The library write function W is translated by l-uwrite, which takes the write function, the current model frame stack, the relabel model, γ, and the unlabel model, φ.

$$\begin{aligned}\text{l-uwrite}(W, \underline{\ddot{\rho}}_2, \gamma, \varphi) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, \hat{v})\ .\ \Sigma_3 \\ \text{where}\quad & \ell = [\![\text{lblterm}(\gamma)]\!]_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_1},\ \varsigma \sqsubseteq \ell \\ & (\Sigma_2, v) = \hat{v}^{\varsigma} \downarrow_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_1} \varphi \\ & \Sigma_3 = W(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_2, v)\end{aligned}$$

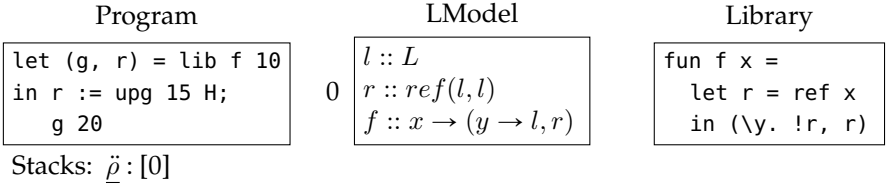


Figure 2.3: Initial structure

The reason l-uwrite takes the relabel model in addition to the unlabel model is that it is used to calculate the label against which the NSU check is made. The label of the stored value is represented by the label term of the relabel model, extracted by the lblterm function, defined in the obvious way by pattern matching. If the write is allowed, the labeled value to be written to the library reference is raised to the context ς, before being unlabeled using the unlabel model, φ. Finally, the unlabeled value is written to the library reference, using the unlabeled write function.

Interaction with the model heap To see how higher-order functions and references interact with the model heap, consider the code snippet below to the right. The program calls the library function `f`, which takes a parameter, and creates a reference `r` initially set to the value of the parameter. `f` returns a pair, where the first element is a function that, given any argument, will dereference the reference and the second element is the actual reference. This pair is stored as `(g, r)`. Thereafter, `r` is assigned the value 15^H, before `g` is called with the parameter 20^L.

```
let (g, r) = lib f 10
in r := upg 15 H;
   g 20
%%
lbl l :: L
mod r :: ref (l, l)
mod f :: x -> (y -> l, r)
fun f x = let r = ref x
          in (\y. !r, r)
```

The following occurs w.r.t. relabeling and unlabeling in the program, where the initial setting can be seen in Figure 2.3.

When `f` is lifted to the program, l-uclos is used to relabel the library closure, which will copy the model frame stack to the wrapped `f` and store the function model $x \to (y \to l, r)$. In the example, the resulting program closure is applied to 10^L, which causes a new model frame to be allocated on the model heap, into which the argument is unlabeled, causing L to be stored in the new model frame as the label for `x`, and the pointer to the new model frame is stored in the model frame stack. After this, the actual unlabeled function is called, which results in the returned pair being relabeled. The relabeling of the pair results in l-uclos being used to relabel `\y. !r` with the model $y \to l$, and l-uread and l-uwrite being used to relabel `r` with the

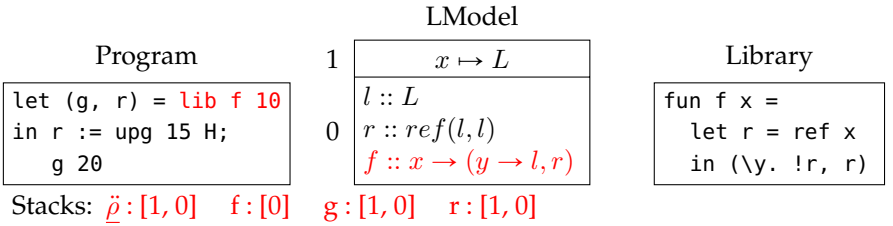


Figure 2.4: Calling relabeled closure

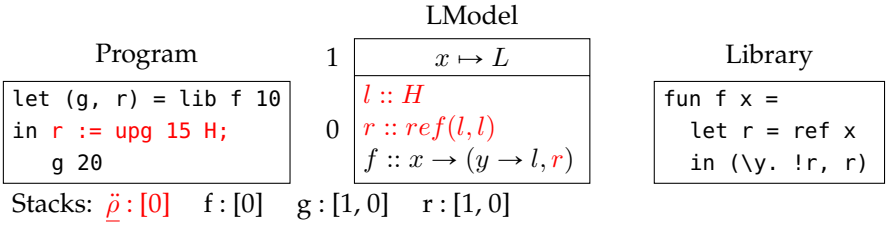


Figure 2.5: Writing to `r`

reference model r. The key here is that the relabeling occurs in the same model state, which means that the produced program function and reference will be bound to the same model frame stack. This causes writes to the reference to modify the model frame shared with the function, ensuring that they have the same view of the model of the reference. The entire process is highlighted in Figure 2.4.

When the program writes to the reference (`r := upg 15 H`), the closure from l-uwrite is triggered, causing l in the shared model frame to be updated to H, which can be seen in Figure 2.5. Note that the pointer to the model frame created from the call to the wrapped `f` is removed from the model frame stack. This ensures any subsequent calls to the wrapped `f`, as well as any created wrappers will not be able to use that model frame, as it belongs only to the first call to the wrapped `f` and the created wrappers *within* the call. When the function `g` is called, it will trigger its l-uclos wrapper and, as can be seen in Figure 2.6, the model $y \to l$ is used in the l-uclos wrapper for `g`, with l being used to relabel the result. Since l was modified by the writing to the reference (Figure 2.5), the shared view of the library model state, will make the function `g` return a secret value.

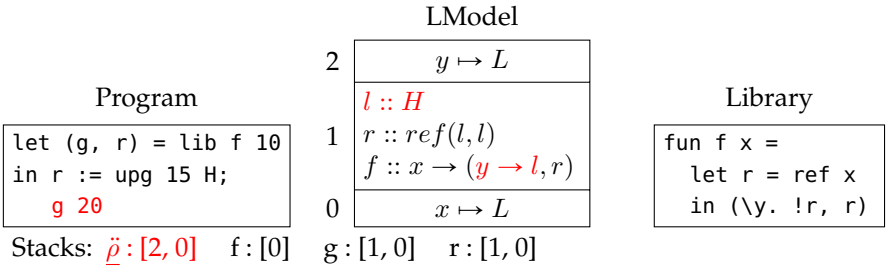


Figure 2.6: Calling `g`

4 Examples

In the following section we provide some examples to highlight how the language would interact with common programming techniques. The language used in this section is an extended version of the language of the paper. The major differences are the addition of records, functions with multiple arguments, a limited form of pattern matching, and optional unlabeling. The extensions are all present as experimental features in the implementation. In all examples, the code above `%%` is the program and the code below is the library.

Writebacks. Returning two or more results from a function can be done in two ways: 1) tupling the result, or 2) by using writebacks. When using writebacks for, e.g., reading a file, the read function is provided a pointer to a place in the memory where the contents of the file should be stored instead of returning a pointer to the data.

In our language, writebacks can be modeled by passing program references to the library as shown to the right. In the example, the program variable `buf` is a program reference. The reference is passed to the library function `action` that writes the result to the buffer. When interacting with a program reference, the reference is given an abstract name (`b` for buffer in this case) that the function interacting with the buffer uses to relabel the interaction.

```
let buf = ref 0
fun main () = (lib action) buf;
              !buf
%%
let data = 42
mod action :: #b -> L {| H |- #b <- H |}
fun action b = b := data; ()
```

In case the function used the writeback under secret control, represented by the model `mod action :: #b -> L {| H |- #b <- H |}`, the example would fail due to NSU. The reason being the value the reference `buf` is pointing to is public, and is not allowed to change label under secret control.

Modifying the declaration of `buf` to be `let buf = ref (upg 0 H)` solves this, as the reference will point to a secret value.

Library state. Libraries often keep state, e.g., error codes, computation results or options set by the program. Typical examples are the predefined object properties `$1,..,$9` from JavaScript RegExp [24].

The example to the right shows how state can be used to store error information. In the example, the function `action` may fail depending on the value of the parameter. The reason it failed is stored into the library reference `errno`, which is modeled by a security label used to relabel program reads and writes of the reference. Since the update of `errno` is conditional, it means that the value of `errno` is dependent of the argument of the `action` function. To model this, the argument label is stored in the model variable `a`, which is used to update the security label of `errno`. Note that the update of the security label is independent on whether the operation fails or not. This is needed to ensure that the label of `errno` is independent of secrets. The label of `errno` indicates that the error code is public. Consider the case where an action sets the error code under secret control, represented by the following model `mod action :: a -> L {| H |- l <- a |}`. If such an action was used our system would halt execution, since the update of the error code would trigger NSU.

```
fun main () =
  (lib action) (upg 42 H);
  print !(lib errno)
%%
lbl l :: L
mod errno :: ref (l, l)
let errno = ref 0

mod action :: a -> L {| l <- a |}
fun action x = if x == 1
               then errno := 1
               else ();
               ()
```

The one-place buffer. In the previous example, the library state is exposed to the program, which can freely read and write to `errno`. Frequently it is good practice to hide the internal state of the library and only allow the program to access it indirectly via the functions of the library. We exemplify this by implementing a simple one-place buffer, seen to the right. While simple, the example captures the essence of, e.g., buffered file access.

Since there is no model for `buf`, it is not accessible from the program. Instead, the state of the library is modeled using the label `l`. This label is used by the operations that give the pro-

```
fun main () = (lib set) (upg 42 H);
              (lib getAsync) print;
              (lib get) ()
%%
lbl l :: L
let buf = ref 0

mod set :: a -> a {| l <- a |}
fun set x = buf := x

mod get :: _ -> l
fun get () = !buf

mod getAsync :: #cb -> L {| #cb l -> _ |}
fun getAsync cb = cb !buf; ()
```

gram access to the buffer contents. When setting the value of the buffer via `set`, the label of the value is used to update the label of the library state. When reading, either via the synchronous function `get` or via the asynchronous function `getAsync`, `l` is used to relabel the dereferenced value from `buf`. In the synchronous case by relabeling the dereferenced value directly, and in the asynchronous case by relabeling the parameter to the callback. Note the use of the wildcard _ to indicate values that are not important for the model.

Stored callbacks. Stored callbacks are callbacks that are saved in the internal state of the library and used, e.g., to signal the occurrence of some event. A typical example of stored callbacks is the event handlers present in many languages.

Consider the program to the right that registers an event handler by storing the event handler (`print` in this case) in the `event` reference of the library. The relabel model of the `event` will unlabel the function and give it the abstract name `event`.

```
fun main () = lib event := print;
              (lib fire) (upg 42 H)
%%
lbl l :: L
mod event :: ref (l, #event^l)
let event = ref 0

mod fire :: a -> L {| #event a -> _ |}
fun fire x = !event x; ()
```

The event is triggered by calling the `fire` function, which takes the event data and passes it to the stored event handler. In the example, the `fire` function may be called from the program. In a practical setting, events may be triggered by interacting with the library (e.g., by adding values to a data structure) or from the library itself to indicate that certain events, such as mouse movement or clicks, have occurred.

In the example, it is not possible to fetch the event handler from the library and call it. In order to allow for this, we have to change the relabel model for the library reference to relabel read interactions as functions, changing the `event` model to be the following.

```
mod event :: ref (a -> b {| #event a -> b |}^l, #event^l)
```

To understand the new relabel model we must recognize that unlabeled program functions that are passed back need to be relabeled as any other library function. In this case, the library function that should be relabeled calls the unlabeled program function, and needs a corresponding call model. The result is a function that unlabels its argument into the label variable `a`, which is used to relabel the argument before calling the program function. The result of calling the program function is unlabeled into the label variable `b`, which in turn is used to relabel the result of the relabeled function.

5 Case study

For case study, we model an API inspired by the fs API of node.js [10]. In the interest of exposition we model the file system state as a single label as shown to the right. The extension of the model to nested records is simple but space demanding.

```
lbl l :: L
mod state :: ref (l, l)
let state = ref 1
```

Examples of functions in the API are the `rmdir` function and its synchronous sibling `rmdirSync`. Both will, given a path, remove the folder pointed to by the path. In addition, `rmdir` also takes a callback that is called with an error if the removal of the folder pointed to by the path fails.

```
mod rmdirSync :: a -> l + a {| l <- a |}
mod rmdir :: (a, #cb) -> L {| l <- a, #cb (l + a) -> b |}
```

We use the name `a` to represent the path and the abstract name `cb` to represent the callback. From a modeling standpoint, we need to ensure that the level of the path is propagated to the state, since removing the folder influences the file system state. We can see this in the effect constraint `l <- a`, where the label of the path is propagated to the label of the state. The success of the operation is depending on the library state and the security label of the path, `l + a`. Where `rmdirSync` returns the result, `rmdir` communicates the result to the callback as an argument, `#cb (l + a)`. The immediate return value of the latter is `undefined`, regardless of the outcome of the operation and hence labeled `L`.

A more complex function in the API is `createWriteStream` that returns a record. Calling `createWriteStream` with a path and an optional argument that defines options (e.g. the encoding) returns a WriteStream.

```
mod createWriteStream :: (a, b)
  -> { path         : a
     , bytesWritten : a + b + l
     , open         : #op -> L {| #op (a + b + l) -> o |}
     , close        : #cl -> L {| #cl (a + b + l) -> c |}
     }
```

The WriteStream has four parts; the fields `path` and `bytesWritten`, as well as the events `open` and `close`. For the model of the returned record, the property `path` is modeled by the argument `a`, which is the label of the path. The property `bytesWritten`, which corresponds to the amount of bytes written so far, is modeled as the least upper bound of `a`, `b` and `l`, i.e., the path, the options and the current library state. The events are modeled as functions that accept (and store) callbacks — the event handler — as modeled by the properties `open` and `close`. When the stream is opened or closed, the path, the options and the current library state all influence the parameter to those callbacks.

To contrast the case study with the examples, note that Section 4 makes the assumption that the source code of the library is available (albeit not supporting the labeled semantics) whereas this section makes the assumption it is not. Both cases are common, and can be modeled in our approach. In case the source code is indeed available an interesting line of future work is to look at the possibilities of automatically deducing models, e.g., using something similar to summary functions [27].

6 Correctness

The correctness of the language is complicated by the fact that it is parameterized over a library model that defines how to marshal values between the program and the library. Since we make no assumption on the implementation language of the library or the availability of the source code we cannot reason about the correctness of the model w.r.t. the library. Instead we assume the correctness of library models in terms of three hypotheses used in the noninterference proof. The low-equivalence definition, the model hypotheses and more information on the proof can be found in Appendix D and Appendix E, respectively.

We prove noninterference assuming that the library model correctly models the library as the preservation of a low-equivalence relation under execution. Apart from covering a larger language, the proof improves over [18] in two important aspects: 1) it significantly weakens the model hypothesis, and 2) the proof has been formalized in Coq [20].

Theorem 4 (Noninterference of labeled execution)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \varsigma, \Gamma \models (\Sigma_1, e) \rightarrow (\Sigma_2, \hat{v}) \ \wedge \\ \varsigma, \Gamma' \models (\Sigma_1', e) \rightarrow (\Sigma_2', \hat{v}') \Rightarrow (\Gamma, \Sigma_2) \simeq (\Gamma', \Sigma_2') \ \wedge \ \hat{v} \simeq \hat{v}'$$

Proof. By mutual induction on labeled and unlabeled evaluation (via u-lclos and l-uclos). The theorem makes use of *confinement*, i.e., that evaluation under high security does not modify the public part of the environment. □

7 Related work

Bielova and Rezk present a comprehensive taxonomy of information flow monitors [4]. Some monitors [16, 15, 14, 3] and secure multi-execution [13, 12, 25, 6, 21] mechanisms have been integrated in a browser. Bichhawat et al. instrumented the WebKit JavaScript interpreter [3]. While taking advantage of the current optimizations in the interpreter, it loses the differentiation

between the program and library execution. FlowFox [13], which implements *secure multi-execution (SME)* [6], modifies the SpiderMonkey engine in two ways: 1) augmenting the internal objects representing the JavaScript context with a current execution level, as well as a boolean indicating if SME is active, and 2) augmenting the internal representation of JavaScript values with a security level. Unfortunately, API calls are only treated as I/O actions. JSFlow [16] is an information-flow aware JavaScript interpreter, augmented with security labels on the JavaScript values. In order to allow for libraries in JSFlow, deep hand-written models must be used, with reimplementation of the libraries as a result [15]. To allow for scaling, JSFlow attempts to automatically wrap libraries, albeit in an ad-hoc manner. While the correctness of simple examples are easy to see, the correctness and scalability when passing, e.g., functions to and from the library remain unclear. Bauer et al. [2] developed a light-weight coarse-grained run-time monitor for Chromium, using taint tracking, to help reasoning about information flow in a fully fledged browser. In this work, formal models of, e.g., cookies, history and the *document object model (DOM)* are defined, as well as event handlers, to model the browser internals and help prove noninterference. Heule et al. [19] provided a theoretical foundation for a language-based approach for coarse-grained dynamic information flow control, that can be applied to any programming language where external effects can be controlled. A first step for handling libraries in environments where dynamic information flow control is not possible was taken by Hedin et al. [18], falling short by not supporting references, and thereby not allowing for first-class mutable state in combination with higher-order functions.

Findler and Feleisen's higher-order contracts [9] address the problem of checking contracts at the boundary between statically type-checked and dynamically type-checked code. The problem relates to the problem of interfacing with libraries where it is impossible to check dynamic information flow control. In particular, when considering function values crossing the boundary, the compliance of such function values with their respective contracts is undecidable. Findler and Feleisen proposed to wrap the function and check the contract at the point where the function is called. This is comparable to how we handle structured data, including references and function values. A question for future work is if we can remove our abstract identifiers for function values and references, and instead inject the unlabeling/relabeling functionality using proxies, similar to how it is done in higher-order contract checking [8]. If a contract is violated, the proper assignment of blame must be given [7, 11]. In static information flow checking, the assignment of blame has been investigated by King et al. for information flow violations [22]. Although our work can be seen as an application of dynamic higher-order contract checking for information flow contracts, we do

not consider assigning blame. Indeed, runtime detection of a library which does not obey the specified contract (i.e. the given model) is not possible in this work.

8 Conclusion

Based on a central idea of a model heap, we have developed a foundation for information flow tracking in the presence of libraries with side effects in a language with higher-order functions, first-class state and lazy-marshaling — three cornerstones of practical libraries. We have implemented a prototype to verify the examples and performed a larger case study that shows that the language is able to model key parts of a real file system library. In addition, we have formalized the language and its correctness proof in Coq.

Future work includes support for model abstraction and application, and dependent models. Thanks to the three cornerstones, we believe modeling JavaScript objects does not require development of new theory, indicating that it is possible to use this technique in tools like JSFlow.

Acknowledgments This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

9 Bibliography

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

[2] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *NDSS*. The Internet Society, 2015.

[3] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in webkit's javascript bytecode. In *POST*, 2014.

[4] N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *POST*, 2016.

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

[6] D. Devriese and F. Piessens. Noninterference Through Secure Multi-Execution. In *S&P*, 2010.

[7] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *POPL*, 2011.

[8] C. Dimoulas, M. S. New, R. B. Findler, and M. Felleisen. Oh Lord, please don't let contracts be misunderstood (functional pearl). In *ICFP*, 2016.

[9] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, 2002.

[10] File System | Node.js v9.2.0 Documentation. `https://nodejs.org/api/fs.html`. accessed: Nov 2017.

[11] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL*, 2010.

[12] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a Web Browser with Flexible and Precise Information Flow Control. In *CCS*, 2012.

[13] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Secure multi-execution of web scripts: Theory and practice. *Journal of Computer Security*, 2014.

[14] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for JavaScript and its APIs. *Journal of Computer Security*, 2015.

[15] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

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

[17] D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *CSFW-19*, 2006.

[18] D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld. A principled approach to tracking information flow in the presence of libraries. In *POST*, 2017.

[19] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *POST*, 2015.

[20] INRIA. The Coq Proof Assistant. `https://coq.inria.fr/`. accessed: Nov 2017.

[21] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *S&P*, 2011.

[22] D. King, T. Jaeger, S. Jha, and S. A. Seshia. Effective blame for information-flow violations. In *FSE*, 2008.

[23] Mozilla Developer Network. Proxy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. accessed: Mar 2018.

[24] Mozilla Developer Network. RegExp. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp`. accessed: Mar 2018.

[25] W. Rafnsson and A. Sabelfeld. Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent. In *CSF*, 2013.

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

[27] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. *Program Flow Analysis: Theory and Applications*, 1981.

[28] A. Sjösten, D. Hedin, and A. Sabelfeld. Information Flow Tracking for Side-effectful Libraries - Full version. `http://www.cse.chalmers.se/research/group/security/side-effectful-libraries/`.

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

A Full syntax

$$
\begin{array}{rcl}
e & ::= & n \mid x \mid \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \mid \mathit{fun}\ x = e \mid e_1\ e_2 \mid \\
& & x_{lib} \mid e_1 \oplus e_2 \mid \ominus e \mid \mathit{head}\ e \mid \mathit{tail}\ e \mid e_1 : e_2 \mid [\,] \mid (e_1, e_2) \mid (\,) \mid \\
& & \mathit{ref}\ e \mid !e \mid \{\ \underline{x} : \underline{e}\ \} \mid e.x \mid e_1 := e_2 \mid e_1\ ;\ e_2 \mid \mathit{upg}\ e\ \ell \\
\ell & ::= & L \mid H \\
d & ::= & \mathit{fun}\ f(x) = e \mid \mathit{let}\ x = e \\
m & ::= & \mathit{mod}\ x :: \gamma \mid \mathit{lbl}\ x :: \kappa \\
\kappa & ::= & \ell \mid \alpha \mid \kappa_1 \sqcup \kappa_2 \\
\gamma & ::= & \kappa \mid (\gamma_1, \gamma_2)^\kappa \mid [\gamma]^\kappa \mid (\varphi \to \gamma, \underline{\zeta})^\kappa \mid \mathsf{ref}(\varphi, \gamma)^\kappa \mid \{\ \}^\kappa \mid \{\ \alpha_1\ :\ \gamma_1,\ \ldots\ \}^\kappa \\
\varphi & ::= & \alpha \mid \#\alpha^\alpha \mid (\varphi_1, \varphi_2)^\alpha \mid [\varphi]^\alpha \mid \{\ \}^\alpha \mid \{\ \alpha_1\ :\ \varphi_1,\ \ldots\ \}^\alpha \\
\zeta & ::= & !\#\alpha \to \varphi \mid \kappa \vdash \#\alpha \leftarrow \gamma \mid \kappa \vdash \#\alpha\ \gamma \to \varphi \mid \kappa \vdash \alpha \leftarrow \kappa \\
\hat{v} & ::= & n^\ell \mid (\hat{v}_1, \hat{v}_2)^\ell \mid ()^\ell \mid \hat{F}^\ell \mid (\hat{H}, \hat{T})^\ell \mid [\,]^\ell \mid (\hat{R}, \hat{W})^\ell \mid \hat{O}^\ell \mid \{\ \}^\ell \\
\ddot{\imath} & ::= & \ell \mid \gamma \mid \zeta
\end{array}
$$

B Full labeled semantics

$$\textsc{int}\ \frac{}{\varsigma, \Gamma \models (\Sigma, n) \to (\Sigma, n^L)} \qquad \textsc{var}\ \frac{\text{lookupL}(\Gamma, \Sigma, x) = \hat{v}}{\varsigma, \Gamma \models (\Sigma, x) \to (\Sigma, \hat{v})}$$

$$\textsc{if-true}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 \neq 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_2^\ell)}$$

$$\textsc{if-false}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^\ell) \qquad v_1 = 0 \qquad \varsigma \sqcup \ell, \Gamma \models (\Sigma_2, e_3) \to (\Sigma_3, \hat{v}_3)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \to (\Sigma_3, \hat{v}_3^\ell)}$$

$$\textsc{fun}\ \frac{}{\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}) \models (\Sigma, \mathit{fun}\ x = e) \to (\Sigma, \text{lclos}(\underline{\hat{\rho}}, x, e)^L)}$$

$$\textsc{app}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{F}^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_1) \qquad \hat{F}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}_1) = (\Sigma_4, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ e_2) \to (\Sigma_4, \hat{v}_2^\ell)}$$

$$\textsc{ref}\ \frac{\varsigma, \Gamma \models (\Sigma, e) \to ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}) \qquad \hat{\rho}\ \text{fresh}}{\varsigma, \Gamma \models (\Sigma, \mathit{ref}\ e) \to ((\hat{\sigma}[\hat{\rho} \mapsto \hat{v}], \sigma, \ddot{\sigma}), (\text{lread}(\hat{\rho}), \text{lwrite}(\hat{\rho}))^L)}$$

$$\textsc{deref}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, !e) \to (\Sigma_3, \hat{v}^\ell)}$$

$$\textsc{assign}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}) \qquad \hat{W}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}) = \Sigma_4}{\varsigma, \Gamma \models (\Sigma_1, e_1 := e_2) \to (\Sigma_4, \hat{v})}$$

$$\textsc{lib}\ \frac{\text{lookupU}(\Gamma, \Sigma, s) = v \qquad \text{lookupM}(\Gamma, \Sigma, s) = \gamma \qquad v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}}{\varsigma, \Gamma \models (\Sigma, s_{lib}) \to (\Sigma, \hat{v})}$$

$$\textsc{tuple}_1\ \frac{}{\varsigma, \Gamma \models (\Sigma, (\,)) \to (\Sigma, (\,)^L)}$$

$$\textsc{tuple}_2\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{v}_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, (e_1, e_2)) \to (\Sigma_3, (\hat{v}_1, \hat{v}_2)^L)}$$

$$\textsc{label}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, \mathit{upg}\ e\ \ell) \to (\Sigma_2, \hat{v}^\ell)}$$

$$\textsc{sequence}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{v}_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ ;\ e_2) \to (\Sigma_3, \hat{v}_2)}$$

$$\texttt{let}\ \frac{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma_1, e_1) \to ((\hat{\sigma}, \sigma, \ddot{\sigma}), \hat{v}_1) \qquad \hat{\rho}\ \text{fresh} \quad \varsigma, (\hat{\underline{\rho}} \cdot \hat{\rho}, \underline{\rho}, \ddot{\underline{\rho}}) \models ((\hat{\sigma}[\hat{\rho} \mapsto \hat{v}_1], \sigma, \ddot{\sigma}), e_2) \to (\Sigma_2, \hat{v}_2)}{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma_1, \mathit{let}\ x = e_1\ \mathit{in}\ e_2) \to (\Sigma_2, \hat{v}_2)}$$

$$\texttt{binop}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, v_1^{\ell_1}) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, v_2^{\ell_2})}{\varsigma, \Gamma \models (\Sigma_1, e_1 \oplus e_2) \to (\Sigma_3, (v_1 \oplus v_2)^{\ell_1 \sqcup \ell_2})}$$

$$\texttt{unop}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, \ominus e) \to (\Sigma_2, \ominus \hat{v})}$$

$$\texttt{record-empty}\ \frac{}{\varsigma, \Gamma \models (\Sigma, \{\ \}) \to (\Sigma, \{\ \}^L)}$$

$$\texttt{record-nonempty}\ \frac{\forall i\ .\ 1 \leqslant i < n \wedge n \geqslant 2 \Rightarrow \varsigma, \Gamma \models (\Sigma_i, e_i) \to (\Sigma_{i+1}, \hat{v}_i)}{\varsigma, \Gamma \models (\Sigma_1, \{\ p_1 : e_1, \dots\ \}) \to (\Sigma_n, \mathsf{lproj}(\{\ p_1 : \hat{v}_1, \dots\ \})^L)}$$

$$\texttt{projection}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{O}^\ell) \qquad \hat{O}(\varsigma \sqcup \ell, \Gamma, \Sigma_2, p) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, e.p) \to (\Sigma_3, \hat{v}^\ell)}$$

$$\texttt{cons}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{v}_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1 : e_2) \to (\Sigma_3, \mathsf{lcons}(\hat{v}_1, \hat{v}_2)^L)}$$

$$\texttt{nil}\ \frac{}{\varsigma, \Gamma \models (\Sigma, [\,]) \to (\Sigma, [\,]^L)}$$

$$\texttt{head}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{H}, \hat{T})^\ell) \qquad \hat{H}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, \mathit{head}\ e) \to (\Sigma_3, \hat{v}^\ell)}$$

$$\texttt{tail}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{H}, \hat{T})^\ell) \qquad \hat{T}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})}{\varsigma, \Gamma \models (\Sigma_1, \mathit{tail}\ e) \to (\Sigma_3, \hat{v}^\ell)}$$

We will now explain the remaining evaluation relations which are not explained in Section 3.

Tuples (`tuple`$_1$ and `tuple`$_2$) are always labeled L. If the tuple is non-empty, the two expressions in the tuple are evaluated, and their resulting values returned (`tuple`$_2$). When using the keyword *upg*, the value will be raised to the given label ℓ (`label`), and evaluating multiple expressions in sequence evaluates the first expression e_1, and uses the modified environment when evaluating e_2, with the final value in the sequence being the returned (`sequence`). With let bindings (`let`), the bound expression e_1 is evaluated. The labeled heap is updated with a new frame, where the value of the evaluation of the bound expression is stored. This new environment is then used when evaluating the body e_2. Evaluation of binary operators (`binop`) evaluates

both expressions, and then applies the binary operator to the result. The result is then raised to the least-upper bound of the two operands. Unary operators (`unop`) are evaluated similarly, but with only one operand.

In order to trap interaction with the different properties of a record, records are represented as functions $\hat{O} : (\varsigma, \Gamma, \Sigma, p) \to (\Sigma, \hat{v})$. Creation of a record is done in two ways: it is either an empty record, labeled L (`record-empty`), or it is a function (`record-nonempty`) which, given the mapping between properties and values, is defined by lproj as follows.

$$\begin{aligned}
&\text{lproj}(rec) = \lambda(\varsigma, \Gamma, \Sigma, id).\ \text{lfind}(\Gamma, \Sigma, rec, id) \\
&\quad \text{where} \\
&\qquad \text{lfind}(\Gamma, \Sigma, \{\,\}, id) = (\Sigma, ()) \\
&\qquad \text{lfind}(\Gamma, \Sigma, \{\, p : \hat{v}, rest \,\}, id) = \begin{cases} (\Sigma, \hat{v}), \text{ if } p = id \\ \text{lfind}(\Gamma, \Sigma, \{\, rest \,\}, id) \text{ otherwise} \end{cases}
\end{aligned}$$

When projecting a record property (`projection`), the projected property p is applied to $\hat{O}$ along with the current execution environment, which uses lfind to find the corresponding labeled value. Creation and interaction with a library record, $O : (\varsigma, \Gamma, \Sigma, p) \to (\Sigma, v)$ is analogous.

A list is a pair of functions, representing a cons cell. One function is for reading the head of the list, $\hat{H} : (\varsigma, \Gamma, \Sigma) \to (\Sigma, \hat{v})$, and one for reading the tail of the list, $\hat{T} : (\varsigma, \Gamma, \Sigma) \to (\Sigma, \hat{v})$. A list can be created in two ways. Either the list is empty (`nil`), and it will be labeled `L`. Or the list is non-empty (`cons`), and the two given values (the head and tail of the cons cell) will be wrapped using lcons to allow for delaying computations as follows.

$$\text{lcons}(\hat{v_1}, \hat{v_2}) = (\lambda(\varsigma, \Gamma, \Sigma).\ (\Sigma, \hat{v}_1), \lambda(\varsigma, \Gamma, \Sigma).\ (\Sigma, \hat{v}_2))$$

Reading the elements of the list is done by the primitives *head* and *tail* (rules `head` and `tail`), which will call $\hat{H}$ and $\hat{T}$ for the head and tail respectively. Upon interaction with $\hat{H}$ and $\hat{T}$, the current execution environment is passed, with the context while evaluating the head or tail function being the least-upper bound of the current context and the label of the cons cell. Creation and use of a library list, $H : (\varsigma, \Gamma, \Sigma) \to (\Sigma, v)$ and $T : (\varsigma, \Gamma, \Sigma) \to (\Sigma, v)$, is analogous, apart from the fact that there is no cons cell label, making the evaluation of H and T occur in the current execution context.

B.1 Marshaling

Primitive values Integers and tuples are eagerly marshaled. The unlabel and relabel models follow the structure of the value, which is fully taken apart in order to construct the labeled or unlabeled counterpart.

Unlabeling. Unlabeling an integer n^ℓ, based on an unlabel model α will bind the label ℓ to the name α in the model heap. Unlabeling of the empty tuple is performed structurally in an analogous way, while unlabeling a non-empty tuple first unlabels the components before binding ℓ to the name α. When unlabeling the two components of the tuple, the updated heap from unlabeling the first component is used when unlabeling the second component. Similarly, it is the updated heap from unlabeling the second component that is used to bind ℓ to α.

$$\begin{array}{rcl} n^\ell \downarrow_{\varsigma,\Gamma,\Sigma} \alpha & = & (\text{updateM}(\varsigma,\Gamma,\Sigma,\alpha,\ell), n) \\ (\,)^\ell \downarrow_{\varsigma,\Gamma,\Sigma} \alpha & = & (\text{updateM}(\varsigma,\Gamma,\Sigma,\alpha,\ell), (\,)) \\ (\hat{v}_1,\hat{v}_2)^\ell \downarrow_{\varsigma,\Gamma,\Sigma_1} (\varphi_1,\varphi_2)^\alpha & = & (\text{updateM}(\varsigma,\Gamma,\Sigma_3,\alpha,\ell), (v_1,v_2)) \\ & & \quad \text{where} \quad (\Sigma_2, v_1) = \hat{v}_1 \downarrow_{\varsigma,\Gamma,\Sigma_1} \varphi_1 \\ & & \qquad\quad (\Sigma_3, v_2) = \hat{v}_2 \downarrow_{\varsigma,\Gamma,\Sigma_2} \varphi_2 \end{array}$$

Relabeling. Relabeling an integer interprets the label term, κ, in the given environment. A tuple is relabeled structurally, by relabeling the components and the tuple itself.

$$\begin{array}{rcl} n \uparrow_{\Gamma,\Sigma} \kappa & = & n^{[\![\kappa]\!]_{\Gamma,\Sigma}} \\ (\,) \uparrow_{\Gamma,\Sigma} \kappa & = & (\,)^{[\![\kappa]\!]_{\Gamma,\Sigma}} \\ (v_1,v_2) \uparrow_{\Gamma,\Sigma} (\gamma_1,\gamma_2)^\kappa & = & (v_1 \uparrow_{\Gamma,\Sigma} \gamma_1, v_2 \uparrow_{\Gamma,\Sigma} \gamma_2)^{[\![\kappa]\!]_{\Gamma,\Sigma}} \end{array}$$

Records When marshaling a record, it is important that only the used properties of a record affects the model state when going back and forth between the program and the library.

Unlabeling. As a record can be either empty or non-empty, there are two unlabel models for records. The unlabel model for an empty record is $\{\,\}^\alpha$, whereas a non-empty record has $\{\,\alpha_1 \;:\; \varphi_1, \ldots\}^\alpha$, where each property is given a separate unlabel model. When passing a program record to the library, the model heap must be altered, by defining the name α to point to the label ℓ, which is the program record's structure label.

$$\begin{array}{l} \{\,\}^\ell \downarrow_{\varsigma,\Gamma,\Sigma} \{\,\}^\alpha = (\text{defineM}(\Gamma,\Sigma,\alpha,\ell), \{\,\})) \\ \hat{O}^\ell \downarrow_{\varsigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma_1} \{\,\alpha_1 \;:\; \varphi_1, \ldots\}^\alpha = \\ \quad (\text{defineM}((\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma,\alpha,\ell), \text{u-lrec}(\hat{O},\ell,\underline{\ddot{\rho}},\{\,\alpha_1 \;:\; \varphi_1, \ldots\})) \end{array}$$

The translation of the program record is performed by u-lrec, which takes the program record, the label of the program record, the model frame pointer stack, and the unlabel models for the properties.

$$\begin{aligned}\text{u-lrec}(\hat{O}, \ell, \underline{\ddot{\rho}}_2, \{\ \alpha_1\ :\ \varphi_1, \ldots\}) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, p).\ (\Sigma_3, v)\\ \text{where}\quad (\Sigma_2, \hat{v}) &= \hat{O}(\varsigma \sqcup \ell, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, p)\\ \varphi &= \text{mfind}(\{\ \alpha_1\ :\ \varphi_1, \ldots\}, p)\\ (\Sigma_3, v) &= \hat{v}^{\ell}\downarrow_{\varsigma\sqcup\ell, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_2), \Sigma_2}\ \varphi\end{aligned}$$

When the resulting library record is interacted with, the original program record is used to get the labeled value corresponding to the property p. The function mfind is then used to get the unlabel model φ corresponding to the property p from the record unlabel model, which dictates how to unlabel the labeled value for property p.

$$\text{mfind}(\{\ p : m, rest\ \}, id) = \begin{cases} m, \text{ if } p = id \\ \text{mfind}(\{\ rest\ \}, id) \text{ otherwise} \end{cases}$$

Due to the structure being the same for record unlabel models and record label models, mfind is generic and will be used both when unlabeling and relabeling a record. Finally the labeled value corresponding to the property p is unlabeled with the unlabel model returned from mfind, using the old model frame stack $\underline{\ddot{\rho}}_2$.

Relabeling. Relabeling a record follows the structure of the record: it is either an empty record, or a record with properties.

$$\begin{aligned}&\{\ \} \uparrow_{\Gamma,\Sigma} \{\ \}^{\kappa} = \{\ \}^{[\![\kappa]\!]_{\Gamma,\Sigma}}\\ &O \uparrow_{\Sigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}})} \{\ p_1\ :\ \gamma_1, \ldots\ \}^{\kappa} = \text{l-urec}(O, \underline{\ddot{\rho}}, \{\ p_1\ :\ \gamma_1, \ldots\ \}^{\kappa})^{[\![\kappa]\!]_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma}}\end{aligned}$$

If the library record is empty, an empty program record is returned, with the label of the interpretation of the label term κ in the current environment. However, if the library record is non-empty, the auxiliary function l-urec is used, and takes the library record, the current model frame pointer stack, and the relabel model of the record. The resulting program record from l-urec is labeled with the interpretation of κ.

$$\begin{aligned}\text{l-urec}(O, \underline{\ddot{\rho}}_2, \{\ p_1\ :\ \gamma_1, \ldots\}) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, p).\ (\Sigma_2, \hat{v})\\ \text{where}\quad (\Sigma_2, v) &= O(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1, p)\\ \gamma &= \text{mfind}(\{\ p_1\ :\ \gamma_1, \ldots\}, p)\\ \hat{v} &= v \uparrow_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_2), \Sigma_2}\ \gamma\end{aligned}$$

When the resulting program record is interacted with, library record is used to get the unlabeled value corresponding to the property p. The resulting unlabeled value is then relabeled with the relabel model γ, corresponding to the relabel model returned from mfind for property p in the record relabel

model. As with unlabeling of records, the relabeling of the value takes place with the old model frame stack $\underline{\ddot{\rho}}_2$, bound when the relabeling was first initiated.

Lists Lists are marshaled using *lazy marshaling*, which dictates only the traversed elements of the list should affect the unlabeling and relabeling of a list.

Unlabeling. A library list is unlabeled as a value, translating the head and tail functions to its unlabeled counterpart, as well as updating the cons cell label in the model heap.

$$
\begin{aligned}
&[\,]^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} [\varphi]^{\alpha} = (\text{updateM}(\varsigma,\Gamma,\Sigma,\alpha,\ell), [\,]) \\
&(\hat{H},\hat{T})^{\ell} \downarrow_{\varsigma,\Gamma,\Sigma} [\varphi]^{\alpha} = \\
&\quad (\text{updateM}(\varsigma,\Gamma,\Sigma,\alpha,\ell), (\text{u-lhead}(\hat{H},\ell,\underline{\ddot{\rho}},\varphi), \text{u-ltail}(\hat{T},\ell,\underline{\ddot{\rho}},[\varphi]^{\alpha})))
\end{aligned}
$$

The translation of a non-empty list is performed by translating the head and tail functions independently. $\hat{H}$ is translated by u-lhead, which takes the program head function, the label of the cons cell, the current model frame pointer stack, an the unlabel model for unlabeling the program value that corresponds to the head of the list.

$$
\begin{aligned}
\text{u-lhead}(\hat{H},\ell,\underline{\ddot{\rho}}_2,\varphi) &= \lambda(\varsigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_1),\Sigma_1).\ (\Sigma_3, v) \\
\text{where} \quad (\Sigma_2,\hat{v}) &= \hat{H}(\varsigma \sqcup \ell, (\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_1),\Sigma_1) \\
(\Sigma_3, v) &= \hat{v}^{\ell} \downarrow_{\varsigma\sqcup\ell,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_2} \varphi
\end{aligned}
$$

When the resulting library head function is called, the program head function $\hat{H}$ is used to get the corresponding labeled value, while being executed in the security context of the least upper bound of the current security context and the label of the cons cell. The resulting labeled value must be unlabeled w.r.t. the unlabel model φ using the model frame pointer stack $\underline{\ddot{\rho}}_2$, which was bounded when the marshaling process started before being returned to the library.

Similarly, the program tail function $\hat{T}$ is translated using u-ltail, which takes the program tail function, the label of the cons cell, the current model frame pointer stack, and the unlabel model of the entire list, as the tail of a list is a list.

$$
\begin{aligned}
\text{u-ltail}(\hat{T},\ell,\underline{\ddot{\rho}}_2,[\varphi]^{\alpha}) &= \lambda(\varsigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_1),\Sigma_1).\ (\Sigma_3, v) \\
\text{where} \quad (\Sigma_2,\hat{v}) &= \hat{T}(\varsigma \sqcup \ell, (\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_1),\Sigma_1) \\
(\Sigma_3, v) &= \hat{v}^{\ell} \downarrow_{\varsigma\sqcup\ell,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_2} [\varphi]^{\alpha}
\end{aligned}
$$

As with unlabeling of $\hat{H}$, when the resulting library tail function is interacted with, the program tail function is used to get the corresponding labeled value.

Since evaluating the program tail function can result in either the empty list [ ] or a new pair $(\hat{H}', \hat{T}')$, the full unlabel model $[\varphi]^{\alpha}$ must be passed to u-ltail. The corresponding labeled value is then unlabeled with the old model frame pointer stack $\ddot{\underline{\rho}}_2$, and the resulting library value along with the updated memory gets returned.

Relabeling. When relabeling a library list, the library list is translated into a program counterpart.

$$[\,] \uparrow_{\Gamma,\Sigma} [\gamma]^{\kappa} = [\,]^{[\![\kappa]\!]_{\Gamma,\Sigma}}$$
$$(H, T) \uparrow_{\Sigma,(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}})} [\gamma]^{\kappa} = (\text{l-uhead}(H, \underline{\ddot{\rho}}, \gamma), \text{l-utail}(T, \underline{\ddot{\rho}}, [\gamma]^{\kappa}))^{[\![\kappa]\!]_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}),\Sigma}}$$

The case for the empty list is simply interpreting the label term κ in the current model state, and the corresponding label is used for labeling the empty program list. When relabeling an unlabeled cons cell, (H, T), each function is relabeled separately, with the corresponding program cons cell being labeled with the interpretation of κ in the current model state. Translating the library head function H is done by l-uhead, which takes the library head function, the current model frame pointer stack and the relabel model for the value corresponding to the head of the list.

$$\begin{aligned} \text{l-uhead}(H, \underline{\ddot{\rho}}_2, \gamma) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1).\ (\Sigma_2, \hat{v}) \\ \text{where} \quad & (\Sigma_2, v) = H(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1) \\ & \hat{v} = v \uparrow_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_2} \gamma \end{aligned}$$

The resulting program head function, when interacted with, takes the current execution environment as parameter. The library head function H is then used to calculate the unlabeled value, which is then relabeled with the element relabel model γ with the old model frame stack $\ddot{\underline{\rho}}_2$.

Relabeling of the library tail function T is done in a similar fashion, using l-utail, which takes the library tail function, the current model frame pointer stack and the list relabel model.

$$\begin{aligned} \text{l-utail}(T, \underline{\ddot{\rho}}_2, [\gamma]^{\kappa}) &= \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma_1).\ (\Sigma_2, \hat{v}) \\ \text{where} \quad & (\Sigma_2, v) = T(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}_1), \Sigma) \\ & \hat{v} = v \uparrow_{(\underline{\hat{\rho}},\underline{\rho},\underline{\ddot{\rho}}_2),\Sigma_2} [\gamma]^{\kappa} \end{aligned}$$

When the resulting program tail function is interacted with, the library tail function T is used to calculate the unlabeled value. As with unlabeling a program tail function, the resulting value from T can be either an empty list [ ], or a new pair of functions (H', T'). This unlabeled value is relabeled using the list relabel model $[\gamma]^{\kappa}$ with the old model frame pointer stack $\ddot{\underline{\rho}}_2$.

It is due to trapping the interaction using the head and tail functions that allow for the lazy marshaling. This lazy marshaling in turn ensures only the traversed elements of a list will affect the resulting labels.

C Full unlabeled semantics

$$\textsf{int}\ \frac{}{\varsigma, \Gamma \models (\Sigma, n) \rightsquigarrow (\Sigma, n)} \qquad \textsf{var}\ \frac{\text{lookupU}(\Gamma, \Sigma, x) = v}{\varsigma, \Gamma \models (\Sigma, x) \rightsquigarrow (\Sigma, v)}$$

$$\textsf{if-true}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad v_1 \neq 0 \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_2)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow (\Sigma_3, v_2)}$$

$$\textsf{if-false}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad v_1 = 0 \qquad \varsigma, \Gamma \models (\Sigma_2, e_3) \rightsquigarrow (\Sigma_3, v_3)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3) \rightsquigarrow (\Sigma_3, v_3)}$$

$$\textsf{fun}\ \frac{}{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma, \mathit{fun}\ x = e) \rightsquigarrow (\Sigma, \text{uclos}(\underline{\rho}, x, e))}$$

$$\textsf{app}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, F) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_1) \qquad F(\varsigma, \Gamma, \Sigma_3, v_1) = (\Sigma_4, v_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ e_2) \rightsquigarrow (\Sigma_4, v_2)}$$

$$\textsf{ref}\ \frac{\varsigma, \Gamma \models (\Sigma, e) \rightsquigarrow ((\hat{\sigma}, \sigma, \ddot{\sigma}), v) \qquad \rho\ \text{fresh}}{\varsigma, \Gamma \models (\Sigma, \mathit{ref}\ e) \rightsquigarrow ((\hat{\sigma}, \sigma[\rho \mapsto v], \ddot{\sigma}), (\text{uread}(\rho), \text{uwrite}(\rho)))}$$

$$\textsf{deref}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, (R, W)) \qquad R(\varsigma, \Gamma, \Sigma_2) = (\Sigma_3, v)}{\varsigma, \Gamma \models (\Sigma_1, !e) \rightsquigarrow (\Sigma_3, v)}$$

$$\textsf{assign}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, (R, W)) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v) \qquad W(\varsigma, \Gamma, \Sigma_3, v) = \Sigma_4}{\varsigma, \Gamma \models (\Sigma_1, e_1 := e_2) \rightsquigarrow (\Sigma_4, v)}$$

$$\textsf{tuple}_1\ \frac{}{\varsigma, \Gamma \models (\Sigma, (\,)) \rightsquigarrow (\Sigma, (\,))}$$

$$\textsf{tuple}_2\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_2)}{\varsigma, \Gamma \models (\Sigma_1, (e_1, e_2)) \rightsquigarrow (\Sigma_3, (v_1, v_2))}$$

$$\textsf{sequence}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1\ ;\ e_2) \rightsquigarrow (\Sigma_3, v_2)}$$

$$\textsf{let}\ \frac{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma_1, e_1) \rightsquigarrow ((\hat{\sigma}, \sigma, \ddot{\sigma}), v_1) \qquad \rho\ \text{fresh} \qquad \varsigma, (\hat{\underline{\rho}}, \rho \cdot \underline{\rho}, \ddot{\underline{\rho}}) \models ((\hat{\sigma}, \sigma[\rho \mapsto v_1], \ddot{\sigma}), e_2) \rightsquigarrow (\Sigma_2, v_2)}{\varsigma, (\hat{\underline{\rho}}, \underline{\rho}, \ddot{\underline{\rho}}) \models (\Sigma_1, \mathit{let}\ x = e_1\ \mathit{in}\ e_2) \rightsquigarrow (\Sigma_2, v_2)}$$

$$\texttt{binop}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1 \oplus e_2) \rightsquigarrow (\Sigma_3, (v_1 \oplus v_2))}$$

$$\texttt{unop}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, v)}{\varsigma, \Gamma \models (\Sigma_1, \ominus e) \rightsquigarrow (\Sigma_2, \ominus v)}$$

$$\texttt{record-empty}\ \frac{}{\varsigma, \Gamma \models (\Sigma, \{\ \}) \rightsquigarrow (\Sigma, \{\ \})}$$

$$\texttt{record-nonempty}\ \frac{\forall i\ .\ 1 \leqslant i < n \wedge n \geqslant 2 \Rightarrow \varsigma, \Gamma \models (\Sigma_i, e_i) \rightsquigarrow (\Sigma_{i+1}, v_i)}{\varsigma, \Gamma \models (\Sigma_1, \{\ p_1 : e_1, \ldots\ \}) \rightarrow (\Sigma_n, \mathsf{uproj}(\{\ p_1 : v_1, \ldots\ \}))}$$

$$\texttt{projection}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, O) \qquad O(\varsigma, \Gamma, \Sigma_2, p) = (\Sigma_3, v)}{\varsigma, \Gamma \models (\Sigma_1, e.p) \rightsquigarrow (\Sigma_3, v)}$$

$$\texttt{cons}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e_1) \rightsquigarrow (\Sigma_2, v_1) \qquad \varsigma, \Gamma \models (\Sigma_2, e_2) \rightsquigarrow (\Sigma_3, v_2)}{\varsigma, \Gamma \models (\Sigma_1, e_1 : e_2) \rightsquigarrow (\Sigma_3, \mathsf{ucons}(v_1, v_2))}$$

$$\texttt{nil}\ \frac{}{\varsigma, \Gamma \models (\Sigma, [\,]) \rightsquigarrow (\Sigma, [\,])}$$

$$\texttt{head}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, (H, T)) \qquad H(\varsigma, \Gamma, \Sigma_2) = (\Sigma_3, v)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{head}\ e) \rightsquigarrow (\Sigma_3, v)}$$

$$\texttt{tail}\ \frac{\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, (H, T)) \qquad T(\varsigma, \Gamma, \Sigma_2) = (\Sigma_3, v)}{\varsigma, \Gamma \models (\Sigma_1, \mathit{tail}\ e) \rightsquigarrow (\Sigma_3, v)}$$

D Low-equivalence

Low-equivalence for labeled values, $\hat{v} \simeq \hat{v}'$, encodes the intuition that public values should be equal. Two environments are low-equivalent, $(\Gamma, \Sigma) \simeq (\Gamma', \Sigma')$ if both the labeled stack and labeled heap and the model frame stack and model heap are low-equivalent. Low-equivalence is defined as follows.

$$\frac{}{n^L \simeq n^L} \qquad \frac{}{v_1^H \simeq v_2^H} \qquad \frac{\hat{v}_1 \simeq \hat{v}_1' \qquad \hat{v}_2 \simeq \hat{v}_2'}{(\hat{v}_1, \hat{v}_2)^L \simeq (\hat{v}_1', \hat{v}_2')^L}$$

$$\frac{}{(\,)^L \simeq (\,)^L} \qquad \frac{}{[\,]^L \simeq [\,]^L} \qquad \frac{\hat{v}_1 \simeq \hat{v}_2 \qquad \hat{v}_1' \simeq \hat{v}_2'}{\mathsf{lcons}(\hat{v}_1, \hat{v}_2)^L \simeq \mathsf{lcons}(\hat{v}_1', \hat{v}_2')^L}$$

$$\frac{\ddot{\underline{\rho}} \simeq \ddot{\underline{\rho}}'}{(\text{l-uhead}(H, \ddot{\underline{\rho}}, \gamma, L), \text{l-utail}(T, \ddot{\underline{\rho}}, [\gamma]^\kappa, L)) \simeq (\text{l-uhead}(H', \ddot{\underline{\rho}}', \gamma, L), \text{l-utail}(T', \ddot{\underline{\rho}}', [\gamma]^\kappa, L))}$$

$$\frac{\underline{\hat{\rho}} \simeq \underline{\hat{\rho}}'}{\text{lclos}(\underline{\hat{\rho}}, x, e)^L \simeq \text{lclos}(\underline{\hat{\rho}}', x, e)^L}$$

$$\frac{\underline{\ddot{\rho}} \simeq \underline{\ddot{\rho}}'}{\text{l-uclos}(F, \underline{\ddot{\rho}}, (\varphi \to \gamma, \underline{\zeta}), L) \simeq \text{l-uclos}(F', \underline{\ddot{\rho}}', (\varphi \to \gamma, \underline{\zeta}), L)}$$

$$\frac{\hat{\rho} \simeq_{\hat{\beta}} \hat{\rho}'}{(\text{lread}(\hat{\rho}), \text{lwrite}(\hat{\rho}))^L \simeq (\text{lread}(\hat{\rho}'), \text{lwrite}(\hat{\rho}'))^L}$$

$$\frac{\underline{\ddot{\rho}} \simeq \underline{\ddot{\rho}}'}{\begin{array}{c}(\text{l-uread}(R, \underline{\ddot{\rho}}, \gamma, L), \text{l-write}(W, \underline{\ddot{\rho}}, ref(\varphi, \gamma), L)) \simeq \\ (\text{l-uread}(R', \underline{\ddot{\rho}}', \gamma, L), \text{l-write}(W', \underline{\ddot{\rho}}', ref(\varphi, \gamma), L))\end{array}}$$

$$\frac{}{[\,] \simeq [\,]} \qquad \frac{\hat{\rho} \simeq_{\hat{\beta}} \hat{\rho}' \qquad \underline{\hat{\rho}} \simeq \underline{\hat{\rho}}'}{\hat{\rho} \cdot \underline{\hat{\rho}} \simeq \hat{\rho}' \cdot \underline{\hat{\rho}}'} \qquad \frac{\ddot{\rho} \simeq_{\ddot{\beta}} \ddot{\rho}' \qquad \underline{\ddot{\rho}} \simeq \underline{\ddot{\rho}}'}{\ddot{\rho} \cdot \underline{\ddot{\rho}} \simeq \ddot{\rho}' \cdot \underline{\ddot{\rho}}'} \qquad \frac{}{\ddot{v} \simeq \ddot{v}}$$

$$\frac{\forall x \,.\, (\hat{\omega}[x] = \hat{v} \wedge \hat{\omega}'[x] = \hat{v}' \wedge \hat{v} \simeq \hat{v}') \vee (\hat{\omega}[x] = \bot \wedge \hat{\omega}'[x] = \bot)}{\hat{\omega} \simeq \hat{\omega}'}$$

$$\frac{\forall x \,.\, (\ddot{\omega}[x] = \ddot{v} \wedge \ddot{\omega}'[x] = \ddot{v}' \wedge \ddot{v} \simeq \ddot{v}') \vee (\ddot{\omega}[x] = \bot \wedge \ddot{\omega}'[x] = \bot)}{\ddot{\omega} \simeq \ddot{\omega}'}$$

$$\frac{\underline{\hat{\rho}} \simeq \underline{\hat{\rho}}' \qquad \begin{array}{c}\forall \hat{\rho}, \hat{\rho}' \,.\, \hat{\rho} \simeq_{\hat{\beta}} \hat{\rho}' \Rightarrow \hat{\sigma}[\hat{\rho}] \simeq \hat{\sigma}'[\hat{\rho}'] \\ \forall \ddot{\rho}, \ddot{\rho}' \,.\, \ddot{\rho} \simeq_{\ddot{\beta}} \ddot{\rho}' \Rightarrow \ddot{\sigma}[\ddot{\rho}] \simeq \ddot{\sigma}'[\ddot{\rho}']\end{array} \qquad \underline{\ddot{\rho}} \simeq \underline{\ddot{\rho}}'}{((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma})) \simeq ((\underline{\hat{\rho}}', \underline{\rho}', \underline{\ddot{\rho}}'), (\hat{\sigma}', \sigma', \ddot{\sigma}'))}$$

$$\frac{}{n \simeq_L n} \qquad \frac{}{v_1 \simeq_H v_2} \qquad \frac{v_1 \simeq_L v_1' \qquad v_2 \simeq_L v_2'}{(v_1, v_2) \simeq_L (v_1', v_2')}$$

$$\frac{}{(\,) \simeq_L (\,)} \qquad \frac{}{[\,] \simeq_L [\,]} \qquad \frac{\hat{v}_1 \simeq \hat{v}_2 \qquad \hat{v}_1' \simeq \hat{v}_2'}{\text{ucons}(v_1, v_2) \simeq_L \text{ucons}(v_1', v_2')}$$

$$\frac{\hat{H} \simeq \hat{H}' \qquad \hat{T} \simeq \hat{T}' \qquad \underline{\ddot{\rho}} \simeq \underline{\ddot{\rho}}'}{\begin{array}{c}(\text{u-lhead}(\hat{H}^L, \underline{\ddot{\rho}}, \varphi), \text{u-ltail}(\hat{T}^L, \underline{\ddot{\rho}}, [\varphi]^\alpha)) \simeq_L \\ (\text{u-lhead}(\hat{H}'^L, \underline{\ddot{\rho}}', \varphi), \text{u-ltail}(\hat{T}'^L, \underline{\ddot{\rho}}', [\varphi]^\alpha))\end{array}}$$

$$\frac{}{\text{uclos}(\underline{\rho}, x, e) \simeq_L \text{uclos}(\underline{\rho}', x, e)}$$

$$\frac{\hat{F} \simeq \hat{F}'}{\text{u-lclos}(\hat{F}, \pi, L) \simeq_L \text{u-lclos}(\hat{F}', \pi, L)}$$

$$\frac{}{(\text{uread}(\rho), \text{uwrite}(\rho)) \simeq_L (\text{uread}(\rho'), \text{uwrite}(\rho'))}$$

$$\frac{\hat{R} \simeq \hat{R}' \qquad \hat{W} \simeq \hat{W}'}{\begin{array}{c}(\text{u-lread}(\hat{R}, \pi, L), \text{u-lwrite}(\hat{W}, \pi, L)) \simeq_L \\ (\text{u-lread}(\hat{R}', \pi, L), \text{u-lwrite}(\hat{W}', \pi, L))\end{array}}$$

E Correctness

The correctness of the language is complicated by the fact that it is parameterized over a library model that defines how to marshal values between the program and the library. Noninterference is only guaranteed given that the library model correctly models the behavior of the library. We handle this by assuming the correctness of the library model in terms of three hypotheses.

The first hypothesis deals with the possibility of relabeling the unlabeled values of the library. It makes use of a non-standard low-equivalence relation for unlabeled values, $v \simeq_\ell v'$, that expresses that v and v' are low-equivalent w.r.t. label ℓ. The hypothesis states that the result of evaluating an expression in the unlabeled semantics returns values that can be correctly labeled with *some* label. The existential abstracts the fact the model selects the label, and the assumption the label is correct.

Hypothesis 1 (Labelability of unlabeled execution)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, v) \ \wedge \\ \varsigma, \Gamma' \models (\Sigma_1', e) \to (\Sigma_2', v') \Rightarrow \exists \ell \,.\, v \simeq_\ell v'$$

Even though the hypothesis speaks about all expressions it should be understood in relation to where it is used: when the labeled semantics explicitly calls the library.

The second hypothesis states that the result of relabeling two labelable unlabeled values is low-equivalent, regardless of the relabel model or model state. This hypothesis expresses the assumption that the library model is correct.

Hypothesis 2 (Unlabeled correctness of library models)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \exists \ell \,.\, v \simeq_\ell v' \ \wedge \ v \uparrow_{\Gamma,\Sigma} \gamma = \hat{v} \ \wedge \ v' \uparrow_{\Gamma',\Sigma'} \gamma = \hat{v}' \Rightarrow \hat{v} \simeq \hat{v}'$$

The third hypothesis states that the library model correctly models program callbacks and side effects.

Hypothesis 3 (Labeled correctness of library models)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, v) \ \wedge \\ \varsigma, \Gamma' \models (\Sigma_1', e) \to (\Sigma_2', v') \Rightarrow (\Gamma, \Sigma_2) \simeq (\Gamma', \Sigma_2')$$

The three hypotheses are used in the proof, when values are passed from the library to the program.

For the remainder of this section, we will further explain some of the key concepts and discuss some limitations of the Coq formalization. The formalization can be obtained at [28].

Records. The implementation of records can be seen as a generalization of lists. To keep the formalization reasonably maintainable, we have opted to exclude records. Lists are part of the formalization and suffice to establish the compatibility of lazy marshaling and the model state.

Noninterference. We prove noninterference as the preservation of low-equivalence under execution.

Theorem 1 (Noninterference of labeled execution)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma', \Sigma_1') \ \wedge \ \varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{v}) \ \wedge \\ \varsigma, \Gamma' \models (\Sigma_1', e) \to (\Sigma_2', \hat{v}') \Rightarrow (\Gamma, \Sigma_2) \simeq (\Gamma', \Sigma_2') \ \wedge \ \hat{v} \simeq_\ell \hat{v}'$$

In addition to confinement the noninterference proof uses noninterference results for basic primitives like lookupL, lookupU, and lookupM as well as form updateL, updateU and updateM. Those constructions are entirely standard, and has been proved correct in previous work [18]. We have chosen to admit their proofs of noninterference in the formalization. For the major constructions of the labeled semantics — values, lists, higher-order functions and references as well as the evaluation relation and the unlabel and relabel functions — noninterference has been formally proved.

Confinement. Confinement expresses that execution under secret control does not modify public parts of the environment. As is common we express confinement in terms of the low-equivalence relation.

Lemma 2 (Confinement of labeled execution)**.**

$$(\Gamma, \Sigma_1) \simeq (\Gamma, \Sigma_1) \wedge H, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, \hat{v}) \Rightarrow (\Gamma, \Sigma_1) \simeq (\Gamma, \Sigma_2)$$

Confinement uses confinement results for basic primitives like lookupL, lookupU, and lookupM as well as form updateL, updateU and updateM. Those constructions are entirely standard, and has been proved correct in previous work [18]. We have chosen to admit their proofs of confinement in

the formalization. For the major constructions of the labeled and unlabeled semantics — values, lists, higher-order functions and references as well as the evaluation relation and the unlabel and relabel functions — confinement has been formally proved.

Low-equivalence. Low-equivalence with heap allocated values either requires the heap to be split into a public and a secret part or perform the correctness argument up to isomorphism of the low-reachable parts of the heap. We have opted for the latter, since it requires no modifications of the semantics. However, reasoning up to bijection on two heaps requires that two bijections are maintained and pushed throughout the proof, which significantly increases the proof burden and drenches the core of the proof in unimportant details. To handle the bijections are entirely standard and has been done formally by Hedin and Sands [17] in their work on opaque pointers. For this reason we have decided to axiomatize the handling of the bijections, while maintaining them in the low-equivalence relation to make the interaction between the frame stacks and the corresponding heaps clear.

In addition, the noninterference proof makes use of symmetry, transitivity and conditional reflexivity of the low-equivalence relation. The relations clearly satisfy these properties, why we have admitted their proofs.

F Heap operations

$$\text{lookupL-1}\ \frac{x \in \hat{\sigma}[\hat{\rho}] \qquad \hat{\sigma}[\hat{\rho}][x] = \hat{v}}{\text{lookupL}((\hat{\rho} \cdot \underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \hat{v}}$$

$$\text{lookupL-2}\ \frac{x \notin \hat{\sigma}[\hat{\rho}] \qquad \text{lookupL}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \hat{v}}{\text{lookupL}((\hat{\rho} \cdot \underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \hat{v}}$$

$$\text{lookupU-1}\ \frac{x \in \sigma[\rho] \qquad \sigma[\rho][x] = v}{\text{lookupU}((\underline{\hat{\rho}}, \rho \cdot \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = v}$$

$$\text{lookupU-2}\ \frac{x \notin \sigma[\rho] \qquad \text{lookupU}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = v}{\text{lookupU}((\underline{\hat{\rho}}, \rho \cdot \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = v}$$

$$\text{lookupM-1}\ \frac{x \in \ddot{\sigma}[\ddot{\rho}] \qquad \ddot{\sigma}[\ddot{\rho}][x] = \ddot{v}}{\text{lookupM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{v}}$$

$$\text{lookupM-2}\ \frac{x \notin \ddot{\sigma}[\ddot{\rho}] \qquad \text{lookupM}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{v}}{\text{lookupM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{v}}$$

$$\text{findM-1}\ \frac{x \in \ddot{\sigma}[\ddot{\rho}]}{\textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{\rho}}$$

$$\text{findM-2}\ \frac{x \notin \ddot{\sigma}[\ddot{\rho}] \qquad \textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{\rho}'}{\textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \ddot{\rho}'}$$

$$\text{findM-3}\ \frac{}{\textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \varnothing), (\hat{\sigma}, \sigma, \ddot{\sigma}), x) = \bot}$$

$$\text{updateM-1}\ \frac{\textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), \alpha) = \bot}{\textsf{updateM}(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), \alpha, \ell) = (\hat{\sigma}, \sigma, \ddot{\sigma}[\ddot{\rho}][\alpha \mapsto \ell])}$$

$$\text{updateM-2}\ \frac{\textsf{findM}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), \alpha) = \ddot{\rho} \qquad \ddot{\sigma}[\ddot{\rho}][\alpha] = \ell' \qquad \varsigma \sqsubseteq \ell'}{\textsf{updateM}(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), \alpha, \ell) = (\hat{\sigma}, \sigma, \ddot{\sigma}[\ddot{\rho}][\alpha \mapsto \ell \sqcup \ell'])}$$

$$\text{defineM}\ \frac{\alpha \notin \ddot{\sigma}[\ddot{\rho}]}{\textsf{defineM}((\underline{\hat{\rho}}, \underline{\rho}, \ddot{\rho} \cdot \underline{\ddot{\rho}}), (\hat{\sigma}, \sigma, \ddot{\sigma}), \alpha, \ddot{v}) = (\hat{\sigma}, \sigma, \ddot{\sigma}[\ddot{\rho}][\alpha \mapsto \ddot{v}])}$$

Paper III

EssentialFP: Exposing the Essence of Browser Fingerprinting

Alexander Sjösten, Daniel Hedin, Andrei Sabelfeld

Under submission

Abstract

Web pages aggressively track users for a variety of purposes from targeted advertisement to enhanced authentication. As browsers move to restrict traditional cookie-based tracking, web pages increasingly move to tracking based on browser fingerprinting. Unfortunately, the state of the art to detect fingerprinting in browsers is often error-prone, resorting to heuristics and crowd-sourced filter lists. This paper presents EssentialFP, a principled approach to detecting fingerprinting on the web. We argue that the pattern of (i) gathering information from a wide browser API surface (multiple browser-specific sources) and (ii) communicating the information to the network (network sink) captures the essence of fingerprinting. This pattern enables us to clearly distinguish fingerprinting from similar types of scripts like analytics and polyfills. We demonstrate that information flow tracking is an excellent fit for exposing this pattern. To implement EssentialFP we leverage, extend, and deploy JSFlow, a state-of-the-art information flow tracker for JavaScript, in a browser. We illustrate the effectiveness of EssentialFP to spot fingerprinting on the web by evaluating it on several categories of web pages: analytics, authentication, bot detection, and fingerprinting-enhanced Alexa top pages.

1 Introduction

Web pages aggressively track users for a variety of purposes such as targeted advertisement, enhanced security, and personalized content [54]. Web tracking is subject to much debate [40, 25] that, in the light of privacy-enhancing legislation [18], has led the major browser vendors to introduce anti-tracking measures.

From cookies to fingerprinting To keep track of users a unique identifier was traditionally stored in a cookie. However, in the face of growing awareness of privacy concerns (e.g., the "do not track" flag [39]), a study by Yen et al. showed 32% of the users could not be completely tracked using only cookies [90].

Increasingly, web pages have been moving to collect *browser fingerprints*, "information related to a user's device from the hardware to the operating system to the browser and its configuration" [69]. Mayer showed in a small-scale study that 96.23% of users could be uniquely identified by combining seemingly benign browser features. Eckersley then showed in a large-scale study that by combining information such as the screen dimensions, the user agent and the installed browser plugins it was possible to uniquely identify 83.6% of the tested browsers looking at only eight attributes (increased to 94.2% if the browsers supported Flash or Java) [53].

Recent studies show that (i) browser fingerprinting is becoming increasingly prevalent [44, 54]; (ii) modern technique include hardware fingerprinting through the Canvas [73] and WebGL [73, 50] APIs; and (iii) on average, a fingerprint can track a browser instance for 54.48 days [89]. From analyzing newer data, researches have also shown that the number of uniquely identifiable users based on the fingerprint has gone down, but that modifying a few features of the fingerprint very probably makes it become unique [57].

Privacy-violating fingerprinting As with tracking in general, web pages fingerprint users for a variety of purposes. To thwart the privacy-violating fingerprinting efforts browser vendors have introduced mitigations, which include randomizing the output of known fingerprinting vectors by Brave [41], using privacy budgets by Chrome [29], blocking third-party requests suspected of being tracking related by Edge [24] and Firefox [17] based on, e.g., the Disconnect list [13], and making more devices look identical by Safari [35]. The diversity of these techniques reflects that they each come with their pros and cons, with no clear principle on how to detect fingerprinting. This reflects in large numbers of both false positives and negatives [2, 3, 4, 5].

In addition to browser vendors taking responsibility in limiting tracking and fingerprinting, a user can install browser extensions like Privacy Badger [30] or Ghostery [19] to help block privacy-intrusive scripts. These extensions use filter lists, which are collections of rules dictating what should be blocked. When generating the filter lists, a large number of community members usually collaborate in identifying undesirable resources, a process known as crowd-sourcing. A canonical example of such a list is the above-mentioned Disconnect list [13]. While this approach is popular, it has the obvious limitation: as fingerprinting scripts are discovered and labeled as such, the attacker has the possibility to easily evade the blacklisting by cloning scripts and serving them from different Internet domains [51].

Recent research approaches to limit fingerprinting include randomization of features [75, 68, 87], modifying the fingerprint per session [86, 55], and making users look identical through virtualization [70, 56]. Yet making every user look unique or making users look identical to prevent fingerprinting also breaks techniques that use fingerprinting to improve security, e.g., to make authentication stronger [67]. We discuss these and further related approaches in Section 6.

Generally, while the above approaches focus on solving the problem for specific vectors, they fall short of addressing the general case: there is currently no uniform solution for identifying fingerprinting. This leads us to our first research question: *RQ1: What is the essence of browser fingerprinting?*

"Bad" vs. "good" fingerprinting What makes the problem of fingerprinting intricate is that not all fingerprinting is "bad" fingerprinting [69]. Indeed,

fingerprinting can be justified to increase security when used to improve e.g. bot detection, fraud detection, and protection against account hijacking [45]. Where to draw the line between "bad" and "good" fingerprinting is an open and arguably subjective question. Approaches that try to draw this line are bound to result in both false positives and negatives.

The stance of this paper is thus neutral: we focus on identifying the *presence* of fingerprinting, hence providing necessary input into the decision process (by the user and browser) on whether to allow it. This motivates our second research question: *RQ2: How do we reliably expose fingerprinting scripts in a principled way, without relying on ad-hoc heuristics and crowd-sourcing, while at the same time not having to judge the fingerprinting as "bad" or "good"?*

Fingerprinting Our key observation is that the essence of fingerprinting can be captured by the pattern: (i) gathering information from a wide browser API surface (multiple browser-specific sources) and (ii) communicating the information to the network (network sink). The communication in (ii) may either be direct (via, e.g., `XMLHttpRequest`) or indirect (via, e.g., the cookie) and may be done by sending the raw data piece by piece or (more commonly) as a precomputed fingerprint.

One natural start for semantic detection of fingerprinting is with the gathered information, i.e, the API surface accessed by fingerprinting code (the *API imprint*). If the information that escapes the browser has a significant overlap with the API imprint this may be indicative of fingerprinting. The *flow* of information is key to reliable detection: we must track how information flows from the API imprint to the network sinks. It does not suffice to compare an application's API access pattern to the API imprint due to the risk for false positives (every use of the API would count towards fingerprinting). In particular, in the presence of *polyfills*, the API access patterns of applications naturally become rather large, thus increasing the risk of false positives significantly. (Polyfills, such as Modernizr [26] and core-js [11], are libraries intended to extend older browsers with support for new features. To be able to do this they probe and enhance the execution environment by injecting any missing features.)

Based on this, we propose EssentialFP, a principled approach that utilizes dynamic *Information-Flow Control (IFC)* as a means to expose fingerprinting.

Lightweight information flow control There are various forms of IFC, all sharing the same fundamental concept but differing in how information is tracked and what security guarantees they provide. We refer the reader to [77, 62] for more background on various flavors of IFC. EssentialFP utilizes a variant of dynamic IFC known as *observable tracking* [48, 83]. In dynamic IFC all values are given a runtime security label computed to reflect the information used in the construction of the value. This is done by tracking

two types of information flows. The first kind of flow, the *explicit flows*, correspond to data flows [52] in traditional program analysis. An explicit flow occurs when one or more values are combined into a new value (Note that copying a value is a special case of this.), e.g., when adding two numbers. `c = a + b;`. After execution, the value stored in `c` is the sum of the values stored in `a` and `b` which is reflected by letting the label of the value of `c` be the join of the labels of the values of `a` and `b`.

The second kind of flow, the *implicit flows*, corresponds to control flows [52] in traditional program analysis. Implicit flows arise between values when one value indirectly influences another via the control flow of the program. Consider the following program

```
if (a) { b = true; } else { b = false; }
```

where the value of `b` is indirectly influenced by the value of `a`, since it controls which branch is taken and, hence, which assignment is executed. Assuming `a` is a boolean variable, the above implicit flow is equivalent to the explicit flow `b = a;` and the label of the value of `b` must be computed to reflect this flow of information. To track implicit flows observable tracking maintains a security label associated with the control flow, the so-called *pc* label. This label is used to ensure that any values influencing the control flow are taken into account when computing the labels of side effects.

Implicit flow are not of theoretical importance only [63, 83]. Unless implicit flows are tracked, important flows are potentially missed. Consider the following code taken from FingerprintJS:

```
var getNavigatorPlatform = function (options)
{
    if (navigator.platform) {
        return navigator.platform
    } else {
        return options.NOT_AVAILABLE
    }
}
```

In case `navigator.platform` is present, there is an observable implicit flow from it to the return value of the function. This code from FingerprintJS represents a common pattern, where the link between the original API source and the sink would be lost in the negative cases.

Given this, observable tracking is an excellent match to capture the essence of fingerprinting. To track how information flows from the API imprint to the network sinks we label the values originating from the API imprint with their access paths and capture (the accumulation) of labels exiting the browser via network sinks. The observable tracking ensures that the flow of information internal to the program correctly reflects any flow of information from the API imprint to the network.

Metrics via aggregated labels To detect fingerprinting we accumulate the labels that reach each of the network sinks and compare the result against a baseline API imprint computed from the API imprints of three major open-source fingerprinting libraries to find the largest sink. The overlap between the largest sink and the API imprint gives us how large part of a full fingerprint that escapes the browser to the fingerprinting party. The hypothesis is that fingerprinting applications have a larger overlap than other non-fingerprinting applications. In addition, we also gather all internally created labels. This set of labels represent the values created during executions and allows us to detect whether the script itself compiles information from the baseline API imprint into one value, i.e., a fingerprint. Having information on how much fingerprinting information that reaches the API and if this information was compiled by the page or sent piece by piece allows us to gain further insights into how fingerprinting operates in the wild.

Implementation and evaluation Our approach leverages JSFlow [61, 60, 59], a state-of-the-art dynamic IFC monitor for ECMA-262 version 5. We extend JSFlow to handle libraries not natively supported by ECMA-262 version 5 and modify Chromium to use JSFlow to execute scripts instead of V8. The resulting browser, EssentialFP, allows us to visit pages while tracking and collecting data about the flows of information on the page.

A large-scale evaluation on thousands of web pages is not in scope of this work because our focus is on providing a platform for experimenting with and deep understanding of JavaScript on web pages. To evaluate the approach we have selected a number of web pages divided into four categories: 1) pages that do analytics, 2) pages that do authentication, 3) pages that do bot detection, and 4) pages that do fingerprinting. Each of those pages was then executed using EssentialFP and the collected data analyzed w.r.t. the baseline API imprint. Our results show a clear distinction between the analytics category and the fingerprinting category of pages where fingerprinting pages reach scores of 20% and above, while all analytics pages fall well under 20%. For bot detection and authentication the results are more subtle due to variations in how much fingerprinting is used to enhance the security.

Contributions In summary, this paper offers the following contributions:

(i) We develop EssentialFP a principled approach to fingerprinting detection based on observable tracking (Section 2). We define the sources and sinks and design a metric that allows us to characterize fingerprinting patterns via aggregated labels.

(ii) We present the design and implementation of EssentialFP to allow JSFlow to track information within web APIs, and how to track label combinations of known fingerprinting patterns (Section 3).

(iii) We present an empirical study, where we visit web pages based on different categories (analytics, authentication, bot detection, and fingerprinting Alexa top pages), demonstrating the effectiveness of EssentialFP (Section 4).

The code of our tool and its benchmarks are available to the reviewers online [14]. We will release them publicly upon publication.

2 Approach

Based on our observation that the essence of fingerprinting can be captured by the pattern of (i) gathering information from a wide browser API surface (multiple browser-specific sources) and (ii) communicating the information to the network (network sink) we suggest detecting fingerprinting by using a variant of dynamic IFC known as observable tracking. Observable tracking uses runtime values that are labeled with security labels. The labels are combined during execution to reflect the flow of information in the program. In brief, our approach relies on the computation of a baseline API imprint capturing all parts of the API that contain fingerprinting-sensitive information, but instead of comparing it to the API access pattern of the application we track the accumulation of fingerprinting-sensitive information at the network sinks. This is done by labeling any information that originates from the baseline API imprint with the access path, causing all fingerprinting-sensitive information to carry its origin as a security label. During the execution, the observable tracking ensures values that reach the network sinks are correctly labeled to reflect the information that was used in their creation. The labels of the escaping values are then accumulated for each network sink and compared to the baseline API imprint to detect the presence of fingerprinting. In our case, the label analysis is done after the execution of the page, but nothing prevents a runtime solution where information would be allowed to escape until a certain threshold has been met. Such a solution would be related to the use of a *privacy budget* [10], with the difference that it measures the budget on the escaping information, rather than the read information.

2.1 Computing the API imprint

We use three known and widely used open-source fingerprinting libraries to compute the baseline API imprint: FingerprintJS [16], ImprintJS [21], and

ClientJS [9]. Of the three FingerprintJS is the best maintained and to a large extent the API imprint of FingerprintJS supersedes the API imprints of the other two. Common to those three fingerprinting libraries is that they are extensively configurable w.r.t. what information to use in the fingerprint. The reason for this configurability is that different fingerprinting uses require different properties of the fingerprint. For instance, for user identification, where the fingerprint is used as an identifier, two properties are important for the fingerprint: 1) uniqueness and 2) time stability [46]. If the produced fingerprint is not unique any users that share the same fingerprint will be mistakenly identified, and if the fingerprint is not stable the same user will appear to be different users. Those two properties are to a certain extent implementation antagonists; a large API imprint promotes the uniqueness of the fingerprint while potentially causing the fingerprint to become less stable, while a smaller API imprint potentially leads to a more stable fingerprint, while, at the same time, less unique. In addition, a large API imprint is easier to detect also for the more crude detection mechanisms that are presently available. Indeed, our empirical study indicates that deployed fingerprinting carefully select which fingerprinting features to use, likely both to avoid detection and to promote stability.

To compute the baseline API imprint we combine the API imprints of FingerprintJS, ImprintJS, and ClientJS running in isolation with all features turned on. This was done by creating specialized web pages for each library containing only the fingerprinting code. Each such page was then executed while labeling the results of all API accesses with their access paths. This way, the API imprint of each of the libraries could be read directly from the label of the resulting fingerprint as a set of access paths. The three resulting sets were then combined to form the final baseline API imprint.

2.2 Detecting fingerprinting

The baseline API imprint identifies the part of the execution environment that contains fingerprinting-sensitive information. By labeling all values originating from this part with the access path, tracking the information flow dynamically and monitoring the label creation and flow during the execution we are able to detect fingerprinting in applications. For each page and API endpoint we accumulate the labels of all values reaching the endpoint. From this set of endpoints we select the potential network sinks, i.e., endpoints that can be used to communicate information, such as `XMLHttpRequest`, or store information, such as `window.localStorage`, `window.sessionStorage`, and `document.cookie`. In addition, we gather all labels of all values created by the application. This allows us to detect if the web page gathers and compiles a fingerprint internally even if the fingerprint is not communicated over the

network. Since this collection is relatively resource intensive when done on actual web pages, we restrict the initial labeling to the API imprint.

The result of the label collection is two maps: one mapping network sinks to the accumulated label of escaping values, and one mapping all script origins to the set of created labels. Those two maps allow us to analyze each webpage for both internal creation of fingerprints and fingerprinting, i.e., where the collected information leaves the browser via a network sink. More precisely we can distinguish between the following uses.

- Traditional use: information is gathered, composed and sent (detected as both internal creation of fingerprints and fingerprinting)
- Piece by piece: information is gathered and sent (detected as fingerprinting without internal creation of fingerprints)
- Local use: information is gathered, composed and used, but not sent (detected as internal creation of fingerprints without fingerprinting)
- No fingerprinting: no information is composed, used or sent (detected as neither internal creation of fingerprints or fingerprinting)

To identify the presence of internal creation of fingerprints, we measure the maximum overlap of the created labels for each script w.r.t. the baseline API imprint, and, to identify the presence of fingerprinting, we compute the overlap of each identified network sink and select the largest as the fingerprinting flow. The reason we do not join the network sinks is that we cannot be sure that all of them go to the same receiver. By only selecting the largest sink we don't overestimate the flow.

In addition to detecting the presence of fingerprinting, it is also interesting to try to identify the kind of fingerprinting that takes place. To this end, we use the per-flag extracted API imprints of FingerprintJS. This gives us the possibility to characterize detected fingerprinting in terms of features and to identify common patterns using heatmaps.

3 Design and implementation

To perform measurements and detect fingerprinting we have created an information flow aware browser: EssentialFP. EssentialFP is implemented as a combination of Chromium [8] and JSFlow [61, 60, 59], a security-enhanced JavaScript interpreter, allowing fine-grained information flow tracking. At the core, EssentialFP is a modified version of Chromium that uses JSFlow instead of V8 to execute JavaScript. JSFlow is deployed as a library [71], allowing JSFlow to execute all script content of a page. The injection is done

by trapping all scripts to be executed and wrapping them in a call to the JSFlow execute function.

3.1 Extending JSFlow

The current release of JSFlow supports ECMA-262 version 5 (ES5) [22] along with the mandated standard runtime environment. In order to use JSFlow to run actual web pages, a number of extensions must be made. JSFlow must be extended to support new features defined in ECMA-262 version 6 [36] and later standards (ES6+). In addition, JSFlow must also be extended to mediate between its own execution environment and the execution environment of the browser, as well as collecting the created and escaping labels seen during page execution.

Extending JSFlow to ES6+ Initial attempts showed that a large portion of web pages use ES6+ features and, thus, do not run fully using an interpreter that only supports ES5. Extending JSFlow with support for ES6+ is a large undertaking that would require both extending the core engine of JSFlow, as well as the standard libraries. Such an extension would require that large portions of JSFlow be rewritten. While preferable, we opted for the reasonable middle ground of adding support for ES6+ in JSFlow by using a combination of transpiling and polyfilling.

Before any script is executed, JSFlow transpiles the code from ES6+ to ES5 using Babel [6]. This will produce a new program that should be semantically equivalent to the original, but only use ES5 features. As an example, assume the following code which uses the arrow function (=>), introduced in ES6.

```
[1,2,3].map(x => x + 1)
```

Since the arrow function does not exist in ES5, Babel transpiles the above piece of code to the following.

```
[1,2,3].map(function (x) {
    return x + 1;
})
```

The result contains only ES5 features and can be executed by JSFlow.

In addition to transpiling, we use polyfills to provide the parts of the ES6 runtime that JSFlow does not implement. Before any code is executed JSFlow executes a runtime bundle containing all support libraries and polyfills needed for proper execution. This runtime bundle extends the JSFlow runtime environment with polyfills from core-js [11] for the ES6+ standard runtime, regenerator-runtime [34] which is needed by some of Babel's transformations, and window-crypto [42] which adds the `Window.crypto` functionality [43].

Connecting the execution environments In order to use JSFlow to execute scripts in a browser environment, it is imperative to connect the JSFlow execution environment to the browser's V8 execution environment. This requires mediation between V8 values and JSFlow values. The mediation needs to be bidirectional: JSFlow values must be mediated to values V8 can interpret and V8 values must be mediated to values JSFlow can interpret. Further, it is crucial that the mediation is connected in the sense that modifications done in either execution environment are reflected in the other. As an example, if a script registers an event handler by assigning a function to a property, the assignment must be pushed from the JSFlow execution environment to the V8 execution environment. and the JSFlow function must be mediated. Since JSFlow functions are not V8 functions using a JSFlow function as a V8 callback would not work and the event handler would not be called when the event occurs.

To mediate between JSFlow and V8, we scale the mediation technique presented by Sjösten et al. [82] to full JavaScript in the browser setting. The mediation differs depending on the type of value and the direction of mediation.

Masquerading JSFlow values as V8 values Mediating primitive values is relatively simple since JSFlow primitive values are pairs of V8 values and security labels. When mediating values from JSFlow to V8, these security labels must be removed: a process we call *unlabeling*. After a primitive value has been unlabeled, it can be directly transferred to V8.

Mediating non-primitive values such as `Functions` or `Objects` requires recursive mediation of their parts. To retain the connection between the original JSFlow entity and the V8 entity we use Proxies [31]. The proxy allows a JSFlow object to masquerade as a V8 object and performs recursive on-the-fly mediation of the JSFlow entity on access.

Masquerading V8 values as JSFlow values Similar to above, primitive values can be passed from the V8 execution environment to the JSFlow execution environment as they are, apart from the need to add a security label: a process called *relabeling*.

Mediating non-primitive values such as `Functions` or `Objects` requires recursive mediation of their parts. This is done by using wrapper objects. These wrappers are special versions of the JSFlow internal Ecma objects, that in addition to the standard object behavior push mediated values between the wrapper and its host. The mediation follows a read-once-write-always semantics. That is, when a property is read, if it is defined on the host object, it is brought from the V8 execution environment, wrapped and cached as an ordinary JSFlow property on the wrapper. Subsequent reads interact with

the wrapper as an ordinary JSFlow object. When a property is written, it is written both unmediated to the wrapper as well as mediated to the host.

The mediation explained above allows for bringing entities from the V8 runtime into the JSFlow runtime and vice versa, which effectively extends JSFlow with the APIs provided by the browser. This allows scripts to interact with the browser as if they are running directly in V8.

Protecting JSFlow Since JSFlow itself runs in the V8 execution environment it is mediating to and from, there is a need to protect the integrity of JSFlow. As the mediation allows the scripts to modify the V8 execution environment, JSFlow must run the scripts defensively, protecting key parts of the environment from being tampered with. JSFlow itself uses (parts of) the standard ES6+ execution environment, and protection is provided by the JSFlow implementation of the ES5 execution environment and the global object as follows. The standard execution environment of JSFlow does not perform mediation, which means that scripts are unable to modify those parts of the V8 execution environment. Parts that are not part of the JSFlow execution environment are all mediated via the JSFlow global window object. This object is a hybrid between a JSFlow global object and a wrapper and provides protection from tampering by hiding sensitive parts of the execution environment. The parts of the execution environment defined by JSFlow via this global object or that is part of the hidden environment will not be mediated and instead implemented by the polyfills. Other parts follow the read-once-write-always mediation provided by the JSFlow wrappers.

Label models The labeling and unlabeling of entities when mediating between JSFlow and V8 rely on label models which provide an abstract view of the computation of the mediated V8 values. Providing such a model for the full execution environment of a browser is not within the scope of this work, and we refer the reader to [82] for an insight into the complexity of creating such models. Instead, we use a simpler label model, where parts of the execution environment can be selected to be labeled with the access path. As an example, reading the V8 runtime property `navigator.userAgent` would label the resulting value with the label `<global.navigator.userAgent>`, where `global` represents the global `window` object. When computing the baseline API imprint of the fingerprinting libraries, we use a model that labels every mediated part of the V8 API. This baseline is then used for a more conservative model used when crawling pages. This model only labels the parts of the V8 API that is part of the baseline.

3.2 Extending Chromium

Modifying and maintaining modifications on a commercial product like Chromium requires a lot of work. The updates are frequent and the changes are often major, potentially requiring a lot of effort to cope with if the modification is substantial. For this reason, we try to keep the Chromium modifications to a minimum.

To be able to inject JSFlow we focus the modifications to the point in Chromium where the script source code is transferred from the rendering engine Blink to V8 as a string for execution. There, the following functionality was inserted.

- If the intercepted script is the first script to execute in the context JSFlow is first injected, followed by the injection of the JSFlow runtime containing polyfills and other supporting libraries.
- When JSFlow has been injected the script is rewritten to contain a call to the main execution method of JSFlow passing the original source code as an argument.

To exemplify the rewriting consider a page containing the following script:

```
<script>
  console.log("Hello World!");
</script>
```

As described above, the injection will wrap the entire script in a call to JSFlow. In this particular case, the result would become

```
<script>
  jsflow.executeAndUnlabelResult(
    "console.log(\"Hello World\");"
  );
</script>
```

The injection works in the same way for inline scrips, scrips fetched via a URL, or retrieved by other means. At the point of injection Chromium has already extracted the script source into a string. This way, all scripts on a page are executed via JSFlow regardless of if they originate from a inline script tag, a script tag using a URL, or an event handler.

3.3 Collecting labels

Every value that is going through JSFlow will be given a label by the label model, and when two values are combined, the corresponding label will be the least upper bound of the labels. In our setting, since labels are the access paths of the information used to create the values, the least upper bound

corresponds to the union of the paths. Take, for example, the following code snippet.

```
let a = navigator.userAgent;
let b = navigator.language;
let c = a + ' ' + b;
```

The value `a` will be labeled `<navigator.userAgent>` and the value `b` will be labeled `<navigator.language>`. Since the value `c` is the aggregation of values `a` and `b` it will, therefore, be labeled with both sources forming the label `<navigator.userAgent, navigator.language>`.

In order to analyze the aggregated labels, JSFlow was extended with the ability to store the labels seen during execution on a script basis. In practice this is done on label creation; whenever the least upper bound is computed in JSFlow, the computed label will be stored internally in a map that maps script origins to label sets. This allows us to detect whether any values that may correspond to a fingerprint were created by the script.

In addition, in order to be able to detect fingerprinting, we are interested in the flow from the API imprint to the network sinks. To be able to track this we also store an accumulated label per API endpoint. Every time a JSFlow value is unlabeled to be passed into the V8 runtime (by an assignment or function call), we add the removed label to a map that maps API endpoints to labels.

During the execution of a page, JSFlow regularly writes the accumulated labels via a function on the global object if such a function exists. This way automated tools, like crawlers, are able to collect labels from JSFlow by providing the extraction function. In our case, we use a Puppeteer-based [32] crawler that stores the collected labels to disk for later analysis.

4 Empirical study

In order to validate our approach, we have conducted an empirical study by crawling web pages belonging to one of four different categories:

- (analytics) web pages that use analytics but not fingerprinting,
- (bot detection) web pages that perform bot detection,
- (authentication) web pages that use some form of fingerprinting as part of their authentication process, and
- (fingerprinting) web pages in Alexa top 100,000 that perform fingerprinting that is not part of bot detection or authentication.

A total of 25 web pages were selected: 5 in each of the categories analytics, bot detection, and authentication, and 10 in the fingerprinting category. Depending on the category different rationals were used in the selection.

Analytics For the analytics category we wanted to find pages that make use of analytics but are free from fingerprinting code. Analytics is widely used and we made the assumption that popular pages would include analytics to analyze the popularity of various parts. Using this as a starting point we selected a few candidates and visited them with the Brave browser [7]. Brave clearly indicates when tracking and analytics scripts have been blocked which allows us to easily identify their presence. To verify that the scrips were used (and not only present) by the pages the blocking was disabled and the pages were revisited to verify that the scripts were executed. To ensure that the analytics pages were free from fingerprinting the method to find fingerprinting pages described below was used in addition to the information given by Brave. A total of five pages containing analytics were selected.

Authentication Since fingerprinting can be used to identify and track users it can also be used at authentication points to protect against, e.g., account hijacking. To find pages that incorporate fingerprinting in their authentication process we selected the pages of a few banks under the assumption that banks both include authentication and have a vested interest in protecting their users. To detect the candidates that may contain fingerprinting we searched for the presence of known fingerprinting libraries by using the following simple syntactic heuristics where we looked for one of the following three features:

1. calls to `toDataURL`, which can be used for canvas fingerprinting,
2. the known pangrams that are part of widely used fingerprinting libraries `"How quickly daft jumping zebras vex."`, `"Cwm fjordbank glyphs vext quiz"`, and `"abcdefghijklmnopqrstuvwxyz"`, and
3. calls to `navigator.plugins`.

If any of these features were found, we manually analyzed the scripts of the page to determine if the page contained fingerprinting or not. Note that this heuristics does not necessarily skew the selection to pages that contain canvas or plugin fingerprinting. In our experience, pages that perform fingerprinting using one of the major fingerprinting libraries do so by configuring the library without modifying it, i.e., all parts of the library remain intact on the page, potentially without ever being used. A total of five pages containing authentication were selected.

Bot detection The use of bots for, e.g., scraping prices from online stores and ticket shops is common, which has prompted web pages to try and detect these bots. There are several companies which offer protection against bots, such as DataDome [12], PerimeterX [28], and Imperva (previously Distil Networks) [20]. In order to find pages that do bot detection, we created a specialized crawler that visits web pages, waits for 30 seconds, and then takes a screenshot. The candidate pages to be visited were collected in three ways. First, companies that offer bot detection often also mention some pages that use their services. We included a selection of promoted pages as candidates. Second, bot detection is often used on booking pages. For this reason, we collected a list of large airline companies and included them as candidates. Third, we included the web pages mentioned by Jonker et al. in their paper on the detection of bot detection [64] as candidates. Each of these pages was visited by the screenshot crawler, and the screenshots were collected and analyzed for bot detection. From the pages where the screenshot indicated potential bot detection five pages using different bot detection mechanisms were manually selected.

Fingerprinting To find web pages that perform fingerprinting which does not match any of the previous categories, we used two different approaches. First, we used information from a crawl of Alexa top 100,000 performed with OpenWPM [1], which recorded blocked fingerprinting resources based on the Disconnect list [13]. We also visited pages in the Alexa top 1,000 and did the same analysis as for authentication pages. Based on these results, we visited a random set of web pages which had fingerprinting resources and ensured these resources were executed. From this set, we picked a total of ten pages.

4.1 Experiment setup

To visit the different web pages, we used a crawler implemented with Puppeteer [32] to control EssentialFP. In order to decrease the overhead of collecting the labels, we used the baseline API imprint created by running three open-source fingerprinting libraries: FingerprintJS [16], ImprintJS [21], and ClientJS [9]. Both FingerprintJS and ImprintJS are easily customizable allowing the user of the libraries to select what features should be used for the fingerprinting. For the full baseline API imprint, we used all available features of FingerprintJS and ImprintJS, while for ClientJS we used the `getFingerprint` function. To do this, we created three different web pages that we visited using the crawler to collect the API imprint of each library. Those API imprints were then combined to form the baseline API imprint. In addition to creating the baseline, we also created API imprints for each

Table 3.1: Table showing the sinks used by web pages to exfiltrate the (potential) fingerprint value. Only sinks that can be used to send data (e.g. `document.cookie`, network requests etc.) is shown. The entries are sorted by category (analytics, authentication, bot detection, and fingerprinting).

Domain	Sinks
microsoft.com	`navigator.sendBeacon`
alexa.com	`Image.src`
stackoverflow.com	`navigator.sendBeacon`
washingtonpost.com	`navigator.sendBeacon`
zoom.us	`HTMLFormElement.setAttribute`
zionsbank.com	`XMLHttpRequest.open`
pnc.com	`global.HTMLFormElement.className`, `document.createElement.src`
citibank.com	`XMLHttpRequest.send`
cit.com	`HTMLFormElement.setAttribute`
credit-suisse.com	`HTMLFormElement.appendChild`
brainly.com	`document.createElement.src`
skyscanner.com	`Blob`
streeteasy.com	`XMLHttpRequest.send`
frankmotorsinc.com	`XMLHttpRequest.send`
lufthansa.com	`XMLHttpRequest.send`
ultimate-guitar.com	`document.createElement.src`, `navigator.sendBeacon`
aktuality.sk	`document.createElement.src`, `XMLHttpRequest.open`
lg.com	`HTMLFormElement.setAttribute`
olx.ua	`document.cookie`
scribd.com	`document.createElement.src`
rezka.ag	`HTMLFormElement.setAttribute`
kinoprofi.vip	`HTMLFormElement.setAttribute`
jd.com	`XMLHttpRequest.send`
sciencedirect.com	`document.cookie`
rei.com	`document.cookie`

individual feature of FingerprintJS and ImprintJS. Those API imprints were used when analyzing the kind of fingerprinting detected.

Using the baseline, each of the selected pages was visited by the crawler for up to 12 hours to ensure that the fingerprinting script was not prevented from running due to performance issues.

4.2 API endpoints

Each of the selected web pages was visited while recording created and escaping labels from the baseline API imprint. For each page and each API endpoint, every label reaching the endpoint was collected in addition to collecting all created labels for each page and script. This allows us to infer what parts of the API imprint did flow to network sinks as well as if the application accumulated fingerprinting information internally.

To define potential network sinks we analyzed the used API endpoints to identify uses that could instigate a network or storage request (which would or could be used to send the information later). For this experiment, the identified network sinks were:

- `navigator.sendBeacon`
- `XMLHttpRequest`
- `fetch`
- `window.postMessage`
- setting `src` attributes of HTML elements such as images and scripts
- setting attributes on `HTMLFormElement`
- `document.cookie`
- `window.localStorage` and `window.sessionStorage`

In addition to this, we also treat the `Blob` class as a network sink, since on one page the data was transmitted using a `Blob` object, a flow that would otherwise have been missed by the current label model of EssentialFP. This is not a limitation of the approach, but rather only the current label model, which does not fully track flows via mediated parts of the API. As expected, the number of flows via the mediated API is small (this was the only occurrence) and developing a full flow model for the Chromium execution environment is too large an undertaking to be presently justifiable.

4.3 Analysis

The complete overlap for the sinks against the combined fingerprinting baseline from FingerprintJS, ImprintJS, and ClientJS for each web page can be found in Figure 3.1 and the exfiltration method for each web page can be seen in Table 3.1.

A key takeaway is that our results confirm our intuition: pages that access a wide surface of sensitive APIs and send consolidated information to the

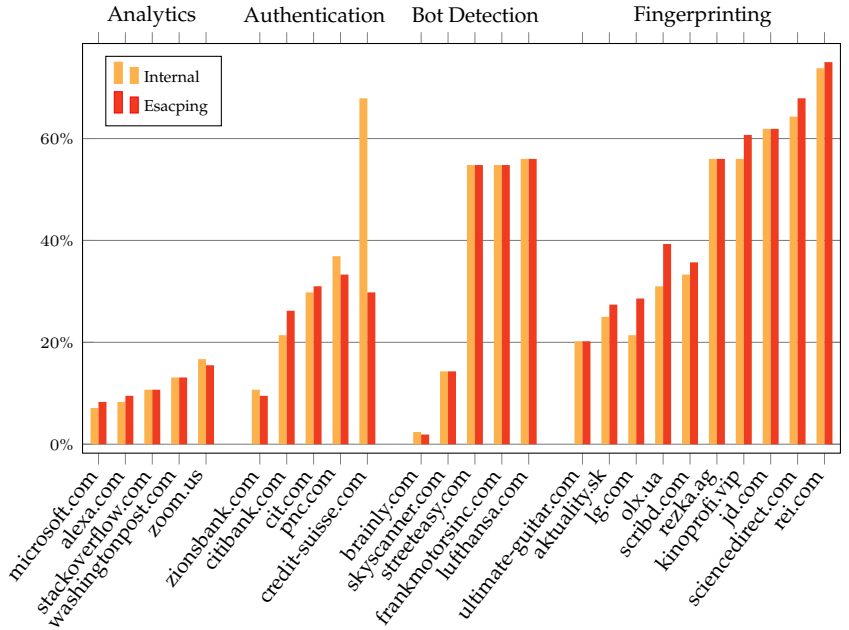


Figure 3.1: Breakdown of the maximum overlap for each web page against the baseline API imprint consisting of the libraries FingerprintJS, ImprintJS, and ClientJS. The web pages are sorted by category, as well as within each category based on the escaping label. The y-axis is the percentage overlap for a web page with the baseline API imprint.

network represent the essence of fingerprinting and are clearly marked as such by their high overlaps with the baseline.

Further, the majority of the visited web pages send sensitive information via network sink, as can be seen in Table 3.1. This is an indication that (partial) compounded data is being sent to an external server. As expected, this occurs on pages belonging to all categories. The difference is the amount of sensitive information being transmitted. This is a strong indicator that it does not suffice to look at the API imprint of the application and whether the application uses the network or not. To detect fingerprinting we must track what information reaches the network sink.

We can also see that the majority of the web pages have close to equivalent internal fingerprints and labels that reach the network sink, which indicates that current fingerprinting scrips accumulate and compute a fingerprint before transmitting it and that piece-by-piece fingerprinting, where the fingerprint is being sent gradually to an external server, is not as common.

When looking at the different categories, an interesting picture emerges.

First, we can see there is a potential cut-off that allows us to distinguish between the analytics category and more intrusive fingerprinting in the fingerprinting category. A larger than 20% overlap with the full baseline indicates the presence of fingerprinting. Indeed, the *maximum* matching overlap for the analytics category came from zoom.us, with an overlap of 15.5%. This can be compared with the *minimum* matching overlap for the fingerprinting category, which was from ultimate-guitar.com with 20.2%.

For the other categories, the situation is more subtle because the extent of fingerprinting is different in authentication and bot detection web sites. This is not surprising since authentication and bot detection have different goals compared to analytics and actual fingerprinting. Another interesting result is the result for brainly.com, which uses DataDome for bot detection. From the small API imprint, it is clear that DataDome did not have to perform any massive fingerprinting in order to distinguish our crawler from a human user. The reason for this is likely the fact that we do not attempt to hide that we are a crawler, as using stealth libraries for Puppeteer (e.g. puppeteer-extra-plugin-stealth [33]) can make properties used for fingerprinting non-accessible, making the amount of fingerprinting information less.

Aside from the pages in the fingerprinting category using cookies as the exfiltration method, there is no real distinction between the exfiltration methods used in the different categories. However, when looking at credit-suisse.com, we can see a large drop between the overlap of the largest internally created label and the exfiltrated label. After analyzing the source code of the web page it is clear that the largest internal label comes from performing fingerprinting, which writes the result to a global variable called `fp2murmur`. This global variable is then never used, which may indicate the result from that specific fingerprinting method is not used. However, we could also see that several fingerprinting attributes are being written gradually to a form element, which is the 29.8% overlap which can be seen in Figure 3.1.

To further analyze the four categories, we have created heatmaps for the different features of FingerprintJS. The heatmaps are found in Figure 3.2, where each row shows the conditional probabilities of the features of each column given the feature of the row interpreted as a gradient between yellow (0% probability) and red (100% probability). Thus, the diagonal shows which features were used by the pages in the category. Four distinct patterns for the different categories emerge. Looking at the heatmaps we see that pages from the analytics category (Figure 3.2a), are not that intrusive when it comes to data collection, while the pages from the authentication category (Figure 3.2b) and the bot detection category (Figure 3.2c) show an increased intensity in the use of fingerprinting features. Finally, Figure 3.2d clearly

shows that the more general, traditional fingerprinting is the most intrusive. Relatively many features are used, which supports the success in detecting fingerprinting by looking at the overlap between the baseline API imprint and the escaped information.

Interestingly enough, pages in the bot detection category use `enumerateDevices`, which probes the list of all available media input and output devices. It is worth to point out that `enumerateDevices` is disabled by default in FingerprintJS, which may explain why it is not used in the traditional fingerprinting category. Similarly, both the authentication and the bot detection categories use the `webdriver` feature, which checks for `navigator.webdriver`. This property indicates if the user agent is controlled by an automatic tool, such as Puppeteer and Selenium, which is logical to check for by these pages.

4.4 Remarks on performance

JSFlow is an information flow aware JavaScript interpreter written in JavaScript. The main goal of JSFlow is to provide a platform for experimentation with various forms of dynamic IFC in a JavaScript setting and as such is not written with performance in mind. On the contrary, in order to ensure correct behavior JSFlow follows the ECMA-262 version 5 standard very closely. Naturally, if one compares JSFlow executing a script on top of V8 to the script running natively in V8 the performance difference is significant.

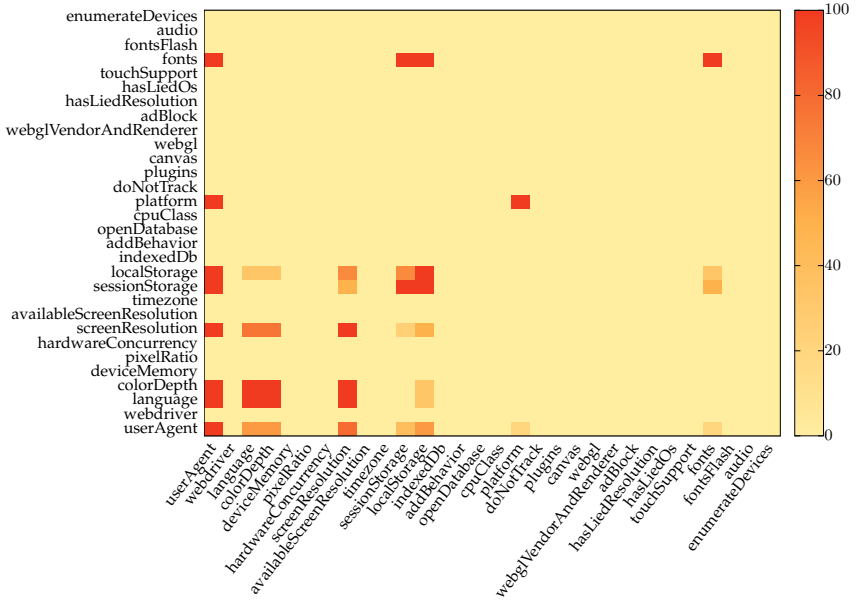
A more interesting question is how large portion of the execution time is due to information flow tracking. While this is not entirely easy to measure, since there is no way to turn off information flow tracking in JSFlow, most of the tracking passes through a limited number of functions (e.g., for joining and comparing labels). Profiling those functions indicates that the labeling incurs a runtime cost of around 5% on the FingerprintJS page used to create the baseline API imprint.

While the 5% overhead might be a reasonable starting point for optimizing a V8-based implementation, this direction is outside the scope of our work. Instead, we have focused on a proof of concept that demonstrates that observable tracking can be leveraged to effectively track fingerprinting patterns. As such, EssentialFP provides a rich platform for experiments and security testing of real-world web pages with dynamic IFC.

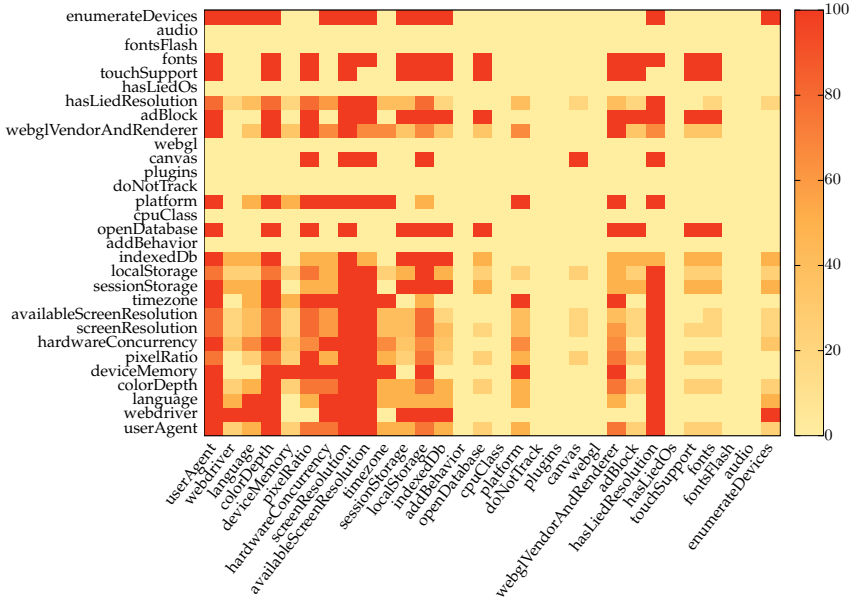
5 Discussion

While our initial experiments show that it is possible to detect fingerprinting by tracking how information flows from a baseline API imprint to network sinks, our work opens up a few interesting avenues of future work.

(a) Analytics



(b) Authentication



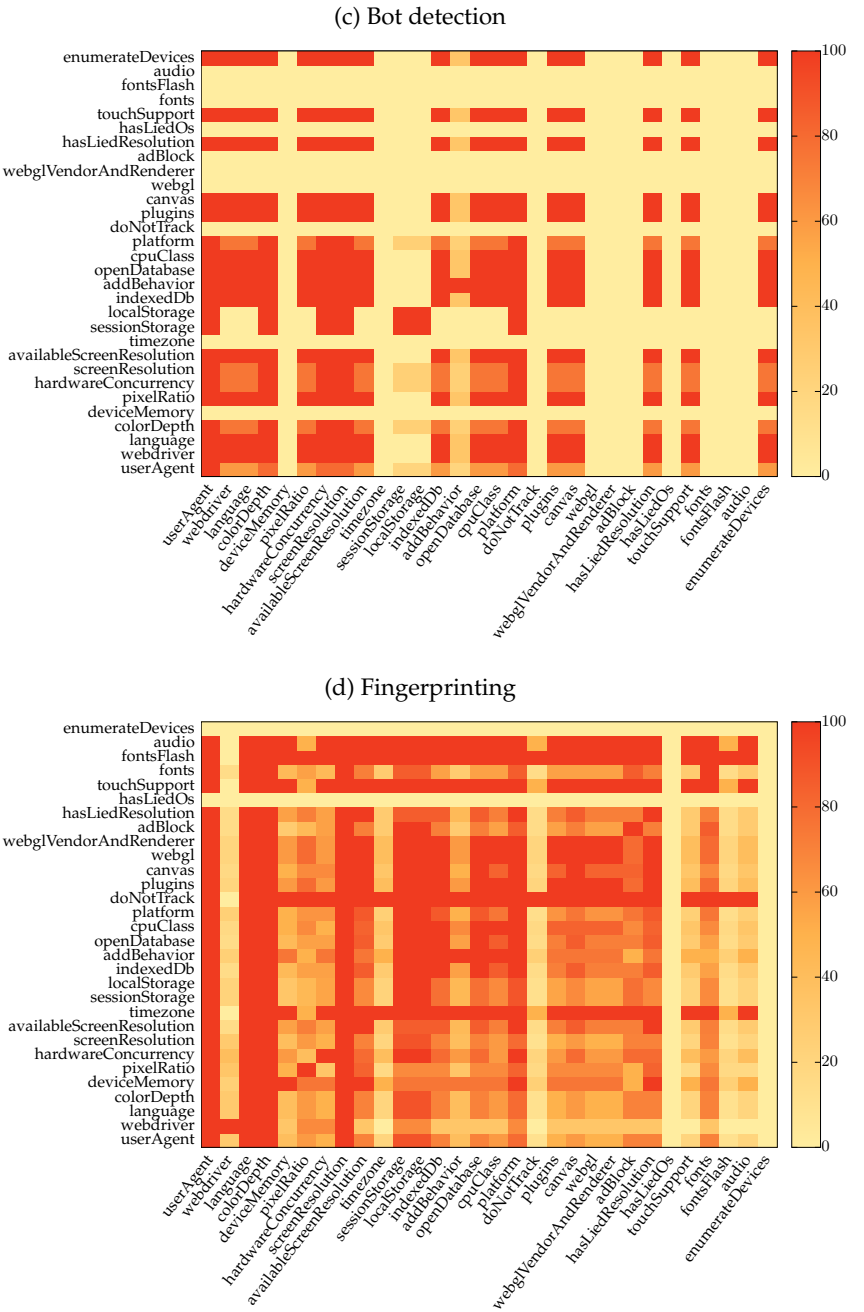


Figure 3.2: Heatmaps for the different categories. For each row the column identifies the conditional probability of the feature of the column given the feature of the row. The probability is interpreted as a gradient between yellow (0% probability) and red (100% probability).

5.1 Label models

Two of the crawled pages used mediated versions of `btoa` to base-64 encode the gathered information. This caused the labels to be lost before reaching the network sink. In this case the remedy was simple. By implementing `btoa` the flows were restored, but this points to the issue of losing labels when using functions that were automatically mediated from the V8 execution environment. An attractive goal for future explorations is to find better-suited label models for standard API interaction that more precisely track flows of information via the mediated API functionality.

5.2 Fingerprinting metrics

Our current solution compares the overlap between the baseline API imprint and the script API access pattern to detect fingerprinting. The results show that this approach is already effective. However, it would be interesting to explore if the choice of metrics can be further improved.

Weighted overlap The unweighted overlap used in this paper does not distinguish between common and uncommon features. For example, the API pattern of canvas fingerprinting is implicitly assigned the same importance as the API pattern of querying the user agent or the screen width of the browser, while the canvas fingerprinting is arguably more indicative of fingerprinting than probing the user agent or screen width. This can be remedied by assigning weights to each source and compare the weighted sum. One interesting starting point to the challenge of deciding on the relative weights could be to use the entropy reported by Panopticlick [27] to generate the weights for each API pattern in the fingerprinting library.

Conditional overlap The weighted overlap assigns more importance to features that are more indicative of fingerprinting, but does not take that combinations of features may be rarer than others, as indicated by our heatmaps, into account. Thus, such combinations should probably be given more weight than the sum of their parts. The conditional probability computed to generate the heatmaps could be a good starting point to finding clusters that identify the various categories.

6 Related work

Much work has been done, regarding both conducting and combating device fingerprinting. This section aims to provide an overview of both categories. For a full description, we refer the reader to a timely survey by Laperdrix et al. [69].

Browser fingerprinting While device fingerprinting has been a known problem for a long time, Mayer noticed a browser could present "quirkiness" which originated from the operating system, the hardware, and the browser configuration [72]. He showed that 96.23% of 1,328 users could be uniquely identified just by looking at `navigator`, `screen`, `navigator.plugins`, and `navigator.mimeTypes`. Eckersley was the first to present how browser fingerprinting could be effective by querying for standard browser features in a large-scale study [53], showing 83.6% of the 470,161 fingerprints were unique. This number rose to 94.2% if Flash or Java were enabled. However, a more recent study by Gómez-Boix et al. [57] showed the amount of uniquely identifiable users have gone down, but that a non-unique fingerprint is probable to become unique if some features change. Since the study of Eckersley, new technology has been added to the browser, and with that new ways of fingerprinting browsers have emerged. Mowery and Shacham showed how the HTML5 canvas feature and the WebGL graphics API could be used to generate stable fingerprints [73]. Cao et al. designed a fingerprinting technique that relied on WebGL, uniquely identifying 99.24% of the 1,903 tested devices [50]. Mulazanni et al. showed how differences in the JavaScript execution engine could be used to fingerprint users based on a test suite on the EcmaScript standard [74], and Nikiforakis et al. [76] showed differences in the `navigator` and `screen` objects can distinguish between different browser families, as well as major and minor versions within the same family. Sanchez-Rola et al. exploited API functions in the HTML Cryptography API in Chromium which, when called, will invoke native functions, allowing to time the internal clock signals of the CPU [79]. Although fingerprinting is privacy-intrusive, research has been made on how it can strengthen security. Alaca and van Oorschot [46] explored device fingerprinting for augmenting authentication, and Laperdrix et al. proposed the use of canvas fingerprinting to strengthen web authentication via a challenge-response mechanism [67]. On a similar vein, Jonker et al. showed that, by using the characterization of the fingerprint surface of 14 web bots, a vast majority of them can be uniquely identifiable through well-known fingerprinting techniques [64]. Amin Azad et al. showed that relying on fingerprinting could be enough to defend web pages against basic bots, but currently fail for less popular browsers in term for automation [47].

Acar et al. conducted a large scale study over browser fingerprinting, showing that 5.5% of the crawled Alexa top 100,000 web pages conducted canvas fingerprinting [44]. Repeating a similar experiment in 2016, Englehardt and Narayanan [54] conducted a large-scale analysis on fingerprinting on Alexa top 1 Million web pages to see the long tail of online tracking. They found that 1.6% of the crawled web pages used canvas fingerprinting,

but they also discovered three techniques not measured before: AudioContext fingerprinting, Canvas-Font fingerprinting, and WebRTC fingerprinting. Lastly, Vastel et al. [89] analyzed the evolution of fingerprints to link fingerprints belonging to the same device over time, showing they could track a user on average 51.8 days, and 26% of devices over 100 days.

Browser extensions There have also been experiments of how browser extensions can be used to enhance fingerprinting. Several studies have shown different methods of discovering browser extensions, ranging from searching for resources in the extension accessible by the web page [81], how they can be used to track users [80], and how one can time the accessing to detect every installed browser extension [78]. Starov and Nikiforakis created XHOUND, showing browser extensions could be uniquely identifiable based on *Document Object Model (DOM)* modifications [85]. They found that 9.2% of the top 10,000 extensions (and 13.2% of the top 1000 extensions) can be uniquely identifiable on any domain based on the DOM modifications alone. These numbers went up to 16.6% for the top 10,000 extensions and 23% for the top 1000 extensions when a domain from the Alexa top 50 was visited. Looking at how browser extensions made page modifications deemed unnecessary for the extension's functionality, Starov et al. [84] showed that 5.7% out of 58,304 extensions were identifiable due to this unnecessary bloat. Gulyás et al. combined browser fingerprinting with additional data of installed extensions and web logins [58]. They conclude 54.86% of users with at least one detectable extension is unique, 19.53% of users with at least one detectable login is unique, and 89.23% of users are unique if they have at least one detectable extension and one detectable login.

Combating fingerprinting In order to prevent browser fingerprinting, several different approaches have been proposed. Browser extensions can be used to randomize, e.g., the user agent, but as Nikiforakis et al. showed [76], this can create inconsistencies between the user agent and other publicly available API (e.g. `navigator.platform`). Similarly, Vastel et al. developed FP-SCANNER [88], a test-suite that explores browser fingerprint inconsistencies to detect potential alterations done by fingerprinting countermeasures tools. They demonstrated FP-SCANNER could not only find these inconsistencies but could also reveal the original values, which in turn could be exploited by fingerprinters to more accurately target browsers with fingerprinting countermeasures.

Instead, as one key aim when conducting device fingerprinting is to link the newly generated fingerprint to an old one, work has been focused on breaking the linkability between different sessions. Both Laperdrix et al. [70] and Gómez-Boix et al. [56] proposed virtualization and modular

architectures to randomly assemble a coherent set of components whenever a user wanted to browse the web. This would break the linkability while not having any inconsistencies between attributes, but the user comfort may go down.

Besson et al. [49] formalized a privacy enforcement based on a randomization defense using quantitative information-flow showing how to synthesize a randomization mechanism that defines the configurations for each user. They found that more efficient privacy enforcement often lead to lower usability, i.e., users have to switch to other configurations often. Ferreira Torres et al. [86] proposed generating unique fingerprints to be used on each visited web page, making it more difficult for third parties to track the same user over multiple web pages. FaizKhademi et al. [55] proposed the detection of fingerprinting by monitoring and recording the activities by a web page from the time it started loading. Based on the recording, they were able to extract metrics related to fingerprinting methods to build a signature of the web page to distinguish normal web pages from fingerprinting web pages. If the web page is deemed to be fingerprinting, the access to the browser is limited e.g. by limiting the number of fonts allowed to be enumerated, by adding randomness to attribute values of the `navigator` and `screen` objects, and noise to canvas images that are generated.

Nikiforakis et al. [75] proposed using randomization policies, which are protection strategies that can be activated when certain criteria are met. In particular, they added randomization to offset measurements of HTML elements (which is used when doing e.g. font enumeration) and plugin enumeration. Similarly, Laperdrix et al. [68] proposed adding randomness to some more complex parts of the DOM API: canvas, web audio API, and the order of JavaScript object properties. Randomization has also been used to combat fingerprinting via browser extensions by Trickel et al. [87] that proposed randomizing paths to web-accessible resources to prevent probing attacks, changing ID and class names which are injected to change the behavioral fingerprint.

Adding randomness to mitigate canvas, WebGL, and AudioContext fingerprinting has now been implemented by the Brave browser [41], adding to their already implemented fingerprinting protections [15]. Other browser vendors are also implementing anti-fingerprinting measures. Firefox introduced Enhanced Tracking Protection [23], which would allow third-party cookies to be blocked. This has later been expanded to also block all third-party requests to companies that are known to participate in fingerprinting [17]; a feature that is also found in Microsoft Edge [24]. Safari applies similar restrictions on cookies as Firefox, and also presents a simplified version of the system configuration to trackers, making more devices look

identical [35]. The Tor browser aims to make all users look identical to resist fingerprinting [38]. Unfortunately, this means that as soon as a user maximizes the browser window or installs a plugin, their fingerprint will divert from the unified Tor browser fingerprint [66]. Similarly, as all Tor browsers aim to look identical, Khattak et al. showed they can be a target for blocking, showing 3.67% of the Alexa top 1,000 pages blocked access to Tor users [65]. Lastly, of the well-known browser vendors, Chrome has announced "The Privacy Sandbox" [37], where they are planing on combat fingerprinting by implementing a privacy budget [29].

7 Conclusion

We have presented EssentialFP, a principled approach to detecting fingerprinting on the web. Coming back to RQ1 on the essence of fingerprinting: EssentialFP identifies the essence of browser fingerprinting by the following pattern: (i) gathering information from a wide browser API surface (multiple browser-specific sources) and (ii) communicating the information to the network (network sink) captures the essence of fingerprinting. This pattern enables us to clearly distinguish fingerprinting from similar types of scripts like analytics and polyfills. Coming back to RQ2 on exposing fingerprinting: EssentialFP exposes the above pattern by monitoring based on observable information flow tracking. To implement EssentialFP we have leveraged, extended, and deployed in a browser JSFlow, a state-of-the-art information flow tracker for JavaScript. We have demonstrated EssentialFP's effectiveness to spot fingerprinting on the web by evaluating it on several categories of web pages: analytics, authentication, bot detection, and fingerprinting-enhanced web pages from the Alexa list. Our results reveal different extent of fingerprinting in these categories: from no evidence of fingerprinting in the analytics pages to evidence of fingerprinting in some of the authentication and bot detection pages and to full-blown evidence in fingerprinting-enhanced pages from the Alexa list.

8 Bibliography

[1] OpenWPM. `https://github.com/mozilla/OpenWPM`, accessed: Jan 2020.

[2] `https://github.com/disconnectme/disconnect-tracking-protection/issues`, accessed: May 2020.

[3] `https://forums.lanik.us/viewforum.php?f=64&sid=3d7d9fed66ba36b96c4b18f3142d0e43`, accessed: May 2020.

[4] `https://github.com/easylist/easylist/issues`, accessed: May 2020.

[5] https://github.com/brave/brave-browser/issues/10000, accessed: May 2020.

[6] Babel. https://babeljs.io/, accessed: May-2020.

[7] Brave Browser. https://brave.com/, accessed: May-2020.

[8] Chromium. https://www.chromium.org/Home, accessed: May 2020.

[9] ClientJS. https://clientjs.org/, accessed: May 2020.

[10] Combating Fingerprinting with a Privacy Budget. https://github.com/bslassey/privacy-budget, accessed: May-2020.

[11] core-js. https://www.npmjs.com/package/core-js, accessed: May-2020.

[12] Datadome. https://datadome.co/, accessed: May 2020.

[13] Disconnect. https://github.com/disconnectme/disconnect-tracking-protection, accessed: May-2020.

[14] EssentialFP code and benchmarks. https://drive.google.com/drive/folders/1Y2YhZAEUMbtAUXMx-c1Y9lwtcK9BZFNU?usp=sharing, accessed: May-2020.

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

[16] FingerprintJS. https://fingerprintjs.com/, accessed: May 2020.

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

[18] General Data Protection Regulation GDPR. https://gdpr-info.eu/, accessed: May 2020.

[19] Ghostery. https://www.ghostery.com/, accessed: May 2020.

[20] Imperva. https://www.imperva.com/, accessed: May 2020.

[21] ImprintJS. https://github.com/mattbrailsford/imprintjs, accessed: May 2020.

[22] Jsflow. http://www.jsflow.net/, accessed: May-2020.

[23] Latest Firefox Rolls Out Enhanced Tracking Protection. https://blog.mozilla.org/blog/2018/10/23/latest-firefox-rolls-out-enhanced-tracking-protection/, accessed: May 2020.

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

[25] Mitigating Browser Fingerprinting in Web Specifications. https://w3c.github.io/fingerprinting-guidance/, accessed: May 2020.

[26] Modernizr. https://modernizr.com/, accessed: May 2020.

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

[28] PerimeterX. https://www.perimeterx.com, accessed: May 2020.

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

[30] Privacy Badger. https://privacybadger.org/, accessed: May 2020.

[31] Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy, accessed: May 2020.

[32] Puppeteer. https://pptr.dev/, accessed: May 2020.

[33] puppeteer-extra-plugin-stealth. https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth, accessed: May 2020.

[34] regenerator-runtime. https://www.npmjs.com/package/regenerator-runtime, accessed: May-2020.

[35] Safari Privacy Overview. https://www.apple.com/safari/docs/Safari_White_Paper_Nov_2019.pdf, accessed: May-2020.

[36] Standard ECMA-262 6th Edition / June 2015. https://www.ecma-international.org/ecma-262/6.0/, accessed: May 2020.

[37] The Privacy Sandbox. https://www.chromium.org/Home/chromium-privacy/privacy-sandbox, accessed: May 2020.

[38] Tor. https://www.torproject.org/, accessed: May 2020.

[39] Tracking Preference Expression (DNT). https://www.w3.org/TR/tracking-dnt/, accessed: May 2020.

[40] Unsanctioned Web Tracking. https://w3ctag.github.io/unsanctioned-tracking/, accessed: May 2020.

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

[42] window-crypto. https://www.npmjs.com/package/window-crypto, accessed: May-2020.

[43] Window.crypto. https://developer.mozilla.org/en-US/docs/Web/API/Window/crypto, accessed: May 2020.

[44] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 674–689, 2014.

[45] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel. FPDetective: dusting the web for fingerprinters. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1129–1140, 2013.

[46] F. Alaca and P. C. van Oorschot. Device fingerprinting for augmenting web authentication: classification and analysis of methods. In *ACSAC*, pages 289–301. ACM, 2016.

[47] B. A. Azad, O. Starov, P. Laperdrix, and N. Nikiforakis. Web Runner 2049: Evaluating Third-Party Anti-bot Services. In *17th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2020), Lisboa, Portugal, June 24-26, 2020*, 2020.

[48] M. Balliu, D. Schoepe, and A. Sabelfeld. We are family: Relating information-flow trackers. In *ESORICS*, volume 10492 of *Lecture Notes in Computer Science*, pages 124–145. Springer, 2017.

[49] F. Besson, N. Bielova, and T. P. Jensen. Browser randomisation against fingerprinting: A quantitative information flow approach. In *Secure IT Systems - 19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, pages 181–196, 2014.

[50] Y. Cao, S. Li, and E. Wijmans. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[51] C. Cimpanu. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/, accessed: May 2020.

[52] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Commun. ACM*, 20(7):504–513, 1977.

[53] P. Eckersley. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, pages 1–18, 2010.

[54] S. Englehardt and A. Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401, 2016.

[55] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. Fpguard: Detection and prevention of browser fingerprinting. In *Data and Applications Security and Privacy XXIX - 29th Annual IFIP WG 11.3 Working Conference, DBSec 2015, Fairfax, VA, USA, July 13-15, 2015, Proceedings*, pages 293–308, 2015.

[56] A. Gómez-Boix, D. Frey, Y. Bromberg, and B. Baudry. A Collaborative Strategy for Mitigating Tracking through Browser Fingerprinting. In *Proceedings of the 6th ACM Workshop on Moving Target Defense, MTD@CCS 2019, London, UK, November 11, 2019*, pages 67–78, 2019.

[57] A. Gómez-Boix, P. Laperdrix, and B. Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*, pages 309–318, 2018.

[58] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia. To extend or not to extend: On the uniqueness of browser extensions and web logins. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 14–27, 2018.

[59] D. Hedin, L. Bello, and A. Sabelfeld. Information-flow security for javascript and its apis. *Journal of Computer Security*, 24(2):181–234, 2016.

[60] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: tracking information flow in javascript and its apis. In *Symposium on Applied*

Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014, pages 1663–1671, 2014.

[61] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 3–18, 2012.

[62] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Software Safety and Security*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[63] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *ACM Conference on Computer and Communications Security*, pages 270–283. ACM, 2010.

[64] H. Jonker, B. Krumnow, and G. Vlot. Fingerprint Surface-Based Detection of Web Bot Detectors. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, pages 586–605, 2019.

[65] S. Khattak, D. Fifield, S. Afroz, M. Javed, S. Sundaresan, D. McCoy, V. Paxson, and S. J. Murdoch. Do You See What I See? Differential Treatment of Anonymous Users. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[66] P. Laperdrix. Browser Fingerprinting: An Introduction and the Challenges Ahead. `https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead`, accessed: May 2020.

[67] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, pages 43–66, 2019.

[68] P. Laperdrix, B. Baudry, and V. Mishra. FPRandom: Randomizing Core Browser Objects to Break Advanced Device Fingerprinting Techniques. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, pages 97–114, 2017.

[69] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. Browser fingerprinting: A survey. *ACM Trans. Web*, 14(2):8:1–8:33, 2020.

[70] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating Browser Fingerprint Tracking: Multi-level Reconfiguration and Diversification. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pages 98–108, 2015.

[71] J. Magazinius, D. Hedin, and A. Sabelfeld. Architectures for inlining security monitors in web applications. In J. Jürjens, F. Piessens, and N. Bielova, editors, *Engineering Secure Software and Systems*, pages 141–160, Cham, 2014. Springer International Publishing.

[72] J. R. Mayer. Any person... a pamphleteer": Internet Anonymity in the Age of Web 2.0. *Undergraduate Senior Thesis, Princeton University*, page 85, 2009.

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

[74] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, and E. Weippl. Fast and Reliable Browser Identification with JavaScript Engine Fingerprinting. In *Web 2.0 Workshop on Security and Privacy (W2SP), Vol. 5*, 2013.

[75] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 820–830, 2015.

[76] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 541–555, 2013.

[77] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003.

[78] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Extension breakdown: Security analysis of browsers extension resources control policies. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 679–694, 2017.

[79] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Clock around the clock: Time-based device fingerprinting. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1502–1514, 2018.

[80] A. Sjösten, S. V. Acker, P. Picazo-Sanchez, and A. Sabelfeld. Latex gloves: Protecting browser extensions from probing and revelation attacks. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019.

[81] A. Sjösten, S. V. Acker, and A. Sabelfeld. Discovering browser extensions via web accessible resources. In *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy, CODASPY 2017, Scottsdale, AZ, USA, March 22-24, 2017*, pages 329–336, 2017.

[82] A. Sjösten, D. Hedin, and A. Sabelfeld. Information flow tracking for side-effectful libraries. In C. Baier and L. Caires, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 38th IFIP WG 6.1 International Conference, FORTE 2018, Held as Part of the 13th International Federated Conference on Distributed Computing Techniques, DisCoTec 2018, Madrid, Spain, June 18-21, 2018, Proceedings*, volume 10854 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2018.

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

[84] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis. Unnecessarily identifiable: Quantifying the fingerprintability of browser extensions due to bloat. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 3244–3250, 2019.

[85] O. Starov and N. Nikiforakis. XHOUND: quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 941–956, 2017.

[86] C. F. Torres, H. L. Jonker, and S. Mauw. Fp-block: Usable web privacy by controlling browser fingerprinting. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 3–19, 2015.

[87] E. Trickel, O. Starov, A. Kapravelos, N. Nikiforakis, and A. Doupé. Everyone is Different: Client-side Diversification for Defending Against Extension Fingerprinting. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1679–1696, 2019.

[88] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. FP-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In

27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018, pages 135–150, 2018.

[89] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. FP-STALKER: Tracking Browser Fingerprint Evolutions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 728–741, 2018.

[90] T. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host fingerprinting and tracking on the web: Privacy and security implications. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

Paper IV

Discovering Browser Extensions via Web Accessible Resources

Alexander Sjösten, Steven Van Acker, Andrei Sabelfeld

Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, March 2017

Abstract

Browser extensions provide a powerful platform to enrich browsing experience. At the same time, they raise important security questions. From the point of view of a website, some browser extensions are invasive, removing intended features and adding unintended ones, e.g. extensions that hijack Facebook likes. Conversely, from the point of view of extensions, some websites are invasive, e.g. websites that bypass ad blockers. Motivated by security goals at clash, this paper explores browser extension discovery, through a non-behavioral technique, based on detecting extensions' web accessible resources. We report on an empirical study with free Chrome and Firefox extensions, being able to detect over 50% of the top 1,000 free Chrome extensions, including popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, and Ghostery. We also conduct an empirical study of non-behavioral extension detection on the Alexa top 100,000 websites. We present the dual measures of making extension detection easier in the interest of websites and making extension detection more difficult in the interest of extensions. Finally, we discuss a browser architecture that allows a user to take control in arbitrating the conflicting security goals.

1 Introduction

Browser extensions provide a powerful platform to enrich browsing experience. The Chrome web store currently contains around 43,000 free extensions, with many of these extensions, such as AdBlock, Adobe Acrobat, and Skype, having more than 10,000,000 users.

From the security point of view, browser extensions are deployed as a "man in the browser" [30], implying that extensions have privileges to arbitrarily alter the behavior of webpages. Naturally, the power of browser extensions creates tension between the security goals of the webpages and those of the extensions themselves. Let us consider some representative scenarios to illustrate the challenges in balancing these goals.

The first and second scenarios present an exclusive point of view of websites, concerned with malicious extensions. The third scenario presents an exclusive view of extensions, concerned with malicious websites. The fourth scenario illustrates legitimate synergies between websites and extensions. Finally, the fifth scenario illustrates the security goals of websites and extensions at outright clash.

Bank scenario Bank webpages manipulate sensitive information whose unauthorized access may lead to financial losses. It is desirable to detect potentially insecure and vulnerable extensions and prevent extensions from injecting third-party scripts into the bank's webpages. The latter technique

is in fact a common practice for many extensions [31, 35]. This scenario motivates the goal of discovering browser extensions, as the knowledge of what extensions run on the webpage can be used for tuning the defense.

Facebook scenario With over a billion daily users [18], Facebook is a popular target for attacks. Since the Facebook application itself is relatively well protected from attacks like cross-site scripting, attackers look for attacks elsewhere. A prevalent threat to user integrity and confidentiality is the use of browser extensions to inject scripts into the Facebook application to gain full access to the user's account [15]. Jagpal et al. [35] identify Facebook as the number one target for malicious extensions, reporting on the proliferation of attacks such as fake content (ad or otherwise) injection and information stealing.

This scenario motivates the need for recognizing browser extensions by webpages. Having an extension detection technique available, the webpage can adapt its behavior to the extensions installed. Research by Facebook's anti-abuse team confirms that this is a realistic scenario [15].

LastPass scenario LastPass [38] is a password manager that permits users to only remember one master password while automatically generating, storing, and filling in passwords for the individual services. The LastPass Chrome extension has currently over 4,000,000 users. Being a sensitive extension, LastPass has been subject to attacks. For example, LostPass [39] is a "pixel-perfect phishing" attack that exploits the fact that LastPass displays its notification in the browser viewport. LostPass fakes a message of an expired session and redirects users to a fake login page where it harvests the master password. (LastPass subsequently responded by interface measures and asking for email confirmation for all logins from new IPs [37].)

This scenario motivates the need to protect sensitive extensions. Being able to detect LastPass is a trigger for phishing attacks via a malicious webpage, as in the case of LostPass. It is in the interest of LastPass to stay undetected. Similar scenarios arise with extensions such as Avast Online Security and Ghostery, popular security- and privacy-critical extensions that can be targeted by malicious websites.

Google Cast scenario Google Cast [29] is a popular extension to play content on a Chromecast device from Chrome. Upon detecting the Google Cast extension, websites like Twitch.tv adjust their functionality and offer richer features.

This scenario highlights the benefit of browser extension detection, as motivated by enriching functionality rather than by security considerations.

AdBlock scenario With over 40,000,000 users, AdBlock is currently the most popular Chrome extension [12]. It is in the very nature of ad blocking

to modify webpages, looking for ads and blocking them. These goals are clearly at odds with the webpages' goals. Consequently, some webpages try to detect ad blockers.

This scenario motivates both the need for extension detection from the point of view of webpages and the need for evading discovery from the ad blockers' point of view. As we detail in Section 2, the state of the art for this scenario is much of a cat-and-mouse game.

Security goals at clash The above scenarios demonstrate that the different stakeholders (websites vs. browser extensions) have different interests, resulting in the clash of the respective security goals. Motivated by these security goals, this paper focuses on discovering browser extensions and pursues the following research questions: (i) How to discover browser extensions from within a webpage, i.e, without modifying the browser? and (ii) How can extensions evade detection?

We emphasize that this paper does not assume the interest of webpages over the interest of extensions or vice versa. Instead, we recognize that these different interests are legitimate, even if conflicting. We seek to better understand these interests, conceptually and empirically, and suggest steps to improve the state of the art on both sides.

Non-behavioral extension discovery We refer as *behavioral* to extension discovery techniques that require analyzing the behavior of browser extensions. Behavioral detection is sometimes desirable, when a particular behavior needs to be detected, regardless of what extension triggers it. On the other hand, *non-behavioral* discovery detects extensions without having to analyze their behavior. Non-behavioral detection is attractive when it can be done with low efforts. This motivates our focus on non-behavioral techniques.

In similar vein, when we consider measures against extension discovery, our goal is to stop non-behavioral detection and force attackers to do behavioral analysis of extensions.

Discovery via web accessible resources We explore a non-behavioral technique for discovering extensions, based on so called *web accessible resources* and implement it for detecting Chrome and Firefox extensions. Web accessible resources are the resources accessible in the context of a webpage. These resources enable interaction of extensions with the user via the underlying webpages.

While there are other, more elaborate, ways to set up this kind of interaction without web accessible resources (see Sections 3.2 and 6.2), web accessible resources provide a straightforward mechanism of direct access via URIs. Indeed, as we will see later, web accessible resources are used by many popular extensions.

Our detection is precise, in the sense of no false positives, and robust, as long as extensions require web accessible resources. While behavioral techniques may mistakenly detect an extension based on a monitored behavior, our technique is based on detecting resources that are bound to unique extension ids, implying that we never report an extension that is not present.

Contributions To the best of our knowledge, this work is the first comprehensive effort on non-behavioral extension detection, putting the spotlight on a largely unexplored area and systematically studying the technique and its applicability at large scale. To this end, the paper offers the following contributions:

Precise non-behavioral extension discovery We investigate a non-behavioral extension detection technique, based on web accessible resources (Section 3). Based on unique extension ids, our detection is precise, in the sense of no false positives, and robust, as long as extensions require web accessible resources.

Empirical studies of Chrome and Firefox extensions We report on a empirical study with Chrome's free extensions where we detect over 50% of the top 1,000 free Chrome extensions, including popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, and Ghostery, and 28% of the Chrome extensions in the study overall (Section 4).

We report on a similar study with Firefox's free extensions (Section 4). Due to Firefox's lax architecture, extensions are not prevented from direct modifications to the UI of the browser. This explains the lesser need for web accessible resources in Firefox extensions and, therefore, lower discovery rates.

Demo webpage for Chrome and Firefox We provide a demo webpage [60] to demonstrate discovery of Chrome and Firefox extensions in practice. This proof-of-concept webpage lists detected extensions once a user visits the page with Chrome or Firefox. This page serves as a starting point, providing a core that can be further developed either as a standalone service or a library for inclusion into other webpages. In fact, our code is already used by INRIA's Browser Extension Experiment [34].

Empirical studies of the Alexa top 100,000 websites We conduct an empirical study of non-behavioral extension discovery on the Alexa top 100,000 websites. Our findings suggest that the technique is not widely

known, although we do discover several websites that try to find extensions for types that include fun, productivity, news, weather, search tools, developer tools, accessibility, and shopping (Section 5).

Measures We discuss two types of measures that correspond to the interests of webpages and extensions, respectively. For webpages, we discuss a solution based on extension whitelisting. For extensions, we have recommendations to restrict APIs related to web accessible resources and webpage whitelisting (Section 6). We also discuss behavioral techniques and argue that to be effective, they need to be extension-specific.

2 State-of-the-art arms race

The state of the art is best illustrated with the arms race between ad blockers and ad blocker detectors, with its rival spirit captured by the (blatantly explicit) naming of the respective libraries.

Whenever an extension manipulates the webpage's DOM, it can be discovered using behavioral analysis. For instance, a webpage can discover an ad blocker when the latter removes an ad from the webpage. Since ad blockers act as good examples of security goals at clash, the rest of this section will focus on the arms race between webpages and ad blockers. Table 4.1 summarizes the steps in this arms race.

A straightforward approach to check for ad blockers is to create a fake ad which sets a global variable and then check for that specific variable. Figure 4.1 displays a current solution [33] which works in AdBlock, AdBlock Plus and AdBlock Pro for Chrome, as well as AdBlock Plus for Firefox, where the default behavior is to block the execution of the file `showads.js`.

Such a useful behavioral technique is often prepackaged as a JavaScript library marketed for detecting ad blockers, called "anti ad blockers". One such example is F***AdBlock (*FAB*) [13], which helps the users do behavioral analysis during a user-specified time interval. If a certain (user defined) amount of negative results in a row occurs, no ad-blocking tools are deemed to be running. This means the check can be run multiple times, making it harder for ad blockers to hide their presence by delaying their interaction.

Just as there are tools designed to help detect ad blockers, there are also tools that detect anti ad blockers. The library F***F***AdBlock (*FFAB*) [41] is an anti anti ad blocker created as a response to the anti ad blocker FAB. FFAB redefines some JavaScript function objects used during FAB's execution, overriding FAB's ad blocker detection mechanism and claims no ad blockers are detected.

But just as FAB is sensitive to behavioral analysis, so is FFAB. In turn, F***F***F***AdBlock (*FFFAB*) [16], is a response to FFAB. FFAB itself is not

```
<script src="showads.js">
<script>
  if(window.canRunAds === undefined)
  {
    // Ad blocking detected
  }
</script>
```

(a) HTML part of fake ad

```
var canRunAds = true;
```

(b) showads.js (fake ad)

Figure 4.1: Ad-blocking behavioral detection

Table 4.1: Ad blocking arms race

AdBlock	Remove ads
FAB	Injects bait for AdBlock and analyzes behavior
FFAB	Exploits global property in window object set by FAB
FFFAB	Detects if FFAB has done anything, reverts the changes

careful enough when overriding FAB's code, which gives FFFAB an opportunity to detect when FAB's code has been tampered with. When FFFAB detects this manipulation, it restores the original FAB functionality.

Detection of extensions by webpages is possible if the extension somehow modifies the DOM. In addition, behavioral detection is usually cross-browser, as the same behavior will take place no matter which browser is used.

If webpages are forced into behavioral extension detection, they cannot easily determine which extension is causing the behavior, and the extension detection loses precision. If they instead find extensions using unique ids, the extension name for Firefox extensions or a 32-character textual token for Chrome extensions, the extension can be uniquely determined and the detection is exact.

As this arms race indicates, behavioral extension detection is both error-prone because it is imprecise, and costly because it requires time and effort to keep up with the latest evasion techniques. These reasons motivate the need for a more robust and cheaper technique, bringing us to the study of non-behavioral extension detection in the following sections.

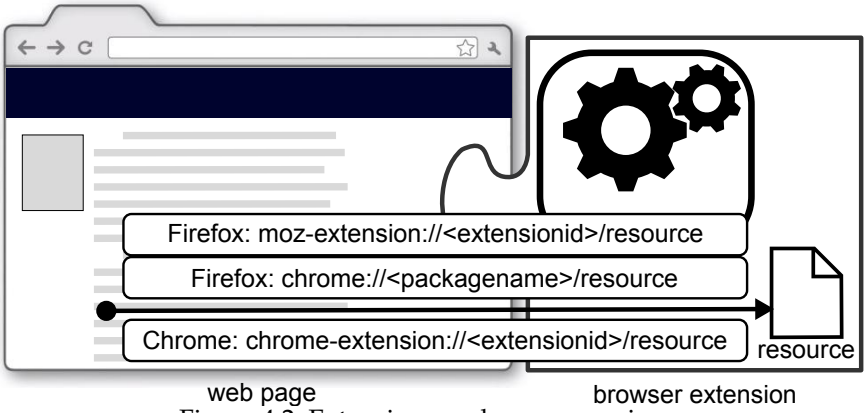


Figure 4.2: Extension - webpage overview

3 Finding extensions via web accessible resources

This section provides background on how browser extensions work in Chrome and Firefox, the role of web accessible resources, how they can be used for finding extensions and the attacker models considered in this work.

3.1 Extensions

An *extension* is a program, typically written in a combination of JavaScript, HTML and CSS to extend the browser functionality. Extensions are not to be confused with browser *plugins*, such as Flash and Java, that are compiled and loadable modules that may live outside the browsers' process space. Extensions may alter the content of a webpage (e.g. ad blockers) or add features such as executing personal scripts (e.g. Greasemonkey). Browser extensions are built using architectures defined by the browser vendors. Mozilla is currently working on *WebExtensions* [52], a new API which will have a similar structure as the Chrome extension API. Figure 4.2 depicts the architecture that connects extensions and a webpage.

Chrome extensions Chrome extensions can consist of three different parts [28]: (i) a background page, which is an invisible page containing the main logic of the extension; (ii) UI pages, ordinary HTML pages that display the extension's UI ("browser actions" [22] and "page actions" [23]); and (iii) a content script, JavaScript which executes in the context of the webpage. The content script makes the interaction with the webpage and runs in an

isolated world [24]. It has access to some Chrome APIs and can communicate with the background page using message passing [27].

Each Chrome extension must have a manifest file, `manifest.json`, which contains important information about the extension [26]. For this work, the only interesting section in the manifest file is *web_accessible_resources*, which defines which resources are accessible in the context of a webpage [25]. The content of the *web_accessible_resources* section is paths to files. They can be URLs or a path to files relative to the package root and can contain wildcards.

Firefox extensions Firefox extensions written using *WebExtensions* will have the same structure as Chrome extensions. This is because Chrome extensions should be easy to port to Firefox [50], as well as having a more unified cross-browser architecture.

For the rest of this section, we will focus on XUL/XPCOM extensions. As this is how most Firefox extensions currently are written, we will refer to them as "Firefox extensions". These extensions also uses manifest files. The extensions automatically read the file `chrome.manifest` in the extension's root [44, 47]. Differently from Chrome, manifest files in Firefox are not mandatory and one manifest file can refer to other manifest files in sub folders.

Similarly to Chrome, a content script can inject and alter content on the webpage and communicate with the background pages using message passing [46, 45]. In the file `chrome.manifest`, a flag `contentaccessible`, which when set to **yes**, makes the specified content web accessible [44].

Differently from Chrome and WebExtensions, Firefox extensions have powerful features such as `overlay`, to describe extra content to the UI [54] and `override`, to override a chrome file provided by the application [44].

3.2 Web accessible resources

Both Chrome and Firefox require that extension resources that are referenced in a regular webpage, are flagged as web accessible in the manifest files. In Chrome and WebExtensions this is done with the key *"web_accessible_resources"* [25, 51] and in Firefox extensions with *"contentaccessible=yes"* [44].

If a Chrome content script injects resources into a webpage, the resource must be flagged as web accessible. This makes the resource available using the following schema: `chrome-extension://<extensionid>/<pathToFile>`, where `<extensionid>` is a unique identifier for each extension and `<pathToFile>` is the same as the relative URL from the package root [28].

Similarly for Firefox, if resources from the extension are to be referenced by an untrusted part using `<img>` or `<script>` tags, the corresponding registered content package must be flagged with `contentaccessible=yes`. Doing

this would allow for the webpage to load resources from the extension, e.g. images to an `<img>` tag [44]. The content can then be accessed using the `chrome://packagename/content/` schema [44], where the `packagename` should be unique for all extensions. For WebExtensions, the content can be accessed with `moz-extension://<extensionid>/<pathToFile>` [51].

Examples of web accessible resources in practice To illustrate web accessible resources and how they differ in Firefox and Chrome, consider two real-world examples: AdBlock and LastPass.

AdBlock for Chrome displays an icon in the browser toolbar which seemingly triggers a popup. This popup is actually an HTML page which loads JavaScript code to interact with the user. Both the HTML and JavaScript files are web accessible resources and must be listed as such [25].

When logging in to a new website with a password, LastPass for Chrome will prompt the user whether this password should be stored. This prompt is actually an "overlay" injected and rendered into the viewport of the visited webpage. The overlay is an HTML resource provided by the extension and marked as web accessible. LastPass for Firefox uses a slightly different approach because Firefox extensions have the ability to modify the browser chrome through *XML User Interface Language (XUL)*. Because this XUL file is only part of the browser chrome it does not need to be accessible from the visited webpage. Therefore, it does not need to be marked as a web accessible resource.

Benefits with web accessible resources While web accessible resources are a convenience, it is possible to do without them. Resources can be represented as strings using data URIs [40], which can be added to the created DOM element before injecting it to the webpage. It is also possible to store the resources on an external server and fetch them from there. However, both of these approaches have disadvantages. Encoding and injecting resources as strings can be difficult to maintain, and storing resources on an external server has potential privacy and security issues.

By using web accessible resources, the resources are stored within the extension. This make them easier to maintain and access with extension APIs.

Finding extensions via web accessible resources Because web accessible resources can be accessed in the context of a given webpage, they can be abused to detect the presence of browser extensions to which the resources belong. As mentioned above, LastPass for Chrome has the overlay file `overlay.html` marked as web accessible, making it possible to make a request for the file using e.g. XMLHttpRequest. If the resource is present,

the request will receive a positive answer, indicating that the extension is installed.

In Firefox, the extension Firebug has `contentaccessible=yes` set. Similarly to LastPass in Chrome, this makes Firebug detectable without behavior analysis, as the resource can be loaded to a `script` tag, using `onsuccess` and `onerror` to check if the extension is present or not.

Note that thanks to the uniqueness of the extension ids, we obtain a detection technique without false positives. While there is no guarantee that the behavioral techniques precisely detect a given extension, we never report an extension that is not present. Compared to behavioral techniques that may have both false positives and negatives, finding extensions via web accessible resources may have false negatives but no false positives.

Using CSP for finding extensions Content Security Policy (*CSP*) allows websites to whitelist where resources are loaded from [64]. One potential way of finding extensions is when they inject their web accessible resources into the webpage. Since one can define where to load e.g. scripts and images from in the CSP, restricting the CSP to not allow for an extension could in theory be possible. However, we found that both Chrome and Firefox allow `chrome-extension://` and `chrome://` URLs respectively to be injected by the extension, no matter what the CSP is, as long as they are flagged as `web_accessible_resources` and `contentaccessible`. If the injected script from the extensions is from a separate server, it will be blocked if it violates the CSP [31]. *WebExtensions* will not enforce CSP for the extensions [53].

3.3 Two attacker models

Recall that we are interested in two perspectives on extension detection: that of a webpage with the goal to enable extension detection (as in the Bank and Facebook scenarios) and that of an extension with the goal to remain hidden (as in the LastPass scenario). Consequently, this yields two attacker models. The first attacker model corresponds to a malicious extension that has been installed on a user's browser, e.g., to leak bank data or hijack likes. The challenge is to detect such extensions. The second attacker model corresponds to a malicious webpage that tries to thwart the functionality of a legitimate extension, e.g., by blocking ads or phishing. The challenge here is to prevent detection of such extensions. In this paper, we address both perspectives, even if their goals are by nature conflicting.

4 Empirical study of Chrome and Firefox extensions

This section reports on an empirical study to analyze how susceptible free extensions are to be found via web accessible resources.

The study was performed by downloading all free extensions from Chrome web store [21] and Mozilla's add-on store [48], extracting and analyzing their manifest files. The extensions were downloaded in September 2016.

4.1 Chrome

As mentioned in Section 3.1, *web_accessible_resources* in the manifest file can be used to determine extension detection via web accessible resources. If the manifest file does not contain the section *web_accessible_resources*, the extension cannot be detected using this technique. If the only accessible resources of an extension are URLs, we deem the extension non-detectable without behavioral analysis.

A total of 43,429 extensions were downloaded. However, the total amount of extensions where the user statistics were found by the scraper was 43,197 (≈99.5% of all downloaded extensions). The reason for this drop is that some extensions were removed from the Chrome web store before the scraper had the time to retrieve the user statistics, whereas some extensions (like Google Cast) did not display user statistics.

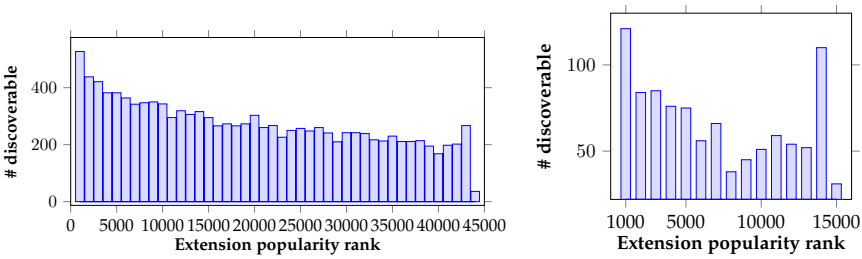
Results Table 4.2 displays the results of testing all downloaded Chrome extensions for *web_accessible_resources*. The parsing of the manifest files yielded parse errors for 36 extensions, for which we manually edited the manifest files to remove the errors.

We note that 148 extensions have *web_accessible_resources* set to an empty array in the manifest file, which implies that these extensions have no web accessible resources. Similarly, the 54 extensions which only have URLs as web accessible resources cannot be found with our technique as they do not have resources that should run in the context of the website stored locally in the extension. The "No accessible resources" in Table 4.2 are all the extensions where the *web_accessible_resources* field was missing in the manifest file, including 146 extensions which had only non-existing resources listed.

In total, 12,154 extensions out of 43,429 could be found using non-behavioral extension detection, which corresponds to ≈28%. Figure 4.3a shows the amount of detectable extensions sorted by popularity, based on the reported number of users in the Google Chrome web store. For this, we only use the set of extensions for which we could find user statistics, yielding 12,112 extensions detectable out of 43,197. We divide the sorted extensions

Table 4.2: Chrome and Firefox extension results

Category	Chrome	Firefox
Empty accessible resources	148	–
Only URLs	54	–
No manifest file	–	7,396
Detectable	12,154	1,003
No accessible resources	31,073	6,497
Total amount of extensions	43,429	14,896



(a) Chrome (b) Firefox

Figure 4.3: Discoverable browser extensions based on popularity

in groups of 1000, which we call "intervals". We find 70% of the top 10, 62% of the top 100 and 52.7% of the top 1000 extensions with a non-behavioral technique. These extensions include popular security- and privacy-critical extensions such as AdBlock, LastPass, Avast Online Security, Ghostery and Disconnect. The graph also shows a descending trend, indicating that more popular extensions have on average more *web_accessible_resources*.

4.2 Firefox

As mentioned in Section 3.1, manifest files for Firefox extensions can be located in several different sub folders of an extension. The manifest files in the sub folders are referenced from `chrome.manifest` in the root directory. For this study, all manifest files were analyzed, including the manifest files in the sub folders.

The `contentaccessible` flag indicates web accessible resources, but we found that a webpage cannot perform a normal `XMLHttpRequest` in order to retrieve the resource. However, it is possible to create a `script` tag with the corresponding `script.src` attribute set to the resource in order to retrieve it. By attaching `onload` and `onerror` event handlers to this `script` element, it is possible to learn whether the resource could be retrieved. In addition, because the absence of a resource is gracefully handled with the `onerror`

handler, no error is reported and this method in Firefox is more discrete than the method used with Chrome.

The amount of Firefox extensions was 17,375. However, some extensions were duplicated in the list on Mozilla's add-on page based on the extension name and the extension id. The scraper found a total of 14,925 unique extensions, but was redirected to a dead link for 29 extensions, yielding the total number of analyzed extensions to 14,896.

Results The results of the study can be seen in Table 4.2. 7,396 did not have a `chrome.manifest` file in the extension's root directory and 6,381 extensions did not have the flag `contentaccessible` in the `chrome.manifest` file in the root directory. 116 out of the 1,119 extensions who had set `contentaccessible` linked it to non-existing files. We also detected a total of 775 extensions who use `WebExtensions`. Out of those 775 extensions, 11 also defined `chrome.manifest`. 221 had `web_accessible_resources` set, indicating $\approx$ 28,5% of those extensions should be detectable. Unfortunately, *WebExtensions* extension ids are not stored publicly. One could, in theory, manually install all those extensions and see if they have e.g. an options page [49], which when browsed to would give the extension id. Due to this, we do not consider WebExtensions detectable in this experiment.

1,003 out of 14,896 can be found with web accessible resources, which corresponds to 6.73%. The trend for the detectable extensions can be seen in Figure 4.3b. The interval with the most extensions that are detectable was the top 1000 extensions with 121 detectable extensions (i.e. 12.1%). These extensions include Firebug, Easy Screenshot and Web of Trust. However, no ad blockers nor the popular script blocker Ghostery can be found in Firefox without behavioral analysis. As explained in Section 3.2, Firefox extensions have the ability to directly add to the UI using XUL, so that they do not require web accessible resources like Chrome extensions. Therefore, Firefox extensions need less web accessible resources.

4.3 Comparison of results

One major difference between Chrome and Firefox is how `XMLHttpRequest` is handled. In Firefox, it is not allowed to access `chrome://` with `XMLHttpRequest`, whereas it is possible to access `moz-extension://` in Firefox and `chrome-extension://` in Chrome. The use of web accessible resources, and with that the percentage of detectable extensions, is higher for Chrome. As a Chrome extension cannot make much modifications to the UI of the browser compared to Firefox, there is a greater need for using web accessible resources in Chrome. Similarities could be found in the trends of accessible resources, where both browsers had the largest interval of detectable exten-

sions in the top 1000 extensions, but Chrome had a more clear decrease over the following intervals compared to Firefox.

5 Browser extension detection in the Alexa top 100,000

It is possible for a webpage to detect some browser extensions in a visitor's browser by attempting to retrieve web accessible resources. This detection technique may be used in a malicious capacity (e.g. fingerprinting or the reconnaissance before an attack), as well as for benign reasons (e.g. to avoid offering the extension again, in case the visitor is already using it).

To determine whether web developers actively use this extension detection technique, we visited the top webpage on the most popular 100,000 web domains according to Alexa, a web traffic analysis company. For each domain, e.g. example.com, we visited its top-most URL, i.e. `http://example.com` and waited a total of one minute for the page to load and any JavaScript to run its course. To determine whether a webpage attempts to access web accessible resource URLs, we created a simple headless JavaScript-enabled browser based on Qt5's QWebView, which uses the WebKit web rendering engine. Because our custom browser does not have any browser extensions, and thus no web accessible resources, any request towards a URL with unknown scheme results in an error. These errors, together with all console output generated by WebKit, were logged for every page visit for later analysis.

To avoid an unnecessary check, a webpage can query the browser's user-agent before deciding to request a certain resource. Therefore, we configured our browser to report a user-agent string associated with the most popular web browser vendors [1]. The list of used user-agent strings was retrieved from a list of commonly occurring user-agent strings [5]. We emulated Google Chrome 47.0, Mozilla Firefox 40.1, Opera 12.16, Apple Safari 7.0.3, Microsoft Internet Explorer 11, and Microsoft Edge 12.246.

Our intent is not to fake the presence of a particular browser, but instead determine whether web developers inspect the browser's user-agent string before attempting to detect browser extensions.

All webpages were visited in September 2016. Of the 100,000 URLs we visited, 91,299 webpages (91.3%) could be visited by at least one of the user-agents.

The data shows attempts to access resource URLs with several different schemes, but we were only interested in Google Chrome's `chrome-extension://` and Mozilla Firefox's `chrome://` and `moz-extension://`. We did not log any attempts to access `moz-extension://`, most likely because WebExtensions is not yet fully implemented and not many Firefox extensions use it yet.

Table 4.3: Which web pages detect which Firefox extensions via simple GET requests through HTML elements, when impersonating **C**hrome, **F**irefox, **S**afari, **O**pera, **M**SIE and **E**dge respectively. No visited web pages attempted to detect extensions using the XMLHttpRequest method, thus these columns are omitted.

Rank	Domain	Ext.id	GET C F S O M E
10018	amaebi.net	Fext_C	✓ ✓ ✓ ✓ ✓ ✓
13138	forum.hr	Fext_D	– ✓ – – – –
17410	ebitsu.net	Fext_C	✓ ✓ ✓ ✓ ✓ ✓
20688	katohika.gr	Fext_D	– ✓ – – – –
22197	881903.com	Fext_F	✓ ✓ ✓ ✓ ✓ ✓
45043	rincondeltibet.com	Fext_B	✓ ✓ ✓ ✓ ✓ ✓
48860	dalmacijanews.hr	Fext_D	– ✓ – – – –
57858	blogsdelagente.com	Fext_E	✓ ✓ ✓ ✓ ✓ ✓
60190	footballmanagerstory.com	Fext_D	– ✓ – – – –
64627	arouraios.gr	Fext_D	– ✓ – – – –
73723	aekfans21.com	Fext_D	– ✓ – – – –
76496	proekt-gaz.ru	Fext_A	✓ ✓ ✓ ✓ ✓ ✓
84514	olagossip.gr	Fext_D	– ✓ – – – –
87870	evrsac.rs	Fext_D	– ✓ – – – –
89329	mikroskopio.gr	Fext_D	– ✓ – – – –
92899	burek.com	Fext_D	– ✓ – – – –
96646	freegossip.gr	Fext_D	– ✓ – – – –
97133	lifenewscy.com	Fext_D	– ✓ – – – –

Table A.1 lists the domains in the Alexa top 100,000 which attempted to access `chrome-extension://` URLs, while Table 4.3 lists the same for `chrome://` URLs. In both of these tables, the "Ext.id" field contains the extension id of the accessed extension. Of the 91,299 webpages we successfully visited, 66 webpages in total attempted to access web accessible resources: 48 and 18 webpages attempted to access `chrome-extension://` and `chrome://` URLs respectively. No webpage attempted to access URLs of both schemes, even when presented with a different user-agent string.

As described in Section 3.2, extensions can be detected by accessing web accessible resources through either XMLHttpRequest or simple GET requests through HTML elements. Of the 48 webpages that detect Chrome

extensions, 23 use the XMLHttpRequest method and 27 use GET requests. Only two webpages, `mon.cat` and `rifftrax.com`, use both techniques. The 18 web pages that detect Firefox extensions all use GET requests, presumably because the web developers know about Firefox's limitation discussed in Section 4.2.

Of the 66 webpages that access web accessible resources, 23 (17 detecting Chrome extensions, six for Firefox extensions) do not change their behavior when presented with a different user-agent string. The majority of 43 webpages (31 Chrome, 12 Firefox) only attempt to access web accessible resources when presented with a specific set of user-agent strings. For the 31 webpages detecting Chrome extensions based on certain user-agent strings, 15 check for a Chrome user-agent string, nine for either Chrome or Edge, two for Opera and five for five different sets of user-agent combinations. The 12 webpages detecting Firefox extensions based for a specific user-agent, all only target the Firefox user-agent.

Table 4.4 lists the extensions probed for during our visit of the Alexa top 100,000 for Chrome and Firefox extensions. Of the 36 Chrome extensions, nine could not be found in the Chrome Web Store, including one (`Cext_AA`) for which we could not find any information at all. None of these Chrome extensions could be labeled as malware with any certainty. The Chrome Web Store categorizes these 36 extensions as: eight "productivity", eight "fun", six "news and weather", five "search tools", three "developer tools", three "accessibility" and two as "shopping". There are seven different versions of Google Cast Chrome extension appears seven time in the list, and eight extensions named "My <something> XP" which are from the same author.

Of the six Firefox extensions in Table 4.4, only one (`Fext_C`) could be found on the Mozilla Add-ons website. Of the five others, two are related to malware. Noteworthy is `Fext_F`, which is a Firefox extension developed in a Firefox extension development tutorial.

Out of the 66 webpages that access web accessible resources, most (49) probe for the existence of a single Chrome or Firefox extension. The other 17 web pages probe for more than one extension, indicating three distinct clusters of extensions in our dataset.

The first cluster contains extensions `Cext_E`, `Cext_K`, `Cext_M`, `Cext_Q` and `Cext_U`. This cluster of five extensions is probed for on nine different domains using only XMLHttpRequests and the extensions are different versions of the Google Cast extension.

The second cluster contains `Cext_E`, `Cext_K`, `Cext_M`, `Cext_Q`, `Cext_P` and `Cext_AI`. This cluster is same as the previous one, but lacks `Cext_U` and adds `Cext_P` and `Cext_AI`. Two webpages test for this cluster and use XMLHttpRequests for the web accessible resources from the previous cluster, but GET

requests for the resources of the two added extensions in the list. All these extensions are again different versions of Google Cast.

Finally, a third cluster consists of `Cext_J`, `Cext_W` and `Cext_AD`. This cluster appears on five webpages using only GET requests to probe for the associated web accessible resources. These three extensions are not versions of the same extensions like in both previous clusters. Instead, the common factor in this case are the webpages probing for the extensions. All five webpages are protected by an F5 BIG-IP APM, which rewrites and obfuscates JavaScript code before transmitting it to the browser. We are uncertain whether this F5 appliance inserts the extension detection code by itself, or whether the web pages happen to serve the same JavaScript.

The results from our experiment on the Alexa top 100,000 domains show that `chrome-extension://` and `chrome://` URLs are sometimes used by webpage developers to identify the presence of a certain extensions, although this practice seems not widespread.

The same technique could also be used to fingerprint visitors for tracking or deanonymization purposes, but we did not find any obvious evidence that suggests that this is a common practice.

The presence of clusters of extension detections such as for the detection of the Google Cast extension and all its versions (first two clusters) follows a pattern that may indicate that web developers are sharing code for this purpose. The reason behind the existence of the third cluster is unclear, since it involves three very different extensions and the webpages deploying the cluster use the same F5 appliance.

6 Measures

Section 6.1 suggests measures in favor of website developers, while Section 6.2 suggests how extensions can prevent being found by webpages. Finally, Section 6.3 concludes with a discussion of how to resolve security goal clashes.

6.1 Measures for webpages: whitelisting extensions

Enabling webpages to specify a whitelist of allowed extensions, would empower them to guarantee a clean web environment for their content. We envision that such a measure can be implemented as a policy specified by the webpage and enforced by the browser.

For a web application handling sensitive information, like a web banking application, an environment known-to-be free from malware would help secure the user's sensitive data. Of course, such a whitelist could be used to block any extension, such as an ad blocker, as well.

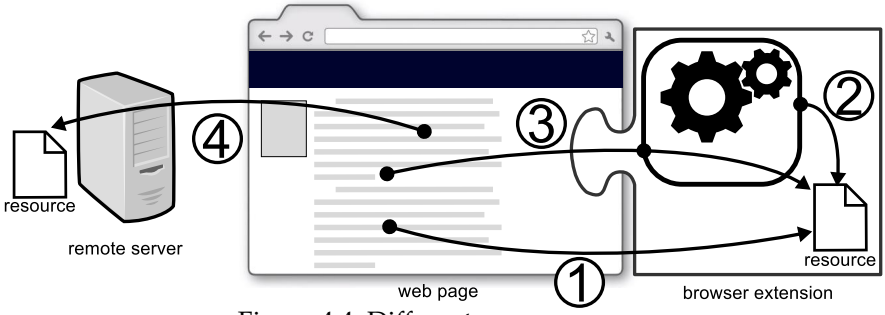


Figure 4.4: Different measures map

We believe it is crucial to not take away control from either party, but rather have both parties agree on a sensible list of extensions that may be used on the webpage. The webpage may suggest the whitelist to indicate its intentions to secure a malware-free environment. One possibility in this design space is to leave the final decision up to the user, endorsing and/or overriding the whitelist, if desirable.

6.2 Measures for extensions

Extensions that are designed to enrich user experience would like to minimize the risk of being found using non-behavioral analysis. The following section will give examples of what such measures could look like. Figure 4.4 illustrates these approaches.

Prevent direct access to extension resources from webpage One natural measure to prevent detection of an extension would be to disable direct access from a webpage to an extension's resource (Arrow #1 in Figure 4.4). Instead, to retrieve an extension's resources, a webpage would then need to communicate with the extension via a message passing API (Arrow #3 in Figure 4.4).

This measure would not prevent detection of an extension entirely, but it would give the extension the opportunity to be involved in the detection process, as desired in e.g, the Google Cast scenario.

No accessible resources Web accessible resources can be avoided by hosting the resources on a remote server or using data URIs (see Section 3.2).

Hosting resources on a remote server (Arrow #4 in Figure 4.4) will cause more network traffic. However, the extra network traffic can be reduced through the browser's caching mechanism. This approach, be it with or without caching, does not fully prevent the extension from being detectable through a timing attack. A webpage trying to detect the presence of the

extension may request the same remote resource and measure its loading time. If the extension is present, the loading time will be small.

In addition to detectability through a timing attack, remotely hosted resources also introduce privacy concerns. Unlike for web accessible resources hosted locally from inside an extension, requests for remotely hosted resources can be monitored by an external party. These requests compromise the privacy of the user by revealing visited URLs and possibly parts of the user's identity.

Using data URIs [40] would effectively remove all arrows but #2 in Figure 4.4 and would remove extensions' dependence on web accessible resources. A disadvantage of this approach, is that hard-coded data URIs can be difficult to maintain.

Track script provenance One could potentially track who injected the script and only allow access to a given set of principals. Tracking the information flow is, however, expensive and can make the system slower, but it would allow for web accessible resources to be used by the content script and scripts on the webpage that originate from the extension, but not be used by the actual webpage itself. With such a system in place, the extension can be seen as a closed entity from the webpage's point of view, and therefore the web accessible resources would not have to be publicly available.

This measure can benefit from recent work on tracking information flow in JavaScript [32] and tracking provenance across the browser's document object model (DOM) [14].

When one looks at tracking script provenance, it is easy to see a scenario where it would be up to the user to decide if a webpage should be allowed to access the extension's resources by prompting the user whenever a script which was not part of the injected scripts from the extension tries to access resources.

This measure would distinguish Arrows #1 and #2 in Figure 4.4, only allowing injected scripts on the webpages to access the resources based on provenance.

Extension ids An extension developer, in order to avoid detection, could change the extension id by e.g. resubmitting the same extension to the extension repository and getting a new id. This by itself would be of limited effect as then the extension with updated id needs to rebuild its userbase.

An extension has other means to retrieve its own resources (Arrow #2 in Figure 4.4) than via web-accessible resources. The only reason to have web accessible resources is for a webpage to load its resource. But the location of this resource does not need to be fixed. Instead of having a fixed extension id which can be used to detect the presence of an extension, the extension could

generate a random token and pass it along to the webpage. A webpage which possesses this token, can use it to gain access to the extension's resources.

Whitelisting webpages Instead of being active on all webpages a browser visits, extensions could be activated on a case-by-case basis. For instance, there is probably no need to enable the Google Cast extension on a banking website. If an extension is not active on a webpage, and its resources not available to this webpage, then it can not be detected through the presence of web accessible resources. A measure such as this one can be implemented through a user-modifiable whitelist in the browser.

6.3 User to resolve conflicting security goals

Because the conflicting security goals are legitimate, it is important to strike a reasonable balance between the interests of the different parties by combining webpage measures with extension measures. For example, allowing webpages to whitelist extensions which can be active in their domain, whereas allowing extensions to whitelist webpages which are allowed to communicate with the extensions would help both webpages and extensions reach their goals.

But who should be the one to resolve the conflicting security goals? As mentioned in Section 6.1, allowing a webpage to provide a whitelist over extensions allowed to execute in their domain can lead to webpages not allowing any extension. This can lead to users losing their ability to customize their user experience when browsing the web.

We resort to the "users > developers > browser" principle, as common in the web community folklore. This principle gives users precedence over developers and browsers in the web setting. Driven by this principle, we designate the user as an arbiter to endorse and/or overwrite whitelists provided by webpages and extensions, respectively.

We currently experiment with a prototype, based on Chromium, to support fine-grained whitelisting policies that give the user the power to temporarily enable and disable extensions depending on what webpages are being visited.

7 Related work

Non-behavioral extension detection has so far received only scarce attention, primarily in the form of scattered blog posts [8, 4, 3, 6, 2, 7], some referring to outdated browser features and some only traceable in Internet archives [8, 4].

To the best of our knowledge, we are the first to systematically study non-behavioral extension discovery at large in both Chrome and Firefox's extension web stores, as well as the Alexa top 100,000 webpages.

There is a large body of work on detection of maliciously behaving browser extensions. The state of the art is well summarized by Jagpal et al. [35]. The rest of this section focuses on detecting extensions and fingerprinting browsers.

7.1 Detecting extensions

Prior work in detecting extensions has focused on behavioral techniques. For instance, Nikiforakis et al. [57] analyze eleven popular browser extensions that hide the real user agent string from visited websites in order to obfuscate a browser's fingerprint, but observe that the these extensions neglect to remove the same information from the JavaScript environment, making the extension detectable by a visited website through its behavior. This detection mechanism is fragile since, as explained in Section 2, extensions may modify their behavior in order to avoid detection, forcing websites to alter their detection method, triggering an arms race. Using another approach, Thomas et al. [61] detect the in-flight alteration of a webpage, by comparing the DOM of the rendered webpage against the expected DOM. This catch-all method detects all DOM modifying extensions as well as proxies and compromised browsers. Such an approach is more robust, since it will detect all extensions that modify the DOM even when they attempt to evade detection. However, since it does not focus on an extension's specific behavior, it is less precise. Non-behavioral extension detection on the other hand, like the technique presented in this paper, uses simple and cheap checks to determine the presence of a specific extension, without false positives. In addition, an extension can not evade detection by altering its behavior. Instead, the only way for an extension to avoid detection is by removing its web accessible resources, which is not always practical as explained in Section 6.2.

Non-behavioral extension discovery via web accessible resources has only received scarce attention in the form of scattered observations, primarily in blog posts [8, 4, 3, 6, 2, 7], some referring to outdated browser features and some only traceable in Internet archives [8, 4].

We go beyond these observations by systematically studying the entire class of extension discovery via web accessible resources, performing an empirical study with discoverability of all free extensions of the two major browsers, preforming a large scale study of discovery by the top 100,000 Alexa webpages, and proposing measures.

7.2 Fingerprinting browsers

There has been much work on browser fingerprinting. INRIA's Browser Extension Experiment [34] is based on our technique and code to enhance browser fingerprinting by detecting extensions. We overview the work on fingerprinting below, noting that the rest of the approaches are less related because they do not address extension detection.

Panopticlick [59] uses such browsers properties as screen resolution, user agent string, timezone, system fonts, and browser plugins to uniquely identify browsers. Browsers can also be fingerprinted through browser quirks [9], canvas fingerprinting [43, 10], dimensions of rendered font glyphs [19], browser histories [58], ECMAScript compliance [55], performance of the JavaScript engine and whitelisted domains in the NoScript extension [42], and more [57, 63].

Nikiforakis et al. [57] detect font probing and flash-based proxy evasion as fingerprinting mechanisms provided by three commercial fingerprinting companies, and find 40 websites in the Alexa top 10,000 make use of them. Acar et al. build FPDetective [11] and find 404 websites in the Alexa top million that use JavaScript-based font probing, as well as 145 websites in the Alexa top 10,000 that use Flash-based font probing to fingerprint visitors. Acar et al. [10] study the Alexa top 100,000 and find that canvas fingerprinting is the most commonly used fingerprinting technique, with 5% of the studied websites using it.

Defending against fingerprinting is difficult, if even possible. There appears to be no one-size-fits-all solution. Several strategies have been suggested. One crude way to address the problem is by simply blocking certain forms of third-party content, such as JavaScript or Flash known to contain fingerprinting code [10, 17, 57, 58, 63]. Similarly crude would be to disable certain functionality in the browser, such as the ability to query pixel-values from a canvas [43].

Instead of blocking third-party content or functionality, a browser could ask for user permission whenever a fingerprintable characteristic of the browser is queried, e.g. reading those pixel-values from a canvas [10, 43, 63].

Yet another approach adds (smart) noise to fingerprintable browser characteristics, thereby randomizing the fingerprint [10, 43, 17, 19, 20, 36, 56, 62, 63]. The reverse approach is to decrease the randomness of the reported browser characteristics by standardizing the set of possible values for fingerprintable resources, such as the list of system fonts, so that all browsers report the same values [19, 43, 57, 63].

Conceding that fingerprinting cannot be stopped, recent work has investigated preventing the exfiltration of the fingerprint itself by monitoring net-

work traffic [62, 19, 55], or even by rewriting a detected fingerprint through a network proxy [65].

8 Conclusion

To the best of our knowledge, we have presented the first comprehensive study of non-behavioral browser extension discovery. We have systematically studied the technique and its applicability at large scale. At the core of our technique is detection of web accessible resources that are associated with extensions via unique extension ids. This yields an effective detection technique with no false positives, which we have instantiated for both Chrome and Firefox. We report on an empirical study with free Chrome and Firefox extensions, detecting over 50% of the top 1,000 free Chrome extensions (including such sensitive extensions as AdBlock and LastPass) and over 28% of the Chrome extensions in the study overall. We have conducted an empirical study of non-behavioral extension detection on the Alexa top 100,000 websites. This study confirms that detecting extensions via web accessible resources is not widely known. Nevertheless, we identify websites that perform extension detection for types of extensions that include fun, productivity, news, weather, search tools, developer tools, accessibility, and shopping. We have presented measures for and against browser extension discovery, catering to the needs of website owners and extension developers, respectively. Finally, we have discussed a browser architecture that allows a user to take control in arbitrating the conflicting security goals.

Our code for discovering browser extensions is already used by INRIA's Browser Extension Experiment [34].

Future work focuses on the measures outlined in Section 6. In particular, our short-term goal is to study whether disallowing GET requests from webpages to extension schemas (Firefox disallows `XMLHttpRequest` apart from for WebExtensions, but not GET from HTML elements such as `script` and `img`, whereas Chrome allows all three) will result in breaking functionality of common extensions. Such a study may provide useful input for the future handling of extensions in Chrome and Firefox. As mentioned earlier, we are also experimenting with a prototype based on Chromium to support fine-grained whitelisting policies that give the user the power to temporarily enable and disable extensions depending on what webpages are being visited.

Acknowledgments Thanks are due to Ioannis Papagiannis for the inspirations and helpful feedback. This work was partly funded by Andrei Sabelfeld's Google Faculty Research Award, Facebook Research and Academic Relations Program Gift, the European Community under the ProSecuToR project, and the Swedish research agency VR.

9 Bibliography

[1] Desktop Browser Market Share. https://www.netmarketshare.com/browser-market-share.aspx.

[2] Detecting Chrome Extensions in 2013. http://gcattani.github.io/201303/detecting-chrome-extensions-in-2013/.

[3] Detecting Firefox Extensions Without Javascript. http://kuza55.blogspot.co.uk/2007/10/detecting-firefox-extension-without.html.

[4] Detecting FireFox Extentions. http://ha.ckers.org/blog/20060823/detecting-firefox-extentions/.

[5] List of User Agent Strings. http://www.useragentstring.com/pages/useragentstring.php.

[6] Sparse Bruteforce Addon Detection. http://www.skeletonscribe.net/2011/07/sparse-bruteforce-addon-scanner.html.

[7] The Evolution of Chrome Extensions Detection. http://blog.beefproject.com/2013/04/the-evolution-of-chrome-extensions.html.

[8] Yet Another Way to Detect Internet Explorer. http://ha.ckers.org/blog/20060821/yet-another-way-to-detect-internet-explorer/.

[9] E. Abgrall, Y. Traon, M. Monperrus, S. Gombault, M. Heiderich, and A. Ribault. XSS-FP: Browser fingerprinting using HTML parser quirks. Technical report, 2012. arXiv:1211.4812 [cs].

[10] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *CCS*, 2014.

[11] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the web for fingerprinters. In *CCS*, 2013.

[12] AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom.

[13] V. Allaire. FuckAdBlock. https://github.com/sitexw/FuckAdBlock.

[14] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, and Y. Tian. Run-time monitoring and formal analysis of information flows in chromium. In *NDSS*, 2015.

[15] Q. Cao, X. Yang, J. Yu, and C. Palow. Uncovering large groups of active malicious accounts in online social networks. In *CCS*, 2014.

[16] clsr. FuckFuckFuckAdBlock. https://gist.github.com/clsr/3f5ca796463a0e6fc8af.

[17] A. FaizKhademi, M. Zulkernine, and K. Weldemariam. FPGuard: Detection and prevention of browser fingerprinting. In *Data and Applications Security and Privacy*, 2015.

[18] http://newsroom.fb.com/company-info/#statistics.

[19] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*, 2015.

[20] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri. Countering browser fingerprinting techniques: Constructing a fake profile with google chrome. In *NBiS*, 2014.

[21] Google. Chrome web store. https://chrome.google.com/webstore/category/extensions?hl=en-GB&_feature=free.

[22] Google. chrome.browserAction. https://developer.chrome.com/extensions/browserAction.

[23] Google. chrome.pageAction. https://developer.chrome.com/extensions/pageAction.

[24] Google. Content Scripts. https://developer.chrome.com/extensions/content_scripts.

[25] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources.

[26] Google. Manifest File Format. https://developer.chrome.com/extensions/manifest.

[27] Google. Message Passing. https://developer.chrome.com/extensions/messaging.

[28] Google. Overview. https://developer.chrome.com/extensions/overview.

[29] Google Cast. https://chrome.google.com/webstore/detail/google-cast/boadgeojelhgndaghljhdicfkmllpafd.

[30] P. Gühring. Concepts against man-in-the-browser attacks. http://www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf, 2006.

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

[32] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and its APIs. In *SAC*, 2014.

[33] How to detect Adblock on my website? http://stackoverflow.com/questions/4869154/how-to-detect-adblock-on-my-website.

[34] INRIA. Browser Extension Experiment. https://extensions.inrialpes.fr.

[35] N. Jagpal, E. Dingle, J. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas. Trends and lessons from three years fighting malicious extensions. In *USENIX Sec.*, 2015.

[36] P. Laperdrix, W. Rudametkin, and B. Baudry. Mitigating browser fingerprint tracking: Multi-level reconfiguration and diversification. In *SEAMS*, 2015.

[37] I read that LastPass is vulnerable to phishing attacks - should I be concerned? https://lastpass.com/support.php?cmd=showfaq&id=10072.

[38] LastPass. https://lastpass.com/.

[39] LostPass. https://www.seancassidy.me/lostpass.html.

[40] L. Masinter. The "data" URL scheme. http://tools.ietf.org/html/rfc2397.

[41] Mechazawa. FuckFuckAdBlock. https://github.com/Mechazawa/FuckFuckAdblock.

[42] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in JavaScript implementations. In *W2SP*, 2011.

[43] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in HTML5. In *W2SP*, 2012.

[44] Mozilla. Chrome registration. https://developer.mozilla.org/en-US/docs/Chrome_Registration.

[45] Mozilla. Communicating using "port". https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Content_Scripts/using_port.

[46] Mozilla. Communicating using "postmessage". https://developer.mozilla.org/en-US/Add-ons/SDK/Guides/Content_Scripts/using_postMessage.

[47] Mozilla. Manifest Files. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Tutorial/Manifest_Files.

[48] Mozilla. Most Popular Extensions. https://addons.mozilla.org/en-US/firefox/extensions/?sort=users.

[49] Mozilla. options_ui. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/options_ui.

[50] Mozilla. Porting a Google Chrome extension. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Porting_a_Google_Chrome_extension.

[51] Mozilla. web_accessible_resources. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/web_accessible_resources.

[52] Mozilla. WebExtensions. https://developer.mozilla.org/en-US/Add-ons/WebExtensions.

[53] Mozilla. WebExtensions - Permission Model. https://wiki.mozilla.org/WebExtensions#Permission_Model.

[54] Mozilla. XUL Overlays. https://developer.mozilla.org/en-US/docs/Mozilla/Tech/XUL/Overlays.

[55] M. Mulazzani, P. Reschl, M. Huber, M. Leithner, S. Schrittwieser, E. Weippl, and F. Wien. Fast and reliable browser identification with JavaScript engine fingerprinting. In *W2SP*, 2013.

[56] N. Nikiforakis, W. Joosen, and B. Livshits. PriVaricator: Deceiving fingerprinters with little white lies. In *WWW*, 2015.

[57] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *S&P*, 2013.

[58] L. Olejnik, C. Castelluccia, and A. Janc. Why johnny can't browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs*, 2012.

[59] Panopticlick. `https://panopticlick.eff.org/`.

[60] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering Browser Extensions via Web Accessible Resources. Full version and code. `http://www.cse.chalmers.se/research/group/security/extensions`.

[61] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. A. Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *S&P*, 2015.

[62] C. F. Torres, H. Jonker, and S. Mauw. FP-block: Usable web privacy by controlling browser fingerprinting. In *ESORICS*, 2015.

[63] R. Upathilake, Y. Li, and A. Matrawy. A classification of web browser fingerprinting techniques. In *NTMS*, 2015.

[64] W3C. Csp2. `https://www.w3.org/TR/CSP2/`.

[65] S. Yokoyama and R. Uda. A proposal of preventive measure of pursuit using a browser fingerprint. In *IMCOM*, 2015.

Table 4.4: Chrome (`Cext_*`) and Firefox (`Fext_*`) extensions requested from Alexa top 100,000 sites

Ext.id	Extension name	count	in web store	malware?	Extension type
Cext_A	Turn Off the Lights	1	✓	–	accessibility
Cext_B	Gismeteo	1	✓	–	news and weather
Cext_C	My Speed Test XP	1	✓	–	productivity
Cext_D	GF Tools	1	✓	–	accessibility
Cext_E	Google cast	11	✓	–	fun
Cext_F	Adblock plus	1	✓	–	productivity
Cext_G	My classifieds XP	1	✓	–	search tools
Cext_H	My maps XP	1	✓	–	search tools
Cext_I	Screen Capture	3	–	?	developer tools
Cext_J	User-Agent Switcher	5	✓	–	productivity
Cext_K	Google Cast Beta	11	–	–	fun
Cext_L	offnews.bg	1	✓	–	news and weather
Cext_M	Google Cast (old)	11	–	–	fun
Cext_N	My email XP	1	✓	–	search tools
Cext_O	My weather XP	1	✓	–	news and weather
Cext_P	Google Cast (old)	2	–	–	fun
Cext_Q	Google Cast (old)	11	–	–	fun
Cext_R	Google Docs Offline	3	✓	–	productivity
Cext_S	Adblock	2	✓	–	productivity
Cext_T	My TV XP	1	✓	–	search tools
Cext_U	Google Cast (old)	9	–	–	fun
Cext_V	My current news XP	1	✓	–	news and weather
Cext_W	Table capture	5	✓	–	developer tools
Cext_X	NetBarg	1	✓	–	shopping
Cext_Y	Galera Video News	1	✓	–	accessibility
Cext_Z	RT News	1	✓	–	news and weather
Cext_AA	???	1	–	?	???
Cext_AB	Enable Copy	1	✓	–	productivity
Cext_AC	Letyshops Cashback	1	✓	–	shopping
Cext_AD	Scraper	5	✓	–	developer tools
Cext_AE	Ghostery	2	✓	–	productivity
Cext_AF	Iomods	1	–	–	fun
Cext_AG	My directions XP	1	✓	–	search tools
Cext_AH	Новости дня СМИ2	1	✓	–	news and weather
Cext_AI	Google Cast (old)	2	–	–	fun
Cext_AJ	Streak GRM for Gmail	2	✓	–	productivity
Fext_A	“depositfiles”	1	–	?	
Fext_B	PiccShare	1	–	✓	adware
Fext_C	S3 Google Translator	2	✓	–	
Fext_D	“searchincognito”	12	–	✓	adware
Fext_E	Skype Extension	1	–	–	
Fext_F	Firefox Toolbar Tutorial	1	–	–	tutorial

Table A.1: Which web pages detect which Chrome extensions, via either XMLHttpRequest or simple GET requests through HTML elements, when impersonating **C**hrome, **F**irefox, **S**afari, **O**pera, **M**SIE and **E**dge respectively.

Rank	Domain	Ext.id	XHR C F S O M E	GET C F S O M E
127	twitch.tv	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓✓✓✓✓	– – – – – –
417	newegg.com	Cext_AE	– – – – – –	✓✓✓✓✓✓
564	gismeteo.ru	Cext_B	– – – – – –	✓ – – – – ✓
1678	smi2.ru	Cext_AH	✓ – – – – ✓	– – – – – –
2012	popmyads.com	Cext_AE	– – – – – –	✓ – – – – ✓
2423	shadbase.com	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓✓✓✓✓	– – – – – –
2486	what-character-are-you.com	Cext_S	– – – – – –	✓✓✓✓✓✓
4726	gdeposylka.ru	Cext_AC	– – – – – –	✓ – – – – –
6486	stc.com.sa	Cext_A	– – – – – –	✓✓✓✓✓✓
10226	netbarg.com	Cext_X	– – – – – ✓	– – – – – –
11157	offnews.bg	Cext_L	– – – – – –	✓ – – – – ✓
14921	moi.gov.qa	Cext_J, Cext_W, Cext_AD	– – – – – –	✓✓✓✓✓✓
15862	gameblog.fr	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓✓✓ – ✓	– – – – – –
21917	myemailxp.com	Cext_N	✓ – – – – –	– – – – – –
23008	takenokosokuhou.com	Cext_I	– – – – – –	– – – ✓ – –
25410	loginfaster.com	Cext_AA	✓ – – – – –	– – – – – –
25647	gorod.dp.ua	Cext_F, Cext_S	– – – – – –	✓✓✓ – – ✓
25787	mailfoogae.appspot.com	Cext_AJ	– – – – – –	✓ – – – – ✓
26908	mon.cat	Cext_E, Cext_K, Cext_M, Cext_Q	✓✓✓✓✓✓	– – – – – –
		Cext_P, Cext_AI	– – – – – –	✓✓✓✓✓✓
29906	landandfarm.com	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓ – – – – –	– – – – – –
33100	dailynews.lk	Cext_Z	✓ – – – – ✓	– – – – – –
36050	amtrakguestrewards.com	Cext_J, Cext_W, Cext_AD	– – – – – –	✓✓✓✓✓✓
42726	wotlabs.net	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓ – ✓ – ✓	– – – – – –
43800	rifftrax.com	Cext_E, Cext_K, Cext_M, Cext_Q	✓✓✓✓✓✓	– – – – – –
		Cext_P, Cext_AI	– – – – – –	✓✓✓✓✓✓
44979	teutorrent.com	Cext_D	– – – – – –	✓✓✓✓ – ✓
45000	dohabank.com.qa	Cext_J, Cext_W, Cext_AD	– – – – – –	✓✓✓✓✓✓
45463	gameworld.gr	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓✓✓✓✓	– – – – – –
45922	myspeedtestxp.com	Cext_C	✓ – – – – –	– – – – – –
48905	mymapsxp.com	Cext_H	✓ – – – – –	– – – – – –
49383	mydrivingdirectionsxp.com	Cext_AG	✓ – – – – –	– – – – – –
50866	samagra.gov.in	Cext_R	– – – – – –	✓ – – – – –
51177	mytelevisionxp.com	Cext_T	✓ – – – – –	– – – – – –
51651	agariomods.com	Cext_AF	– – – – – –	✓ – – – – ✓
52003	cal-online.co.il	Cext_J, Cext_W, Cext_AD	– – – – – –	✓✓✓✓✓✓
53310	magine.com	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓ – – – – ✓	– – – – – –
56422	globalgamejam.org	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓✓✓✓✓✓	– – – – – –
56759	connectdirectlink.com	Cext_I	– – – – – –	– – – ✓ – –
62515	sorteiefb.com.br	Cext_I	– – – – – –	✓✓✓✓✓✓
65826	emsisoft.com	Cext_R	– – – – – –	✓ – – – – –
67549	deepdiscount.com	Cext_J, Cext_W, Cext_AD	– – – – – –	✓✓✓✓✓✓
72167	streak.com	Cext_AJ	– – – – – –	✓ – – – – ✓
73173	galerafilmes.com	Cext_Y	– – – – – –	✓✓✓✓✓✓
77437	mycurrentnewsxp.com	Cext_V	✓ – – – – –	– – – – – –
78429	chuckhawks.com	Cext_E, Cext_K, Cext_M, Cext_Q, Cext_U	✓ – – – – –	– – – – – –
81724	zjw.cn	Cext_AB	– – – – – –	✓✓✓✓✓✓
91408	myweatherxp.com	Cext_O	✓ – – – – –	– – – – – –
92146	myclassifiedsxp.com	Cext_G	✓ – – – – –	– – – – – –
93774	freehomeschooldeals.com	Cext_R	– – – – – –	✓ – – – – –

Paper V

Latex Gloves: Protecting Browser Extensions from Probing and Revelation Attacks

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

Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, February 2019

Abstract

Browser extensions enable rich experience for the users of today's web. Being deployed with elevated privileges, extensions are given the power to overrule web pages. As a result, web pages often seek to detect the installed extensions, sometimes for benign adoption of their behavior but sometimes as part of privacy-violating user fingerprinting. Researchers have studied a class of attacks that allow detecting extensions by *probing* for Web Accessible Resources (WARs) via URLs that include public extension IDs. Realizing privacy risks associated with WARs, Firefox has recently moved to randomize a browser extension's ID, prompting the Chrome team to plan for following the same path. However, rather than mitigating the issue, the randomized IDs can in fact exacerbate the extension detection problem, enabling attackers to use a randomized ID as a reliable fingerprint of a user. We study a class of extension *revelation* attacks, where extensions reveal themselves by injecting their code on web pages. We demonstrate how a combination of revelation and probing can uniquely identify 90% out of all extensions injecting content, in spite of a randomization scheme. We perform a series of large-scale studies to estimate possible implications of both classes of attacks. As a countermeasure, we propose a browser-based mechanism that enables control over which extensions are loaded on which web pages and present a proof of concept implementation which blocks both classes of attacks.

1 Introduction

Browser extensions, or simply extensions, enable rich experience for the users of today's web. Since the introduction of browser extensions in Microsoft Internet Explorer 5 in 1999 [42], they have been an important tool to customize the browsing experience for all major browser vendors. Today, the most popular extensions have millions of users, e.g. AdBlock [10] has over 10,000,000 downloads in the Chrome Web Store [24]. All major web browsers now support browser extensions. Mozilla and Chrome provide popular platforms for browser extensions, with Mozilla having over 11.78%, and Chrome over 66.1% of the browser's market share (April 2018) [57].

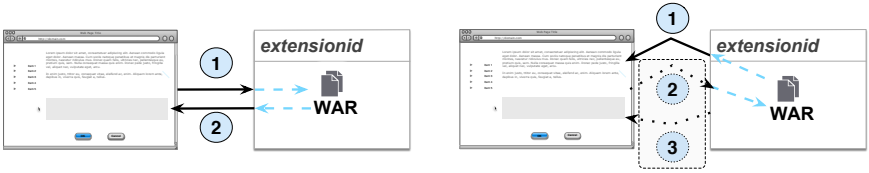
Power of extensions Firefox and Chrome provide their extensions with elevated privileges [41]. As such, the extensions have access to a vast amount of information, such as reading and modifying the network traffic, the ability to make arbitrary modifications to the DOM, or having the possibility to access a user's private information from the browsing history or the cookies. The extension models for both Firefox and Chrome allow extensions to read

and modify the DOM of the currently loaded web page [44, 26]. In addition to the aforementioned scenarios, some browser extensions like password managers, have access to sensitive data such as the user's passwords, which can include credentials to email accounts or social networks.

Detecting extensions Due to the increased power that browser extensions possess, they have been target for detection from web pages. Today, Chrome comes with a built-in ChromeCast extension [31], which has *Web Accessible Resources (WARs)*, public files which exist in the extension and can be accessible from the context of the web page. Web pages, such as video streaming pages, can then probe for the ChromeCast extension, and add a cast button which would allow to cast the video player to the connected ChromeCast. By doing this, the browsing experience of the user is improved. On the other side, a web page might want to prevent DOM modifications (e.g. by detecting ad blockers), prepare for an attack against the user of a browser extension with sensitive information (e.g. by performing a phishing attack [16]), or even to gain access to the elevated APIs the browser extension has access to [3]. With the possibility of detecting browser extensions by web pages, users can be tracked based on their installed browser extensions [22, 55, 53]. This motivates the focus of this paper on the problem of protecting browser extensions from detection attacks.

Probing attack Previous works [55, 53] have focused on *non-behavioral detection*, based on a browser extension's listed WARs. The WARs are public resources which can be fetched from the context of a web page using a predefined URL, consisting of a public extension ID (or *Universally Unique Identifier (UUID)*) and the path to that resource. With the predefined URL to fetch a WAR from an extension, a web page can mount a *probing attack*, designed to detect an extension by probing for WARs, since a response with the probed WAR indicates the corresponding extension is installed. This attack can be seen in Figure 5.1a where ① denotes the requests made by the attacker to probe for an installed browser extension. If the browser extension is in the browser context, the attacker will get a response consisting of the requested WAR (denoted by ②). This attack can be magnified by probing for a set of browser extensions' resources, thereby enumerating many or even all installed browser extensions.

Firefox defense against probing As the probing attack is possible when the URLs of a browser extension's WARs are fixed and known beforehand, Firefox implements a randomization scheme for the WAR URLs in their new browser extension model, *WebExtensions*. To make the probing attack



(a) Probing attack. (b) Revelation attack.

Figure 5.1: Schematic overview of the extension probing attack and extension revelation attacks. In the probing attack, a web page probes for the presence of an extension. In the revelation attack, the extension reveals itself to the attacker by injecting content in the web page.

infeasible, each browser extension is given a random UUID, as it *"prevents websites from fingerprinting a browser by examining the extensions it has installed"* [50]. The Chrome developers are considering to implement a similar randomization scheme, when they have *"the opportunity to make a breaking change"* [8].

Revelation attack Starov and Nikiforakis [56] show that browser extensions can introduce unique DOM modifications, which allows an attacker to determine which extension is active based on the DOM modification. In contrast to probing attacks, these attacks are *behavioral* attacks because they are based on detecting behavior of a browser extension via, e.g., DOM modifications.

This work puts the spotlight on *revelation attacks*, an important subclass of behavioral attacks, first introduced by Sánchez-Rola et al. in the context of Safari extensions [53]. The core of a revelation attack is to trick an extension to inject content via WAR URLs, thereby giving up its random UUID and provide a unique identifier of the victim. This attack is displayed in Figure 5.1b. When the WAR is injected by the browser extension (①), the URL with the random UUID becomes known to the attacker, who is monitoring changes to the web page through JavaScript. With the random UUID known, an attacker can construct WAR URLs to known resources by initiating a probing attack (② and ③). The probing in this case will be done for known unique resources for browser extensions which have the injected WAR as a resource, a set which can be precomputed by the attacker. Upon finding one of the resources in this precomputed set, the attacker can deduce which browser extension injected the information, allowing derandomization of browser extensions.

Starov and Nikiforakis [56] show that browser extensions can provide unique DOM modifications, allowing an attacker to determine the active

extension. However, it is not possible to uniquely identify the victim only based on the browser extensions [33]. This is the crucial part of the revelation attack: as the random UUID becomes known to the attacker, it enables them to uniquely identify the victim, based on that installed extension alone. Furthermore, in most cases these random WAR URLs can easily be used to derandomize an extension, indicating the UUID randomization does not prevent extension fingerprinting. In fact, since a malicious web page in many situations can not only figure out which browser extension has the random UUID, but also uniquely identify the user, the randomization of UUIDs amplifies the effect of a revelation attack rather than mitigating detection possibilities. The problem with randomization of UUIDs is known, and has been a topic of discussions among browser developers [1], as well as presented as an attack against a built-in browser extension which takes screenshots for Firefox [13]. Although this attack requires user interaction, it is important to study how many of the Firefox and Chrome extensions can be exploited without the need for user interaction.

Empirical studies To see how many extensions are susceptible to the revelation attack without user interaction, and how many web pages probe for extensions, we conduct several empirical studies.

- We download all extensions for Firefox and Chrome and determine that, in theory, 1,301 (≈94.41%) and 10,459 (≈89.91%) of the Firefox and Chrome extensions respectively that might inject content are susceptible to the revelation attack.

- We check how many of the extensions susceptible to the revelation attack actually reveal themselves, where the attacker model is a generic web developer with the ability to host a web page visited by the victim. While the victim is on the attacker web page, the attacker will attempt to make the installed browser extensions inject content to make them reveal themselves, with the hope of determining exactly which browser extensions are being executed based on the injected content. If the randomized token proves stable enough, the attacker may also use it to track the victim on the Web. This attacker model fits a wide range of possible attackers, from small and obscure web pages, to top-ranked web applications. To emulate this, we check how many extensions reveal themselves based on where the extension is defined to inject content, and whether the actual content on the web page matters, showing that 2,906 out of 13,011 (≈22.3%) extensions reveal themselves on actual pages.

- We visit the most popular 20 web pages for each of the Alexa top 10,000 domains, and find that 2,572 out of those 10,000 domains probe for WARs.

"Latex Gloves" mitigation approach In popular culture, crime scene investigators frequently use latex gloves to avoid contaminating a crime scene with fingerprints. In this work, our goal is to prevent that extensions leave any "fingerprints" that are detectable by an attacker web page, be it through a probing attack or a revelation attack. For this reason, we named our approach "Latex Gloves" for extensions.

A key feature of our approach is its generality. The mechanism is parametric in how whitelists (or, dually, blacklists) are defined, with possibilities of both web pages and extensions having their say. Extension manifest files can be used for automatic generation of whitelists already. While it might be suitable to let the advanced user affecting the whitelists, the goal is to relieve the average user from understanding the workings and effects of web pages and browser extensions. For the whitelist, which defines which extensions are allowed to reveal themselves to the web page, there are several options, each with its own benefits and drawbacks. For example, a mechanism similar to Google Safe Browsing [28] can be employed, where browser vendors can provide blacklists for our mechanism containing web pages known to perform extension fingerprinting. This would put the burden on the browser vendors to keep the blacklist up to date. Another option would be to allow web pages to specify a whitelist, similar to how a *Content Security Policy (CSP)* [58] is defined. Naturally, there is a big risk web pages would simply try to deny all extensions any access, greatly limiting a user's intentions. Another option is a simple interface that allows users to classify websites into sensitive (e.g., bank) and insensitive (e.g., news portal), so that it is possible to configure whether an extension is triggered on a(n) (in)sensitive website. Yet another option is an all-or-nothing policy: either all extensions are triggered on all insensitive websites or no extensions are triggered on any sensitive websites. This would keep interaction with the user to a minimum. Each option has advantages and disadvantages, and usability studies can help determine the most suitable alternatives.

Our vision is to have direct browser support for Latex Gloves. However, in order to aid evaluation of the general mechanism, we present a proof-of-concept prototype consisting of a Chromium browser modification, a Chrome extension and a web proxy. This prototype allows the whitelisting of those web pages that are allowed to probe for extensions, and the whitelisting of those extensions that are allowed to reveal themselves to web pages.

Contributions In this work, we present the first large-scale empirical study of browser extensions on both Firefox and Chrome based on the revelation attack, in order to determine how fingerprintable the browser extensions — and the users of browser extensions — are, in the presence of a random WAR URL scheme. Additionally, we propose a countermeasure based on two whitelists, defining which web pages may interact with which extensions and vice versa, thus allowing users to avoid being fingerprinted or tracked by untrusted web sites. We finally give some guidelines to avoid this security issue for browser developers.

The main contributions of this paper are:

Revelation attack on Firefox. We demonstrate how to derandomize Firefox extensions through revelation attacks (Section 4).

Empirical studies of Firefox and Chrome extensions. We present large-scale empirical studies of Firefox and Chrome extensions regarding revelation attacks (Section 4), where we determine how $\approx 90\%$ out of all extensions injecting content can be uniquely identified in spite of a randomization scheme, as well as evaluating how many extensions can be detected with a revelation attack, based on the attacker model.

Empirical study of the Alexa top 10,000. We report on an empirical study over the Alexa top 10,000 domains, with up to 20 of the most popular pages per domain to determine how widely the probing attack (Section 3) is used on the Web.

Resetting Firefox random UUID. We investigate the user actions required to reset the random UUID of a Firefox extension, in order to remove a unique fingerprint accidentally introduced by Mozilla, on the most prominent operating systems: Windows, Mac OSX and Linux.

Design of a mechanism against the two attacks. We give the design for "Latex Gloves" (Section 5), a mechanism against both probing and revelation attacks using whitelists to specify which web sites are allowed to interact with which extension's WARs, and which extensions are allowed to interact with which web sites.

Proof of concept prototype. We implement a proof of concept prototype (Section 6) consisting of a modified Chromium browser, a browser extension and a web proxy, all based on the whitelisting mechanism. Our prototype is evaluated (Section 7) against two known attacks (extension enumeration [55] and timing attack [53]).

Recommendations for browser developers. We use key insights from our empirical studies to give recommendations (Section 8) to browser developers for a browser extension resource URL scheme.

2 Background

An extension is a program, typically written in a combination of JavaScript, HTML and CSS. Browser extensions have become a vital piece in the modern browser as they allow users to customize their browsing experience by enriching the browser functionality, e.g. by altering the DOM or executing arbitrary scripts in the context of a web page.

JavaScript code in a browser extension can roughly be classified as *background pages* and *content scripts*. Background pages are executed in the browser context and cannot access the DOM of the web page. Instead, they are allowed to access the same resources as the browser, e.g. cookies, history, web requests, tabs and geolocation. However, in order to make use of these capabilities the user has to explicitly grant most of them.

Content scripts are files that is executed in the context of a web page. Although the content scripts live in isolated worlds, allowing them to make changes to their JavaScript environment without conflicting with the web page or any other content scripts, they have access to the same DOM structure as the main web content. As content scripts are executed in the context of the web page, the content scripts can read and modify the DOM of the web page the browser is visiting, as well as inject data such as images and other scripts into the web page [44, 26]. Content scripts can only use a subset of the extension API calls ("extension", "i18n", "runtime" and "storage"), neither of which need approval from the user. In case the content scripts need access to more privileged extension APIs, they can only access them indirectly by communicating with the background pages through message passing. As the access of the privileged API calls goes through the background page via message passing, the user must approve them upon installing the extension.

The structure of an extension is defined in a *manifest file*, called *manifest.json*, which is a mandatory file placed in the extension's root folder [46, 30]. The manifest file contains, among other things, which files belong to the background page, which files belong to the content script, which permissions the extension requires, and which resources can be injected into the web page. An example of a manifest file can be seen in Figure 5.2, which specifies the background page to be the JavaScript file `background.js` and the content script (`content_scripts`) to use the JavaScript file `content_script.js`, and be executed on all domains that matches the domain `example.com`. It defines two WARs (`web_accessible_resources`), which are resources that can be injected

```
{
  "manifest_version": 2,
  "name": "Example",
  "version": "1.0",
  "background": {
    "scripts": ["background.js"]
  },
  "content_scripts": [
    {
      "matches": ["*://*.example.com/*"],
      "js": ["content_script.js"]
    }
  ],
  "web_accessible_resources": [
    "images/img.png",
    "scripts/myscript.js"
  ],
  "permissions": ["webRequest"]
}
```

Figure 5.2: Example of a *manifest.json* file

into the web page from the content script. The path for the WARs is the path from the extension's root folder to the resources. The extension also asks for the permission `webRequest`, which indicates the extension's background page want the ability to intercept, block and modify web requests.

Browser extensions scope In the particular case of content scripts, browser extensions insert their JavaScript files in those web pages explicitly defined by the extension's developers in the manifest file. Concretely, there is a mandatory property named `matches` which indicate the web pages the content script should be injected into. URLs can be defined following a match pattern syntax, which is reminiscent of regular expressions, operating on a `<scheme>://<host><path>` pattern [18]. Background pages are not affected by the `matches` property. Instead, they remain idle until a JavaScript event such as a network request or message passing coming from an arbitrary content script, triggers their code, after which they return to an idle state.

Web Accessible Resources If an extension wants to inject a resource, such as an image or a script, into a web page, the recommended way is to make the resource "web accessible". WARs are files that exist in a browser extension

but can be used in the context of a web page. A browser extension must explicitly list all WARs through the `web_accessible_resources` property in the manifest file [50, 29].

WAR URLs are different for Firefox and Chrome: `moz-extension://<ext-UUID>/<path>` and `chrome-extension://<ext-UUID>/<path>` in Firefox and Chrome, respectively. In Firefox, `<ext-UUID>` is a randomly generated UUID for each browser instance, and is generated when the extension is installed [50]. However, for Chrome, `<ext-UUID>` is a publicly known 32 character string derived from the RSA public key with which the extension is signed, encoded using the "mpdecimal" scheme. WAR URLs in Chrome have the `<ext-UUID>` hardcoded as the "hostname" part. For both Firefox and Chrome, the recommended way of getting the URL of the resource is to use the built-in API, which is `browser.extension.getURL("path")` in the case of Firefox [45], and `chrome.runtime.getURL("path")` for Chrome [25]. Since Chrome extensions have a publicly known extension UUID, an attacker could enumerate all installed extensions which have WARs (See Section 3).

Browser profiles and extension UUIDs In Chrome and Firefox, data such as bookmarks, passwords and installed extensions is stored in a *browser profile* [49]. A browser installation may have several browser profiles, each with its own data. Because Firefox's extension UUIDs are randomized, the same extension installed in multiple browser profiles will have a different UUID for each profile. In Chrome, which uses fixed extension UUIDs, an extension installed in multiple browser profiles will use the same extension UUID in each profile.

3 Probing attack

When probing for an extension, JavaScript running in a web page tries to determine the presence of a browser extension in the browser in which the web page has been loaded.

One way of performing the extension probing is by requesting a browser extension's WARs through the publicly known URLs for these resources. This is schematically shown in Figure 5.1a where ① denotes the request made by the web page to probe for a browser extension's WAR. A successful response to this request (denoted by ②) indicates the presence of the extension to which the WAR belongs.

Probing for an extension in itself does not mean an attack is taking place. It is not an attack if, e.g., Google probes for the ChromeCast extension on YouTube.com since this is the extension developer who probes for their own extension. However, if it is not the extension developer who is probing for

Table 5.1: Alexa top 10,000 domains probing for Chrome extensions. Note that a domain may appear in several rows and/or columns.

	same domain	other domain	YouTube
top frame	185	15	4
sub frame	36	2,399	2,277
Total	2,572		

the browser extension, but rather a third party with the intent of discovering installed extensions to, e.g., increase the entropy for browser fingerprinting, the probing becomes a *probing attack*. Attackers may use a probing attack to detect the presence of any of the known browser extensions, thereby enumerating the installed browser extensions in a victim's browser.

Sjösten et al. [55] explore the Alexa top 100,000 domains to examine how many of them probe for WARs on their front page and their reasons for doing so. Their research shows that web developers and their applications may probe for WARs for legitimate reasons. They find only 66 domains, none in the top 10,000, and surmise that this is caused by the technique not being widely known.

We repeat the experiment using a different detection method, in order to study how this problem has developed over time. Instead of the top 100,000, we limit ourselves to the top 10,000, but perform a deeper study by visiting up to twenty of the most popular web pages on each domain. We also gather metrics that indicate whether the probing is due to a third-party web origin, or whether it originates from the domain itself.

Setup We use a modified version of Chromium 63.0.3239.84, which allows us to monitor requests for WAR URLs from a Chrome extension, as described in Section 6. The entire process is automated using Selenium 3.8.1.

When visiting a web page, we wait for up to 10 minutes for the web page to load. Once loaded, we wait an additional 20 seconds in order for any JavaScript on the web page to execute.

During this time, a custom browser extension monitors any requests made towards `chrome-extension://` URLs and logs them. In addition to the WAR URL itself, we also log whether the request came from the parent frame or a sub frame, as well as the web origin from which the request occurred.

Results Starting from the list of top 10,000 domains according to Alexa, we queried Bing to retrieve the most popular twenty pages per domain. Bing returned 180,471 URLs for 9,640 domains. We further disregard domains for which Bing did not return any results. Of the 180,471 URLs, we were able to visit 179,952 spread over 9,639 domains.

An overview of the results is shown in Table 5.1. In total, out of the 10,000 domains, 2,572 probed for 45 different extensions from either the top frame or a sub frame. Of the domains that requested a WAR from the top frame, 185 had not redirected the browser to another domain, while 15 did. In the latter case, 4 redirected to YouTube.com. In the other cases, WARs were requested from a sub frame: 36 domains loaded the sub frame from the same domain, while 2,399 loaded it from a third-party domain. Strikingly, 2,277 of those sub frames originated on YouTube.com where most of these requests were probing for the ChromeCast browser extension.

Our results are different from Sjösten et al. [55], which may be attributed to the different methodology or an increase in extension probing. No matter the reason for the discrepancy, probing is both common and relevant. Although YouTube.com probing for ChromeCast is not a probing attack, most of the remaining extensions being probed for (e.g. popular extensions such as AdBlock [10], AdBlock Plus [2] and Ghostery [6]) constitute probing attacks.

4 Revelation attack

In an effort to eliminate the extension probing attack, Mozilla implemented a randomization scheme in its extensions' UUIDs. Since each extension is given a random UUID upon installation, it is impossible to compose the URL of a WAR to launch a probing attack without knowing that random UUID. However, it is possible for an attacker to learn the random UUID of an extension through an extension revelation attack.

In an extension revelation attack, JavaScript running in a web page tries to determine the presence of a browser extension by monitoring the web page for new content which references WARs. Although any introduced DOM modification might uniquely identify an extension [56], an injected WAR URL contain a unique UUID for each profile, which in turn can be used to track users. Also, due to the nature of the WAR URLs, a vast majority of all extensions injecting content with WAR URLs can still be uniquely identifiable, in spite of the randomization scheme, indicating it might make more harm than good.

Figure 5.1b displays the revelation attack. JavaScript in a web page detects that a browser extension has inserted a reference to a WAR (①), and can now deduce the presence of this extension.

In the case of Firefox, the revelation attack reveals a WAR URL, which consists of a random UUID and a path component. While the random UUID itself is insufficient to derandomize the extension, it can be used as a basis for a probing attack (② and ③).

It is important to realize that a probing attack may not be needed in order to derandomize Firefox's random UUIDs. In Section 4.1, we show that the path component of the WAR URL, which is not randomized in Firefox, contains enough information to derandomize an extension's random UUID in many cases. In addition, because an attacker can retrieve the content of a WAR and compute a hash over it, it is possible to derandomize an extension even if the full WAR URL is randomized.

Furthermore, because the random UUID is unique per "browser instance", it can also be used as a unique fingerprint to deanonymize web users through the revelation attack. As we show in Section 4.2, it is not trivial to remove this unique fingerprint from the browser.

The developers of Google's Chrome browser have expressed interest in implementing a similar randomization scheme [8]. In Section 4.3, we study the impact of adopting this randomization scheme on Chrome extensions. The results of both Section 4.1 and Section 4.3 are summarized in Table 5.2, where "Path" is the amount of extensions that can be derandomized based on the path, "Hash" based on the sha256 hash digest of the content of the WARs, and "Path $\cup$ Hash" the union of those sets.

Finally, in Section 4.4 we perform an empirical study of all available Firefox and Chrome extensions to determine how many of them are affected by the revelation attack, revealing themselves and their users to attackers simply by visiting an attacker's web page.

4.1 Derandomizing Firefox extensions

Since Firefox employs random UUIDs, the enumeration techniques presented in [55, 53] cannot be used. Instead, the extension must reveal itself for an attacker to get hold of the random UUID. In order to derandomize a Firefox extension, the extension must meet the following criteria. First, the extension must have at least one defined WAR, indicating it might inject a resource. Second, the extension must make a call to either of the functions `browser.extension.getURL`, `chrome.extension.getURL` or `chrome.runtime.getURL`, which are functions that, given an absolute path from the root of the extension to the WAR, will return the full `moz-extension://<ext-UUID>/<path>` URL. For the rest of this section, we will group those functions together as `getURL()`. Although these API functions are executed in the context of the extension, i.e. they cannot be called directly from the web page, if the extension injects the WAR in this manner, the random UUID will be revealed to the web page as part of the WAR URL. If this happens, and the attacker gets the UUID, then how many extensions can be uniquely identified based on the injected WAR URL?

Table 5.2: Breakdown of the uniqueness detectability for browser extensions, assuming a randomized schema with the ability to probe.

	Extensions total	Path	Hash	Path ∪ Hash
Firefox	1,378	1,107 (80.33%)	1,292 (93.76%)	1,301 (94.41%)
Chrome	11,633	7,214 (62.01%)	10,355 (89.01%)	10,459 (89.91%)
Total	13,011	8,321 (63.95%)	11,647 (89.52%)	11,760 (90.39%)

To determine this, we scraped and downloaded all free Firefox extensions from the Mozilla add-on store [47]. The extensions are valid for Firefox 57 and above, as it is the first Firefox version to only support WebExtensions [51], indicating all will receive a random UUID when installed. The scrape was done on February 23, 2018, giving us 8,646 extensions. All of these extensions were unpacked, and their manifest file examined for the `web_accessible_resources` key, resulting in 1,742 extensions having at least one defined WAR. The mere presence of a WAR in an extension does not mean that this resource will ever be injected. We took the 1,742 extensions with declared WARs, and checked how many of them call a `getURL()` function, as this will construct the WAR URL to be injected to the web page. This resulted in a total of 1,378 extensions, indicating ≈79.10% of all Firefox extensions with declared WARs can reveal their random UUID.

Having access to only the random UUID is not sufficient. The path component present in a WAR URL can give away the identity of the extension, if there is a mapping between a path and the corresponding extension. Out of the 1,378 extensions that call a `getURL()` function, 1,107 extensions provide at least one unique path, i.e. the full path to a resource. Aside from the WAR URL, a potential attacker also has access to the contents of the WAR. We investigated the contents of the extensions' WARs to determine how unique they are by calculating a hash digest over the contents. A total of 1,292 browser extensions have a unique digest when hashing their WARs, where a different hash digest indicate a difference in content between the WARs of the different browser extensions. We then took the union of the two sets of browser extensions with at least one unique path and a unique digest, yielding a total of 1,301 browser extensions to be uniquely identifiable. Although only ≈15.05% of all extensions can be uniquely identified, it is ≈94.41% of all extensions that have the possibility to inject a WAR.

4.2 Resetting Firefox's random UUID

For Firefox, each UUID is "randomly generated for every browser instance" [50]. However, it is not clear what "browser instance" means in this setting. In order to determine when the random UUID of a browser extension is being reset in Firefox, we tried different approaches on three operating

systems: Windows 10, Linux (Debian) and Mac OSX. The approaches were restarting, updating and re-installing the browser, updating and re-installing the extension, switching the browser tab to incognito mode and clearing the cache and cookies of the browser. The result can be found in Table 5.3, and for the rest of this subsection, we will briefly cover the differences between the operating systems.

None of the operating systems change the internal UUIDs upon restarting the browser, indicating "browser instance" from the documentation does not mean "started browser process". When re-installing the browser, the default behavior for the Windows 10 installer is to reset the standard options, which includes removing the old browser extensions. As this would force a user to re-install the browser extensions, each browser extension would get a new random UUID. However, a user has the option of not resetting the standard options, along with not removing the old browser extensions. Hence, uninstalling Firefox on Windows keeps all settings, and it is up to the user to decide to keep or remove them when re-installing the browser. This is not the case for Linux and Mac OSX. For both operating systems, it is up to the user to manually remove the profile folder (default is `.mozilla` in the home folder for Linux, and `Library/ApplicationSupport/Firefox` in Mac OSX) in order to remove the old browser extensions upon re-installing the browser, as they are not prompted about a default option of resetting the standard options.

For all operating systems, the UUID was regenerated when reinstalling the extension, given that the browser was restarted between uninstalling and reinstalling the extension. If the browser was not restarted, the profile file containing the data would not change, giving the new installation the same UUID.

On all platforms, clearing the profile (i.e. removing the actual profile folders) would force a user to re-install all extensions, which means they would get a new random UUID.

4.3 Derandomizing Chrome extensions

As Chrome does not employ random UUIDs, the technique presented by Sjösten et al. [55] still works. However, as Chromium developers plan to employ random UUIDs, we performed the same experiment as for Firefox. In total, we scraped 62,994 free extensions from the Chrome Web Store [24]. Out of those, 16,280 defined `web_accessible_resources` with at least one corresponding WAR. The amount of extension that called either `chrome.runtime.getURL` or `chrome.extension.getURL` was 10,764. We also checked the extensions that called `chrome.runtime.id` (728 extensions), which return the extension's UUID, and the ones that hardcoded their extension UUID into a resource URL (141

Table 5.3: Actions which result in UUID regeneration for each of the major operating systems. "Yes" or "No" means that the action did or did not cause UUID regeneration respectively. Notes: (*) Firefox's installer on Windows prompts the user to reset settings and remove extensions, which is enabled by default, whereas for Linux and Mac OSX ($^+$), the default is to keep all settings.

		Linux	Mac OSX	Windows
Restarting browser		No		
Updating browser		No		
Re-installing browser		No$^+$		Yes*
Updating extension		No		
Re-installing extension	w/ browser restart	Yes		
	w/o browser restart	No		
Incognito mode		No		
Clearing cache and cookies		No		
Clearing the profile		Yes		

extensions), with the assumption they will change to call `getURL()` if Chrome adopts random UUIDs. With this, the total amount of detectable extensions would be 11,633 extensions, which corresponds to ≈71.46% of all extensions with at least one WAR declared. Assuming random UUIDs for Chrome, we must check if a path can uniquely identify an extension. We applied the same uniqueness procedure as in Section 4.1, finding 7,214 extensions being unique without the need for any content hashing. When hashing the content of the WARs, we got a total of 10,355 browser extensions, and the union of those two sets yield a total of 10,459 uniquely identifiable browser extensions. While only being ≈16.60% of all extensions, it is ≈89.91% of all browser extensions that have the possibility to inject a WAR.

4.4 Extensions revealing themselves to web pages

As browser extensions can inject WARs into a web page to allow it access in the domain of the web page, the WARs are visible to JavaScript executed in the origin of this web page. A web page can scan for these WARs in order to reveal installed browser extensions, as well as to deanonymize the visitor: from the WARs, an attacker can infer the installed extension, and from Firefox browser extensions' random UUIDs, the attacker can identify the visitor.

For this experiment, we consider all 8,646 Firefox extensions, but are also interested in the 62,994 Chrome extensions. As Chrome are considering random UUIDs, the findings are relevant to their future development plans.

Setup We use Selenium 3.9.1 with Firefox 58.0.1 and Chromium 64.0.3282.167 to automate the process.

For each browser extension, we visit a web page through mitmproxy 2.0.2 [21] with a custom addon script. In order to be able to manipulate web pages served over HTTPS, both Firefox and Chromium were configured to allow untrusted SSL certificates.

The mitmproxy addon script injects a piece of attacker JavaScript code in the web page which walks through the HTML tree and extracts any attributes that contain `chrome-extension://` or `moz-extension://` present in the web page. In addition, because the CSP may prevent the execution of injected JavaScript, the mitmproxy addon script disables CSP if present.

Because browser extensions may inject content only after a while, the attacker script also installs a mutation observer which repeats the scan every time a change to the web page is detected. With this setup, we can detect the injection of WARs at any point in the web page's lifetime. For every page visit, we wait for up to one minute for the page to load before aborting that page visit. When a page is successfully loaded, we wait for five seconds to let any JavaScript on the page run its course.

Dataset extensions Because of the way Firefox extensions work, we only consider those extensions which seemingly make a call to `getURL()` and which have web accessible resources. After this filtering step, 1,378 out of the 8,646 Firefox extensions remain for our study.

Similarly for Chrome, we retain 11,633 out of the total 62,994 Chrome extensions.

Dataset URLs These 13,011 extensions (1,378 Firefox + 11,633 Chrome) will only execute on a web page if the URL matches the regular expressions in their manifest file. For instance, an extension which lists `http://example.com/*` in its manifest file, will not execute when visiting, e.g., `http://attacker.invalid/index.html`. Extensions can only reveal themselves when they are executing on a web page they were designed for, e.g by checking for the presence of a certain keyword in the URL. Because of this, it is important to visit the right URLs.

To determine the set of URLs we should visit for a particular extension, we make use of the CommonCrawl dataset [5]. This dataset contains data about ≈4.57 billion URLs from a wide variety of domains. From the 13,011 extensions, we extracted 24,398 unique regular expressions and matched them against the CommonCrawl dataset using the regular expression matching rules specific to the manifest file specification. For each regular expression, we only consider the first 100 matches. For each extension, which can have

many regular expressions in its manifest, we combine all matching URLs and take a random subset of maximum 1,000 URLs. In total we obtained 506,215 unique URLs from the CommonCrawl dataset that match the regular expressions from the extensions' manifest files. We call this set of URLs the "real" URLs.

From the "real" URLs, we derive two extra sets of URLs by considering that an attacker can host a copy of a real web page on a different web host. For instance, the web page at `http://www.example.com/abc` could be hosted on an attacker-controlled `http://www.attacker.invalid/abc`. We call this cloned set of "real" URLs, where the hostname has been replaced by `attacker.invalid`, the "attackerhost" URLs.

Extensions with more fine-grained regular expressions may require the attacker to register a domain in DNS. For instance, a regular expression `http://*.com/abc` does not match the `attacker.invalid` domain which we assume is under attacker control. Therefore, we also consider a URL set where the hostname in each URL has been replaced by a hostname with the same top-level domain, but with an attacker-controlled domain name. For instance, for `http://www.example.com/abc` we also consider `http://www.attacker.com/abc`. Naturally, we chose a domain name of sufficient length and consisting of random letters, to make sure it was not registered yet. We call this cloned set of "real" URLs, the "buydns" URLs.

In addition to the real CommonCrawl URLs which match the regular expressions, we also generate URLs based on those regular expressions by replacing all "*" characters with "anystring". For instance, we generate the URL `http://*.example.com/anystring` for the regular expression `http://*.example.com/*`. We call this set of URLs the "generated" URLs.

Dataset web page content Aside from expecting a certain URL, an extension may also depend on certain HTML elements, HTML structure or particular text present on a visited web page. To determine whether this is the case, each web page visited through a URL in the "real" URLs set, as well as the derived "attackerhost" and "buydns" sets, is also visited with all content removed. We visit each of these URLs twice: once with the real content, and once serving an empty page instead of the real content. For the "generated" URL set, we only serve empty pages, since there is no way to determine what type of content should be present on such a URL. A known practice from previous work is to use "Honey Pages", empty pages that create the DOM content of a web page dynamically, based on what the extension is querying [56, 35]. While "Honey Pages" can provide useful information to, e.g., find malicious extensions, some extension behavior can be difficult to trigger in an automated way, as it may not be only nested

Table 5.4: Breakdown of Chrome and Firefox extensions, indicating which how many extensions revealed themselves, how many didn't, and how many we were unable to analyze (broken).

	Revealed	Broken	Not revealed	Total
Chromium	2,684	412	8,537	11,633
Firefox	222	150	1,006	1,378
Total	2,906	562	9,543	13,011

Table 5.5: Breakdown of extensions that reveal themselves. The number between brackets indicates the amount of potentially affected users, assuming no overlaps.

	Content-dependent						Any content						Total	
	"real" URL		"attackerhost" URL		"buydns" URL		"real" URL		"attackerhost" URL		"buydns" URL			
Chromium	289	(3,227,947)	217	(2,680,324)	2	(110)	1,281	(17,301,512)	891	(14,601,057)	4	(1,172)	2,684	(37,812,122)
Firefox	49	(39,780)	19	(75,940)	0	(0)	138	(649,236)	16	(27,082)	0	(0)	222	(792,038)
Either browser	338	(3,267,727)	236	(2,756,264)	2	(110)	1,419	(17,950,748)	907	(14,628,139)	4	(1,172)	2,906	(38,604,160)

Table 5.6: Breakdown of revealing Chrome and Firefox extensions, indicating how many of the extensions revealing themselves that could be uniquely identified, either through the path, through the content of the WARs, and the union of those sets.

	Revealed	Unique path	Unique hash	Unique path $\cup$ hash
Chromium	2,684	2,063	2,603	2,606 (97.09%)
Firefox	222	198	216	216 (97.30%)
Total	2,906	2,261	2,819	2,822 (97.11%)

DOM structures, but also events an extension acts on. In this light, "Honey Pages" may not be representative of the operation of actual web pages. As we are interested in whether web pages would be able to employ a revelation attack with their current structure, our experiments are not using "Honey Pages". Instead, we look at the current interaction between web pages and extensions, providing an indication of how many extensions that are currently vulnerable. For the best coverage, it would be interesting to combine our results with "Honey Pages", but we leave that for future work.

Results The results of the experiment are shown in Tables 5.4 to 5.6.

Out of 13,011 extensions, 2,906 revealed themselves on actual pages. We suppose this behavior is intentional, but it can be abused by the website owners to track the users. 9,543 did not reveal themselves and 562 could not be used in our experiment because of issues with the third-party software we used in our setup (Selenium, browser-specific or addon-specific issues).

The other remaining 9,543 extensions which call `getURL()` and have WARs, seemingly do not inject any WARs into the web page, or probably more accurately: we did not trigger the correct code path in the extension that results in a WAR being injected into a web page. Analyzing these remaining

extensions via "Honey Pages" could reveal they also inject WARs under the right circumstances, although none of the web pages we visited would make them inject content. Nevertheless, our analysis of web page and extension interaction succeeded in exposing 2,906 extensions which reveal themselves on web pages.

Of these 2,906 extensions triggered by real URLs, 2,330 depend only on the URL of the web page visited, and do not depend on the content of that page, since they execute even when the presented web page is empty. Moreover, out of the 2,906 extensions that reveal themselves on the right URLs, 1,149 can be tricked into executing on attacker-controlled web pages. Only for 6 Chrome extensions (but none of the Firefox extensions) does the attacker potentially have to register a new domain to host the malicious website on.

Moreover, for 1,149 of the extensions that can be tricked to execute on an attacker URL, 911 do not depend on the page content, further easing the life of the attacker.

The numbers between brackets in Table 5.5 denotes the total number of extension users affected by these revealing extensions. Assuming there are no overlaps between the users of the revealing extensions, a total of 38,604,160 web users are vulnerable to the revelation attack through their installed extensions. For the 792,038 affected Firefox users, this means that they are uniquely identifiable through the unique fingerprint exposed by their revealing extensions. The 37,812,122 affected Chrome users do not suffer from this issue at this point in time, but would also be uniquely identifiable if the Google Chrome developers adopt Firefox's UUID randomization scheme.

Furthermore, as seen in Table 5.6, out of the 2,906 revealing extensions, 2,261 have at least one unique path, and 2,819 have at least one WAR with a unique content. The union of those sets contains 2,822 extensions, indicating that 97.11% of the 2,906 (97.09% of Chrome and 97.30% of Firefox) revealing extensions can be uniquely identified.

5 Mitigation design

From the introductory example in Section 1, it is clear that there is a legitimate use-case for being able to probe for WARs. Extensions that want to be detectable through their WARs, e.g. ChromeCast, would become dysfunctional if probing for WARs was blocked in general. Therefore, preventing the extension probing attack through a blanket ban on extension probing, is not an option.

In similar vein, preventing extensions from revealing themselves to web pages is also not an option. The data from Section 4.1 implies that many extensions may inject content into a web page, and could become dysfunctional if this functionality was no longer available. Extensions ill intent on revealing themselves may be unstoppable, and we consider them out of scope, only focusing on those extensions that accidentally reveal themselves.

Our experiments show the different ways through which extensions reveal themselves by injecting content. From an unrandomized WAR URL injected in a page, as is the case for Chrome extensions, it is trivial to extract the UUID to determine the installed extension. As is shown in Table 5.2, from a WAR URL where just the UUID has been randomized and probing is possible, as is the case for Firefox extensions, we can deduce the installed extension with a 80.33% accuracy by considering only the path of the URL, and the paths tied to each extension. Similarly, we would be able to deduce the installed extension with a 93.76% accuracy by only looking at the contents of the resources tied to the extensions, and combining the two approaches, we can deduce the installed extension with a 94.41% accuracy. Similarly, we detect Chrome extensions with a 62.01% accuracy based on the path, 89.01% accuracy based on the content of the resource, and 89.91% accuracy when we combine the path and the content.

Without breaking the intended functionality provided by existing extensions, we cannot prevent extension probing attacks and extension revelation attacks in general.

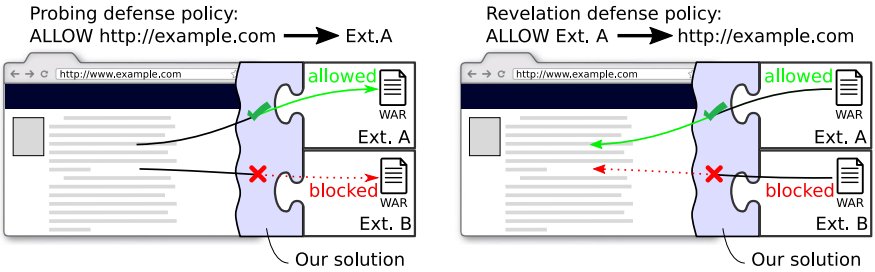
Our envisioned solution, which we call "Latex Gloves" since the goal is to prevent extensions from leaving fingerprints, is depicted in Figure 5.3.

We prevent extension probing attacks (Figure 5.3a) by allowing a whitelist to specify a set of web pages that may probe for each individual extension.

For instance, YouTube.com may be allowed to probe for the ChromeCast extension, so that the extension's functionality can be used with YouTube videos. In that case, a request for a WAR in the ChromeCast extension will be allowed by the policy. However, when the same WAR is requested by another web page, such as attacker.com, the request is blocked. Similarly, if YouTube.com would request a WAR from another extension, e.g. AdBlock, it would be blocked with this particular policy.

We prevent extension revelation attacks (Figure 5.3b) by allowing a whitelist to specify a set of web pages on which each extension is allowed to execute.

For instance, the AdBlock extension may be allowed to run on example.com. In that case, when example.com is visited, the AdBlock extension can remove any advertisements from the page. However, the same extension



(a) Probing defense (b) Revelation defense

Figure 5.3: Concept design of our proposed defenses for the extension probing and revelation attacks. Our solution mediates access from the web page to the extension WARs for the probing defense, and from the extensions to web pages for the revelation defense. In each case, access is mediated based on a specified policy.

may be disallowed from running on a website which is trusted by the whitelist policy, thereby not interfering with the revenue stream of that website. Similar to the probing defense example, the policy here also blocks other extensions from executing — and thereby potentially revealing themselves — on example.com.

Conceptually, the policies for both defenses can be visualized in a matrix, with extensions and web origins as rows and columns respectively. Each element in this matrix would then indicate whether access is allowed between the extension and the web origin.

However, such a matrix would make the assumption that policies for the probing and revelation defenses cannot conflict, which is not necessarily the case.

For instance, consider a configuration where AdBlock is installed, and a banking website bank.com, which is trusted by the whitelist policy. Because this trust, bank.com should be allowed to probe for AdBlock. However, due to the sensitive nature of the data on bank.com, the whitelist policy does not allow AdBlock to operate on the bank.com web pages, although AdBlock want to execute on every web page.

This conflict between the policies for a particular web origin and extension illustrates the need for separate whitelisting mechanisms for both the probing and revelation defenses.

6 Proof of concept implementation

Our prototype implements defenses against both the extension probing and extension revelation attacks as a proof of concept. Because changing

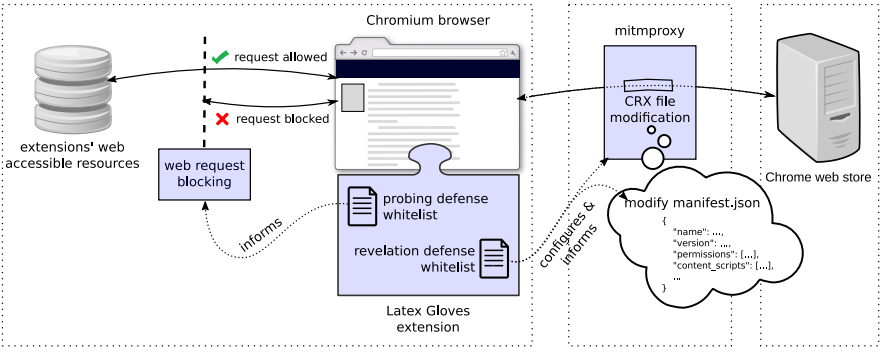


Figure 5.4: Overview of the prototype implementation of our proposed defenses: a modified Chromium browser with the Latex Gloves extension and mitmproxy.

browser code can quickly get very complicated, we opted to implement only the core functionality in the actual browser code, while the bulk of our prototype is implemented separately as a browser extension and a web proxy. For adoption in the real world, the full implementation should of course be embedded in the web browser's C++ code. However, our proof of concept implementation still allows to test the effectiveness of our solution. For simplicity, the proof of concept is designed to allow a security-aware end user to arbitrarily modify the whitelists. While this is not something one should assume an arbitrary user would do, we deem it to be good in order to show the functionality of the whitelisting mechanisms. In a full implementation, the end user should be queried as little as possible.

As depicted in Figure 5.4, our prototype implementation consists of three components: a slightly modified Chromium browser, a browser extension named "Latex Gloves" and a web proxy based on mitmproxy. Our modifications to the Chromium 65.0.3325.181 code consist of nine lines of code spread over four files. The patches to Chromium, as well as binary packages compiled for Ubuntu 16.04, our browser extension and our addon script for mitmproxy 3.0.4 are available upon request to the authors.

6.1 Preventing the probing attack

Chrome extensions can use the webRequest API to observe, modify and block requests from web pages. The requests that an extension can observe through the webRequest API, include requests with the `chrome-extension://` scheme. However, requests to `chrome-extension://<ext-UUID>` URIs where `<ext-UUID>` is not its own extension ID, will be hidden. Even though requests to non-installed extension resources, or to `chrome-extension://` URIs with an

invalid extension ID are hidden from observation with the webRequest API, those URIs are replaced by `chrome-extension://invalid` internally.

Our prototype needs the ability to monitor requests to all `chrome-extension://` URIs, even for other installed extensions, non-installed extensions or invalid extension IDs. In addition, we also want to avoid that Chromium replaces the URI with `chrome-extension://invalid`, since we are interested in the originally requested URI.

To achieve this, we modified the Chromium source code and changed just two lines of code in two files. First, we disable the check that determines whether the extension ID of the requested URI matches that of the extension observing the request. Second, we disable Chromium's behavior of replacing invalid `chrome-extension://` URIs.

The remainder of this part of the prototype is implemented as a browser extension which uses this modified webRequest API. Requests to all `chrome-extension://` URIs are monitored by the extension and matched against a predefined but customizable whitelist. The whitelist maps a web origin O to a list of allowed extension IDs L. When the browser visits a web page located in the given web origin O, the extension checks any requested `chrome-extension://` URIs and determines whether they target an extension in L. In case of a match, the request is allowed, otherwise it is canceled. In the latter case, it will appear to the web page as if the requested resource is not accessible, whether the extension is installed or not.

6.2 Preventing the revelation attack

By design, Chrome extensions can specify which URLs they want to operate on, by listing those URLs in the `permissions` and `content_scripts` properties of the *manifest.json* file. Restricting the list of URLs on which an extension is allowed to operate, would help prevent the extension revelation attack on arbitrary attacker pages, since the extension would not execute on those pages, and thus not reveal itself. However, this whitelist of URLs is at the discretion of the extension developer and cannot easily be altered by the whitelist policy provider.

Our implementation, schematically depicted on the right side of Figure 5.4, exposes the whitelist on which URLs the extension operates to the whitelist policy provider, allowing the restriction of the set of URLs on which the extension operates. Instead of implementing new functionality in the browser to modify this whitelist, and then exposing it to our browser extension, we decided to modify the browser extension CRX [19] files, which are packaged and signed versions of browser extensions, "in flight" when they are installed or updated from the Chrome web store.

Because extensions from the Chrome web store are signed with a private key, which we cannot obtain, we modified the Chromium browser to not strictly verify an extension's signature. This modification consists of six lines of code in a single file, and disables signature verification on both version 2 and 3 of the CRX file format. It is important to note that, for a real-world implementation, this should not be done, but rather have the full mechanism implemented in the browser. We only use this to show and evaluate the core whitelisting mechanism in the proof of concept prototype.

Since the browser no longer verifies CRX signatures, we are free to modify web traffic between the browser and the Chrome web store, and can update the manifest files in extensions' CRX files "in flight" and restrict the `permissions` and `content_scripts` properties according to the wishes of the whitelist. This CRX rewriting process is implemented in a web proxy as a mitmproxy addon script.

When the policy changes the hostname whitelist associated with an extension, the new whitelist is communicated to the proxy. When the auto-update process in the browser queries the Chrome web store whether the extension has been updated, we inform the browser that a new version exists. The browser then downloads the new version of the extension from the Chrome web store, which gets rewritten by our mitmproxy addon script, and includes the new whitelist.

Taking over the extension auto-update process for our proof of concept prototype in this manner, requires us to make more frequent changes to the version number of an extension than the extension's developer would. Because of the way the versioning system works, we need to keep track of a parallel versioning scheme that is only visible between the browser and the proxy. The details of this process are too technical to detail in this paper, but require us to change the `version` property of the manifest file in addition to the `permissions` and `content_scripts` properties.

By default, the Chromium auto-update process can take up to seven days, which we deem too infrequent to be of practical use in our proof of concept. An optional modification of one line of code in one file of the Chromium source code changes this update interval to five seconds, so that updates to the policy whitelist are implemented more promptly.

In addition, it should be noted that the original extension update mechanism will prompt the end user whenever the extension requests additional permissions compared to the previous version. Our proof of concept implementation does not alter this default behavior.

6.3 Discussion and future work

Our prototype implementation is a proof of concept, showing that it is possible to use whitelisting policies to defend against extension probing and revelation attacks. As mentioned before, an actual production-quality implementation of these defenses would require more changes to browser code and result in better performance and a nicer user experience with regards to e.g. the user interface.

A real-world implementation in the browser would not need to rewrite the extensions on the fly, and would not have to disable security checks. Similar to how the browser checks if, e.g., a WAR should be allowed to be injected, the browser can check if the extension should be allowed to execute on any given domain.

Recently, Google released the plan to allow end users to restrict the host permissions for an extension [7], indicating the core mechanism for modifying browser extension behavior within the browser is possible, and something which can be used to control the extension whitelist. In this case, the browser extension can provide a whitelist which can be modified without the need to re-install the extension.

It is also crucial for a real-world implementation to not have an early-out mechanism, which is what was exploited in the timing attack presented by Sánchez-Rola et al. [53], and subsequently removed [20]. In the situation an attacker is allowed to probe for an extension, and that extension is present, an early-out from the whitelisting mechanism during a probing attack would allow for the attacker to measure the elapsed time, and deduce whether the request was blocked based on the whitelist. If an attacker knows the time it takes to get a response from an installed extension which they are allowed to probe for, and an extension which is blocked by the whitelist, the attacker can, for each negative probing attempt, deduce which extensions that are not installed, and which that are blocked based on the whitelist.

For our prototype, we made the rather arbitrary choice to limit whitelists to web origins and hostnames in the probing and revelation defense respectively. While these choices serve us well for a proof of concept, it could prove interesting to refine these whitelists to use e.g. regular expressions on URLs instead.

Additionally, for the probing defense, when a web page contains an embedded subframe, we disregard the web origin of the subframe and enforce the whitelist associated with the web origin of the main frame. Our prototype is very well capable of applying a different whitelist for the subframe, in case the end user would wish to do so. However, we regarded this particular refinement of the prototype as out of scope for a proof of concept implementation.

In our proof of concept implementation, only the end-user can specify policy whitelists for both the probing and revelation defenses. In a production implementation, one should consider a system where both web applications and browser extensions can suggest a policy, which the end-user could then refine or even override. Another possibility is to have a system similar to Google Safe Browsing [28], keeping the user interaction to a minimum.

Finally, our prototype implementation displays information to the user about which extensions are being probed for on any visited web page. We do not display similar information regarding revelation attacks. We also consider these visual markers to be out of scope to prove the functionality of the concept.

7 Evaluation

We have evaluated the functionality of our proof of concept implementation to ensure that it works as intended. Using the data from Sections 3 and 4, we randomly selected and visited several dozen web pages that perform probing attacks, and also visited our attacker web page with the top ten (Chrome) extensions that reveal themselves on any web page with any content. As expected, our proof of concept implementation stops both the probing attacks and revelation attacks.

We also perform two evaluations against known old attacks, the enumerating probing attack presented by Sjösten et al. [55] (Section 7.1) and the enumerating timing probing attack presented by Sánchez-Rola et al. [53] (Section 7.2).

7.1 Enumerating probing attack

We visited two known web pages that employed the enumerating probing attack [54, 32] twice: the first time with an unmodified Chromium browser, and the second time with the modified Chromium browser and with our browser extension installed. We used browser extensions which we know can be detected both times: AdBlock [10], Avast Online Security [4], Ghostery [6] and LastPass [39]. When visiting with the modified Chromium browser with our browser extension, we set the policy to a "block all" policy, meaning we expect no WARs to be accessible to the web page.

As expected, with our unmodified Chromium browser, the probing attack was successful against all four extensions. Note that although the database was last updated in December 2016 for [54], it could still detect the popular extensions, which might indicate browser extensions do not change internally very often. Using our proof of concept implementation, the probing

Table 5.7: Enumeration timing probing attack.

	Chromium 53.0.2785.135	Patch	Patch + Extension
<realExtUUID>/<realPath>	8.53ms	9.67ms	8.95ms
<realExtUUID>/<fakePath>	12.59ms	9.71ms	9.17ms
<fakeExtUUID>/<fakePath>	7.86ms	10.16ms	9.3ms

attacks failed for all extensions. Although the execution time increased significantly, due to the handling of over 11,000 requests for our JavaScript code in the browser extension, we note that this is something that will improve if the mechanism is fully implemented in the source language of the browser. We also set policies to allow for the probing of each extension, one at a time, indicating that the overall idea explained in Section 5 is sound.

7.2 Enumerating timing probing attack

To be consistent with prior work, we determined whether our modification of Chromium's core might reintroduce the enumerating timing probing attack — already fixed from versions higher than 61.0.3155.0 — presented by Sánchez-Rola et al. [53]. This timing attack makes a distinction between two types of requests: 1. chrome-extension://<fakeExtUUID>/<fakePath>, and; 2. chrome-extension://<realExt-UUID>/<fakePath>. The attacker uses the User Timing API [59], which allows to take time measurements with high precision, to check the response times for each of these requests. If the measured times do not differ more than 5%, the attacker can conclude that the requested extension is not installed in the client's browser.

In order to reproduce this timing attack, we downloaded and built Chromium 53.0.2785.135 on a virtual machine with Ubuntu 16.04.

We identified three scenarios: 1. using the original Chromium 53.0.2785.135 source code; 2. Chromium 66.0.3359.117 with our patch applied, but without the Latex Gloves extension, and; 3. Chromium 66.0.3359.117 with our patch applied and the Latex Gloves extension installed.

For each scenario, we had Avast Online Security installed and used it as the <realExt-UUID>. When executing with our patch and Latex Gloves installed, we had set the whitelist to allow all requests to extension WARs, apart from to Avast Online Security and AdBlock. Table 5.7 shows the results of our experiment, where the time measurement for each request was averaged over 1,000 runs. From these results, it is clear that Chromium 53.0.2785.135 is vulnerable to the timing attack, since there is more than 5% difference between the time measurement for an existing extension and a

Table 5.8: Breakdown of the amount of Chrome and Firefox extensions that would be uniquely identifiable through the content of a WAR, given that no probing could take place.

	Extensions	Total WARs	Unique WARs	Detection probability
Firefox	1,378	95,920	23,687	24.69%
Chromium	11,633	12,499,335	127,054	1.02%
Revealing	2,906	4,027,046	35,478	0.88%

non-existing extension. However, with our modification (with or without extension), that difference is no longer present.

8 Recommendations

Based on the experiments in Sections 3 and 4, we recommend several improvements to the browser extension ecosystem, addressed to browser developers and extension developers.

Recommendations for browser developers Chrome extensions are vulnerable to the extension probing attack because their UUIDs are static and publicly known. Firefox extensions combat this vulnerability by having randomized extension UUIDs. However, Firefox extensions can still be identified through the revelation attack. Worse, because Firefox's random UUIDs are not easily changed after an extension is installed, they can be used to fingerprint the extension user.

Our first recommendation is to re-generate Firefox's random UUIDs more often, either upon starting the browser or for each domain visited. Similarly, if a user enables private browsing mode [48, 23], each active browser extension should be provided with a new random UUID. Although this would not prevent detecting which browser extensions are executed, it would limit the tracking to a specific instance, making it infeasible to use this technique for long-term tracking of users.

Our second recommendation is to randomize the full URL of a WAR, and not just the UUID. With this change, a WAR URL seen by an attacker would be shaped as `moz-extension://<random-UUID>/<random-path>` for Firefox and `chrome-extension://<random-UUID>/<random-path>` for Chrome. Without any recognizable path components, the attacker would be forced to read and fingerprint the contents of the WAR to determine which extension is installed. As depicted in Table 5.8, without the ability to probe, this would decrease the probability of detecting Firefox extensions to 24.69% (compared to 93.76%, as shown in Table 5.2), and 1.02% for Chrome (compared to 89.01%) and probability of detecting the extensions we know reveal themselves would

drop to 0.88% from 89.52%. The random path approach can be taken one step further by implementing the WAR URLs to be of single use, i.e. the same WAR will have different paths each time it is injected or fetched. Such a change to core extension infrastructure would make it impossible for an attacker to fetch a recently injected resource in order to analyze the content. However, it would also require an overhaul of the browser implementation and possibly most browser extensions, which is very impractical.

Recommendations for browser extension developers Both Mozilla [43] and Google [27] provide guidelines for browser extension developers, e.g. "never ask for more permission than needed", and "properly secure sensitive or personal data when transmitting over the network". However, neither provide specific guidelines on how to handle WARs in a secure way.

Our only recommendations fall in the "least privilege" category, where no more privileges than needed to perform a certain task should be requested. Firstly, to help prevent the revelation attack, extension developers should not arbitrarily inject content with the random UUID. As seen in Table 5.5, several extensions currently inject content on any arbitrary web page, including blank pages. Secondly, to help prevent the probing attack, extensions should *not expose unused WARs*. A non-existent WAR cannot be used in a probing attack, thus reducing the chances that an extension can be identified through a probing attack.

9 Related work

User fingerprinting by using web browsers has been widely studied in the literature [12, 9, 11, 38, 15, 34]. As an example, Cao et al. [15] were able to fingerprint 99.24% of web users — being completely web browser agnostic — by using hardware features such as those from GPUs or CPUs. More recently, Gómez-Boix et al. [34] performed a large scale experiment to determine whether fingerprinting is still possible nowadays. They reached the conclusion that in desktop web browsers, both plugins (e.g. Flash, NPAPI, etc) and fonts are the most representative features to fingerprint users. However, none of the aforementioned works have taken browser extensions into consideration.

Nikiforakis et al. [52] showed that implementation differences between browsers can be fingerprinted. There exist several extensions that attempt to erase those fingerprints, but those extensions in turn allow a user to also be fingerprinted. In the same vein, Acar et al. [9] state that browser extensions can be exploited to fingerprint and track users on the Web.

Starov and Nikiforakis [56] presented a method to fingerprint browser extensions using a behavioral attack. They show browser extensions can provide unique, arbitrary DOM modifications, and analyzes the top 10,000 of most downloaded browser extensions, concluding 9.2% to 23% of those extensions are detectable. Contrarily to the experiments they performed — they only analyzed the manifest file of 1,665 browser extensions and they found that more than a 40% of them do make use of WARs, in this work we have scrutinized 62,994 browser extensions and concluded that 16,280 explicitly declare some WARs in their *manifest.json* file (≈26%).

In 2011, Kettle [36] demonstrated that all Chrome extensions could be enumerated by requesting their manifest file, which was explained in 2012 by Kotowicz [37]. Google solved this problem by introducing WARs, but Sjösten et al. [55] showed that all Chrome extensions with WARs can be enumerated without them being active on the attacker page. They demonstrated that approximately 28% of all Chrome extensions and approximately 6.7% of all non-WebExtension Firefox extensions could be enumerated from a web page. Gulyás et al. [33] combine known fingerprinting techniques with the Chrome extension enumeration attack presented by Sjösten et al. [55], along with a login-leak which determines the web pages that a user is logged in to [40]. They conclude that 54.86% of users which have installed at least one detectable extension and 19.53% of users which have at least one detectable active login, are unique. A combination of at least one detectable extension installed, and at least one detectable active login make the uniqueness number go up to 89.23%, indicating that installed browser extensions can make a good fingerprint, further showing the necessity of a mechanism to prevent extension fingerprinting.

Sánchez-Rola et al. [53] presented a timing attack against Chrome and Firefox by using the fact that the internal branching time for WARs differs between installed and non-installed extensions, thus detecting 100% of all extensions. A temporary solution has been implemented in Chrome [20], and the plan is to implement a randomization scheme similar to Firefox's, when they can make "a breaking change" [8]. In [53], Sánchez-Rola et al. also presented the revelation attack against Safari, which was the first browser to use randomized UUIDs. Based on a static analysis of 718 extensions, they estimated more than 40% of the extensions could leak the random UUID. They manually analyzed 68 security extensions, finding one false negative and 20 out of 29 extensions flagged as suspicious indeed leaked the random UUID. Contrarily to Sánchez-Rola et al, we investigate all Chrome and Firefox extensions to see which leak their UUID on actual web pages.

Chen and Kapravelos [17] developed a taint analysis framework for browser extensions to study their privacy practices. From sources, such as

DOM API calls (e.g. `document.location`), and extension API calls (e.g. `chrome.history`), they find 2.13% of Chrome and Opera extensions to potentially be leaking privacy-sensitive information to sinks such as `XMLHttpRequest` and `chrome.storage`. However, they do not seem to consider extension UUIDs as part of the privacy-sensitive information.

Finally, it is worth mentioning that an attacker might use any of the attacks presented in this paper to detect browser extensions and thus, perform more harmful attacks. Buyukkayhan et al. [14] for instance, exploit the lack of non-isolation worlds on the previous version of the Firefox add-ons architecture, allowing legitimate extensions which make use of *Cross Platform Component Object Model (XPCOM)* to access system resources such as the file system and the network. A prerequisite for this attack is that there must be a mechanism to disclose installed extensions in the victim's browser. Thus, the attacks described in our work may be used as a stepping stone to escalate the attacker's privileges in the browser.

10 Conclusion

We have investigated the problem of detecting browser extensions by web pages. With the intention to prevent probing for browser extensions by web pages, Mozilla Firefox recently introduced randomized extension UUIDs. A similar move is currently being discussed by the Google Chrome developers. We have demonstrated that the randomized UUIDs can in fact hurt user privacy rather than protect it. To this end, we have studied a class of attacks, which we call revelation attacks, allowing web pages to detect the randomized browser extension UUIDs in the code injected by extensions into the web pages, which, due to the design of the randomization of UUIDs, giving the ability to uniquely track users.

We have conducted an empirical study assessing the feasibility of revelation attacks. Our experiments show that combining revelation and probing attacks, it is possible to uniquely identify 90% out of all extensions injecting content, in spite of a randomization scheme. Furthermore, we have conducted a large-scale study assessing the pervasiveness of probing attacks on the Alexa top 10,000 domains, providing new evidence for probing beyond what was captured by previous work.

As a countermeasure, we have designed a mechanism that controls what extensions are enabled on what pages. As such, our mechanism supports two types of whitelists: specifying which web pages are allowed to probe for which extensions and specifying which extensions are allowed to inject content on which web pages. We have presented a proof of concept prototype

that blocks both probing and revelation attacks, unless explicitly allowed in the whitelists.

For future work, it would be interesting to consider XHOUND [56] and Hulk [35] to make a comparison on the different extensions that provide arbitrary DOM modifications (XHOUND), extensions that are deemed malicious (Hulk), and that inject WAR URLs. Unfortunately, the tools are unavailable at present.

Next steps for Firefox and Chrome We have reported the details of our study and our suggestions for mitigation to both involved browser vendors.

The issue with the randomized UUIDs has been confirmed by Firefox developers [1]. They agree that attacks like the revelation attack defeat anti-fingerprinting measures. While the problem is clear to the developers, the discussion on countermeasures is still ongoing.

As mentioned earlier, Google has recently announced that Chrome will allow users to restrict extensions from accessing websites by a whitelisting mechanism in line with ours [7]. Users will be able to restrict the host permissions for an extension, paving the way for an in-browser mechanism to control the extension whitelist.

Acknowledgments This work was partly funded by the Swedish Foundation for Strategic Research (SSF) under the WebSec project and the Swedish Research Council (VR) under the PrinSec and PolUser projects.

11 Bibliography

[1] https://bugzilla.mozilla.org/show_bug.cgi?format=default&id=1372288. accessed July-2018.

[2] AdBlock Plus. https://chrome.google.com/webstore/detail/adblock-plus/cfhdojbkjhnklbpkdaibdccddilifddb. accessed Aug-2018.

[3] Adobe: Adobe Acrobat Force-Installed Vulnerable Chrome Extension. https://bugs.chromium.org/p/project-zero/issues/detail?id=1088. accessed May-2018.

[4] Avast Online Security. https://chrome.google.com/webstore/detail/avast-online-security/gomekmidlodglbbmalcneegieacbdmki. accessed May-2018.

[5] Common Crawl. http://commoncrawl.org/. accessed May-2018.

[6] Ghostery – Privacy Ad Blocker. https://chrome.google.com/webstore/detail/ghostery---privacy-ad-blo/mlomiejdfkolichcflejclcbmpeaniij. accessed Aug-2018.

[7] Trustworthy Chrome Extensions, by Default. https://security.googleblog.com/2018/10/trustworthy-chrome-extensions-by-default.html. accessed Nov-2018.

[8] WebAccessibleResources take too long to make a decision about loading if the extension is installed. https://bugs.chromium.org/p/chromium/issues/detail?id=611420#c19. accessed Feb-2018.

[9] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. FPDetective: Dusting the Web for Fingerprinters. In *CCS*, pages 1129–1140, 2013.

[10] AdBlock. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom. accessed Aug-2018.

[11] P. Baumann, S. Katzenbeisser, M. Stopczynski, and E. Tews. Disguised Chromium Browser: Robust Browser, Flash and Canvas Fingerprinting Protection. In *WPES*, pages 37–46, 2016.

[12] K. Boda, A. M. Földes, G. G. Gulyás, and S. Imre. User Tracking on the Web via Cross-browser Fingerprinting. In *NordSec*, pages 31–46, 2012.

[13] M. Brinkmann. Firefox WebExtensions may be used to identify you on the Internet. https://www.ghacks.net/2017/08/30/firefox-webextensions-may-identify-you-on-the-internet/, 2017.

[14] A. S. Buyukkayhan, K. Onarlioglu, W. K. Robertson, and E. Kirda. CrossFire: An Analysis of Firefox Extension-Reuse Vulnerabilities. In *NDSS*, 2016.

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

[16] S. Cassidy. LostPass. https://www.seancassidy.me/lostpass.html, 2018.

[17] Q. Chen and A. Kapravelos. Mystique: Uncovering Information Leakage from Browser Extensions. In *CCS 2018*, pages 1687–1700, 2018.

[18] Chrome. Match Patterns. https://developer.chrome.com/extensions/match_patterns. accessed Apr-2018.

[19] Chrome. Webstore Hosting and Updating. https://developer.chrome.com/extensions/hosting. accessed Apr-2018.

[20] Chromium Code Reviews. Issue 2958343002: [Extensions] Change renderer-side web accessible resource determination (Closed). accessed Feb-2018.

[21] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010–. [Version 3.0], accessed May-2018.

[22] U. Fiore, A. Castiglione, A. De Santis, and F. Palmieri. Countering Browser Fingerprinting Techniques: Constructing a Fake Profile with Google Chrome. In *NBiS*, pages 355–360, 2014.

[23] Google. Browse in private. https://support.google.com/chrome/answer/95464. accessed May-2018.

[24] Google. Chrome Web Store. https://chrome.google.com/webstore/category/extensions?_feature=free. accessed Feb-2018.

[25] Google. chrome.runtime. https://developer.chrome.com/extensions/runtime#method-getURL. accessed Feb-2018.

[26] Google. Content Scripts. https://developer.chrome.com/extensions/content_scripts. accessed Feb-2018.

[27] Google. Developer Program Policies. https://developer.chrome.com/webstore/program_policies. accessed May-2018.

[28] Google. Google Safe Browsing. https://safebrowsing.google.com/. accessed July-2018.

[29] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources. accessed Apr-2018.

[30] Google. Manifest File Format. https://developer.chrome.com/extensions/manifest. accessed Feb-2018.

[31] Google. New Cast functionality in Chrome. https://support.google.com/chromecast/answer/6398952. accessed Apr-2018.

[32] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia. Browser Extension and Login-Leak Experiment. https://extensions.inrialpes.fr/. accessed Apr-2018.

[33] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia. To Extend or not to Extend: On the Uniqueness of Browser Extensions and Web Logins. In *WPES@CCS*, pages 14–27, 2018.

[34] A. Gómez-Boix, P. Laperdrix, and B. Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *WWW*, 2018.

[35] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *USENIX Sec.*, pages 641–654, 2014.

[36] J. Kettle. Sparse Bruteforce Addon Detection. `http://www.skeletonscribe.net/2011/07/sparse-bruteforce-addon-scanner.html`, 2011.

[37] K. Kotowicz. Intro to Chrome addons hacking: fingerprinting. `http://blog.kotowicz.net/2012/02/intro-to-chrome-addons-hacking.html`, 2012.

[38] P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *S&P*, pages 878–894, 2016.

[39] LastPass. LastPass: Free Password Manager. `https://chrome.google.com/webstore/detail/lastpass-free-password-ma/hdokiejnpimakedhajhdlcegeplioahd`. accessed May-2018.

[40] R. Linus. Your Social Media Fingerprint. `https://robinlinus.github.io/socialmedia-leak/`, 2016.

[41] L. Liu, X. Zhang, V. Inc, G. Yan, and S. Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[42] Microsoft. Internet Explorer Browser Extensions. `https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa753587(v%3dvs.85)`, 2018.

[43] Mozilla. Add-on Policies. `https://developer.mozilla.org/en-US/Add-ons/AMO/Policy/Reviews`. accessed May-2018.

[44] Mozilla. content_scripts. `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/content_scripts`. accessed Feb-2018.

[45] Mozilla. extension.geturl(). `https://developer.mozilla.org/en-US/Add-ons/WebExtensions/API/extension/getURL`. accessed Feb-2018.

[46] Mozilla. manifest.json. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json. accessed Feb-2018.

[47] Mozilla. Most Popular Extensions. https://addons.mozilla.org/en-US/firefox/search/?sort=updated&type=extension. accessed Feb-2018.

[48] Mozilla. Private Browsing - Use Firefox without saving history. https://support.mozilla.org/en-US/kb/private-browsing-use-firefox-without-history. accessed May-2018.

[49] Mozilla. Profiles - Where Firefox stores your bookmarks, passwords and other user data. https://support.mozilla.org/en-US/kb/profiles-where-firefox-stores-user-data/. accessed Mar-2018.

[50] Mozilla. web_accessible_resoruces. https://developer.mozilla.org/en-US/Add-ons/WebExtensions/manifest.json/web_accessible_resources. accessed Feb-2018.

[51] Mozilla Add-ons Blog. WebExtensions in Firefox 57. https://blog.mozilla.org/addons/2017/09/28/webextensions-in-firefox-57/. accessed Feb-2018.

[52] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *S&P*, pages 541–555, 2013.

[53] I. Sánchez-Rola, I. Santos, and D. Balzarotti. Extension Breakdown: Security Analysis of Browsers Extension Resources Control Policies. In *USENIX Security Symposium*, pages 679–694, 2017.

[54] A. Sjösten, S. Van Acker, and A. Sabelfeld. Non-behavioral extension detector. http://blueberry-cobbler-11673.herokuapp.com. accessed May-2018.

[55] A. Sjösten, S. Van Acker, and A. Sabelfeld. Discovering Browser Extensions via Web Accessible Resources. In *CODASPY*, pages 329–336. ACM, 2017.

[56] O. Starov and N. Nikiforakis. XHOUND: Quantifying the Fingerprintability of Browser Extensions. In *S&P*, pages 941–956, May 2017.

[57] StatCounter. Desktop Browser Market Share Worldwide. http://gs.statcounter.com/browser-market-share/desktop/worldwide. accessed May-2018.

[58] W3C. CSP2. https://www.w3.org/TR/CSP2/. accessed Nov-2018.

[59] W3C. User Timing. https://www.w3.org/TR/user-timing. accessed May-2018.

Paper VI

Filter List Generation for Underserved Regions

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

Proceedings of the Web Conference (WWW), Taipei, Taiwan, April 2020

Abstract

Filter lists play a large and growing role in protecting and assisting web users. The vast majority of popular filter lists are crowd-sourced, where a large number of people manually label resources related to undesirable web resources (e.g. ads, trackers, paywall libraries), so that they can be blocked by browsers and extensions.

Because only a small percentage of web users participate in the generation of filter lists, a crowd-sourcing strategy works well for blocking either uncommon resources that appear on "popular" websites, or resources that appear on a large number of "unpopular" websites. A crowd-sourcing strategy will perform poorly for parts of the web with small "crowds", such as regions of the web serving languages with (relatively) few speakers.

This work addresses this problem through the combination of two novel techniques: (i) deep browser instrumentation that allows for the accurate generation of request chains, in a way that is robust in situations that confuse existing measurement techniques, and (ii) an ad classifier that uniquely combines perceptual and page-context features to remain accurate across multiple languages.

We apply our unique two-step filter list generation pipeline to three regions of the web that currently have poorly maintained filter lists: Sri Lanka, Hungary, and Albania. We generate new filter lists that complement existing filter lists. Our complementary lists block an additional 3,349 of ad and ad-related resources (1,771 unique) when applied to 6,475 pages targeting these three regions.

We hope that this work can be part of an increased effort at ensuring that the security, privacy, and performance benefits of web resource blocking can be shared with all users, and not only those in dominant linguistic or economic regions.

1 Introduction

Hundreds of millions of web users (i.e. 30% of all internet users [25]) use filter lists to maintain a secure, private, performant, and appealing web. Prior work has shown that filter lists, and the types of content blocking they enable, significantly reduce data use [37], protect users from malware [36], improve browser performance [28, 38] and significantly reduce how often and persistently users are tracked on the web.

Most filter lists are generated through crowd-sourcing, where a large number of people collaborate to identify undesirable online resources (e.g. ads, user tracking libraries, anti-adblocking scripts etc..) and generate sets of rules to identify those resources. Crowd-sourcing the generation of these

lists has proven a useful strategy, as evidenced by the fact that the most popular lists are quite frequently used and frequently updated [41, 33].

The most popular filter lists (e.g. EasyList, EasyPrivacy) target "global" sites, which in practice means either websites in English, or resources popular enough to appear on English-speaking sites *in addition to* sites targeting speakers of other languages. Non-English speaking web users face different, generally less appealing options for content blocking. Web users who visit non-English websites that target relatively wealthy users generally have access to well maintained, language-specific lists. Indeed, the French [9], German [11], and Japanese [17] specific filter lists are representative examples of well-maintained, popular filter lists targeting non-English web users. Similarly, linguistic regions with very large numbers of speakers also generally have well maintained filter lists. Examples here include well maintained filter lists targeting Hindi [15], Russian [21], Chinese [5], and Portuguese [20, 4] websites.

Sadly, users who visit websites in languages with fewer speakers, or with less wealthy users, have worse options. Put differently, the usefulness of crowd-sourced filter lists depends on having a large or affluent crowd; filter lists targeting parts of the web with less, or less affluent, users are left with filter lists that are smaller, less well-maintained, or both. Visitors speaking these less-commonly-spoken languages have degraded web experiences, and are exposed to all the web maladies that filter lists are designed to fix.

Compounding the problem, in many cases, users in these regions are the ones who could benefit most from robust filter lists, as network connections may be slower, data may be more expensive, the frequency of undesirable web resources may be higher. An example which motivates this work and illustrates the inability of current filter lists to adequately block ads on a regional website in Albania can be seen in the screenshot in Figure 6.1. In this example, we browse the website *gazetatema.net* while using AdBlock Plus (which uses EasyList, a "global" targeting filter list).

While there has been significant prior work on automating the generation of filter lists [34, 29, 24, 26], this existing work is focused on replicating and extending the most popular English and globally-focused filter lists, with little to no evaluation on, or applicability to, non-English web regions. In this paper, we target the problem of improving filter lists for web users in regions with small numbers of speakers (relative to prominent global languages). We select three regions as representative of the problem in general: Albania, Hungary and Sri Lanka, using a methodology presented in Section 4.1.

We describe a two-pronged strategy for identifying long-tail resources on websites that target under-served linguistic regions on the web: (i) a classifier that can identify advertisements in a way that generalizes well across

languages, and (ii) a method for accurately determining how advertisements end up in pages (as determined by either existing filter lists or our classifier), and by using this information, generate new, generalized filter rules.

We use this novel instrumentation to both build *inclusions chains* (i.e. measurements of how every remote resource wound up in a web page), and determine how high in each inclusion chain blocking can begin. This allows us to (i) generate generalized filter rules (i.e. rules that target scripts that include ad images on each page, instead of rules that target URLs for individual advertisements), and to (ii) ensure we do not block new resources that will break the website in other ways.

Contributions. In summary, this paper makes the following contributions to the problem of blocking unwanted resources on websites targeting audiences with smaller linguistic audiences.

1. The implementation and evaluation of an **image classifier for automatically detecting advertisements on the web** which relies on a mix of perceptual and contextual features. This classifier is designed to be robust across many languages (and particularly those overlooked by existing research) and achieves accuracy of 97.6 % in identifying images and iframes related to advertising.

2. Novel, open source **browser instrumentation, implemented as modifications to the Blink and V8 run-times** that allows for determining the cause of every web request in a page, in a way that is far more accurate than existing tools. This instrumentation also allows us to accurately attribute every DOM modification to its cause, which in turn allows us to predict whether blocking a resource would break a page.

3. The design of a **novel, two stage pipeline for identifying advertising resources on websites**, using the previously mentioned classifier and instrumentation, to identify long-tail advertising resources targeting web users who do not speak languages with large global communities.

4. A **real world evaluation** of our pipeline on sites that are popular with languages that are (relatively) uncommon online. We find that our approach is successful in significantly improving the quality of filter lists for web users without large, language-specific crowd sourced lists. As our evaluation shows, our generated lists block an additional 3,349 of ad and ad-related resources (1,771 unique) when applied to 6,475 pages targeting these three regions.

2 Solution Requirements

A successful contribution to the problem of improving the quality of filter lists in small web regions should account for the following issues:

Scalability. The primary difficulty of generating effective blocking rules for small-region web users is the reduced number of people who can participate in a crowd-sourced list generation. While portions of the web are targeted at large audiences (e.g. sites in English language, or web regions with a large number of language speakers) can count on a large number of users to report unwanted resources, or generally distribute the task of list generation, regions of the web targeting only a small number of users (e.g. languages with less speakers) do not have this luxury. A successful solution therefore likely requires some kind of automation to augment the efforts of regional list generators.

Generalize-ability. In most of the cases, ads are rendered by scripts. In addition, every time an ad-slot is filled, the embedded ad image may have come from a different URL. Approaches that directly target the URLs serving ad-related resources are then likely to become stale very quickly. An effective solution to the problem would instead target the "root cause" of the unwanted resources being included in the page, in this case the script, which determines what image URLs to load. Approaches that attempt to only build lists of URLs of ad-related images are therefore unlikely to be useful solutions to the problem for the long term (as seen also from the screenshot in Figure 6.1).

Web compatibility. Content blocking necessarily requires modifying the execution of a page from what the site-author intended, to something hopefully more closely aligned with the visitor's goals and preferences. Modifying the page's execution in this way (e.g. by changing what resources to load, by preventing scripts from executing, etc.) frequently cause pages to break, and users to abandon content blocking tools. While filter lists targeting large audiences can rely on the crowd to report breaking sites to the list authors, (so that they can tailor the rules accordingly), filter lists targeting smaller parts of the web often do not have enough users to maintain this positive feedback loop. An effective system for programmatically augmenting small-region filter lists must therefore take extra care to ensure that new rules will not break sites.

3 Methodology

This section presents a methodology for programmatically identifying advertising and other unwanted web resources in under-served regions. This section proceeds by describing (i) a high-level overview of our approach,

Figure 6.1: Motivating example of current filter lists' regional inefficiency. Screenshot of Albanian website browsed with Adblock Plus.

(ii) a hybrid classifier used to identify image-based web advertisements, (iii) unique browser instrumentation used in our approach, (iv) how we identify ad-libraries and other "upstream" resources for blocking, (v) how we determined if a request was safe to block (i.e. would not break desirable functionality on the page), and (vi) how we generated filter list rules from the gathered ad URLs.

3.1 Overview

Our solution to improving filter lists for under-served regions consists of the combination of two unique strategies. First, we designed a system for programmatically determining whether an image is an ad, in a cross-language, highly precise way. We use this classifier to identify ad images that are missed by crowd-sourced filter list maintainers.

Second we developed a technique for identifying additional resources that should be blocked, by considering the request chains that brought the ad into the page, and finding instances where we can block earlier in that request chain. We then apply this "blocking earlier in the chain" principle to both ads identified by existing filter lists, and new ads identified by our classifier, to maximize the number of resources that can be safely blocked. This approach also allows us to generate generalized blocking rules that target the *causes* of ads being included in the page, instead of only the "symptoms": the specific, frequently-changing image URLs.

We note that this approach could be applied to any region of the web, including both popular and under-served regions. However, since popular parts of the web are already well-served by crowd-sourced approaches, we expect the marginal improvement of applying this technique will be greatest for under-served regions, where there are comparatively few manual labelers.

The following subsections describe the implementation of each piece in our filter list generation pipeline. Section 4 describes the evaluation of how successful this approach was at generating new filter list rules for under-served regions.

3.2 Hybrid Perceptual/Contextual Classifier

First, our approach requires an oracle for determining if a page element is an advertisement, without human labeling. To solve this problem, we designed and trained a unique hybrid image classifier that considers both the image's pixel data, and page context an image request occurred in, when predicting if a page element is an advertisement. Our classifier targets both images (i.e. `<img>`) and sub-documents (i.e. `<iframe>`). Our classifier prefers precision over recall, since for filter list it is more important to *only* block ads, instead of blocking *every* ad.

Comparison to Existing Approaches While there is significant existing work on image based (i.e. perceptual) web ad classification, we were not able to use existing approaches for two reasons. First, we had disappointing results when applying existing perceptual classifiers to the web at large. The existing approaches we considered did very well on the data sets they were trained on, but did a relatively poor job when applied to new, random crawls of the web.

Second, we were concerned that relying on perceptual features alone would reduce the classifier's ability to generalize across languages. We expected that adding contextual features (e.g. the surrounding elements in the page, whether the image request was triggered by JavaScript or the document parser, attributes on the element displaying the image) would make the classifier generalize better.

Classifier Design Our approach combines both perceptual and contextual page features, each building on existing work. The perceptual features are similar to those described in the Percival [40] paper, while the contextual features are extensions of those used in the AdGraph [34] project. The probability estimated by the perceptual module is then used as an input to the contextual classifier.

	Initial Alexa 10k Data Set			Alexa 10k Recrawl		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
Perceptual-only	95.9 %	95.5 %	96.4 %	77.0 %	48.8 %	87.4 %
Hybrid	-	-	-	97.6 %	92 %	75 %

Figure 6.2: Comparison of classification strategies. "Perceptual-only" refers to the approach by Percival [40] and variants (best numbers reported). "Hybrid" uses both perceptual and contextual features, and performed much better on our independent sampling of images and frames from the Alexa 10k, especially with regards to precision.

	# Images	% Ads	# Frames	% Ads
Initial Alexa 10k Set	7,685	48 %	465	77 %
Alexa 10k Recrawl	2,610	14 %	1,034	41 %

Figure 6.3: Comparison of the distribution of ads for images and frames collected in each data set.

Perceptual Sub-module. The perceptual part of our classifier expands Percival's SqueezeNet based CNN into a larger network, ResNet18 [30]. While the Percival project used a smaller network for fast online, in-browser classification, our classifier is designed for offline classification, and so faces no such constraint. We instead use the larger ResNet18 approach to increase predictive power. Otherwise, our approach is the same as that described in [40].

Contextual Sub-Module. The contextual part of our classifier does not consider the image's pixel data, but instead how the image loaded in the web page, and the context of the page the image or subdocument would be displayed in. Examples of contextual features include whether the resource being requested is served on the same domain as the requesting website, and the number of sibling DOM nodes of the `img` or `iframe` element initiating the request. These features are similar to those described in the AdGraph paper, and detailed in Figure 6.4. The browser instrumentation needed to extract these features is described in detailed in Section 3.3.

Classifier Evaluation We built our image classifier in two steps. First we built a purely perceptual classifier, using approaches described in existing work. Second, when we found the perceptual classifier did not generalize well when applied to a new, independent sampling of images, we moved to a hybrid approach. In this hybrid approach, the output of the perceptual classifier is just one feature among many other contextual features. We found this hybrid approach performed much better on our new, manually labeled,

Content features
Height & Width
Is image size a standard ad size?
Resource URL length
Is resource from subdomain?
Is resource from third party?
Presence of a semi-colon in query string?
Resource type (image or iframe)
Perceptual classifier ad probability
Structural features
Resource load time from start
Degree of the resource node (in, out, in+out)
Is the resource modified by script?
Parent node degree (in, out, in+out)
Is parent node modified by script?
Average degree connectivity

Figure 6.4: Partial feature set of the contextual classifier.

random crawl of the Alexa 10k. The rest of this subsection describes each stage in this process.

Initially, we built a classifier using an approach nearly identical to the perceptual approach described in [40]. We evaluated this model on a combination of data provided by the paper's authors, augmented with a small amount of additional data labeled by ourselves. This data set is referred to in Figure 6.3 as the "Initial Alexa 10k Set". When we applied the training method described in [40] to this data set, we received very accurate results, reported in Figure 6.2.

Later, while building the pipeline described in this paper, we generated a second manually labeled data set of images and frames, randomly sampled a new crawl of the Alexa 10k. This data set is referred to in Figure 6.3 as "Alexa 10k Recrawl", and was collected between 2-6 months later than the previous data set[1]. When we applied the prior purely-perceptual approach to this new data set, we received greatly reduced accuracy. Most alarming of which, for our purposes, was the dramatically reduced precision. These numbers are also reported in Figure 6.2.

We concluded that perceptual features alone were insufficient to handle the breath of advertisements found on the web, and so wanted to augment the prior perceptual approach with additional, contextual features we ex-

[1] the date range here is due to the majority of this data set being collected by the Percival authors, 6 months before our work, with a smaller additional amount of data being collected by ourselves later on.

pected to generalize better, both across languages and across time. A subset of these contextual features are presented in Figure 6.4, and are heavily based on the contextual ad-identification features discussed in the AdGraph [34] project.

After constructing our hybrid classifier from the combination of perceptual and contextual features, we achieved greatly increased precision, though at the expense of some recall. We used a Random Forest approach to combine the perceptual and contextual features, and after conducting a 5-fold cross-validation, achieved mean precision of 92 % and mean recall of 75 %, again summarized in Figure 6.2. Our hybrid classifier could not be evaluated against the initial Alexa 10k data set because the data set 1) programmatically determined some labels, and 2) was collected without our browser instrumentation, meaning we could not extract the required features.

3.3 Browser Instrumentation

In this subsection we present PageGraph, a system for representing and recording web page execution as a graph. PageGraph allows us to correctly attribute every page modification and network request to its cause in the page (usually, the responsible JavaScript unit). We use this instrumentation both to extract the contextual features described in Section 3.2, and to accurately understand what page modifications and downstream requests each JavaScript unit is responsible for.

Our approach is similar to the AdGraph [34] project, but is more robust (i.e. corrects categories of attribution errors) and broader (i.e. cover an even greater set of page behaviors). PageGraph is implemented as a large set of patches and modifications to Blink and V8 (approximately 12K LOC). The code for PageGraph is open source and actively maintained, and can be found at [19], along with information on how other researchers can use the tool.

The remainder of this subsection provides a high-level summary of the graph-based approach used by PageGraph, and how it differs from existing work.

Graph Representation of Page Execution We use PageGraph to represent the execution of each page as a directed graph. This graph is available both at run-time, and offline (serialized as graphml [22]) for after-the-fact analysis. PageGraph uses nodes to represent elements in a web page (e.g. DOM elements, resources requested, executing JavaScript units, child frames) and edges representing the interaction between these elements in the page (e.g. an edge from a script to a node might depict the script modifying an attribute

on the node, an edge from a DOM element node to a resource node might depict a file being fetched because of a `img` element's `src` attribute, etc.). All such page behaviors in the top-level frame, and child-local-frames, are captured in the graph.

We use PageGraph's context-rich recording of page execution for several purposes in this work. First, it allows to accurately and efficiently understand how a JavaScript unit's execution modified the page; we can easily determine which scripts made a lot of modifications to the page, and which had only "invisible" effects to e.g. fingerprint the user. Second, the graph allows us to determine how each element ended up in a page. For example, the graph representation makes it easy to determine if an image was injected in the page by a script, if so *what* other script, and how that script was included in the page, etc. Being able to accurately determine what page element is responsible for the inclusion of each script, frame or image element is particularly valuable to this work, as described in the following subsections.

Differences from Existing Work The most relevant related work to PageGraph is the AdGraph project, which also modifies the Blink and V8 systems in Chromium to build a graph-representation of page execution. PageGraph differs from AdGraph in several significant ways.

Improved Attribution Accuracy. PageGraph significantly improves cause-attribution in the graph, or correctly determining which JavaScript unit is responsible for each modification. We observed a non-trivial number of corner cases where AdGraph would attribute modifications to the wrong script unit, such as when the script was executed as a result of an element attribute (e.g. `onerror="do_something()"`), or when the JavaScript stack is reset through events like timer callbacks (e.g. `setTimeout(do_something,1)`). PageGraph correctly handles these and a large number of similar corner cases.

Increased Attribution Breadth. PageGraph significantly increases the set of page events tracked in the graph, beyond what AdGraph records. For example, PageGraph tracks image requests initiated because of CSS rules and prefetch instructions, records modifications made in local sub-documents, and tracks failed network requests, among many others. This additional attribution allows for greater understanding of the context scripts execute in.

3.4 Generalizing Filter Rules

We next discuss how we generate generalized filter rules from the data gathered by the previously described image classifier and browser instrumentation. The general approach is to find URLs serving ad images and

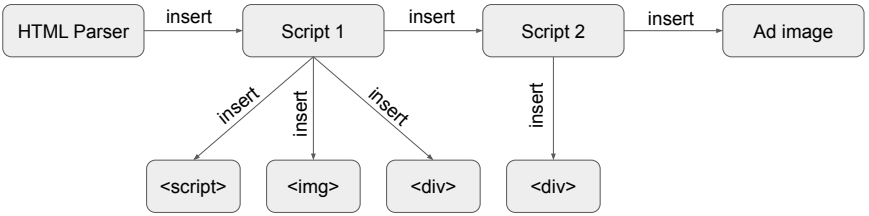


Figure 6.5: Example of a request chain, ending in an inserted ad image.

frames using the classifier, use the browser instrumentation to build the entire request chain that caused the advertisement to be included in the page (e.g. the script that fetched the script that inserted the image), and then again use the browser instrumentation to determine how far up each request chain we can block without breaking the page.

We build these request chains for both images (and frames) our classifier identifies as an ad, and for resources identified by network rules in existing filter lists (i.e. EasyList, EasyPrivacy and the most up to date applicable regional list). The former allows us to generalize the benefits of our image classifier, the latter allows us to maximize the benefits of existing filter lists.

Motivation Blocking higher in the request chain has several benefits. First, and most importantly, targeting URLs higher in the request chain yields a more consistent set of URLs. While the specific images that an ad library loads will change frequently, the URL of the ad library itself will rarely change. Approaches that target the frequently changing image URLs will result in filter list rules that quickly go stale; rules that target ad library scripts (as one example) are more likely to be useful over time, and to a wider range of users. Moving higher in the request chain means we are more likely to programmatically identify ad libraries *in addition to* frequently changing, one-off image URLs.

Second, blocking higher in request chains reduces the total number of requests, bringing privacy and performance improvements. Blocking a single "upstream" ad library may prevent the browser from needing to consider several "downstream" requests.

Building Request Chains To generate optimized filter list rules, we target not only the ad images and ad frames in each page, but the scripts that injected those images and frames (and, potentially, the scripts that injected those scripts, etc.). We refer to the cause of a request as being "upstream", and the thing being requested "downstream". We refer to the list of elements that participated in an advertisement being included as its "request chain."

For each `<img>`, `<iframe>` and `<script>` in a page, we determine the request chain as follows:

1. Locate each element in the PageGraph generated graph structure. Call this element `X`.

2. Use the graph edges to determine how `X` was inserted in the document. If `X` was inserted by the parser (i.e. it appeared in the initial HTML text) then stop.

3. Otherwise, append the script element `X` into the request chain for `X`, set the responsible script element as the new `X` and continue from #2 above.

A simplified result of this process is depicted in Figure 6.5. The figure shows a simplified request chain, where a script was included in the initial HTML ("script one"), that script programmatically inserted another script element into the document ("script two") and that second script inserted an advertising image into the page.

We use these request chains to determine the optimal place to start blocking, using the approach described in the following section.

3.5 Safe Blocking in Request Chains

This subsection describes how we determine whether blocking a script request is likely to break a page. We use this technique to determine how "high" in each request chain we can block, with the goal of determining the earliest "upstream" request we can block in a request chain without breaking the page. Our approach is "conservative" (i.e. prefers false negatives over false positives), under the intuition that users would prefer a working, ad-filled page, over a broken, ad-less page.

Determining Page Breakage We use a pair of simple heuristics to determine whether blocking a script is likely to break a page. These heuristics are designed to distinguish scripts that only inject ads into pages from scripts that perform more complex, and hopefully user serving, page operations.

1. If a script creates more than two subtrees in the document, we consider it **unsafe** to block.

2. If a script inserts another script that matches condition #1, we consider it **unsafe** to block.

3. We consider all other scripts **safe** to block.

Less formally, if a script makes no modifications to the structure of a page, or the modifications to the page are isolated to one or two parts of the page (e.g. one or two ads, an ad and an "ad choices" annotation, etc.) we consider it safe to block. Scripts that add elements to more than two parts of the page, or include scripts that do the same, are considered too risky to block, and too likely to break desirable page functionality.

We note that this is only a heuristic, one that matches our experience building and debugging advertising and tracking-blocking tools, but still only a heuristic. Although heuristics can be fooled by an attacker, they are being used in current tools to identify unwanted code. If the script for injecting ad content is updated to evade the heuristic, the heuristic can be updated. We choose the conservative figure of allowing modifications to a maximum of two regions of the page to favor false negatives over false positives (i.e. we'd rather allow an ad than break a page). The larger problem of predicting whether any given page modification breaks a site *in the subjective determination of the browser user* is an open research question, and one that would be its own complicated project.

Application to Filter List Generation We use the above-described heuristics to determine the highest point in a request chain that can be blocked. For each request chain describing how an advertising image or frame was included in a page, we select the "highest" script request we can block, that will not break the page. Put differently, we want to select the earliest point in each chain to block, that will have no "downstream" breaking scripts. If there are no elements in the request chain that can be blocked, the last loaded resource (i.e. the ad) will be blocked only.

As a demonstration, consider Figure 6.5. Our system would generate two filter rules for this request chain, one targeting the "ad image", and one targeting "script 2". Our system begins by considering the most "downstream" request, the image element at the far right. This image has been identified as an ad, either by our classifier, or by existing filter lists. Using the browser instrumentation described in Section 3.3, we build the request chain for this image.

Next, we try to consider the earliest point in the request chain we can begin blocking. We observe that "script 2" only modifies one other part of the document (inserting a single `<div>` element) and so we consider this script safe to block. The next element in the request chain, "script 1" inserts elements into more than two parts of the document, and so we consider it "unsafe" for blocking.

3.6 Rule Generation

Finally, we describe how we turn the set of identified ad-serving URLs into filter list rules. We do so through the following four steps for each URL we determine to be blockable:

1. Reduce the URL's domain to its `eTLD+1` root.
2. Remove the query parameter portion of the URL.
3. Remove the fragment portion of the URL.
4. Remove the protocol from the URL.

We then record the modified (i.e. reduced) version of each URL as a right-rooted AdBlock Plus format filter rule [2]. For example, the URL `https://a.good.example.com/ad.html?id=3` would be recorded as `||example.com/ad.html`, and would block requests to `https://good.example.com/another-ad.html` and `http://a.b.good.example.com/ad.html?id=4`, but would not block requests to `https://other.domain.com/ad.html`. This approach is designed to generalize some (i.e. match other similar requests, even when irrelevant details like tracking related query parameters change) but not so much so that unrelated materials served on the same host are blocked.

4 Evaluation

In this section, we evaluate the approach to regional filter list generation described in Section 3 by applying the technique to three representative under-served web regions. We find that the technique is successful, and generates 1310 new rules that identify 1,771 advertising URLs missed by existing filter lists. These new rules, when applied in addition to existing filter list rules, results in 27.1% more advertising resource being blocked than when using existing filter lists alone. We also find that our technique is useful in all three measured regions, though to varying degrees.

This section proceeds by first describing how we selected the three under-served regions used in this evaluation, then presents the crawling methodology we used to measure popular websites in each selected region, and follows by presenting the results of applying our methodology to each of these regions. The section concludes by presenting the output of our measurements (i.e. the newly generated filter list rules), so that they can be used by existing content blockers.

Country	# Rules	# Network Rules	Last Update	Source
Albania	201	118	8	[1]
Hungary	1,407	662	3	[16]
Sri Lanka	69	42	22	[3]

Figure 6.6: Regional crawling data. The "Last Update" column gives the number of months since the last update, relative to October 2019.

Country	Commits	Start (Month-Year)	Average	Source
Albania	3	02-2019	0.27	[1]
Hungary	542	12-2014	8.9	[16]
Sri Lanka	16	03-2016	0.35	[3]
India	1,637	05-2018	81.85	[13]
Germany	11,982	01-2014	166.4	[12]
Japan	1,687	05-2014	24.8	[14]

Figure 6.7: Filter list activity. "Average" gives the average number of commits per month since the start of the git repo, relative to Jan. 2020.

4.1 Selecting Regions for Evaluation

We evaluated our approach on three regions under-served by existing crowd-sourced filter lists: Albania, Hungary, and Sri Lanka. We selected these regions after looking for regions that matched six criteria.

1. The national language was not a major world language.
2. There existed at least one filter list for the region.
3. The amount of updates to the regional supplementary list is significantly lower compared to more popular lists.
4. The region has seen a vast increase of Internet usage in the past decade[2].
5. Had a popular sites listing on Alexa Top Sites.
6. We could purchase or gain access to a VPN with an exit point in the country.

Figure 6.6 presents the regions we selected for this evaluation, along with measurements of the existing best-maintained regional filter list. For each region we identified the best maintained filter list for the region by consulting both the EasyList selection of regional filter lists [18], and the filter

[2]Statistics gathered from `www.internetlivestats.com`

Country	# Domains	# Pages	# Images	# Frames
Albania	740	1,805	38,763	578
Hungary	935	2,287	46,687	1,318
Sri Lanka	890	2,196	47,092	844
Total	2,565	6,288	132,542	2,740

Figure 6.8: Measurements of data gathered from crawls of popular sites in selected under-served regions. Given numbers are counts of unique image and frame URLs.

lists indexed on a popular, crowd-sourced site of regional filter lists [8]. To further illustrate the lack of updates, Figure 6.7 illustrates how much activity there is on average each month in the respective GitHub repos between the selected regions and some popular filter lists.

4.2 Crawl Data Set

We next built a data set of popular websites and pages for each of the three selected regions. We use this data set for two purposes: first to approximate how internet users in these regions experience the web, and second to determine how much advertising content was being missed by existing content blocking options.

For each region, we first fetched the 1,000 most popular domains for the region, as determined by the Alexa Top Lists. Next, we purchased VPN access from ExpressVPN [6], a commercial VPN service, that provided an IP address in each region. Third, we configured a crawler to visit each domain and select two random child links with the same `eTLD+1`. We then configured our crawler to use our PageGraph-instrumented browser to visit the domain of each site and each selected child page, each for 30 seconds. All crawling was conducted from the VPN end point, to as closely as possible approximate how the page would run for a local visitor.

After 30 seconds, we recorded the PageGraph data for each page (note, the PageGraph data includes information about all network requests issued during the page's execution, in addition to the cause of each request). We also record all images and scripts fetched in the top-and-local frames (i.e. `<iframe>`s with the same domain as the top level frame) during each page's execution, along with screenshots of each remote child-frame (i.e. `<iframe>`s of third-party domains).

Figure 6.8 presents the results of our automated crawl. For each of the regions we encountered a significant number of non-responsive domains, which comprise the difference between the number in the "# Domains" column and the 1,000 domains identified by Alexa. While the 10%

Country	# Ad Images	%	# Ad Frames	%
Albania	2,553	6.6%	164	28.4%
Hungary	1,479	3.2%	209	15.9%
Sri Lanka	1,451	3.1%	296	35.1%
Total	5,483	4.1%	669	24.4%

Figure 6.9: Measurements of how many unique image and iframe ads are currently identified by existing filter lists (e.g. EasyList, EasyPrivacy, and the best maintained filter list for each region.).

non-responsive rate for Hungary and Sri Lanka is inline with prior web-studies [31] that find around 11% error rate for automated crawls of the web, the even higher rate of non-responsive sites in Albania was surprising. On manual evaluation of a sample of these domains, we found a small number of cases were due to anti-crawler countermeasures or apparent IP blacklisting of the VPN end point. In a surprising number of cases though, domains seemed to be abandoned and hosting no web content at all. We note this as a point for future study.

4.3 Ads Identified by Existing Filter Lists

Next, we measured how successful existing filter lists are at blocking advertisements on popular sites in our selected regions. We treated this measurement as our baseline when measuring how much additional blocking benefit our approach provides. Figure 6.9 shows the number of ads identified by existing filter lists.

We measured the amount of ad resources identified by existing lists in two steps. First, we combined the best maintained regional filter list for each region (listed in Figure 6.6) with the two most popular "global" filter lists, EasyList and EasyPrivacy. Then, we applied these combined filter lists to the image and iframe requests encountered when crawling each region, using a popular AdBlock filter list library [23]. We then noted which images and iframe requests would be blocked by the current best filter lists available to people in each region.

4.4 Ads Identified by Hybrid Classifier

Next, we identified how many advertising images and iframes we missed by existing filter lists. We found a significant number of both; our classifier identified 1,497 ad images and 47 ad frames that were missed by existing filter lists. Put differently, our approach identified 27.3% more ad images, and 7% more ad frames, than existing filter lists.

Country	# Ad Images	%	# Ad Frames	%
Albania	440	17.2%	20	12.2%
Hungary	547	37%	10	4.8%
Sri Lanka	510	35.1%	17	5.7%
Total	1,497	27.3%	47	7%

Figure 6.10: Measurements of how many unique image and iframe ads the classifier described in Section 3.2 identified that *were not identified* as ads by existing filter lists.

Country	Current Lists	Classifier	$\cup$ Chains	Δ
Albania	2,850	460	511	18%
Hungary	1,819	557	586	32.2%
Sri Lanka	1,872	527	674	36%
Total	6,541	1,544	1,771	27.1%

Figure 6.11: Additions to filter lists when applying all steps of our methodology. "Current lists" gives the number of ad-resources found by existing filter lists, "classifier" describes ad-resources found by our hybrid classifier but not existing filter lists. "$\cup$ chains" gives the number of new ad-resources found by applying our "upstream" approach to ad-resources found by either current filter lists or the hybrid classifier. The "Δ" column gives the overall increase in identified ad-resources provided by our techniques, compared to existing filter lists.

We note that these figures only include ad images and frames missed by filter lists; we observed a significant amount of overlap between the two approaches. Figure 6.10 summarizes the additional ad resources our classifier identified.

We measured how many advertising images and frames current approaches miss in two steps. First, we identified each image or frame in the corresponding PageGraph graph data, and used that contextual information to extract the contextual features described in Section 3.2. Second, we used the pixel data of each resource to feed the perceptual part of the classifier. For images, we used the image file directly; for frames we used a screenshot of the frame taken during the crawling step.

4.5 Generalizing Rules with Request Chains

Next, we identified additional resources that should be blocked by examining the request chain for each ad image or frame, and finding the earliest point in each chain that could be blocked without breaking the page. We applied this "upstream request chain" blocking technique to both ad resources labeled by existing filter lists and ad resources newly identified by

Country	Network rules
Albania	387
Hungary	551
Sri Lanka	372
Total	1310

Figure 6.12: Number of new filter list rules.

our hybrid classifier. Doing so allowed us to not only identify specific images and frames that should be blocked, but to programmatically identify the "upstream" libraries that caused those images and frames to be included.

We were able to identify 1,771 additional advertising URLs by analyzing the request chains in this manner, an improvement of 27.1% in advertisement blocking in these regions. We note that by following the methodology described in Section 3.5, targeting these additional resources will result in more generalizable filter rules by identifying both the individual ad image URLs *and* the ad libraries that determine what ads to load. The approach described in Section 3.5 also gives us a high degree of confidence that this "upstream" blocking will not break pages.

Figure 6.11 shows the final results of our regional filter list methodology when applied to the Albanian, Hungarian and Sri Lankan web regions. The "current lists" column presents the number of ad resources existing filter lists identify in each region in our data set. The "classifier" column gives the number of images and frames our hybrid classifier identifies as ad-related *that are not identified* by existing filter lists. The "$\cup$ chains" column gives the total number of ad resources identified by applying the request chain approach (Section 3.4) to images and frames identified as advertisements by *either* existing filter lists or the hybrid classifier. The final "Δ" column gives the percent-increase in resources identified by our combined methodology, when compared to using only existing filter lists.

4.6 Generated Filter-Lists

Finally, we generated filter lists in AdBlock Plus format to block the advertising resources identified in the previous steps. We used the rule generation methodology described in Section 3.6 to generate 1310 new filter rules. We have made the filter lists available [10]. Counts of the total number of new rules for each region are presented in Figure 6.12.

5 Discussion

In this section, we discuss broader issues related to the problem of filter list generation and ad blocking, including possible next steps and extensions

for the described approach, and some limitations and concerns for future researchers to consider.

5.1 Ad Ambiguity

A reoccurring issue in identifying and blocking online advertisements is that many images and ads are context specific. An ad in one context might be core content in another. For example, an image of shoes with the name of the shoe maker might be perceived as an advertisement when positioned next to a news article, but the same image might be desirable when placed in the middle of a page on a shoes selling website.

We encountered an even more difficult case when labeling and debugging the pipeline described in this work. We found a movie sharing forum that used a number of banner ads (like the one presented in Figure 6.13) from elsewhere on the web as a table of contents, to show which movies had most recently been added on the site. In such cases, the "ad-ness" of an image is not ambiguous, its explicitly both an ad and desirable page content!

Crowd-sourced filter list generation approaches rely on the subjective intuition of list contributors to resolve such difficult situations. Programmatic solutions have no such option, and so must address a two tiered problem: first, how to identify images that look like advertisements, and second, how to model subjective user expectations of when an advertisement is desirable to users.

We find the problem presented by the intersection of these two issues is unaddressed by existing literature (current work included). Our resolution to this issue was "ads are ads"', and it is the job of ad blocking tools to block ads, and if a user is in a scenario where they wish to see and ad, they should disable the ad-blocking tool. How satisfying such an approach is will likely be task specific. Thankfully, the number of ambiguous ads we encountered was low enough that it did not affect the main focus of the work, but we mention it as an interesting area for future research.

5.2 Alternative Image-Identification Oracles

This work used a unique hybrid approach for determining whether an image or a frame was an advertisement. This is only one of an infinite number of possible oracles the same pipeline could adopt. While we designed our approach to be conservative in identifying images (as described in Section 3.2), one could instead use a much more aggressive oracle, if one was willing to accept a greater false-positive rate in ad-identification, or was willing to accept sites breaking, for additional data savings and privacy protections.

Figure 6.13: Example of an ambiguous ad image we encountered on a movie sharing forum in Sri Lanka as part of the its table of contents.

In this sense, our oracle represents just one web use preference (less advertisements, but with a low tolerance for error). The same broad approach, as described in this paper, could be used with other oracles, such as those targeting just certain types of advertisements (i.e. blocking adult ads), or certain types of web content in general (i.e. blocking violent images).

While our goal in this work was to improve web browsing for people in under-served regions, the described approach is not specific to advertising. The identify-and-prune-the-request-chain approach could be helpful in addressing many web problems where human labelers are lacking.

5.3 Possible Extensions

We considered many additional features and approaches when designing the methodology in this work. Here, we briefly describe a variety of improvements we considered, but did not implement because of time, cost or complexity. We list them as possible suggestions to other researchers addressing similar problems.

Predicting Page Breakage. An important part of this work was generating and testing useful heuristics for whether blocking a script would break a page. The heuristics discussed in Section 3.5 have proven useful for us, but could be improved. One could, for example, also consider the number and type of Web API calls a script makes, whether the script sets or reads storage, or any number of other behavioral characteristics when trying to predict whether blocking a script would break a page.

Tracking Protections. This work improves blocking *advertisements* in under-

served regions, but similar approach could be taken to target *tracking scripts*. Instead of building a classifier to determine if an image is an advertisement, one would instead need an oracle to determine whether a script was privacy violating. This might be an easier task, since determining the privacy implications of a script's execution is in many cases easier that predicting the subjective evaluation of whether an image is an advertisement.

Improving Other Filter Lists. The approach in this paper was designed to help web users in under-served regions. However, the same approach could likely be used to improve filter lists in general, including "global" popular ones like EasyList and EasyPrivacy. Though the marginal improvement would likely be lower, since the relative popularity of such sites likely means a higher percentage of ad resources have been identified by filter list contributors, our approach could still be useful in improving blocking on less popular, or frequently changing sites.

5.4 Limitations

Finally, we note some limitations of this approach, in the hopes that future work might address them. First, while our approach was successful on the three selected regions, its difficult to know if these findings would generalize to all under-served regions on the web. While such a measurement is beyond our ability to carry out, it would be interesting to better understand how similar web advertising is across the web generally.

Second, our approach relies on automated, manual crawls of websites to identify ad-related resources. It is possible that the kinds of advertisements reachable by automated tools are different from the kinds of advertisements humans experience when on parts of the web not reachable by crawlers, such as within web applications, behind paywalls, or within account-requiring portions of websites. This limitation is a subset of a larger open problem in web measurement, of understanding how well automated crawls approximate human user experiences.

Finally, the types of advertisements targeted in this work (i.e. image based web advertisements) are just one of many types of advertisements web users face. A partial list includes audio ads, video interstitials, native text ads, and interactive advertisements. If image- and frame- targeting ad blockers continue to become more popular, we can expect advertisers to adopt to these alternative advertising approaches. Researchers will in turn need to come up with new ad blocking techniques to preserve a usable, performant, privacy respecting web.

6 Related Work

Below we cover the existing work related to filter lists (Section 6.1), resource blocking (Section 6.2), and the importance of different vantage points for web measurements (Section 6.3).

6.1 Filter Lists

In [41], Vastel et al. explored the accumulation of dead rules by studying EasyList, the most popular filter list. Results of their study show that the list has grown from several hundred rules, to well over 60,000 rules, within 9 years, when 90.16% of the resource blocking rules are not useful. Finally, authors, propose optimizations for popular ad-blocking tools, that allow EasyList to be applied on performance constrained mobile devices, and improve desktop performance by 62.5%.

Gugelmann et al. in [29] investigated how to detect privacy-intrusive trackers and services from passive measurements and propose an automated approach that relies on a set of web traffic features to identify such services and thus help developers maintaining filter lists. Pujol et al. in [38] used Adblock Plus filter lists for passive network classification. By analyzing data from a major European ISP authors show that 22% of the active users have Adblock Plus deployed. Also they found that 56% and 35% of the ad-related requests are blacklisted by EasyList and EasyPrivacy, respectively.

Iqbal et al., in [33], studied the anti-adblock filter lists that ad blockers use to remove anti-adblock scripts. By analyzing the evolution of two popular anti-adblock filter lists, authors show that their coverage considerably improved the last 3 years and they are able to detect anti-adblockers on about 9% of Alexa top-5K websites. Finally authors proposed a machine learning based method to automatically detect anti-adblocking scripts.

6.2 Resource Blocking

Iqbal et al., in [34], proposed AdGraph: a graph-based machine learning approach for detecting advertising and tracking resources on the web. Contrary to filter list based approaches AdGraph builds a graph representation of the HTML structure, network requests, and JavaScript behavior of a webpage, and uses this unique representation to train a classifier for identifying advertising and tracking resources. AdGraph can replicate the labels of human-generated filter lists with 95.33% accuracy.

In [39], Storey et al. discussed the future of ad blocking by modelling it as a state space with four states and six state transitions, which correspond to techniques that can be deployed by either publishers or ad blockers.

They also proposed several new ad blocking techniques, including ones that borrow ideas from rootkits to prevent detection by anti-ad blocking scripts. Zhu et al., in [42], proposed ShadowBlock: a new Chromium-based ad-blocking browser that can hide traces of ad-blocking activities from anti-ad blockers. ShadowBlock leverages existing filter lists and hides all ad elements stealthily so anti-ad blocking scripts cannot detect any tampering of the ads (e.g. , absence of ad elements). Performance evaluation on Alexa top-1K websites shows that their approach successfully blocks 98.3% of all visible ads while only causing minor breakage on less than 0.6% of the websites.

Garimella et al. in [28] measured the performance and privacy aspects of popular ad-blocking tools. Their findings show that (i) uBlock has the best performance, in terms of ad and third party tracker filtering, and least privacy tracking. They also found that the time to load pages is not necessarily faster when using adblockers, and this happens due to additional functionality introduced by the adblocking tools. In [40], Din at al. proposed Percival: a deep learning based perceptual ad blocker that aims to replace filter list based adblocking. Percival runs within the browser's image rendering pipeline, intercepts images during page execution and by performing image classification, it blocks ads. Percival can replicate EasyList rules with an accuracy of 96.76% when it imposes a rendering performance overhead of 4.55%.

6.3 Internet Vantage Points

Selecting different vantage points to browse Internet from is a quite common technique in order to understand the different view of the web different users may have. Jueckstock et al. in [35] design and deploy a synchronized multi-vantage point web measurement study to explore the comparability of web measurements across different Internet vantage points. In [27] Fruchter et al. proposed a method for investigating tracking behavior by analyzing cookies and HTTP requests from browsing sessions from different countries. Results show that websites track users differently, and to varying degrees, based on the regulations of the country the visitor's IP is based in.

Iordanou et al. in [32] proposed a system for measuring how e-commerce websites discriminate between users. Authors consider several different motivations for discrimination, including geography, prior browsing behavior (e.g., tracking-derived PPI) of the user, and site A/B testing. They found that the first and third motivations explain more website "discrimination" than the second.

7 Conclusions

In this work we address the problem of augmenting filter lists for users of under-served, linguistically-small parts of the web. The approach described in this work is amenable to full automation, and with sufficient computation resources could be applied to any number of additional under-served populations of web users. Further, we expect the same approach could be used to block tracking-related resources too, improving privacy for under-served web users too.

The problem of poorly maintained filter lists in under-served regions is significant. First, the current predominant approach to filter list generation (i.e. crowd-sourcing) is poorly suited for these web-regions, which by definition have less users, and so smaller "crowds." Second, in many cases, under-served areas of the web target users with less income, and with less access to cheap, high speed data; the users who would benefit most from ad blocking are often the poorest served by current filter list generating strategies. Third, existing filter list generation strategies that *do not* rely on crowd-sourcing fail to consider web compatibility (i.e. breaking sites), leaving under-served users with the unappealing trade-off between data-draining, privacy-harming browsing, or, alternatively, breaking web sites.

This work proposes a novel approach for generating filter rules for under-served regions of the web. Our approach determines whether images and frames are advertisements by considering perceptual and contextual aspects of the underlying image (or frame), and then using deep browser instrumentation to determine where in the request chain we can optimally begin blocking requests.

We apply this approach to popular websites in three regions currently poorly served by crowd-sourced filter lists, Sri Lanka, Hungary, and Albania. Our approach is successful at improving blocking without breaking websites. We generate 1310 new filter list rules that identify 27.1% new advertising resources that should be blocked, improving blocking by 30.1% over the existing best options for these regions. We are also releasing our generated filter lists so that web users in these regions can benefit from them [10], along with the source code for our hybrid image classifier [7] and our PageGraph browser instrumentation [19]. We hope this work advances the goal of improving the web for all users, no matter their location or linguistic-community.

Acknowledgements This work was partly funded by the Swedish Foundation for Strategic Research (SSF) and the Swedish Research Council (VR).

8 Bibliography

[1] Adblock list for albania. https://github.com/anxh3l0/blocklist. accessed: Oct-2019.

[2] Adblock Plus filters explained. https://adblockplus.org/filter-cheatsheet/. accessed: Jan-2020.

[3] Adblock sri lanka. https://github.com/miyurusankalpa/adblock-list-sri-lanka. accessed: Oct-2019.

[4] Brazilian filterlist. https://raw.githubusercontent.com/easylistbrasil/easylistbrasil/filtro/easylistbrasil.txt. accessed: Oct-2019.

[5] Chinese filterlist. https://easylist-downloads.adblockplus.org/easylistchina.txt. accessed: Oct-2019.

[6] ExpressVPN. accessed: Oct-2019.

[7] Filterlist Generator GitHub repo. https://github.com/brave-experiments/regional-filterlist-gen. accessed: Jan-2020.

[8] FilterLists. https://filterlists.com/. accessed: Jan-2020.

[9] French filterlist. https://easylist-downloads.adblockplus.org/liste_fr.txt. accessed: Oct-2019.

[10] Generated Filter Lists. https://sites.google.com/site/longtailfilterlists/. accessed: Jan-2020.

[11] German filterlist. https://easylist.to/easylistgermany/easylistgermany.txt. accessed: Oct-2019.

[12] Github repo for german filter list. https://github.com/easylist/easylistgermany. accessed: Jan-2020.

[13] Github repo for hindi filter list. https://github.com/mediumkreation/IndianList. accessed: Jan-2020.

[14] Github repo for japanese filter list. https://github.com/k2jp/abp-japanese-filters. accessed: Jan-2020.

[15] Hindi filterlist. https://easylist-downloads.adblockplus.org/indianlist.txt. accessed: Oct-2019.

[16] hufilter. https://github.com/hufilter/hufilter/wiki. accessed: Oct-2019.

[17] Japanese filterlist. https://raw.githubusercontent.com/k2jp/abp-japanese-filters/master/abpjf.txt. accessed: Oct-2019.

[18] Other Supplementary Filter Lists and EasyList Variants. https://easylist.to/pages/other-supplementary-filter-lists-and-easylist-variants.html. accessed: Jan-2020.

[19] PageGraph. https://github.com/brave/brave-core/wiki/PageGraph. accessed: Jan-2020.

[20] Portugese filterlist. https://easylist-downloads.adblockplus.org/easylistportuguese.txt. accessed: Oct-2019.

[21] Russian filterlist. https://easylist-downloads.adblockplus.org/advblock.txt. accessed: Oct-2019.

[22] The GraphML File Format. http://graphml.graphdrawing.org/. accessed: Jan-2020.

[23] B. L. Andrius Aucinas. Brave improves its ad-blocker performance by 69x with new engine implementation in rust. https://brave.com/improved-ad-blocker-performance/, 2019.

[24] S. Bhagavatula, C. Dunn, C. Kanich, M. Gupta, and B. Ziebart. Leveraging machine learning to improve unwanted resource filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 95–102. ACM, 2014.

[25] Business Insider Intelligence. 30% of all internet users will ad block by 2018. https://www.businessinsider.com/30-of-all-internet-users-will-ad-block-by-2018-2017-3, 2017.

[26] V. Dudykevych and V. Nechypor. Detecting third-party user trackers with cookie files. In *2016 Third International Scientific-Practical Conference Problems of Infocommunications Science and Technology (PIC S T)*, pages 78–80, Oct 2016.

[27] N. Fruchter, H. Miao, S. Stevenson, and R. Balebako. Variations in tracking in relation to geographic location. *CoRR*, abs/1506.04103, 2015.

[28] K. Garimella, O. Kostakis, and M. Mathioudakis. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *WebSci*, pages 259–262. ACM, 2017.

[29] D. Gugelmann, M. Happe, B. Ager, and V. Lenders. An automated approach for complementing ad blockers' blacklists. *Proceedings on Privacy Enhancing Technologies*, 2015(2):282–298, 2015.

[30] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] L. Invernizzi, K. Thomas, A. Kapravelos, O. Comanescu, J.-M. Picod, and E. Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 743–758. IEEE, 2016.

[32] C. Iordanou, C. Soriente, M. Sirivianos, and N. Laoutaris. Who is fiddling with prices?: Building and deploying a watchdog service for e-commerce. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 376–389, New York, NY, USA, 2017. ACM.

[33] U. Iqbal, Z. Shafiq, and Z. Qian. The ad wars: Retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pages 171–183, New York, NY, USA, 2017. ACM.

[34] U. Iqbal, Z. Shafiq, P. Snyder, S. Zhu, Z. Qian, and B. Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *CoRR*, abs/1805.09155, 2018.

[35] J. Jueckstock, S. Sarker, P. Snyder, P. Papadopoulos, M. Varvello, B. Livshits, and A. Kapravelos. The blind men and the internet: Multi-vantage point web measurements. *CoRR*, abs/1905.08767, 2019.

[36] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 674–686, 2012.

[37] 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: Oct-2019.

[38] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, IMC '15, pages 93–106, New York, NY, USA, 2015. ACM.

[39] G. Storey, D. Reisman, J. Mayer, and A. Narayanan. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568*, 2017.

[40] Z. ul Abi Din, P. Tigas, S. T. King, and B. Livshits. Percival: Making in-browser perceptual ad blocking practical with deep learning. *CoRR*, abs/1905.07444, 2019.

[41] A. Vastel, P. Snyder, and B. Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *CoRR*, abs/1810.09160, 2018.

[42] S. Zhu, U. Iqbal, Z. Wang, Z. Qian, Z. Shafiq, and W. Chen. Shadowblock: A lightweight and stealthy adblocking browser. In *The World Wide Web Conference*, WWW '19, pages 2483–2493, New York, NY, USA, 2019. ACM.