THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

ALEXANDER SJÖSTEN

CHALMERS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries
ALEXANDER SJÖSTEN

Technical report 175L
ISSN 1652-876X
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 2018

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

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

Abstract

In modern software development, the use of libraries is prevalent. Libraries pose a big security challenge. How can we ensure that sensitive data is not being leaked through libraries? This is the first question of the thesis. We propose the use of information-flow control, by developing a principled approach for allowing information-flow tracking in libraries, even if they are written in a language not supporting information-flow control. With this approach, we allow for library functions to have *unlabel* and *relabel* models, explaining how values are unlabeled and relabeled when being marshaled between the labeled program and the library. These models are used in combination with *lazy marshaling* to handle structured data such as lists and records, higher-order functions and references.

Modern browsers allow for browser modifications through *browser extensions*, which have special privileges and can, e.g., modify the DOM. As extensions can be intrusive, it is in a webpage's interest to know which extensions are installed in a browser. The second question of the thesis is if it is possible for a webpage to know which extensions are installed in the browser? We conduct a large-scale study to determine how many extensions that are detectable from a webpage based on the extension's resources, showing over 50% of the top 1000 Chrome extensions can be detected, as well as how many of the Alexa top 100,000 webpages employ the technique of the paper.

Keywords: information-flow control, language-based security, side-effectful libraries, web security, browser extensions, large-scale study

Acknowledgements

First and foremost, I want to thank my supervisor Andrei Sabelfeld for all the support, great advice about coffee places, lunch runs and tips about the academic life. Without his support, this work would not have been possible.

I also want to thank my co-supervisor Daniel Hedin for great collaborations, advice, gaming nights, beers, coffees and for leading me down the path to frustration. Praise the Sun, Daniel!

Thank you Steven, who was the co-author on one of the papers for all the advice you have given me about organising the research.

A big thank to my office mates, both old and new, for all the discussions and for making the office a truly awesome working environment: Pablo, Daniel S, Per, Jeff and Iulia.

To all my other colleagues at Chalmers, my deepest and sincerest thank you for all the coffees, beers, lunches and dinners. You all help make Chalmers such a nice place to work at, and I count you all as not just colleagues, but also friends.

To my former study mates from the bachelor's and master's education whom I still keep contact with, thank you for sticking around and sharing coffees, lunches, gaming nights, hockey nights, role playing sessions and whatnot. You help disconnecting the brain from work and I simply have a great time hanging out with you.

To my family, thank you so much for all encouragement and support you have given throughout the years. Although we have had some disagreements (it is what a son and brother is for, right?), you have always been there when I needed it the most, just a phone call away.

Last, but definitely not least: a big thanks goes to Pauline. I don't understand how you can stand to live with me, but you give me so much love and support. You are always there to pick me up when I fall down, supporting me every step I take. You are truly awesome!

Contents

Guarding the Boundary: Information Flow Tracking in the Presence of Libraries

In society today, most business sectors are completely reliant on information technology, and our day-to-day lives are moving online at an astonishing pace. For example, we use computers to talk to friends, read newspapers, watch movies, schedule events, make bank transfers, and buy merchandise. With more and more of our private information going online, the need to protect this information increases.

Unfortunately, it is hard to ensure that private information is not leaked to unintended recipients – even for domain experts. Adding a simple feature, for example tracking how users use a web application, can lead to private information being leaked [28]. In recent years, there have been many reported breaches of security in big companies, where sensitive information, such as credit card information, passwords and emails, has been stolen [13, 3, 27, 22, 26, 21, 24, 8], leading to financial losses and even loss of life [7].

A difficult problem to handle in modern applications is the use of *libraries*. With growing code bases, and potentially different languages getting access to sensitive information, securing the boundaries between a program and its libraries is a big challenge. This challenge is the focus of the first two papers of this thesis: tracking how information is flowing in a library used by a trusted program.

With services coming in the shapes of web applications, we need *web browsers* to access these services. But each individual has their own preferences on how the browsing should be. The users are therefore given the opportunity to extend the browser with functionality via *browser extensions*. There are cases where webpages try to determine what extensions are running, as the existence of an extension can lead to, for example, financial losses due to ad blockers. Another reason can be because the webpage want a clean environment due to handling of sensitive information, or because the webpage is malicious and tries to fingerprint a user. The main goal of the third paper of this thesis is to determine how many extensions for Chrome and Firefox are susceptible to webpages trying to determine their existence.

The first two papers in this thesis are theoretical, whereas the third paper is practical. The future goal is to enable end-to-end security for web applications in the presence of libraries. Since e.g. extensions can manipulate the DOM using JavaScript, "libraries" in this setting corresponds to the DOM API in the browser.

The rest of this chapter is laid out as follows. Section 1 gives an introduction to *information-flow control*, which is the main security mechanism used in this thesis. Section 2 gives a brief introduction to the information security challenges of libraries. Section 3 gives an introduction to browser extensions before Section 4 lists the main contributions of the papers, along with the

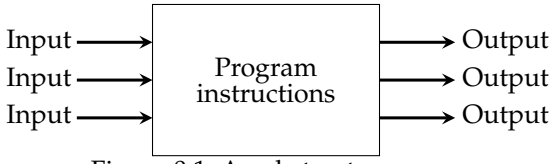


Figure 0.1: An abstract program

contributions made by the author.

1 Information-Flow Control

In software development, the most common ways of checking if an application is correct is through testing and code reviews. Although testing and code reviews can find some security vulnerabilities, many are missed; see for instance Heartbleed [33] and Shellshock [32].

Language-based security is a means to express security policies and enforcement mechanisms using programming language techniques [30]. Frequently, the goal of language-based security is to be provably secure. In this thesis, we will focus on the area of language-based security called *information-flow control (IFC)*. As is common, we work in a batch model of programs, meaning that we see programs as black boxes, treating them like functions from input to output (see Figure 0.1). We call the *inputs* to the program *sources*, and the *outputs* of the program *sinks*. For all useful programs, the outputs are dependent on the inputs. The dependencies from sources to sinks are defined by the program source code, which is written in some programming language.

In a *multi-level system* [9], information is classified into different *levels*, based on a *lattice*. Typical example levels are *unclassified* $\sqsubseteq$ *classified* $\sqsubseteq$ *secret* $\sqsubseteq$ *top secret*, where $\sqsubseteq$ is a relation defining how information is allowed to flow. In the example above, *unclassified* data can flow anywhere, and *classified* data can flow anywhere but to *unclassified*. IFC aims to enforce that the information flow respects the $\sqsubseteq$-relation. Without loss of generality, we use a two-level lattice, $L \sqsubseteq H$, where L is *public* (low) data and H is *secret* (high) data. In this simplified setting, the security property we want to enforce is that sources marked as H does not go to sinks marked as L, which is known as noninterference [14].

1.1 Noninterference

Intuitively, noninterference is achieved if all runs of a program, where the only difference is the high inputs, do not differ in the low output. This means the crossed out dashed red line in Figure 0.2 is not allowed.

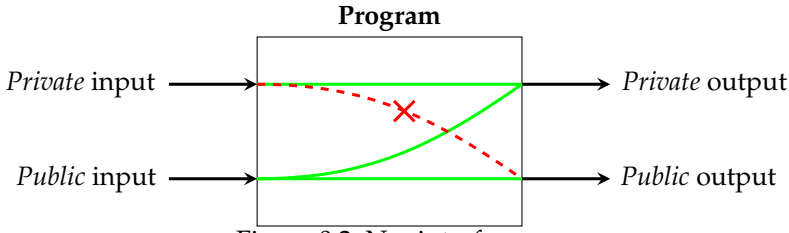


Figure 0.2: Noninterference

In this thesis, we will only consider *termination-insensitive noninterference (TINI)* [31]. The implication of enforcement through TINI is that information leakage through *termination channels* is not considered. Assuming `hi` is an integer labeled H and `print` outputs on a public channel, the following program is deemed secure by state-of-the-art IFC tools that do not take termination into account.

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

As the for-loop is not dependent on the secret variable, the output on the public channel is allowed. However, once `i == hi`, the program will be stuck in an infinite loop, ensuring the final printed value on the public channel to be the secret value, indicating there exists an implicit flow through a termination channel [4]. TINI gives no guarantees about non-terminating runs, since it is formulated in terms of terminating runs. Hence, TINI would be unable to classify the program above as insecure.

1.2 Explicit and implicit flows

In order to achieve noninterference, we must track how information flows within the program. There are two different kinds of flows: *explicit* and *implicit* flows. An example of an explicit flow is when secret data is written directly to a public sink or variable. Imagine two variables, `lo`, which is labeled L, and `hi`, which is labeled H. An explicit flow would be `lo := hi`, as the secret data in `hi` is written directly to the public variable `lo`. In contrast to explicit flows, an implicit flow has no data being written directly from a secret to a public sink or variable. Instead, the control flow of the application is used to learn something about the secret data. As an example, consider the following program that leaks if the variable `hi` is even or odd through the control flow.

```
1 lo := false;
2 if (hi `mod` 2 == 0) then lo := true;
```

Note that although no information about the secret variable `hi` is explicitly written to the public variable `lo`, we can still learn information about it.

1.3 Enforcing Information-Flow Control

Within IFC, there are two different main approaches of enforcement [29]. On one hand, there is *static* enforcement, which is based on static analysis of a program before it is executed. Volpano et al. [34] presented a *type system* with the property that all well-typed programs in this system are sound with respect to noninterference.

On the other hand, *dynamic* enforcement is executed at run-time, using a modified semantics of the language to allow for security checking. Having full access to the run-time environment and the run-time values, dynamic enforcement often leads to a more permissive enforcement compared to the static counterparts for dynamic languages such as JavaScript.

Additionally, there exists combinations of the static and dynamic enforcements known as *hybrid* enforcement. With hybrid enforcement, a static enforcement mechanism can insert annotations during the compilation phase, which can be checked at runtime [11, 12]. Similarly, a dynamic enforcement mechanism can perform static analyses on parts of the program during execution [19].

1.4 Dynamic Information-Flow Control

This thesis is based on purely dynamic IFC. The main reason we chose dynamic IFC is because dynamic languages, such as JavaScript, are widespread and popular, especially on the web.

In dynamic IFC, all runtime values are augmented with a representation of security labels. These labels are copied and joined to reflect the different computations of the program. During execution, a *program counter (pc)* is used to keep track of which level the current execution occurs in (known as the *security context*). If secret data is used when computing the condition in an if-statement, the pc is updated to reflect this, and the body of the if-statement is executed under *secret control*. While under secret control, no public side-effects should be allowed to take place. It is crucial that the handling of side-effects under secret control is done in a safe manner. Otherwise there is a risk of implicit flows into the labels. Consider the following example from [6], where `lo` and `tm` are labeled L and `hi` is labeled H at the start of the execution. What would be the implications of allowing labels to change

freely, i.e. upgrading the security labels on the assigned variables on lines 3 and 4 if the assignment occurs?

```
1 lo := true;
2 tm := true;
3 if (hi == true) then tm := false;
4 if (tm == true) then lo := false;
```

If `hi` is `true`, then `tm` will be `false` based on the assignment on line 3, and its security label upgraded to H, due to the pc being H. Since `tm` is `false`, the condition in the if-statement on line 4 will be `false`, making no assignment to `lo`, which means `lo` will continue to be `true` and labeled L. But if `hi` was `false`, then no assignment would be made to `tm` on line 3, making `tm` remain `true` and labeled L. This would make the assignment to `lo` on line 4 occur in a low context due to the pc being L , making `lo false` and labeled L. The end result in both situations is the value of `hi` being the same as `lo`, while `lo` retains the label L. In other words, `hi` was leaked into `lo`!

The most direct way of preventing the problem in the previous example and avoiding the implicit flows into the labels is to base the enforcement on *no sensitive-upgrades (NSU)*, which disallows upgrading low variables when branching on secret data [5, 35]. With NSU, the assignment on line 3 would not be allowed, as there is a low upgrade under high control, causing a termination of the program before the information leakage occurs.

A problem with NSU is that it can sometimes be too restrictive and mark valid programs as invalid. One could argue the program

```
1 lo := true;
2 if (hi == false) then lo := false;
```

is secure if the low variable `lo` is never written to a public sink, hence not visible to an attacker. The program

```
1 lo := false;
2 if (hi == false) then lo := false;
```

can also be deemed secure, since an attacker will not learn anything about `hi` since the value of `lo` is never changed. This is known as value sensitivity [10] and is not in scope of this thesis. The enforcement mechanisms of this thesis would consider those programs insecure if `hi` is indeed `false`.

2 Libraries

One focus of this thesis is how to handle libraries with respect to information-flow tracking. A major challenge is when the library is not written in the same language as the labeled program. This usually happens in one of two

cases: 1) the library is part of the standard execution environment, and 2) the library is brought into the language using a *foreign function interface (FFI)*. An FFI can therefore be used to extend a programming language with features, such as network communication, not natively in the programming language. When that happens, the values going between the labeled program and the library must be translated, a process known as *marshaling*. This poses a big challenge, since the security labels from the labeled program values must be removed when values go into the library, and put back when values are going from the library to the labeled program. Unlike standard marshaling, which is lossless, this means that we lose data when marshaling between a labeled program and an unlabeled library; we will not be able to compute the labels of the return values without knowing the labels of the arguments. To solve this, we need to keep a *state* in the marshaling process, where we store the labels that are removed. The labels are stored with respect to a *function model* for the library function, which defines how labels are removed (through an *unlabel model*) and how they are re-attached (through a *relabel model*). This interaction is what Papers I and II address.

2.1 Lazy marshaling

When marshaling structured values, such as lists, the question of how to do it effectively arises. It can be done strictly, where the full value is marshaled directly. This is expensive for large structured data, as we might marshal more than is needed for the computation. Another way is to do it lazily, which allows for marshaling on a need basis. With *lazy marshaling*, only the traversed elements of the structured data affect the computed return label. Imagine having a list of ten elements, but only the first two are needed for the computation. Strict marshaling would marshal the full list, but lazy marshaling would marshal exactly the two needed elements.

This notion of lazy marshaling extends naturally to all types of structured data, including records and objects as well.

3 Browser extensions

In order to increase web browser functionality, users install browser extensions. Examples of browser extensions are ad blockers to block advertisement on webpages, anti-tracking extensions to avoid tracking from tracking software, and password managers to make it easier to have unique passwords for all services. But it comes at a cost, as extensions are given permissions greater than those of a webpage. For instance, an extension can inject arbitrary code [16], with some malicious extensions actually injecting tracking software to track their users on every webpage they visit [18]. Even worse, if

an extension has a vulnerability, it might allow webpages to execute arbitrary code with elevated privileges [2, 1]. It is in a webpage's interest to know which extensions a user has installed, as it can lead to, for instance, financial losses due to less advertisements being showed, or to prevent arbitrary code being injected when paying bills over an internet bank.

Webpages can detect extensions using *behavioral analysis*, where one tries to detect extensions by looking for the effects of the extension. An example of this would be to check for the presence or absence of elements on the webpage. It is, however, difficult to use behavioral analysis to identify a specific extension – there are, for instance, several different ad blockers that have the same behavior. Behavioral analysis is also costly, as it requires time and effort to analyse and keep up-to-date with extension updates. Is there an easier way to determine which extensions a user has installed? This is the question tackled in Paper III.

4 Contributions

This thesis presents three papers, where Paper I and Paper III are published in peer-reviewed conferences. Paper II is currently under submission. All three papers presented are extended versions.

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

In this paper, we explore and develop an approach for tracking the information flow in a program that uses libraries. The program is assumed to be written in an information-flow aware language, whereas the library is not. The development is made gradually, starting with a small core language which is then extended with lists and higher-order functions. The general idea is based on *unlabel* and *relabel* models, which defines how labels are removed when marshaling to the library, and how labels are re-attached when marshaling back to the labeled world. An important part in the paper is the *lazy marshaling*, which increases the precision in the tracking as only the used parts of for example a list will affect the resulting label when marshaling back to the labeled world. The system is proven sound with respect to noninterference.

Statement of contribution This paper was co-authored with Daniel Hedin, Frank Piessens and Andrei Sabelfeld. Alexander's contributions was to define the syntax and semantics together with Daniel, implement prototypes

for testing the ideas, and prove soundness of the different systems. All authors contributed to the writing of the paper equally.

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

4.2 Information Flow Tracking for Side-effectful Libraries

The second paper of the thesis is a continuation of the first paper, where the major contribution is the addition of references and with that, side-effects. The core system was overhauled, introducing a *model heap* instead of passing the model state as an (implicit) parameter. With the use of a persistent heap structure, the list and higher-order functions from Paper I had to be modified, along with an extension of records, references and side-effects. The lazy marshaling remained for lists, as well as for the newly implemented records, but the model language was extended to also contain *side-effect constraints*, where the side-effect constraints are models for how the side-effects can manipulate data. The theoretical work is formalised in Coq [20], showing the system is sound with respect to noninterference.

This work, along with Paper I provides a core for how to track information flow in stateful libraries with structured data and higher-order functions.

Statement of contribution This paper was co-authored with Daniel Hedin and Andrei Sabelfeld. Alexander's contributions was to define the syntax and semantics, conduct the case study on a file system library, creating the examples and implementing the prototype. He also wrote an initial version of the paper, which served as the base when turning it to a coherent paper. All authors contributed equally in the latter process.

4.3 Discovering Browser Extensions via Web Accessible Resources

Webpages can perform browser fingerprinting, by detecting specific configurations of the hardware, see for instance Panopticlick [25]. But can extensions be detected by webpages, without the need of analysing their behavior? The third paper explores what knowledge can be gained from a webpage about a user's installed browser extensions. It takes advantage of *web accessible resources (WARs)*, which are public resources for an extension [17]. These WARs can be fetched from any webpage, indicating that all extensions that have at least one listed WAR can easily be detected.

This work includes downloading all free extensions for Chrome [15] and Mozilla [23], as well as crawling the Alexa top 100,000 pages and analyse the requests made, to determine if this technique is widely used. It also includes potential measures one can implement in order to avoid this kind of detection.

Statement of contribution This paper was co-authored with Steven Van Acker and Andrei Sabelfeld. Alexander's contributions was the extensions experiment (all but the Alexa part), as well as defining the measures and develop the prototype for detecting extensions. All authors contributed equally to writing the paper.

Appeared in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy (CODASPY), Scottsdale, AZ, USA, March 2017*

5 Differences between Paper I and Paper II

Although the first two papers of this thesis handle the same topic, there are differences between them. At a high level, Paper II is a superset of Paper I, where the main difference is the addition of more language features, such as records and references. However, when looking at the semantic modelling, there are several key differences that enable the combination of first class mutable state and higher-order functions.

In Paper I, models are defined for library functions, explaining how to *unlabel* the parameters and *relabel* the result, where the removed labels are stored in a *model state*, which is a map from identifiers to labels. Updates to the models occur when data is being marshaled from the program to the library. With the addition of structured data and higher-order functions, the model state is tied to the wrapper functions via copying when the relabeling operation occurs. Unfortunately, this is not extendable to references, as they require a *shared mutable state*. Consider the following code snippet, where the code above `%%` is the program code and the code below is the library code.

```
1  let (g, r) = lib f 10
2  in r := upg 15 H;
3     g 10
4  %%
5  f x =
6    let r = ref x
7    in (\y . !r, r)
```

The library function creates a reference to `x` on line 6, and uses this for a returned tuple with the first element being a callback function which dereferences the reference, and the second element is the actual reference on line 7. What would be the effect in Paper I when the program code is being executed? On line 1, the program binds the returned tuple from calling `f` to `(g, r)`. It then writes the high value 15 to `r` (line 2), before calling the returned function `g` (line 3). Both values in the tuple should work *on the same reference*. However, Paper I would fail to model this due to the state not being shared. When relabeling, the wrappers for `g` and `r` would be given a copy of the model, making the model update from writing to `r` happen locally in the wrapper for `r`. The end result would be `r` being the secret value 15, and the result of calling `g 10` would be the public value 15, as the value is written to the reference, but the model of `g` is not updated.

The conclusion is that reference models must be shared between all values that have access to the reference. To do this, Paper II moves to a stack/heap based structure. The stack contains pointers to model frames, which reside on the heap. The frames on the heap represent scopes, and form scope chains in combination with the stack. In Paper II, the wrapper functions receives pointers to model frames, which allows for a shared view of the model frames residing on the heap. References and callbacks returned from the library are now tied to the local scope of the function. The example above would now work differently. As the wrappers for `g` and `r` are defined in the same function, the system in Paper II would copy the same frame pointer stack to both wrappers. When the writing to the returned reference occurs on line 2, the written labeled value would be unlabeled, updating the model frame pointed to by the frame pointer with the secret label. As `g` and `r` have copies of the same pointers, the updated model of the reference in the library is seen when `g 10` is called, returning a secret value.

In Paper II, the model state is divided into two parts: the *library model state* and the *call model state*. The library model state contains the information needed to lift a library function to the labeled world, i.e. the library model state is used to lift the library function `f` on line 1. As the lifted library function expects unlabeled parameters when it is invoked, the call model state will hold the labels of the parameters, i.e. the call model state will hold the labels from lines 2 and 3. Everything defined in the library will share the same library model state and the call model states are linked via the stack of pointers. Since the created wrappers can have copies of the same stack of model frame pointers, modifications to the model frames pointed to in the shared library model state residing on the heap is seen by everyone who has a model frame pointer to that model frame. This ensures the same view of the library state, even in the presence of mutability.

6 Bibliography

[1] Adobe: Adobe Acrobat Force-Installed Vulnerable Chrome Extension. https://bugs.chromium.org/p/project-zero/issues/detail?id=1088. accessed: March 2018.

[2] C. S. Advisory. Cisco WebEx Browser Extension Remote Code Execution Vulnerability. https://tools.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-20170717-webex. accessed: March 2018.

[3] A. Agarwal. Security update and new features. accessed: March 2018.

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

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

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

[7] C. Baraniuk. Ashley Madison: 'Suicides' over website hack. http://www.bbc.com/news/technology-34044506. accessed: March 2018.

[8] BBC News. Adobe hack: At least 38 million accounts breached. http://www.bbc.com/news/technology-24740873. accessed: March 2018.

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

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

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

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

[13] S. Gibbs. Dropbox hack leads to leaking of 68m user passwords on the internet. accessed: March 2018.

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

[15] Google. Chrome web store. https://chrome.google.com/webstore/category/extensions?hl=en-GB&_feature=free. accessed: March 2018.

[16] Google. Content Scripts. https://developer.chrome.com/extensions/content_scripts. accessed: March 2018.

[17] Google. Manifest - Web Accessible Resources. https://developer.chrome.com/extensions/manifest/web_accessible_resources. accessed: March 2016.

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

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

[20] INRIA. The Coq Proof Assistant. https://coq.inria.fr/. accessed: March 2018.

[21] J. Keane. Security researcher dumps 427 million hacked Myspace passwords online. https://www.digitaltrends.com/social-media/myspace-hack-password-dump/. accessed: March 2018.

[22] B. Krebs. Online Cheating Site AshleyMadison Hacked. https://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/. accessed: March 2018.

[23] Mozilla. Most Popular Extensions. https://addons.mozilla.org/en-US/firefox/extensions. accessed: March 2018.

[24] J. Pagliery. Hackers selling 117 million LinkedIn passwords. http://money.cnn.com/2016/05/19/technology/linkedin-hack/index.html. accessed: March 2018.

[25] Panopticlick. https://panopticlick.eff.org/.

[26] A. Peterson. https://www.washingtonpost.com/news/the-switch/wp/2014/12/18/the-sony-pictures-hack-explained/?utm_term=.b648dc649bac. accessed: March 2018.

[27] B. Quinn and C. Arthur. PlayStation Network hackers access data of 77 million users. https://www.theguardian.com/technology/2011/apr/26/playstation-network-hackers-data. accessed: March 2018.

[28] O. Räisänen. Trackers leaking bank account data. http://www.windytan.com/2015/04/trackers-and-bank-accounts.html. accessed: March 2018.

[29] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF*, 2010.

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

[31] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, pages 40–58, 1999.

[32] Symantec Official Blog. Shellshock: All you need to know about the Bash Bug vulnerability. `https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability`. accessed: March 2018.

[33] Synopsys. The Heartbleed Bug. `http://heartbleed.com/`. accessed: March 2018.

[34] D. M. Volpano, C. E. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[35] 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

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

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.

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

[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

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

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

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

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

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

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

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

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

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

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)**.**

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

Proof. By induction on the height of the derivation tree $\hat{\delta} \models e \to \hat{v}$. 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.

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

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.

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

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 $[\,\varphi\,]_{\alpha}$.

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

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.

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

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.

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

Unlabeled and labeled semantics The additions to the labeled semantics, found in Figure 1.3, are straightforward given the above modeling. Let $\mathsf{lcons}(\hat{v}_1, \hat{v}_2) = (\lambda().\,\hat{v}_1, \lambda().\,\hat{v}_2)$ be the creation of labeled cons cells[2], used in

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

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

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

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

Figure 1.3: Labeled semantics of lists

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.*

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

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

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

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

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

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

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

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

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

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

Figure 1.4: Unlabeled semantics of lists

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, $x :: \gamma$.

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

$$\begin{array}{rl} d ::= & \mathit{let}\ x = e \\ m ::= & x :: \gamma \end{array}$$

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

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

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

[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.

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.

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

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 $\boldsymbol{\gamma} \to \varphi$ 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.

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

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

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

Figure 1.5: Labeled semantics for higher-order functions

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.

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

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.

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

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.

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

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

Figure 1.6: Unlabeled semantics for higher-order functions

.

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.

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

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 $\texttt{f}_{lib}$ `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)**.**

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

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

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

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

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

  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** *let* $x = e_1$ *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:

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

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:

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

  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`, `if`$_1$, `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:

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

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:

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

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

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

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:

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

  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:

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

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

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

  Have $\hat{F}^{\ell} = \mathrm{label}(F, \xi_0, \boldsymbol{\varphi} \to \gamma, \boldsymbol{\zeta})^{[\![\kappa]\!]_{\xi_0}} = \hat{F}'^{\ell'}$ 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 $\hat{\boldsymbol{v}} \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}, \xi')$, $[\![\boldsymbol{\zeta}]\!]_{\xi_0 \amalg \xi'} = \xi_2$, $F(\xi_2, \boldsymbol{v}) = (\xi_3, v)$ from expansion of label in 5.

Have 11) $v' \uparrow_{\xi'_3} \gamma = \hat{F}'(\hat{\boldsymbol{v}}')$, where $\hat{\boldsymbol{v}}' \downarrow \boldsymbol{\varphi} = (\boldsymbol{v}', \xi')$, $[\![\boldsymbol{\zeta}]\!]_{\xi_0 \amalg \xi'} = \xi_2$, $F(\xi_2, \boldsymbol{v}') = (\xi'_3, v)$ 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 0) $\boldsymbol{x} = x \cdot \boldsymbol{x}_r$

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

Under submission

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 [25]. 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* [6], 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 [15]. 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 [14], and experimental extensions of JSFlow [16]. 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. initiate a framework for tracking information flow in libraries [19]. 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. This 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 and 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 in [19]. 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.

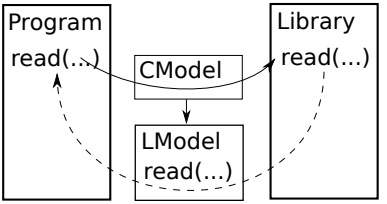


Figure 2.1: Model heap illustration

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 is illustrated by the dotted arrow in the figure by means of wrapping and results in a 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.

Our work explores the boundary between shallow and deep models. The library model language we present allows for modeling information flow inside the library functions and library state. Since our models have access to runtime labels when marshaling they can be characterized as being in between shallow and deep models. Further, it is relatively easy to extend our system to allow models to use the runtime values allowing for *dependent models* [19].

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 val-

ues (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 [11], 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 [21].

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 [1].

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.

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

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 $(\underline{d}, \underline{d}, \underline{m})$, 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 functions or 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), or 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, label 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 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 $\varsigma, \Gamma \models (\Sigma_1, e) \rightsquigarrow (\Sigma_2, v)$.

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.

Function closures are represented as functions, $\hat{F} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \rightarrow (\Sigma_2, \hat{v})$,

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

$$\text{deref}\ \frac{\begin{array}{c}\varsigma, \Gamma \models (\Sigma_1, e) \to (\Sigma_2, (\hat{R}, \hat{W})^\ell) \\ \hat{R}(\varsigma \sqcup \ell, \Gamma, \Sigma_2) = (\Sigma_3, \hat{v})\end{array}}{\varsigma, \Gamma \models (\Sigma_1, !e) \to (\Sigma_3, \hat{v}^\ell)}$$

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

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

$$\text{app}\ \frac{\begin{array}{c}\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{F}^\ell) \\ \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_1) \\ \hat{F}(\varsigma \sqcup \ell, \Gamma, \Sigma_3, \hat{v}_1) = (\Sigma_4, \hat{v}_2)\end{array}}{\varsigma, \Gamma \models (\Sigma_1, e_1\ e_2) \to (\Sigma_4, \hat{v}_2^\ell)} \qquad \text{lib}\ \frac{\begin{array}{c}\mathsf{lookupU}(\Gamma, \Sigma, x) = v \\ \mathsf{lookupM}(\Gamma, \Sigma, x) = \gamma \\ v \uparrow_{\Gamma, \Sigma} \gamma = \hat{v}\end{array}}{\varsigma, \Gamma \models (\Sigma, x_{lib}) \to (\Sigma, \hat{v})}$$

Figure 2.2: Selected labeled semantics

created by lclos (`fun`) in the following way

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

When applying a function closure (`app`), the right environment is created and the body of the function is executed in the program semantics. Creation and application of library closures, $F : (\varsigma, \Gamma, \Sigma_1, v) \to (\Sigma_2, v)$, is analogous.

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, $\hat{W} : (\varsigma, \Gamma, \Sigma_1, \hat{v}) \to \Sigma_2$. Creation of references given a fresh pointer into the labeled heap is defined by lread and lwrite as follows.

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

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. 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 is lower or equal to the label of the reference, $\varsigma \sqsubseteq \ell$. Allowing labels of values to change freely leads to an unsound system, due to the possibility of implicit flows into the labels themselves [2, 27].

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 an 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.

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

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.

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

The process is controlled by the function relabel model, $(\varphi \rightarrow \gamma, \underline{\zeta})^{\kappa}$, 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 stack, the unlabel model for the parameters, φ, the relabel model for the return value, γ, and the effect constraints, $\underline{\zeta}$.

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

When called the program closure produces a fresh frame pointer, pointing to a new 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 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.

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

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.

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

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.

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

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.

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

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 labeling the result.

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

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

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

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

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 stack, the unlabel model, φ, and the relabel model, γ.

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

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.

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 In a language like C, where it is slightly cumbersome to return more than one result, the use of writebacks is relatively common. For instance, when reading a file the read function is provided with 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 label the interaction.

```
let buf = ref 0
fun main () = (lib action) buf;
              !buf
%%
let data = 42
mod action :: #b -> L {| #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. A typical example is `errno`, found in the standard C library. `errno` is used by library functions, where it is possible to indicate an error using the return value, but where the error code would overlap with the normal return values of the function.

```
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 ();
               ()
```

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 label 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.

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 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 [11]. 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 events `open` and `close`, as well as the fields `bytesWritten` and `path`. 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 [26].

6 Correctness

We prove non-interference as the preservation of a low-equivalence relation under execution. Apart from covering a larger language, the proof improves over [19] in two important aspects: 1) it significantly weakens the model hypothesis, and 2) the proof has been formalized in Coq [21].

Theorem 4 (Non-interference of labeled execution)**.**

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

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. It also makes assumptions on the library models. The low-equivalence definition and more information on the proof can be found in Appendix E and Appendix F, respectively. □

7 Related work

Bielova and Rezk present a comprehensive taxonomy of information flow monitors [5]. Some monitors [17, 16, 15, 4] and secure multi-execution [14, 13, 24, 7, 22] mechanisms have been integrated in a browser. Bichhawat et al. instrumented the WebKit JavaScript interpreter [4]. While taking advantage of the current optimizations in the interpreter, it loses the differentiation between the program and library execution. FlowFox [14], which implements *secure multi-execution (SME)* [7], 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 [14]. JSFlow [17] is a 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 [16]. To allow for scaling, JSFlow attempts to automatically wrap libraries, albeit in a 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. In [3], Bauer et al. 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 non-interference. Heule et al. 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 [20]. A first step for handling libraries in environments where dynamic information flow control is not possible was taken by Hedin et al. [19], 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 [10] 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 [9]. If a contract is violated, the proper assignment of blame must be given [8, 12]. In static information flow checking, the assignment of blame has been investigated by King et al. for information flow violations [23]. 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.

9 Bibliography

[1] Information Flow Tracking for Side-effectful Libraries - Full version. `https://sites.google.com/site/iftfselextra/`.

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

[3] 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.

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

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

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

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

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

[9] 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.

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

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

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

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

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

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

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

[20] 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.

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

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

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

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

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

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

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

A Full syntax

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

B Full labeled semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\text{cons}\ \frac{\begin{array}{c}\varsigma, \Gamma \models (\Sigma_1, e_1) \to (\Sigma_2, \hat{v}_1) \\ \varsigma, \Gamma \models (\Sigma_2, e_2) \to (\Sigma_3, \hat{v}_2)\end{array}}{\varsigma, \Gamma \models (\Sigma_1, e_1 : e_2) \to (\Sigma_3, \mathsf{lcons}(\hat{v}_1, \hat{v}_2)^L)}$$

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

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

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

C Full unlabeled semantics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D Remaining constructs

In this section, we will cover the remaining constructs; primitive values (integers and tuples), lists and records. First, we define the evaluation relations for lists and records, before explaining how the marshaling is done for the different values.

D.1 Evaluation relations

Lists are represented in a similar fashion as references. 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, $\hat{T} : (\varsigma, \Gamma, \Sigma) \to (\Sigma, \hat{v})$. Creation of lists, given two labeled values, is defined by `lcons` as follows.

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

Lists 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 values (the head and tail of the cons cell) will be wrapped using lcons, to allow for delaying computations. 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 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 almost analogous. The difference is when calling H and T. As there is no cons cell label, H and T is called with the current execution environment only.

Records are represented as functions $\hat{O} : (\varsigma, \Gamma, \Sigma, p) \to (\Sigma, \hat{v})$, created by lproj (`record`) in the following way.

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

When interaction with a record (`projection`), the searched 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.

D.2 Marshaling

This section will explain how the marshaling is done for the remaining constructs, namely primitive values, lists and records.

Primitive values

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 while threading the memory, before binding ℓ to the name α.

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

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.

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

Lists

When marshaling lists, it is important that it is done lazily. Lazy marshaling of a list dictates that only the traversed elements should affect the unlabeling and relabeling of a list. The marshaling must also ensure that the corresponding values from the head and tail functions are unlabeled and relabeled accordingly.

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.

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

So far, there is no real sign of the lazy marshaling, but there is a sign of accumulation. However, the lazy marshaling can be seen when looking at the actual translation, done by u-lhead and u-ltail. The translation of $\hat{H}$ uses u-lhead, which takes the labeled head function, the current model stack, an unlabel model for unlabeling the resulting labeled value, and the label of

the cons cell.

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

Whenever the resulting unlabeled head function is called, the labeled head function $\hat{H}$ is used to get the corresponding labeled value, while being executed in the current context environment. The resulting labeled value must be unlabeled w.r.t. the bounded unlabel model φ before being returned to the library. Note that while the call to the labeled head function uses the current context environment, the unlabeling uses the bounded model state $\underline{\ddot{\rho}}'$.

Similarly, the labeled tail function $\hat{T}$ is translated using u-ltail.

$$\begin{aligned}&\text{u-ltail}(\hat{T}, \ell, \underline{\ddot{\rho}}', [\varphi]^{\alpha}) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma_1).\ (\Sigma_3, v)\\ &\quad \text{where}\\ &\qquad (\Sigma_2, \hat{v}) = \hat{T}(\varsigma \sqcup \ell, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma_1)\\ &\qquad (\Sigma_3, v) = \hat{v}^{\ell} \downarrow_{\varsigma \sqcup \ell, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}'), \Sigma_2} [\varphi]^{\alpha}\end{aligned}$$

As with unlabeling of $\hat{H}$, when the unlabeled tail function is called, the labeled tail function is used to get the corresponding labeled value. Since evaluating the labeled 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 stack $\underline{\ddot{\rho}}'$, and the unlabeled value along with the updated memory gets returned.

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

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

The case for the empty list is trivial. The label term κ is interpreted in the current model state, and the corresponding label is used for labeling the list. When relabeling a unlabeled cons cell, (H, T), each function is relabeled separately, with the pair of labeled function being tagged with the interpretation of κ. Translating the unlabeled head function H is done by using l-uhead.

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

When interacting with the labeled result, the current execution environment is taken as parameter. The unlabeled 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 stack $\underline{\ddot{\rho}}'$.

Relabeling of the unlabeled tail function T is done in a similar fashion.

$$\begin{aligned}&\text{l-utail}(T, \underline{\ddot{\rho}}', [\gamma]^\kappa) = \lambda(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma_1).\ (\Sigma_2, \hat{v})\\ &\quad \text{where}\\ &\qquad (\Sigma_2, v) = T(\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma)\\ &\qquad \hat{v} = v \uparrow_{(\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}'), \Sigma_2} [\gamma]^\kappa\end{aligned}$$

Upon interaction with the labeled result, the unlabeled tail function T is used to calculate the unlabeled value. As with unlabeling a labeled 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 then relabeled with the old model stack $\underline{\ddot{\rho}}'$, with the list relabel model $[\gamma]^\kappa$.

Marshaling records

When marshaling a record, as with lists, it is important that only the used properties of a record affects the model state when going back and forth between the labeled and the unlabeled world.

Unlabeling When passing a labeled record to the library, the model heap must be altered, by defining the name α to point to the label ℓ, which is the labeled records structure label.

$$\begin{aligned}&\hat{O}^\ell \downarrow_{\varsigma, (\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma_1} \{\ \alpha_1\ :\ \varphi_1, \ldots\}^\alpha =\\ &\qquad (\text{defineM}((\underline{\hat{\rho}}, \underline{\rho}, \underline{\ddot{\rho}}), \Sigma, \alpha, \ell), \text{u-lrec}(\hat{O}, \ell, \underline{\ddot{\rho}}, \{\ \alpha_1\ :\ \varphi_1, \ldots\}))\end{aligned}$$

The actual translation of the labeled record takes place using u-lrec, which creates the unlabeled counterpart.

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

When the resulting unlabeled record is called, the original labeled record is used to collect the labeled value corresponding to the property p. The record model is then used to collect the unlabel model φ, dictating how to unlabel the labeled value for property p. This is done with a generic function

mfind, which takes a record model and a property *id*, and returns the model corresponding to that property.

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

Note that, due to the structure being the same for record unlabel models and record label models, the function mfind will be used both when unlabeling and relabeling a record. Lastly, the labeled value is unlabeled with the unlabel model returned from mfind, using the old model stack $\underline{\ddot{\rho}}'$.

Relabeling Similar to lists, there are two types of relabeling for records; it can be an empty record or a record with properties.

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

If the record is empty, an empty record is returned, tagged with the interpretation of the label term κ in the current environment. However, if the record is non-empty, the auxiliary function l-urec is used, with the resulting labeled version being tagged with the interpretation of κ.

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

When interacted with, the relabeled record will use the unlabeled version to get the unlabeled value corresponding to the property p. The resulting unlabeled value is then labeled with the relabel model γ, corresponding to the relabel model returned from mfind for property p in the record relabel model. As with lists and unlabeling of records, the relabeling of the value takes place with the old model stack $\underline{\ddot{\rho}}'$, bound when the relabeling was first initiated.

E 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 stack and

model heap are low-equivalent. Low-equivalence is defined as follows.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

F 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. Non-interference 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 [1].

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 formalizations and suffice to establish the compatibility of lazy marshalling and the model state.

Non-interference We prove non-interference as the preservation of low-equivalence under execution.

Theorem 5 (Non-interference 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 non-interference proof uses non-interference 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 [19]. We have chosen to admit their proofs of non-interference 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 — non-interference 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 [19]. 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 [18] 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 non-interference 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.

G Heap operations

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

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

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

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

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

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

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

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

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

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

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

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

Paper III

Discovering Browser Extensions via Web Accessible Resources

Alexander Sjösten, Steven Van Acker, Andrei Sabelfeld

Proceedings of the Seventh ACM on 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

```html
<script src="showads.js">
<script>
  if(window.canRunAds === undefined)
  {
    // Ad blocking detected
  }
</script>
```

(a) HTML part of fake ad

```javascript
var canRunAds = true;
```

(b) showads.js (fake ad)

Figure 3.1: Ad-blocking behavioral detection

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 3.1 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 3.1 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

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

Table 3.1: Ad blocking arms race

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 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.

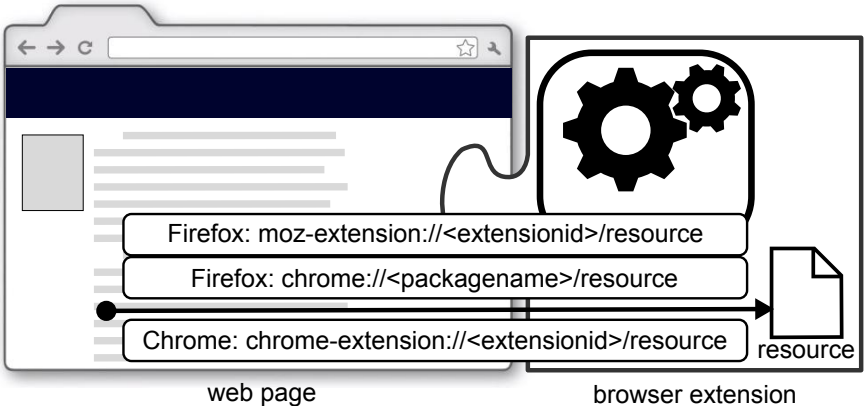


Figure 3.2: Extension - webpage overview

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 3.2 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 3.2 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.

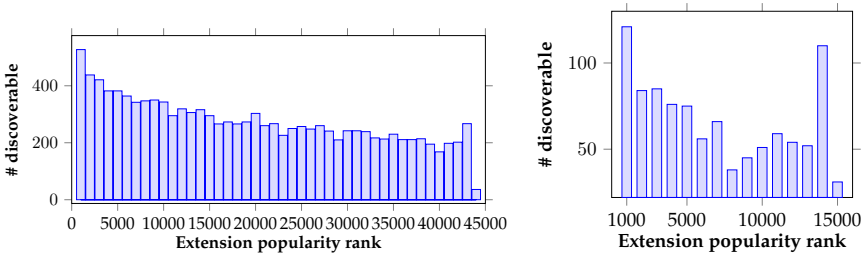
We note that 148 extensions have *web_accessible_resources* set to an empty array in the manifest file, which implies that these extensions have no web accessible resources. Similarly, the 54 extensions which only have URLs as web accessible resources cannot be found with our technique as they do not have resources that should run in the context of the website stored locally in the extension. The "No accessible resources" in Table 3.2 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 3.3a 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

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

Table 3.2: Chrome and Firefox extension results

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*.



(a) Chrome (b) Firefox

Figure 3.3: Discoverable browser extensions based on popularity

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 3.2. 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 3.3b. 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.

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 3.3: 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.

Table A.1 lists the domains in the Alexa top 100,000 which attempted to access `chrome-extension://` URLs, while Table 3.3 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 3.3 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 3.3, 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.

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

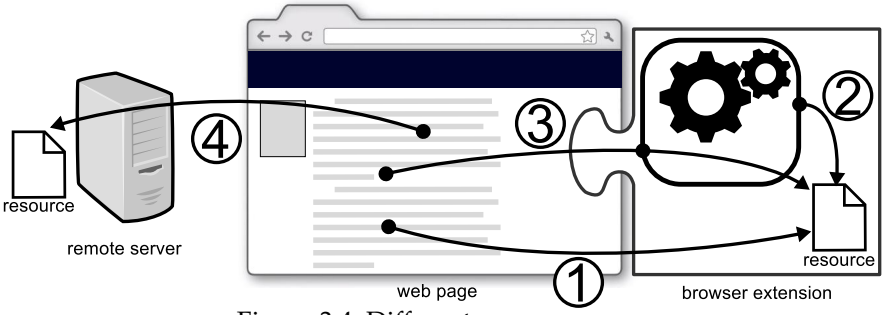


Figure 3.4: Different measures map

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 3.4 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 3.4). 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 3.4).

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 3.4) 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 3.4 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 3.4, 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 3.4) 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 network 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.

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 3.3: Chrome (Cext_*) and Firefox (Fext_*) extensions requested from Alexa top 100,000 sites

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	– – – – – –	✓ – – – – –

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.