Abstract

In many software systems as modern web browsers the user and his sensitive data often interact with the untrusted outer world. This scenario can pose a serious threat to the user's private data and gives new relevance to an old story in computer science: providing controlled access to untrusted components, while preserving usability and ease of interaction. To address the threats of untrusted components, modern web browsers propose privilege-separated architectures, which isolate components that manage critical tasks and data from components which handle untrusted inputs. The former components are given strong permissions, possibly coinciding with the full set of permissions granted to the user, while the untrusted components are granted only limited privileges, to limit possible malicious behaviours: all the interactions between trusted and untrusted components is handled via message passing. In this thesis we introduce a formal semantics for privilege-separated architectures and we provide a general definition of privilege separation: we discuss how different privilege-separated architectures can be evaluated in our framework, identifying how different security threats can be avoided, mitigated or disregarded. Specifically, we evaluate in detail the existing Google Chrome Extension Architecture in our formal model and we discuss how its design can mitigate serious security risks, with only limited impact on the user experience.

Contents

List of Tables

Chapter 1

Introduction

In the last few years there has been an increment in the usage of web applications such HTML5 apps and browser extensions. These increment also enhanced the popularity of the JavaScript language used to develop them. Unfortunately, while traditional languages has lots of tools made explicitly to help the programmer in the development and in the process of validating programs, JavaScript has few of them. Indeed, actual tools for JavaScript are limited and their purpose is no more than syntax highlighting and code completion. Moreover there are almost no static analysis tool to check and validate web applications. This lack of analyzer resulted in few controls on the software that must be done by programmers. An other aspect that we have to consider is that usually these programmers are not even security experts. This situation is very dangerous, even because web applications and browser extensions have often to interact with the untrusted outer world and with user sensitive data.

According to the need of security warranties, in this years various solutions has been adopted in different fields.

1.1 Background

1.1.1 Privilege separation

Privilege separated architectures are built to force developers to split the application in components giving to each only limited permissions. This choice reduces the attack surface of the application, because each component is isolated from the others, and reduces the impact of an attack since a compromised component has only limited privileges.

To preserve interaction between isolated components, the latter can communicate using a tight message passing interface. Such message passing interface constitutes a relevant attack surface against privilege-separated applications, because it may lead to privilege escalation attacks: a compromised, or malicious component that does not have a permission, can send messages asking other components to trigger security-sensitive operations.

Other interesting features of some privilege separated architectures are:

- permissions are given statically before the execution of the application in order to avoid privilege escalation attacks;

- and there is a strict punctual mapping between components and permissions in order to reduce privileges acquired by an attacker.

Privilege separated architectures are adopted in various fields, for example Google Chrome extension frameworks, Android and others.

1.1.2 Privilege escalation attacks

An attacker of a privilege separated architecture compromising a component can exercise only certain permissions, and not all the permissions given to the application. But the problem is that he can try to trigger execution of other privileges that the compromised component do not have. To do this he can use the message passing interface asking to another component to exercise high privileges. Let us have an example: we have two components A, B. Component A runs with high privileges and receives messages from B; B has instead low privileges and it asks to A to perform security sensitive operations. Assume also that the attacker can only compromise B. Compromising B, the attacker gains low privilege, and so his potentiality are limited. But he can ask to component A to exercise high privileges, and if A agreed, B can indirectly exercise such high privilege defeating the aim of privilege separation. These attacks are very common in various privilege separated architecture and are studied deeply in scientific literature [8, 11].

1.2 Chrome extension architecture overview

Chrome by Google, as all actual-days browsers, provides a powerful extension framework. This gives to developers a huge architecture made explicitly to extend the core browser potentiality in order to build small programs that enhance user-experience. In Chrome web store there are lots of extensions with very various behaviors like security enhancers, theme changers, organizers or other utilities, multimedia visualizer, games and others. For example, we point out AdBlock (one of the top downloaded), an extension made to block all ads on websites and ShareMeNot that "protects the user against being tracked from third-party social media buttons while still allowing it to use them"[5]. As we can notice, extensions have different purposes, and many of them has to interact massively with web pages. This creates a very large attack surface for attackers and it is a big threat for the user. Moreover, many extensions are written by developers that are not security experts so, even if their behavior is not malign, the bugs that can appear in them can be easily exploited by attackers.

To mitigate this threat, as deeply discussed in [7], the extension framework is built to force programmers to develop the software using privilege separation, least privilege and strong isolation:

1. Privilege separation, as explained in 1.1.1, forces the developer to split the application in components and it gives a message passing interface to permit the communication among them;

2. least privilege gives to the app the least set of permission needed through the execution of the extension;

3. the strong isolation separates the heaps of the various components of the extension running them in different processes in order to block any possible escalation and direct delegation.

This reduces the attack surface because while the least privilege sets a static upper bound on the possible permission exercised by the extension, the privilege separated architecture and strong isolation reduce the possible malignant operations performed by the attacker.

More specifically, Google Chrome extension framework [3] splits the extension in two sets components: content scripts and background pages.

The content scripts are injected in every page on which the extension is running, using the same origin. They run with no privileges except the one used to send messages to the background, and they cannot exchange pointers with the page, except to the standard field of the DOM.

Background pages have instead only one instance for each extension, are totally separated from the opened pages, have the full set of privilege granted at install time and, if it is allowed from the manifest, they can inject new content scripts to pages. Unfortunately they can communicate with the content scripts only via message passing.

1.3 Bundling

Studying some real Chrome extensions we notice that programmers tend to concentrate most of the privileges of the application in a single component. This is dangerous because if the attacker compromises that component, he escalates all privileges of the extension. Moreover Chrome extension permission system gives all permissions to the extension core, and no one to content script, so it is implicitly bundled [6]. Since the architecture suffers this problem extensions written by programmers that are not security experts sometimes suffer some kind of bugs that are exploitable by the attacker. In section 2.2 we will deeply discuss problems derived by bundling.

1.4 Purposes and methodology

The main aims of this work are to introduce an analysis of JavaScript code of real Chrome extensions, able to find security warranties on it and to build a real tool able to perform such analysis on the source.

Our work is composed by a first part in which we carried out a deep study of the state-of-the-art about Chrome Extensions framework looking both implementation specific details and scientific works. Then we worked pairwise on the formal analysis and on the implementation of a tool for statically checking an extension. Working together on both aspects of the analysis (formal and practical) helped us because some problems arisen in the proofs of the formal part guided the refinement of both the analysis and the tools. Moreover similarly problems found in the implementation also influenced both analysis.

In chapter 3 we will present a sound analysis that is able to determine which capability an extension leaks in extension in presence of an attacker, according to the power of the attacker. In other words having an extension with some components that exercise some

privileges, the analysis shows which privilege an attacker gains if it compromises the different components. In this way we can guarantee that an extension never leaks a privilege to a potential attacker. For better understanding the importance of our finding imagine a bank that wants to develop an extension to enhance user experience and safety of its online banking website. With our analysis the bank can statically know which permission are given to a possible attacker and be sure that an extension never allows the attacker to do a critical task.

We also developed a tool in Microsoft F# for statically checking real extensions. The tool starts from the analysis of the manifest of the extension, desugar the JavaScript source in the more succinct λ_{JS}(described below), parse the desugared sources and perform the analysis returning the possible pairs attacker permission, escalate permission. Since the analysis is sound, the tool gives an upper bound. So if the tool validate an extension, this will work for sure, otherwise if the tool fails, then the extension may fail.

Chapter 2

Background

2.1 Chrome extension architecture

As already discussed before, a Chrome extension is a software that extends the potentiality of the browser and enhances the user experience. Such extensions are not stand-alone software, but are integrated in the browser. The integration is done through a powerful API that expose to the developer lots of functionality of the browser. Since Chrome extensions interact with web pages and with browser them are developed using the classic web-style language: JavaScript. Extension can have even HTML or CSS files and can contains various resources that can be either local or remote resources as typical in the web.

As showed in [3] a Chrome Extension is an archive containing files of various kind like JavaScript, HTML, JSON, images and others that extends the browser features.

A basic extension is composed by a manifest file and one or more JavaScript or Html files.

2.1.1 Manifest

The manifest file `manifest.json` is a JSON-formatted file. It contains all the specification of the extension and for this reason is the entry-point of the extension. Indeed when an extension is load, the loader finds the manifest file and from it create the components of the extension. It contains two mandatory fields: `name` and `version` respectively containing the name and the version of the extension. Other important fields are:

- `background`: contains an object with either `script` or `page` field. The former contains the source of the content script, while the other the source of an HTML page. If the `script` field is used, the scripts are injected in a empty extension core page, while if it is used `page` the HTML document with all its elements (e.g., scripts) composes the extension core;

- `content_scripts`: contains a list of content script objects. Each object contains the field `matches`, a list of match patterns (Match patterns are explained below), and a field `js` containing the list of JavaScript source files to be injected;

Table 2.1 Url pattern syntax. Table taken from [2]

```
<url-pattern> := <scheme>://<host><path>
<scheme> := '*' | 'http' | 'https' | 'file' | 'ftp' | 'chrome-extension'
<host> := '*' | '*.' <any char except '/' and '*'>+
<path> := '/' <any chars>
```

- `permissions`: contains a list of privileges that are requested by the extension. These can be either a host match pattern for XHR request or the name of the API needed.

Another possible field is `optional_permissions`. It contains the list of optional permissions that the extension could require. It is used to restrict the privileges granted to the app. To use one of this permissions the background page has to explicitly require it and, after having used it, the permission has to be released. A program using the optional permissions can reduce the possible privileges escalated by an attacker.

A match pattern is a string composed of three parts: `scheme`, `host` and `path`. Each part can contain a value, or `"*"` that means all possible values. In table 2.1 is shown the syntax of the URL patterns; more details are reported in [2]. In this way we can decide to inject some content scripts only on pages derived from a given match. This is used when a content script of the extension has to interact with only certain pages. For example `"*://*/*"` means all pages; `"https://*/*"` means all HTTPS pages; `"https://*.google.com/*"` means all HTTPS domains that are subdomains of google with all their possible path (e.g., `mail.google.com`, `www.google.com`, `docs.google.com/mine/index.html`).

In table 2.2 we can see a manifest of a simple Chrome extension that expands the feature of moodle. We can see that the extension has an empty background page on which the file `background.js` is injected. It also has permissions `tabs` and `download`, and can execute XHR to pages with every path in `https://moodle.dsi.unive.it/`. It has also one content script that is injected in all subpages of `https://moodle.dsi.unive.it/`.

2.1.2 Content scripts

Content scripts are JavaScript source files that are automatically injected to the web page if this matches with the pattern defined in the manifest. Otherwise it can be programmatically injected by a background page using the `chrome.tabs.executeScript` call (the function requires `tabs` permission). In the example of table 2.1 the JavaScript file `myscript.js` is injected to all sub-pages of `https://moodle.dsi.unive.it/`. In the extension framework content scripts are designed to interact with pages. Since this interaction could be the entry point for an attacker, content scripts have no permissions except the one used to communicate with the extension core. In order to reduce injection of code in the content script from a malign page, there is a strong isolation between the heaps of these two. Content scripts of the same extension are run together in their own address space, and the only way they have to interact with the page on which they are injected is via the DOM API. DOM API lets the content scripts access and modify only standard fields of the DOM object, while other changes are kept locally[7]. This strong

Table 2.2 A manifest file

```
{
  "manifest_version": 2,
  "name":"Moodle expander",
  "description":"Download homework and uploads marks from a JSON
     string",
  "version":"1",
  "background": { "scripts": ["background.js"] },
  "permissions":
    [
      "tabs",
      "downloads",
      "https://moodle.dsi.unive.it/*"
    ],
  "content_scripts":
    [
      {
        "matches": ["https://moodle.dsi.unive.it/*"],
        "js": ["myscript.js"]
      }
    ]
}
```

isolation mitigate the risk of code injection since it blocks almost completely pointer exchange. In figure 2.1 is showed such relation.

In order to keep functionality of extensions, communication between content scripts and extension core is done using a message passing interface. The message passing interface has crucial importance in this work since it is the only way for a content script to trigger execution of a privilege. We will discuss it later in 2.1.4.

2.1.3 Extension core

The extension core is the most critical part of the application. It is executed in a unique origin like `chrome-extension://hcdmlbjlcojpbbinplfgbjodclfijhce` in order to prevent cross origin attacks, but it can communicate with all origins that match with one of the host permission defined in the manifest. In this environment are executed all scripts defined in the background field of the manifest. Since background pages can have remote resources of every kind (even scripts), they can also request to the web such resources, but this can be very dangerous. In fact if the resources are on HTTP connections these can be altered by an attacker. [9] describes how to enforce the security policy in order to avoid such possible weakness. Background pages can interact with content scripts via message passing.

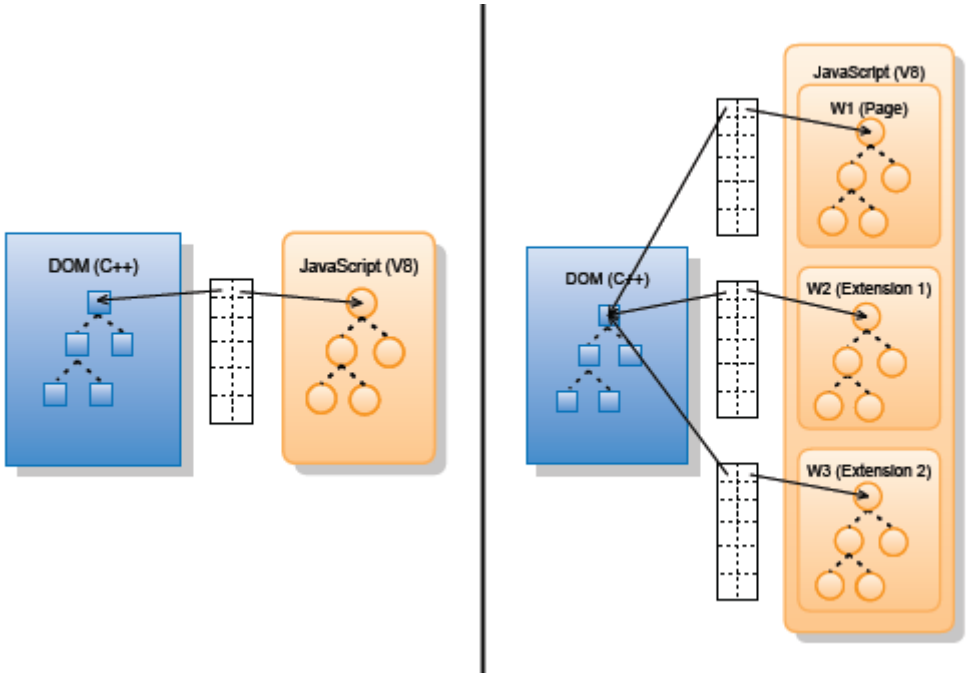

Figure 2.1: On the left the normal one-to-one relation between DOM implementation and JavaScript representation; On the right the one-to-many relation caused by running content scripts in isolated worlds. Figure taken from [7]

2.1.4 Message passing API

Every content script of the extension can use the message passing interface. To do this it has to use the functions contained in the `chrome.runtime` object [4] that exposes such API.

The main way to send a message to the extension core is invoking the method `chrome.runtime.sendMessage`. Like all Chrome APIs even the message passing is asynchronous. As primary arguments it takes the message that can be of any kind and a callback function that is triggered if someone answer to the message. Before sending, the message is marshaled using a JSON serializer.

In order to listen to incoming messages, a component has to register a function on the `chrome.runtime.onMessage` event. This function will be triggered when a message arrives. Its arguments are the message (unmarshalled by the API), the sender and an optional callback used to send response to the sender of the message. The sender field is very important because it is the only way to know the real identity of the sender. In fact the message may not be used to decide the sender, because it can be of every kind.

Since content scripts are multiple and injected in various pages (tabs), the extension core for sending a message has to use the `sendMessage` method of the `tab` object to which the message has to be sent. Its behavior is the same of the `chrome.runtime.sendMessage` method.

In table 2.3 we can see how to use the simple message passing interface. A component simply sends the message and wait for a response. The other registers `onMessage` function in the event listener `onMessage`. When the handler is triggered by an incoming message `onMessage` function it checks the message and decides to compute something according

to the request or refuses the message doing nothing.

Another way to communicate, that is more secure, is using channels as in table 2.4. In the message passing API there is a method called `connect` that triggers the corresponding event listener `onConnect` and returns a port. It has as optional arguments the name of the channel that is creating. A port object is a bidirectional channel that can be used to communicate. It contains the methods `postMessage`, `disconnect` and the events `onMessage` and `onDisconnect`. Communication using ports instead of the classical `chrome.runtime.sendMessage` is more secure, because only who has one of the port endpoint can communicate. Obviously ports are not serializable, so it is impossible to leak the ownership of a port. Ports provide a guarantee for the sender of the message.

Table 2.3 Sending a message.

Sender

```
var info = "hello";
var callback =
  function(response)
  {
    console.log("get response
       : " + response);
  };
chrome.runtime.sendMessage(
   info, callback);
```

Receiver

```
var onMessage =
  function(message, sender,
     sendResponse)
  {
    if (message = "hello")
    {
        //compute message
      sendResponse("hi");
    }
    else
      console.log("connection
          refused from"+
         sender);
  };
chrome.runtime.onMessage.
   addListener(onMessage);
```

2.2 Permission bundling

Modern privilege-separated architectures mitigates various attacks coming from the external untrusted world, but they still have weakness. A great part of this weakness derive from bad-practice of developers that often are not security experts. One of this is called bundling. Bundling is, as the name suggests, the practice of clustering in the same component different privileges. This can be very dangerous because an attacker that compromise the bundled component can escalate all privileges clustered in it.

In Chrome extensions sometimes programmers tend to aggregate in a single function various privileges that are delegated to different components. Moreover, often, the bundled function is the `onMessage` listener in the background, that is the entry point for an attacker that has compromised a content script. The `onMessage` function is, indeed,

Table 2.4 Port creation.

Port opening active

```
var port = chrome.runtime.
   connect({name: "cs1"});
port.onMessage.addListener(
   onMessage)
port.postMessage("hi")
```

Port opening passive

```
var scriptPort = null;
var onConnect =
  function(port)
  {
    if (port.name = "cs1")
    {
      scriptPort = port;
      port.onMessage.
         addListener(
         onMessage);
    }
    else
    {
      console.log("connection
          refused");
      port.disconnect();
    }
  };
chrome.runtime.onConnect.
   addListener(onConnect)
```

critical because it receives messages from a possible compromised content script, since attacker cannot access directly the background page from the web thanks to the privilege separated and strong isolation behavior of the architecture. The practice of permission bundling, especially in the `onMessage` function is very dangerous because an attacker that compromises a content script can directly trigger the listener in order to escalate privileges.

To reduce the attack surface exposed by bundled components, is important to base the decision of which privilege has to be exercised on trusted elements. Indeed, as seen in table 2.3, the choice taken by the bundled component when a message is received can depend on various factors decided by the programmer.

Let us explain the example in 2.5: suppose to have three components Background, CS1 and CS2. CS1 can only send messages that has `"getPasswd"` as title and CS2 only `"executeXHR"`. Here the Background deduct the sender checking the title of the messages instead of of explicitly checking the argument `sender`. According to the check decides which privilege has to be executed. This practice exposes all the weakness of the bundling and is very dangerous because an attacker can compromise just one of the two content scripts and from that one can forge messages with any form to escalate a permission that it does not have in the original setting.

To mitigate such weakness, in chrome extensions, is important to check the sender field of the `onMessage` function in order to be sure of the sender. This cannot be enough

because, as discussed before, contents script that are injected on the same page share their memory, tab and origin, and the message passing interface are does not distinguish them. The fix of this weakness is to use ports instead of the `chrome.runtime.sendMessage` function in order to have different listener for each content script. In this way we unbundle the `onMessage` function, separating in the various listeners privileges. In table 2.6 are showed a not dangerous bundled code, and an unbundled code.

2.3 Flow logic

The goal of this work is to develop an analysis that is able to detect statically absence of bundling in real Chrome extensions. Moreover we want to do this automatically, so without any effort for the developer (e.g., code annotations or similar). Statically typed languages are easier to check because the typing discipline provide strong foundation for detecting the behavior of a program. On the contrary, the weak dynamically typed nature of JavaScript code makes the analysis more difficult. Since extension are written in JavaScript we have to face with problem deriving from a dynamic and weak typing discipline. Indeed, in JavaScript there are lots of quirks that made the analysis very hard with a classical typing approach. For example local and global scoping, passage of functions, and access of a property of an object using a string are very hard to handle statically. To achieve our purpose the analysis must track the flow of both control and data during the execution. In this scenario we used the flow logic approach because of its flexibility, high potential and easiness to use.

Flow logic, introduced in [23], is a static analysis approach that derives from state of the art in program verification and has been successfully used in research projects [16, 15]. It has its root in classical approaches of static program analysis [21] like control flow analysis [12], abstract interpretation, constraint based analysis and data flow analysis. Flow logic lets the specification to focus on when an analysis estimate is acceptable, instead of how to compute such estimate. Another property is that, like structural operational semantics, is adaptable to lots of programming paradigms. Finally it can be used with various levels of abstraction according to the implementation details that are needed, but can be easily translated from one level to another.

The principal levels of abstraction are grouped in some possible approaches: abstract versus compositional and succinct versus verbose. The abstract style is closer to standard semantics while the compositional one is more syntax directed. The succinct approach is similar to the typical style of type systems because it focuses the top part of the analysis, while the verbose approach traces all the internal information in cashes and are typical of the implementation of control flow analysis and constraint based analysis.

The modularity fits very well for analysis, because the abstract succinct style is very clean and expressive without dealing with implementation details, and from such specification is easy to commute it to a compositional verbose specification. From the latter is possible to build an algorithm for generating the set of constraints of a program and combining it with a simple constraint solver like the worklist algorithm [21] or with a more sophisticated ones like the succinct solver [22] or the BANSHEE solver [1], is possible to compute the estimate for a program.

Let us have an example: suppose to have a program and “magically” an estimate for

it: using flow logic judgments we are able to establish if such estimate respect our analysis for the program, or not. This decision is granted by the fact that one of the constraint of generated by the constraint generator is not satisfied. Moreover we can compute starting from the constraint an estimate that satisfy the analysis.

In this work, is used the flow logic, and as described before, the various specification-to-implementation steps are done. In chapter 3 is used an abstract-succinct approach, and in chapter 4 the analysis is expanded in the compositional-verbose one; from this the algorithm for constraint-generation is built and finally is used a worklist algorithm to solve the constraints and to find an estimate for a program.

2.4 Lambda JS

Since JavaScript is a real world programming language that has high level constructs, weak and dynamic typing discipline and unconventional semantics, it is very complex to analyze. We reduce JavaScript to λ_{JS} as described in [13]. λ_{JS} is a dialect of Scheme, with a small-step operational semantics, that, in contrast with JavaScript, has few standard constructs taken from the lambda calculus. λ_{JS} models features of JavaScript in a way that corresponds closely to well known languages semantics.

Thanks to this we can simplify a lot the analysis without losing expressiveness because λ_{JS} contains just few construct with simple semantics.

Unfortunately λ_{JS} is not proved to be a sound representation of JavaScript, but all test on desugared file showed that its semantic coincide with JavaScript. In figure 2.2 is shown the testing method adopted to validate semantics of λ_{JS}.

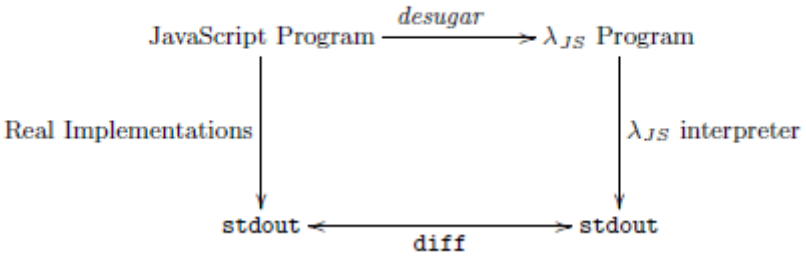

Figure 2.2: The λ_{JS} test. Figure taken from [13]

Table 2.5 Bundled code.

Background

```
function onMessage(message, sender, response)
{
  switch (message.title) {
    /* Requests from content script 1 */
    case "getPasswd":
    // get passwords
    response(passwd)
    break;

    /* Requests from content script 2 */
    case "executeXHR":
    var host = message.host
    var m = message.content;
    // execute XHR on args
    break;

    default:
    throw "Invalid request from contentScript";
  }
}
```

Content script CS1

```
var mess = {title: "getPasswd"};
chrome.runtime.sendMessage(mess);
```

Content script CS2

```
var mess = {title: "executeXHR", host: "www.google.com",
   content: "hi there"};
chrome.runtime.sendMessage(mess);
```

Table 2.6 Unbundling.

Unbundling checking sender

```
function onMessage(message, sender, response)
{
  switch (sender) {
    /* Requests from content script 1 */
    case CS1:
    // get passwords
    response(passwd)
    break;

    /* Requests from content script 2 */
    case CS2:
    var host = message.host
    var m = message.content;
    // execute XHR on args
    break;

    default:
    throw "Invalid request from contentScript";
  }
}
```

Unbundling using ports.

```
// Handler for messages from CS1
function onMessage_cs1(message, sender, response)
{
  /* Requests is content script 1 since it is on its port */
  // get passwords
  response(passwd)
}
// Handler for messages from CS2
function onMessage_cs2(message, sender, response)
{
  /* Requests is content script 2 since it is on its port */
  var host = message.host
  var m = message.content;
  // execute XHR on args
}
port_cs1.onMessage.addListener(onMessage_cs1);
port_cs2.onMessage.addListener(onMessage_cs2);
```

Chapter 3

Formalization

This chapter is about the formal part of the work and explains the calculus, the safety property, the analysis specification, theorem and requirements for correctness. It is part of the work done together with Stefano Calzavara.

3.1 Calculus

In this section we introduce the language used to study privilege escalation. The core of the calculus models JavaScript essential features and is a subset of λ_{JS}. λ_{JS} is a Scheme dialect described in [13] and used to desugar JavaScript in order to simplify it in few construct with easy semantic behavior. It has been used to do static analysis on JavaScript code in [14] and [17]. It admit functions, object (i.e., records) and mutable references. Here we are not using exceptions and break statements for the sake of simplicity. On the other hand, we added specific constructs to explicitly deal with privilege-based access control and privilege escalation. We rely on a channel-based communication model based on asynchronous message exchanges and handlers. Send expression and its corresponding handler are similar to the ones used in [8].

The desugaring function has an interesting feature since it translates JavaScript, that is not lexically scoped, to λ_{JS} that is lexically scoped. This simplifies the analysis because it removes the complexity of dealing with a scoping different to the statical one. In [13] is shown how the desugaring function transforms correctly these two different scoping approaches.

3.1.1 Syntax

Now we introduce the syntax of our calculus starting from values, expressions, memories, handlers, instances and systems. We then have a example to show how it works.

We assume denumerable sets of names $\mathcal{N}$ (ranged over by a, b, m, n) and variables $\mathcal{V}$ (ranged over by x, y, z). We let c range over constants, including numbers, strings, boolean values, unit and the "undefined" value; we also let r range over references in $\mathcal{R}$, i.e., memory locations. The calculus is parametric with respect to an arbitrary lattice of permissions $(\mathcal{P}, \sqsubseteq)$ and we let ρ range over $\mathcal{P}$. Finally, we assume a denumerable set of labels $\mathcal{L}$ (ranged over by ℓ) to support our static analysis. All the sets above are assumed pairwise disjoint.

Variables can be either bound or free. Expressions λ, "let", the handler $a(x \triangleleft \rho : \rho').e$ are binding operators for variables: the notions of free variables fv arise as expected.

Values.

We let u, v range over *values*, defined by the following productions:

$$c ::= num \mid str \mid bool \mid \mathbf{unit} \mid \mathbf{undefined},$$
$$u, v ::= n \mid x \mid c \mid r_\ell \mid \lambda x.e \mid \{\overrightarrow{str_i : v_i}\}.$$

A value could be either a name, a variable, a constant, a reference, a lambda or a record. A constant is either number, a string, boolean, unit (that means nothing) or undefined. Undefined is a JavaScript special value returned from an invalid lookup on an object field. Records are maps from string to a value and we consider them *closed*, without loss of expressiveness. This means that all lambdas in a record are closed, i.e., without free variables. Notice that references contains a label ℓ. This is needed in our static analysis, but has no role in the semantics.

Definition 1 (Serializable Value)**.** *A value v is* serializable *if and only if:*

- *v is a name, a constant, or a reference;*
- *$v = \{\overrightarrow{str_i : v_i}\}$ and each v_i is serializable.*

This means that a serializable value are only names, constants references and record containing only serializable values. Functions and variables cannot be serialized. This fits the model of Chrome extension message passing interface described in 2.1.4 because it let only to send JSON-serialized objects, or strings.

Expressions.

Since the calculus is functional there are no statements, but just expressions. We let e range over *expressions*, defined by the following productions:

$$\begin{array}{rcl} e, f & ::= & v \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid e\,e \mid op(\overrightarrow{e_i}) \mid \mathbf{if}\ (e)\ \{\ e\ \}\ \mathbf{else}\ \{\ e\ \} \mid \mathbf{while}\ (e)\ \{\ e\ \} \\ & \mid & e; e \mid e[e] \mid e[e] = e \mid \mathbf{delete}\ e[e] \mid \mathbf{ref}_\ell\ e \mid \mathbf{deref}\ e \mid e = e \mid \overline{e}\langle e \triangleright \rho \rangle \\ & \mid & \mathbf{exercise}(\rho). \end{array}$$

The operations $op(\overrightarrow{e_i})$ is used for all arithmetic, boolean and string operations. It includes also string equality, denoted by ==. The creation of new references comes with an annotation: $\mathbf{ref}_\ell\ e$ creates a fresh reference r_ℓ labelled by ℓ. As said before, ℓ plays no role in the semantics: annotating the reference with the program point where it has been created is useful for our static analysis.

We discuss the non-standard expressions. The expression $\overline{a}\langle v \triangleright \rho \rangle$ sends the value v on channel a: the value can be received by any handler listening on a, provided that it is granted permission ρ (this allows the sender to protect the message). The expression $\mathbf{exercise}(\rho)$ exercises the permission ρ. Indeed, in order to keep simple the calculus and to more clearly state our security property, we abstract any security sensitive expression (such as the call of a function library) with the generic exercise of the correspondent privilege. So, since the execution of a privilege is only used in API calls, the $\mathbf{exercise}(\rho)$ expression is placed in such libraries just for marking the permission execution.

Memories.

We let μ range over *memories*, defined by the following productions:

$$\mu ::= \emptyset \mid \mu, r_\ell \overset{\rho}{\mapsto} v.$$

A memory is a partial map from (labelled) references to values, implementing an access control policy. Specifically, if $r_\ell \overset{\rho}{\mapsto} v \in \mu$, then permission ρ is required to have read/write access on the reference r in μ. Given a memory μ, we let $dom(\mu) = \{r \mid r_\ell \overset{\rho}{\mapsto} v \in \mu\}$.

Handlers.

We let h range over multisets of *handlers*, defined by the following productions:

$$h ::= \emptyset \mid h, a(x \triangleleft \rho : \rho').e.$$

The handler $a(x \triangleleft \rho : \rho').e$ contains an expression e, which is granted permission ρ'. The handler is guarded by a channel a, which requires permission ρ for write access: this protect the receiver against untrusted senders. When a message is sent over a, the handler is triggered and the expression e will be disclosed and a new *instance* with e is created.

In other words when a $\overline{a}\langle v \triangleright \rho_s \rangle$ is executed on a channel a all handler $a(x \triangleleft \rho : \rho').e$ on the same channel a with permission $\rho' \sqsubseteq \rho_s$ may be triggered. Triggering a handler means that the message value v is bound to variable x in the environment of e and e is executed in a new *instance*.

Instances.

Instances are the active part of a system. They are spawned when a message is received by a handler and the function in it is executed. Instances contains a permission ρ that is the same granted to the handler that has spawned it. We let i range over pools of running *instances*.Instances are multisets defined as follows:

$$i, j ::= \emptyset \mid i, a\{\!|e|\!\}_\rho$$

Instances are annotated with the channel name corresponding to the handler which spawned them: this is convenient for our static analysis, but it is not important for the semantics.

Systems.

A *system* is defined as a triple $s = \mu; h; i$. It is the representation of a running extension in a certain moment. Its components are the memory μ that contains all the data referenced in the program, all the handler registered in it and all the running instances: listeners that have been triggered and that have not yet finished their execution. An initial state is a state where there are no instances $s_{initial} = \mu; h; \emptyset$.

Example.

Handlers can be used to model the single entry point of a Chrome component, which is represented by the the function `onMessage`. To understand the programming model, let's consider a simple protocol:

$$A \to B : \{tag : "init", val : x\}$$
$$B \to A : y$$
$$A \to B : \{tag : "okay", other : z\}$$

Here the component A sends to B a message containing $\{tag : "init", val : x\}$. Then B reply to A with y and finally A respond to B with $\{tag : "okay", other : z\}$ In Chrome, and in λ_{JS}, the handler of the component B is programmed as in table 3.1.

Table 3.1 onMessage handler in JavaScript and in λ_{JS}

JavaScript

```
void onMessage (Message m) {
    if (m.tag == "init")
        process_request (m.val) >> rho;
    else if (m.tag == "okay")
        process_other (m.other) >> rho';
    else
        do nothing;
};
chrome.runtime.onMessage.addListener(onMessage);
```

λ_{JS}

```
a(x <| SEND: BACK).
  if (== (x["tag"], "init"))
  {
    process_request (x["val"])
    exercise(rho)
  }
  else if (== (x["tag"], "okay"))
  {
    process_other (x["other"])
    exercise(rho')
  }
  else
    do_nothing
```

3.1.2 Semantics

In this section we introduce the semantic of our calculus. The small-step operational semantics is defined as a labelled reduction relation between systems, i.e., $s \xrightarrow{\alpha} s'$. The auxiliary reduction relation between expressions that is directly inherited from λ_{JS} semantic [13], i.e., $\mu; e \hookrightarrow_\rho \mu'; e'$. We associate labels to reduction steps just to state our security property easily and to provide additional informations in the proofs, however labels have no impact on the semantics.

Tables 3.2 and 3.3 collect the reduction rules for systems and expressions, where the syntax of labels α is defined as follows:

$$\alpha ::= \cdot \mid a : \rho_a \gg \rho \mid \langle a : \rho_a, b : \rho_b \rangle.$$

The step $s \xrightarrow{a:\rho_a \gg \rho} s'$ identifies the exercise of the privilege ρ by a system component a with privileges ρ_a, while the step $s \xrightarrow{\langle a:\rho_a, b:\rho_b \rangle} s'$ records the fact that an instance a with privilege ρ_a sends a message to an handler b allowing the spawning of a new b-instance running with privilege ρ_b. Any other reduction step is characterized by $s \xrightarrow{\cdot} s'$. We write $\stackrel{\vec{\alpha}}{\Rightarrow}$ for the reflexive-transitive closure of $\xrightarrow{\alpha}$.

Table 3.2 Small-step operational semantics $s \xrightarrow{\alpha} s'$

(R-Sync)

$$\frac{h = h', b(x \triangleleft \rho_s : \rho_b).e \qquad \rho_s \sqsubseteq \rho_a \qquad \rho_r \sqsubseteq \rho_b \qquad v \text{ is serializable}}{\mu; h; a\{\!|E\langle \overline{b}\langle v \triangleright \rho_r \rangle \rangle|\!\}_{\rho_a} \xrightarrow{\langle a:\rho_a, b:\rho_b \rangle} \mu; h; a\{\!|E\langle \mathbf{unit} \rangle|\!\}_{\rho_a}, b\{\!|e[v/x]|\!\}_{\rho_b}}$$

(R-Exercise)

$$\frac{\rho \sqsubseteq \rho_a}{\mu; h; a\{\!|E\langle \mathbf{exercise}(\rho) \rangle|\!\}_{\rho_a} \xrightarrow{a:\rho_a \gg \rho} \mu; h; a\{\!|E\langle \mathbf{unit} \rangle|\!\}_{\rho_a}}$$

(R-Set)

$$\frac{\mu; h; i \xrightarrow{\alpha} \mu'; h'; i'}{\mu; h; i, i'' \xrightarrow{\alpha} \mu'; h'; i', i''}$$

(R-Basic)

$$\frac{\mu; e \hookrightarrow_\rho \mu'; e'}{\mu; h; a\{\!|e|\!\}_\rho \xrightarrow{\cdot} \mu'; h; a\{\!|e'|\!\}_\rho}$$

Rule (R-Sync) implements a security cross-check between sender and receiver: by specifying a permission ρ_r on the send expression, the sender can require the receiver to have at least that permission, while specifying a permission ρ_s in the handler, the receiver can require the sender to have at least that permission. If the security check succeeds, a new instance is created and the sent value is substituted to the bound variable in the handler.

Evaluation contexts are defined by the following productions:

$$\begin{aligned} E \;::=\; & \bullet \mid \mathbf{let}\ x = E\ \mathbf{in}\ e \mid E\,e \mid v\,E \mid op(\overrightarrow{v_i}, E, \overrightarrow{e_j}) \mid \mathbf{if}\ (E)\ \{\ e\ \}\ \mathbf{else}\ \{\ e\ \} \mid E[e] \\ \mid\; & v[E] \mid E[e] = e \mid v[E] = e \mid v[v] = E \mid \mathbf{delete}\ E[e] \mid \mathbf{delete}\ v[E] \mid \mathbf{ref}_\ell\ E \\ \mid\; & \mathbf{deref}\ E \mid E = e \mid v = E \mid E; e \mid \overline{E}\langle e \triangleright \rho \rangle \mid \overline{v}\langle E \triangleright \rho \rangle. \end{aligned}$$

The full reduction semantics λ_{JS} is given in Table 3.3. The semantics comprises two layers: the basic reduction $e \hookrightarrow e'$ does not include references and thus permissions play

no role there; the internal reduction $\mu; e \hookrightarrow_\rho \mu'; e'$ builds on the simpler relation. Labels on references do not play any role at runtime: to formally prove it, we can define an unlabelled semantics (i.e., a semantics over unlabelled references) and show that, for any expression and any reduction step, we can preserve a bijection between labelled references and unlabelled ones, which respects the values stored therein. Intuitively, this is a consequence of (JS-Ref), which never introduces two references with the same name. Hence, there might be two references with the same label but different names, but no pair of references with the same name and two different labels.

We discuss some important points: in rule (JS-PrimOp) we assume a δ function, which defines the behaviour of primitives operations. In rule (JS-Ref) we ensure that running instances can only create memory cells they can access; in rule (JS-Deref) and (JS-SetRef) we perform the expected access control checks. For simplicity we excluded the rules for prototype inheritance of λ_{JS} with no impact on the analysis, but them are included in the implementation. The prototype inheritance of JavaScript is modeled in λ_{JS}as a recursion on the `__proto__` field if an attribute is not found in the current object, and if the `__proto__` field does not exist is returned the value `undefined`.

3.2 Safety despite compromise

Definition 2 (Exercise)**.**

- *A system* s exercises ρ *if and only if there exists* s' *such that* $s \stackrel{\overrightarrow{\alpha}}{\Rightarrow} s'$ *and* $a : \rho_a \gg \rho \in \{\overrightarrow{\alpha}\}$.
- *A system* s exercises at most ρ *iff* $\forall s', \overrightarrow{\alpha}$ *such that* $s \stackrel{\overrightarrow{\alpha}}{\Rightarrow} s'$*, if* $a : \rho_a \gg \rho' \in \{\overrightarrow{\alpha}\}$ *then* $\rho' \sqsubseteq \rho$.

This means that a system *exercises* ρ if and only if through its execution (reduction steps) a permission ρ is exercised and that a system *exercises at most* ρ if and only if all the permission required during all possible executions are lower than ρ. The second statement gives an upper bound on the permission required by the system.

We now introduce our threat model. We partition the set of variables $\mathcal{V}$ into two sets $\mathcal{V}_t$ (trusted variables) and $\mathcal{V}_u$ (untrusted variables). We say that all the variables occurring in a system we analyse are drawn from $\mathcal{V}_t$, while all the variables occurring in the opponent code are drawn from $\mathcal{V}_u$.

Definition 3 (Opponent)**.** *A* ρ-opponent *is a closed pair* (h, i) *such that:*

- *for any handler* $a(x \triangleleft \rho : \rho').e \in h$*, we have* $\rho' \sqsubseteq \rho$*;*
- *for any instance* $a\{\!|e|\!\}_{\rho'} \in i$*, we have* $\rho' \sqsubseteq \rho$*;*
- *for any* $x \in vars(h) \cup vars(i)$*, we have* $x \in \mathcal{V}_u$.

So an ρ-opponent is a pair of handlers and instances such that for each expression in the instances or in the handlers of it, the expression *exercise at most* ρ and all the variable used in the expression by the opponent are untrusted since it can modify their value.

Table 3.3 Small-step operational semantics of λ_{JS}

Basic Reduction:

(JS-PrimOp)
$op(\overrightarrow{c_i}) \hookrightarrow \delta(op, \overrightarrow{c_i})$

(JS-Let)
$\textbf{let } x = v \textbf{ in } e \hookrightarrow e[v/x]$

(JS-App)
$(\lambda x.e)\, v \hookrightarrow e[v/x]$

(JS-GetField)
$\{\overrightarrow{str_i : v_i}, str : v, \overrightarrow{str'_j : v'_j}\}[str] \hookrightarrow v$

(JS-GetNotFound)
$$\frac{str \notin \{str_1, \ldots, str_n\}}{\{\overrightarrow{str_i : v_i}\}[str] \hookrightarrow \textbf{undefined}}$$

(JS-UpdateField)
$\{\overrightarrow{str_i : v_i}, str : v, \overrightarrow{str'_j : v'_j}\}[str] = v' \hookrightarrow \{\overrightarrow{str_i : v_i}, str : v', \overrightarrow{str'_j : v'_j}\}$

(JS-CreateField)
$$\frac{str \notin \{str_1, \ldots, str_n\}}{\{\overrightarrow{str_i : v_i}\}[str] = v \hookrightarrow \{str : v, \overrightarrow{str_i : v_i}\}}$$

(JS-DeleteField)
$\textbf{delete } \{\overrightarrow{str_i : v_i}, str : v, \overrightarrow{str'_j : v'_j}\}[str] \hookrightarrow \{\overrightarrow{str_i : v_i}, \overrightarrow{str'_j : v'_j}\}$

(JS-DeleteNotFound)
$$\frac{str \notin \{str_1, \ldots, str_n\}}{\textbf{delete } \{\overrightarrow{str_i : v_i}\}[str] \hookrightarrow \{\overrightarrow{str_i : v_i}\}}$$

(JS-CondTrue)
$\textbf{if } (\textbf{true}) \{\ e_1\ \} \textbf{ else } \{\ e_2\ \} \hookrightarrow e_1$

(JS-CondFalse)
$\textbf{if } (\textbf{false}) \{\ e_1\ \} \textbf{ else } \{\ e_2\ \} \hookrightarrow e_2$

(JS-Discard)
$v; e \hookrightarrow e$

(JS-While)
$\textbf{while } (e_1) \{\ e_2\ \} \hookrightarrow \textbf{if } (e_1) \{\ e_2; \textbf{while } (e_1) \{\ e_2\ \}\ \} \textbf{ else } \{\ \textbf{undefined}\ \}$

Internal Reduction:

(JS-Expr)
$$\frac{e_1 \hookrightarrow e_2}{\mu; e_1 \hookrightarrow_\rho \mu; e_2}$$

(JS-Ref)
$$\frac{r \notin dom(\mu) \qquad \mu' = \mu, r_\ell \overset{\rho}{\mapsto} v}{\mu; \textbf{ref}_\ell\ v \hookrightarrow_\rho \mu'; r_\ell}$$

(JS-Deref)
$$\frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v}{\mu; \textbf{deref } r_\ell \hookrightarrow_\rho \mu; v}$$

(JS-SetRef)
$$\frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v'}{\mu; r_\ell = v \hookrightarrow_\rho \mu', r_\ell \overset{\rho}{\mapsto} v; v}$$

(JS-Context)
$$\frac{\mu; e_1 \hookrightarrow_\rho \mu'; e_2}{\mu; E\langle e_1 \rangle \hookrightarrow_\rho \mu'; E\langle e_2 \rangle}$$

Our security property is given over *initial* systems, i.e., a system with no running instances, since we are interested in understanding the interplay between the exercised permissions and the message passing interface exposed by the handlers. In particular, we want to understand how many privileges the opponent can escalate by leveraging existing handlers.

Definition 4 (Safety Despite Compromise)**.** *A system $s = \mu; h; \emptyset$ is* ρ-safe despite ρ' *(with $\rho \not\sqsubseteq \rho'$) if and only if, for any ρ'-opponent (h_o, i_o), the system $s' = \mu; h, h_o; i_o$ exercises at most ρ.*

In other words, This crucial definition state that an *initial* system is *ρ-safe despite ρ'* if each ρ'-opponent cannot alter the system in order to access to a privilege bigger than ρ.

3.3 Example

Consider an extension made of two content scripts $CS1, CS2$ and a background page B. Assume that $CS1$ sends only messages with tag `Message1` and $CS2$ sends only messages with tag `Message2`.

A simple encoding of the Google Chrome extension in our calculus is the following:

```
Content script 1:
  cs1(x <| CS1: SEND).b̄⟨{tag: "Message1"}▷BACK⟩
Content script 2:
  cs2(x <| CS2: SEND).b̄⟨{tag: "Message2"}▷BACK⟩
Background:
  b(x <| SEND: BACK).
    if (== (x["tag"], "Message1")) then exercise(ρ)
    else exercise(ρ′)
```

Assume that both ρ and ρ' are bounded above by BACK, while all the other permissions are unrelated. More sensible encodings are possible, but this is enough to present the analysis.

This scenario is composed by two content scripts and a background. Each content script is modeled by a handler that receives messages respectively on channel `cs1` and `cs2`, requires permissions `CS1` and `CS2` and exercises permission `SEND`. When triggered each content script sends a message to the background with tag `Message1` or `Message2` on channel `b` with permission `BACK`. The background waits for messages on channel `b` having at least permission `BACK`. Then, according to the tag of the message, exercise ρ or ρ'.

3.3.1 Privilege escalation analysis.

The idea is that each handler has a "type" which describes the permissions which are needed to access it, and the permissions which will be exercised (also transitively) by the handler. For instance, the example above is acceptable according to the following assumptions:

```
cs1: CS1   ---> ρ ⊔ ρ′
cs2: CS2   ---> ρ ⊔ ρ′
b:   SEND  ---> ρ ⊔ ρ′
```

These assumptions environment tells us that a caller with permission SEND can escalate up to $\rho \sqcup \rho'$. All these aspects are formalized in the *abstract stack* we introduce below and our novel notion of *permission leakage*, which quantifies the attack surface of the message passing interface.

3.3.2 Refining the analysis.

While it is perfectly sensible that an opponent with permission SEND can escalate both ρ and ρ', the typing above may appear too conservative if we focus, for instance, on an opponent with permission CS1. Indeed, an opponent with CS1 can access the first content script, but not directly the background page: since CS1 sends only messages of the first type, it would be safe to state that the opponent can only escalate ρ rather than $\rho \sqcup \rho'$, which is not entailed by the typing above.

Our analysis is precise, though, since it keeps track also of an abstract network, which approximates the incoming messages for all the handlers. In the example above we have:

```
cs1: TOP
cs2: BOTTOM
b:   {{tag:"Message1"}}
```

where TOP signifies that $cs1$ can be accessed by the opponent (hence any value can be sent to it), while BOTTOM denotes that $cs2$ will never be called. Having BOTTOM for $cs2$ is important, since our static analysis will not analyse the body of $cs2$, hence there is no need to include {`tag:"Message2"`} among the messages processed by b. Since the "else" branch in b is unreachable, we can admit the more precise typing:

```
cs1: CS1 ---> rho
b:   SEND ---> rho
```

which captures the correct information for a CS1-opponent (i.e., a CS1-opponent can only escalate ρ).

3.4 Analysis

In the analysis we predict statically which privileges an opponent can escalate through the message passing interface. To do this, as in abstract analysis, we approximate values an expression may evaluate to using abstract representation of concrete values. The abstract-succinct style flow logic specification that follows consists of a set of clauses defining a judgement expressing acceptability of an analysis estimate for a given program fragment.

In this section, the main judgement for the flow analysis of systems will be $\mathcal{C} \Vdash s$ **despite** ρ, meaning that $\mathcal{C}$ represents an acceptable analysis for s, even when s interacts with a ρ-opponent. We will prove in the following that this implies that any ρ-opponent interacting with s will at most escalate privileges according to an upper bound which we can immediately compute from $\mathcal{C}$.

3.4.1 Abstract Values and Abstract Operations.

Here we show the abstract values semantic, but we do not fix any specific representation leaving in the implementation the decision of the domains. Later we will state some properties required by abstract domains in order to prove soundness. Moreover we assume that abstract values are ordered by a pre-order $\sqsubseteq$.

In chapter 4 we describe the actual choice of the abstract domains used in the implementation and the operations on them and how they respect properties listed in section 3.6.

Let $\hat{V}$ stand for the set of the abstract values $\hat{v}$, defined as sets of abstract prevalues according to the following productions[1]:

$$\begin{array}{lrcl} \textit{Abstract prevalues} & \hat{u} & ::= & n \mid \hat{c} \mid \ell \mid \lambda x^{\rho} \mid \langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}, \\ \textit{Abstract values} & \hat{v} & ::= & \{\hat{u}_1, \ldots, \hat{u}_n\}. \end{array}$$

The abstract value $\hat{c}$ stands for the abstraction of the constant c. We dispense from listing all the abstract pre-values corresponding to the constants of our calculus, but we assume that they include **true**, **false**, **unit** and **undefined**.

A function $\lambda x.e$ is abstracted into the simpler representation λx^{ρ}, keeping track of the escalated privileges ρ. Since our operational semantics is substitution-based, having this more succinct representation is important to prove soundness. In the following we let $\Lambda = \{\lambda x \mid x \in \mathcal{V}\}$.

The abstract value $\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}$ is the abstract representation of the concrete record $\{\overrightarrow{str_i : v_i}\}$ in the environment $\mathcal{C}$, assuming permissions ρ. As said before, we do not fix any apriori abstract representation for records, i.e., both field-sensitive and field-insensitive analyses are fine.

We associate to each concrete operation op an abstract counterpart $\widehat{op}$ operating on abstract values. We also assume three abstract operations $\widehat{get}$, $\widehat{set}$ and $\widehat{del}$, mirroring the standard get field, set field and delete field operations on records. These abstract operations can be chosen arbitrarily, but they have to satisfy the conditions needed for the proofs.

3.4.2 Judgements.

The judgements of the analysis are specified relative to an abstract environment $\mathcal{C}$. The abstract environment is global meaning that it is going to represent *all* the environments that may arise during the evaluation of the system. We let $\mathcal{C} = \hat{\Upsilon}; \hat{\Phi}; \hat{\Gamma}; \hat{\mu}$, that is, the abstract environment is a four-tuple made of the following components:

$$\begin{array}{lrcl} \textit{Abstract variable environment} & \hat{\Gamma} & : & \mathcal{V} \cup \Lambda \to \hat{V} \\ \textit{Abstract memory} & \hat{\mu} & : & \mathcal{L} \times \mathcal{P} \to \hat{V} \\ \textit{Abstract stack} & \hat{\Upsilon} & : & \mathcal{N} \times \mathcal{P} \to \mathcal{P} \times \mathcal{P} \\ \textit{Abstract network} & \hat{\Phi} & : & \mathcal{N} \times \mathcal{P} \to \hat{V}. \end{array}$$

The abstract variable environment is standard: it associate abstract values to variables and to abstract functions. Abstract memory is also standard: if associate abstract values

[1] We occasionally omit brackets around singleton abstract values for the sake of readability.

to labels denoting references, but they also keep track of some permission information to make the analysis more precise. Specifically, if $\hat{\mu}(\ell, \rho) = \hat{v}$, then all the references labelled with ℓ contain the abstract value $\hat{v}$, and are protected with permission ρ.

Abstract stack is used to keep track of the permissions required to access a given handler and the permissions which are exercised (also transitively, i.e., via a call stack) by the handler itself. Specifically, if we have $\hat{\Upsilon}(a, \rho_a) = (\rho_s, \rho_e)$, then the handler a with permission ρ_a can be accessed by any component with permission ρ_s and it will be able to escalate privileges up to ρ_e, even by calling other handlers in the system.

Also abstract network is novel and it is used to keep track of the messages exchanged between handlers. For instance, if we have $\hat{\Phi}(a, \rho_a) = \hat{v}$, then $\hat{v}$ is a sound abstraction of any message received by the handler a with permission ρ_a.

To lighten the notation, we denote by $\mathcal{C}_{\hat{\Gamma}}, \mathcal{C}_{\hat{\mu}}, \mathcal{C}_{\hat{\Upsilon}}, \mathcal{C}_{\hat{\Phi}}$ the four components of the abstract environment $\mathcal{C}$.

Table 3.4 Flow analysis for values

$$\text{(PV-Name)}\ \frac{n \in \hat{v}}{\mathcal{C} \Vdash_\rho n \rightsquigarrow \hat{v}} \qquad \text{(PV-Var)}\ \frac{\mathcal{C}_{\hat{\Gamma}}(x) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_\rho x \rightsquigarrow \hat{v}} \qquad \text{(PV-Cons)}\ \frac{\{\hat{c}\} \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_\rho c \rightsquigarrow \hat{v}} \qquad \text{(PV-Ref)}\ \frac{\ell \in \hat{v}}{\mathcal{C} \Vdash_\rho r_\ell \rightsquigarrow \hat{v}}$$

$$\text{(PV-Fun)}\ \frac{\lambda x^{\rho_e} \in \hat{v} \qquad \mathcal{C} \Vdash_\rho e : \hat{v}' \gg \rho' \qquad \hat{v}' \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(\lambda x) \qquad \rho' \sqsubseteq \rho_e}{\mathcal{C} \Vdash_\rho \lambda x.e \rightsquigarrow \hat{v}} \qquad \text{(PV-Rec)}\ \frac{\{\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}\} \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_\rho \{\overrightarrow{str_i : v_i}\} \rightsquigarrow \hat{v}}$$

The judgements have one of the following form:

- $\mathcal{C} \Vdash_\rho v \rightsquigarrow \hat{v}$
  meaning that, assuming permission ρ, the concrete value v is mapped to the abstract value $\hat{v}$ in the abstract environment $\mathcal{C}$. The rules to derive these judgements are collected in Table 3.4.

- $\mathcal{C} \Vdash_\rho e : \hat{v} \gg \rho'$
  meaning that in the context of an handler/instance with permission ρ, and under the abstract environment $\mathcal{C}$, the expression e may evaluate to a value abstracted by $\hat{v}$ and it will escalate (i.e., it will transitively exercise) at most ρ'. The rules for these judgements are collected in Table 3.5.

- $\mathcal{C} \Vdash \mu$ **despite** ρ, $\mathcal{C} \Vdash h$ **despite** ρ, $\mathcal{C} \Vdash i$ **despite** ρ, $\mathcal{C} \Vdash s$ **despite** ρ
  meaning that the respective pieces of syntax are safe w.r.t. a ρ-opponent under the abstract environment $\mathcal{C}$.

The formal definitions of the last judgements are in Table 3.6, where we put in place the required constraints to ensure opponent acceptability, while keeping the analysis sound. We also employ two additional definitions.

Table 3.5 Flow analysis for expressions

$$\text{(PE-Val)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} v \rightsquigarrow \hat{v}}{\mathcal{C} \Vdash_{\rho_s} v : \hat{v} \gg \rho}$$

$$\text{(PE-Let)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(x) \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho}{\mathcal{C} \Vdash_{\rho_s} \textbf{let } x = e_1 \textbf{ in } e_2 : \hat{v} \gg \rho}$$

$$\text{(PE-App)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \qquad \forall \lambda x^{\rho_e} \in \hat{v}_1 : \hat{v}_2 \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(x) \wedge \mathcal{C}_{\hat{\Gamma}}(\lambda x) \sqsubseteq \hat{v} \wedge \rho_e \sqsubseteq \rho}{\mathcal{C} \Vdash_{\rho_s} e_1\, e_2 : \hat{v} \gg \rho}$$

$$\text{(PE-Seq)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho}{\mathcal{C} \Vdash_{\rho_s} e_1; e_2 : \hat{v} \gg \rho}$$

$$\text{(PE-Op)}\quad \frac{\forall i : \mathcal{C} \Vdash_{\rho_s} e_i : \hat{v}_i \gg \rho_i \sqsubseteq \rho \qquad \widehat{op}(\overrightarrow{\hat{v}_i}) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_{\rho_s} op(\overrightarrow{e_i}) : \hat{v} \gg \rho}$$

$$\text{(PE-Cond)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \sqsubseteq \rho \qquad \textbf{true} \in \hat{v}_0 \Rightarrow \mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \hat{v} \gg \rho_1 \sqsubseteq \rho \qquad \textbf{false} \in \hat{v}_0 \Rightarrow \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho}{\mathcal{C} \Vdash_{\rho_s} \textbf{if } (e_0) \{ e_1 \} \textbf{ else } \{ e_2 \} : \hat{v} \gg \rho}$$

$$\text{(PE-While)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \textbf{true} \in \hat{v}_1 \Rightarrow \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \qquad \textbf{false} \in \hat{v}_1 \Rightarrow \textbf{undefined} \in \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \textbf{while } (e_1) \{ e_2 \} : \hat{v} \gg \rho}$$

$$\text{(PE-GetField)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \qquad \widehat{get}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_{\rho_s} e_1[e_2] : \hat{v} \gg \rho}$$

$$\text{(PE-SetField)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \qquad \widehat{set}(\hat{v}_0, \hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_{\rho_s} e_0[e_1] = e_2 : \hat{v} \gg \rho}$$

$$\text{(PE-DelField)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho \qquad \widehat{del}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \textbf{delete } e_1[e_2] : \hat{v} \gg \rho}$$

$$\text{(PE-Ref)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e : \hat{v}' \gg \rho' \sqsubseteq \rho \qquad \hat{v}' \sqsubseteq \mathcal{C}_{\hat{\mu}}(\ell, \rho_s) \qquad \ell \in \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \textbf{ref}_\ell\ e : \hat{v} \gg \rho}$$

$$\text{(PE-Deref)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e : \hat{v}' \gg \rho' \sqsubseteq \rho \qquad \forall \ell \in \hat{v}' : \mathcal{C}_{\hat{\mu}}(\ell, \rho_s) \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \textbf{deref } e : \hat{v} \gg \rho}$$

$$\text{(PE-SetRef)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho \qquad \forall \ell \in \hat{v}_1 : \hat{v}_2 \sqsubseteq \mathcal{C}_{\hat{\mu}}(\ell, \rho_s)}{\mathcal{C} \Vdash_{\rho_s} e_1 = e_2 : \hat{v} \gg \rho}$$

$$\text{(PE-Send)}\quad \frac{\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \sqsubseteq \rho' \qquad \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \sqsubseteq \rho' \qquad \forall m \in \hat{v}_1 : \forall \rho_m \sqsupseteq \rho : \mathcal{C}_{\hat{\Upsilon}}(m, \rho_m) = (\rho_r, \rho_e) \wedge \rho_r \sqsubseteq \rho_s \Rightarrow \rho_e \sqsubseteq \rho' \wedge \hat{v}_2 \sqsubseteq \mathcal{C}_{\hat{\Phi}}(m, \rho_m) \wedge \textbf{unit} \in \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \overline{e_1}\langle e_2 \triangleright \rho \rangle : \hat{v} \gg \rho'}$$

$$\text{(PE-Exercise)}\quad \frac{\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \rho' \wedge \textbf{unit} \in \hat{v}}{\mathcal{C} \Vdash_{\rho_s} \textbf{exercise}(\rho) : \hat{v} \gg \rho'}$$

Definition 5 (Permission Leak)**.** *Given an abstract environment* $\mathcal{C}$*, we let its* permission leak *against* ρ *be:*

$$Leak_\rho(\mathcal{C}) = \bigsqcup_{\rho_e \in L} \rho_e, \text{ with } L = \{\rho_e \mid \exists a, \rho_a, \rho_s : \mathcal{C}_{\hat{\Upsilon}}(a, \rho_a) = (\rho_s, \rho_e) \wedge \rho_s \sqsubseteq \rho\}$$

Remind that $\hat{\Upsilon}(a, \rho_a) = (\rho_s, \rho_e)$ means that the handler a can be called by any component with privileges ρ_s and it *transitively* exercises up to ρ_e privileges. Then, intuitively the permission leak is a sound over-approximation of the permissions which can be escalated by the opponent in an initial system.

Let $\mathcal{C}$ be an abstract environment and pick a ρ-opponent. We define the set $\mathcal{V}_\rho(\mathcal{C})$ as follows:

$$\mathcal{V}_\rho(\mathcal{C}) = \mathcal{V}_u \cup \{x \mid \exists \ell, \rho_r \sqsubseteq \rho, \rho_e : \lambda x^{\rho_e} \in \mathcal{C}_{\hat{\mu}}(\ell, \rho_r)\}.$$

We let $\hat{v}_\rho(\mathcal{C}) = \{\hat{u} \mid vars(\hat{u}) \subseteq \mathcal{V}_\rho(\mathcal{C})\}$. Intuitively, this is a sound abstraction of any value which can be generated by/flow to the opponent (the second component of the union above corresponds to functions generated by the trusted components, which may be actually called by the opponent at runtime).

Definition 6 (Conservative Abstract Environment)**.** *An abstract environment* $\mathcal{C}$ *is* ρ-conservative *if and only if all the following conditions hold true:*

1. $\forall n \in \mathcal{N} : \forall \rho' \sqsubseteq \rho : \mathcal{C}_{\hat{\Upsilon}}(n, \rho') = (\bot, Leak_\rho(\mathcal{C}))$;
2. $\forall n \in \mathcal{N} : \forall \rho_n, \rho_s, \rho_e : \mathcal{C}_{\hat{\Upsilon}}(n, \rho_n) = (\rho_s, \rho_e) \wedge \rho_s \sqsubseteq \rho \Rightarrow \mathcal{C}_{\hat{\Phi}}(n, \rho_n) = \hat{v}_\rho(\mathcal{C})$;
3. $\forall n \in \mathcal{N} : \forall \rho' \sqsubseteq \rho : \mathcal{C}_{\hat{\Phi}}(n, \rho') = \hat{v}_\rho(\mathcal{C})$;
4. $\forall \ell \in \mathcal{L} : \forall \rho' \sqsubseteq \rho : \mathcal{C}_{\hat{\mu}}(\ell, \rho') = \hat{v}_\rho(\mathcal{C})$;
5. $\forall x \in \mathcal{V}_\rho(\mathcal{C}) : \mathcal{C}_{\hat{\Gamma}}(x) = \mathcal{C}_{\hat{\Gamma}}(\lambda x) = \hat{v}_\rho(\mathcal{C})$.

In words, an abstract environment is conservative whenever any code that can be run by the opponent is (soundly) assumed to escalate up to the maximal privilege $Leak_\rho(\mathcal{C})$ (1) and any reference under the control of the opponent is assumed to contain any possible value (4). Moreover, the parameter of any function which could be called by the opponent should be assumed to contain any possible value and similarly these functions can return any value (5). Finally, handlers which can be contacted by the opponent and handlers registered by the opponent may receive any value (2) and (3).

3.5 Theorem

Now we introduce the main finding of our analysis. This theorem gives a quantitative measure on how a system is safe against the attack of an opponent.

Theorem 1 (Safety Despite Compromise)**.** *Let* $s = \mu; h; \emptyset$*. If* $\mathcal{C} \Vdash s$ **despite** ρ*, then* s *is* ρ'*-safe despite* ρ *for* $\rho' = Leak_\rho(\mathcal{C})$*.*

Table 3.6 Flow analysis for systems

$$\text{(PM-Empty)}\quad \mathcal{C} \Vdash \emptyset \textbf{ despite } \rho$$

$$\text{(PM-Ref)}\quad \frac{\mathcal{C} \Vdash_{\rho_r} v \rightsquigarrow \hat{v} \qquad \hat{v} \sqsubseteq \mathcal{C}_{\hat{\mu}}(\ell, \rho_r)}{\mathcal{C} \Vdash r_\ell \overset{\rho_r}{\mapsto} v \textbf{ despite } \rho}$$

$$\text{(PM-Mem)}\quad \frac{\mathcal{C} \Vdash \mu_1 \textbf{ despite } \rho \quad \mathcal{C} \Vdash \mu_2 \textbf{ despite } \rho}{\mathcal{C} \Vdash \mu_1, \mu_2 \textbf{ despite } \rho}$$

$$\text{(PH-Empty)}\quad \mathcal{C} \Vdash \emptyset \textbf{ despite } \rho$$

$$\text{(PH-Single)}\quad \frac{\mathcal{C}_{\hat{\Upsilon}}(a, \rho_a) = (\rho'_s, \rho'_e) \qquad \rho_a \not\sqsubseteq \rho \Rightarrow \rho'_s = \rho_s \qquad \mathcal{C}_{\hat{\Phi}}(a, \rho_a) \neq \emptyset \Rightarrow \mathcal{C}_{\hat{\Gamma}}(x) \sqsupseteq \mathcal{C}_{\hat{\Phi}}(a, \rho_a) \wedge \mathcal{C} \Vdash_{\rho_a} e : \hat{v} \gg \rho_e \wedge (\rho_a \not\sqsubseteq \rho \Rightarrow \rho'_e = \rho_e)}{\mathcal{C} \Vdash a(x \triangleleft \rho_s : \rho_a).e \textbf{ despite } \rho}$$

$$\text{(PH-Many)}\quad \frac{\mathcal{C} \Vdash h \textbf{ despite } \rho \quad \mathcal{C} \Vdash h' \textbf{ despite } \rho}{\mathcal{C} \Vdash h, h' \textbf{ despite } \rho}$$

$$\text{(PI-Empty)}\quad \mathcal{C} \Vdash \emptyset \textbf{ despite } \rho$$

$$\text{(PI-Single)}\quad \frac{\mathcal{C} \Vdash_{\rho_a} e : \hat{v} \gg \rho_e \qquad \rho_a \not\sqsubseteq \rho \Rightarrow \exists \rho_s : \mathcal{C}_{\hat{\Upsilon}}(a, \rho_a) = (\rho_s, \rho_e)}{\mathcal{C} \Vdash a\{\!|e|\!\}_{\rho_a} \textbf{ despite } \rho}$$

$$\text{(PI-Many)}\quad \frac{\mathcal{C} \Vdash i \textbf{ despite } \rho \quad \mathcal{C} \Vdash i' \textbf{ despite } \rho}{\mathcal{C} \Vdash i, i' \textbf{ despite } \rho}$$

$$\text{(PS-Sys)}\quad \frac{\mathcal{C} \Vdash \mu \textbf{ despite } \rho \qquad \mathcal{C} \Vdash h \textbf{ despite } \rho \qquad \mathcal{C} \Vdash i \textbf{ despite } \rho \qquad \mathcal{C} \text{ is } \rho\text{-conservative}}{\mathcal{C} \Vdash \mu; h; i \textbf{ despite } \rho}$$

In other words, given ρ (permission of the attacker), is possible to compute an acceptable estimate of the system **despite** ρ (behavior of the system with the opponent into). And from such estimate, is possible to predict an upper bound of all permissions that such ρ opponent can escalate.

In section 3.3 we showed how an opponent that compromises cs1 was able to escalate only permission ρ because $Leak_\rho(\mathcal{C}) = \rho$, while an opponent with permission `SEND` was able to escalate up to $\rho \sqcup \rho'$ because $Leak_{SEND}(\mathcal{C}) = \rho \sqcup \rho'$.

The importance of this theorem is showed by the fact that given an real Chrome extension, with this theorem and the analysis, we are able to show all possible privileges that an attacker can escalate to. This is very important because we can prove that a certain extension exposes to the attacker only certain permissions, protecting the other.

3.6 Requirements for correctness

In the previous part we abstract the choice of the abstract domains used in the analysis. Here we state the requirement that the abstract domains must satisfy in order to keep the analysis sound.

Assumption 1 (Abstracting Finite Domains)**.** $\forall c \in \{\textbf{true}, \textbf{false}, \textbf{unit}, \textbf{undefined}\} : \hat{c} = c$.

This means that an element of a finite domain has the abstraction that coincide with itself. For example **true** $\rightsquigarrow$ **true** or **undefined** $\rightsquigarrow$ **undefined**.

Assumption 2 (Soundness of Abstract Operations)**.** $\forall op : \forall \overrightarrow{c_i} : \forall c : \delta(op, \overrightarrow{c_i}) = c \Rightarrow \{\hat{c}\} \sqsubseteq \widehat{op}(\overrightarrow{\hat{c}_i})$.

In words, we say that each concrete operation *op* has its counterpart that is less precise since the abstraction of the result of the concrete operation is contained ($\sqsubseteq$) in the result of the abstract operation on the abstracted arguments.

Assumption 3 (Soundness of Abstract Record Operations)**.** *All the following properties hold true:*

1. $\{\overrightarrow{str_i : v_i}\}[str] \hookrightarrow v \wedge \widehat{get}(\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}, \widehat{str}) = \hat{v}' \Rightarrow \exists \hat{v} \sqsubseteq \hat{v}' : \mathcal{C} \Vdash_\rho v \rightsquigarrow \hat{v}$;
2. $\{\overrightarrow{str_i : v_i}\}[str] = v' \hookrightarrow v \wedge \mathcal{C} \Vdash_\rho v' \rightsquigarrow \hat{v}' \wedge \widehat{set}(\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}, \widehat{str}, \hat{v}') = \hat{v}'' \Rightarrow \exists \hat{v} \sqsubseteq \hat{v}'' : \mathcal{C} \Vdash_\rho v \rightsquigarrow \hat{v}$;
3. **delete** $\{\overrightarrow{str_i : v_i}\}[str] \hookrightarrow v \wedge \widehat{del}(\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}, \widehat{str}) = \hat{v}' \Rightarrow \exists \hat{v} \sqsubseteq \hat{v}' : \mathcal{C} \Vdash_\rho v \rightsquigarrow \hat{v}$.

As in the case of the abstract operation in 2, the result of the concrete record operations must be abstracted to at least one value must that is contained ($\subseteq$) in the result of counterpart operation on the abstracted arguments. We say "must be abstracted to at least one value that" because a concrete record can be abstracted to many different representation (e.g., $\{\} \rightsquigarrow \{\}$, even $\{\} \rightsquigarrow \top$, but not $\{"a" : 5\} \not\rightsquigarrow \{\}$) and here we say that at least one of them must be contained in the result.

Assumption 4 (Monotonicity of Abstract Operations)**.** *The following property holds true:*

$$\forall \widehat{op}^* \in \{\widehat{op}, \widehat{get}, \widehat{set}, \widehat{del}\} : \forall \overrightarrow{\hat{v}_i} : \forall \overrightarrow{\hat{v}'_i} : (\forall i : \hat{v}_i \sqsubseteq \hat{v}'_i \Rightarrow \widehat{op}^*(\overrightarrow{\hat{v}_i}) \sqsubseteq \widehat{op}^*(\overrightarrow{\hat{v}'_i})).$$

We say that the same abstract operation on two different arguments with a partial-order relation between them, must preserve the partial-order on the respective results.

Assumption 5 (Totality of Abstract Operations)**.** $\forall \widehat{op}^* \in \{\widehat{op}, \widehat{get}, \widehat{set}, \widehat{del}\} : \forall \overrightarrow{\hat{v}_i} : \exists \hat{v} : \widehat{op}^*(\overrightarrow{\hat{v}_i}) = \hat{v}$.

All abstract operation are closed in the abstract domain. Means that each operation is always applicable on abstract arguments, no matter which these are, and the result is an abstract value too. So even operation that with some concrete values fails in the abstract domain never fails; in the worst case the result is $\perp$

Assumption 6 (Ordering Abstract Values)**.** *The relation $\sqsubseteq$ over $\hat{V} \times \hat{V}$ is a pre-order such that:*

1. $\forall \hat{v}, \hat{v}' : \hat{v} \subseteq \hat{v}' \Rightarrow \hat{v} \sqsubseteq \hat{v}'$;
2. $\forall \hat{v} : \hat{v} \sqsubseteq \emptyset \Rightarrow \hat{v} = \emptyset$;
3. $\forall n : \forall \hat{v} : \{n\} \sqsubseteq \hat{v} \Rightarrow n \in \hat{v}$;
4. $\forall \ell : \forall \hat{v} : \{\ell\} \sqsubseteq \hat{v} \Rightarrow \ell \in \hat{v}$;
5. $\forall \lambda x^{\rho} : \forall \hat{v} : \{\lambda x^{\rho}\} \sqsubseteq \hat{v} \Rightarrow \exists \rho' \sqsupseteq \rho : \lambda x^{\rho'} \in \hat{v}$;
6. $\forall c \in \{\mathbf{true}, \mathbf{false}, \mathbf{unit}, \mathbf{undefined}\} : \forall \hat{v} : \{\hat{c}\} \sqsubseteq \hat{v} \Rightarrow \hat{c} \in \hat{v}$.

Here is stated an ordering on abstract values:

1. the subset inclusion $\subseteq$ between two abstract values always implies an ordering $\sqsubseteq$ on them. So if an abstract value is contained in another then its representation is more precise than the second one (e.g., $\mathbf{true} \in \hat{v} \wedge \mathbf{true} \in \hat{v}' \wedge \mathbf{undefined} \in \hat{v}' \Rightarrow \hat{v} \sqsubseteq \hat{v}'$);
2. if an abstract value is contained in the empty set it must be the empty set.
3. if the singleton $\{n\}$ is contained in an abstract value then n is contained in the set represented by it;
4. the same as (3), but with labels;
5. the singleton $\{\lambda x^{\rho}\}$ is contained in a value if exists a permission ρ' bigger than ρ (i.e., $\rho \sqsubseteq \rho'$) such that the lambda using ρ' is contained in the set represented by the abstract value;
6. the same as (3), but with finite domains;

Assumption 7 (Abstracting Serializable Records)**.** *If $\{\overrightarrow{str_i : v_i}\}$ is serializable, then for any $\mathcal{C}$, ρ_a and ρ_b we have $(\!|\overrightarrow{str_i : v_i}|\!)_{\mathcal{C},\rho_a} = (\!|\overrightarrow{str_i : v_i}|\!)_{\mathcal{C},\rho_b}$.*

In other words, if a record is serializable then its abstract representation do not depend by the permission and cache with whom it is abstracted. This holds because any serializable record does not contains any lambda, but only elements that do not holds permissions.

Assumption 8 (Variables)**.** *All the following properties hold true:*

1. $\forall \hat{c} : vars(\hat{c}) = \emptyset$;
2. $\forall \widehat{op} : \forall \overrightarrow{\hat{v}_i} : vars(\widehat{op}(\overrightarrow{\hat{v}_i})) = \emptyset$;
3. $\forall \hat{v}_1, \hat{v}_2 : vars(\widehat{get}(\hat{v}_1, \hat{v}_2)) \subseteq vars(\hat{v}_1)$;
4. $\forall \hat{v}_0, \hat{v}_1, \hat{v}_2 : vars(\widehat{set}(\hat{v}_0, \hat{v}_1, \hat{v}_2)) \subseteq vars(\hat{v}_0) \cup vars(\hat{v}_2)$;
5. $\forall \hat{v}_1, \hat{v}_2 : vars(\widehat{del}(\hat{v}_1, \hat{v}_2)) \subseteq vars(\hat{v}_1)$.

This last assumption states that no abstract constants contains variables (1), that every abstract operation return a value that is not a lambda neither an object, and so do not contains any variable (2), that, since a record can contains lambdas (the only possible value where a variable can occur), all abstract record operations do not add any variable to the result that is not contained in the arguments. More specifically the abstract get return a value with variables contained in the initial record (3), abstract set do not add variable that are not in the original record or in the setted value (4), and the abstract delete do not add to the result variables that are not in the original record (5).

Chapter 4

Implementation

In this chapter we will show how the calculus and the analysis of chapter 3 has been implemented. We developed a tool written in F# that from a real chrome extension is able to detect which privileges ca be obtained by an attacker that infect a content script.

In order to apply the analysis we had to desugar the JavaScript sources in λ_{JS}. To this purpose we use the desugarer prototype that is a tool written in Haskell and described in [13]. Then we parse the desugared λ_{JS} file and we start the analysis.

Since we are using the 0-CFA approach for the analysis, we do not have any context, so to enhance the precision of the analysis (soundness is kept) we alpha-rename the bound variables in order to distinguish them. We also mark the nodes of the abstract syntax tree with unambiguous labels and we annotate references with ℓ. After that the AST is passed to the algorithm that generates the constraints producing a set of constraints. After that, given an abstract value representation, the constraint are resolved without using the AST any more producing an estimate for the initial program. Finally analyzing the estimate we show the permission that an attacker can escalate if it infect a content script.

4.1 Analysis specification

As explained in section 2.3 the flow logic can be done using various approaches. In the analysis of section 3.4 it is used an abstract succinct approach, but for the implementation of the analysis it is needed a compositional verbose one. The difference between abstract and compositional approach are that the former is closer to the semantic and tend to be simpler, while the latter is more syntax directed. Moreover the abstract approach lets the analysis of lambdas in the application point, while the other analyze it at definition point. The function call in the compositional approach just link the arguments to the formal parameters of the lambda, and links the result of the lambda to the value of the call node.

The differences between a succinct and a verbose analysis are that the succinct analysis focus on the top-level part of the analysis estimate, while the verbose, as in data flow analysis and constraint based analysis, reports all the internal flow data. This second approach is done using caches that holds the analysis informations.

In the following sections we translate the analysis of section 3.4 to a compositional verbose one and then we transform the judgments to a set of constraints.

4.1.1 Compositional Verbose

In compositional verbose approach we add an unambiguous label $\alpha \in A$ to all expression in the syntax and the result of each expression are stored in an abstract cache $\hat{C}$ that its a map from nodes to abstract values. Even the permissions of each expression are stored in caches $\hat{P}$.

A labeled expression is e^{α} has the following property:

$$\text{Let be } e^{\alpha} \text{ and } e'^{\alpha'} \text{ two expressions:}$$
$$\alpha = \alpha' \qquad \text{iff } e = e'$$

This property says that two different labeled expression has different labels.

To enhance readability we let $\mathcal{CV} = \hat{C}, \hat{P}, \hat{\Gamma}, \hat{\mu}, \hat{\Upsilon}, \hat{\Phi}$ to be the the compositional verbose environment that is a six-tuple made of the following components:

Abstract cache	$\hat{C}$	:	$A \to \hat{V}$
Permission cache	$\hat{P}$	:	$\mathcal{P} \to \mathcal{P}$
Abstract variable environment	$\hat{\Gamma}$	:	$\mathcal{V} \to \hat{V}$
Abstract memory	$\hat{\mu}$	:	$\mathcal{L} \times \mathcal{P} \to \hat{V}$
Abstract stack	$\hat{\Upsilon}$	:	$\mathcal{N} \times \mathcal{P} \to \mathcal{P} \times \mathcal{P}$
Abstract network	$\hat{\Phi}$	:	$\mathcal{N} \times \mathcal{P} \to \hat{V}$.

While $\hat{\Gamma}, \hat{\mu}, \hat{\Upsilon}, \hat{\Phi}$ are the same of the previous chapter, the abstract cache $\hat{C}$, and permission cache $\hat{P}$ are specific of the compositional approach.

Table 4.1 4.2 contains the rules for the compositional verbose analysis. Note that are very similar to the abstract succinct except that the expression of the form $\mathcal{C} \Vdash_{\rho} e : v \gg \rho$ do not produce v and ρ, but analysis store them in the cache and became $\mathcal{CV} \Vdash_{cv,\rho} (e^{\alpha_1})^{\alpha}$ so $\mathcal{CV}_{\hat{C}}(\alpha) = v$ and $\mathcal{CV}_{\hat{P}}(\alpha) = \rho$.

4.2 Constraint generation

Now since we have an analysis that is compositional-verbose, we can design an algorithm to compute the set of constraints derived from the analysis. Then such set is given to a constraint solver algorithm to compute an estimate for the program.

4.2.1 Constraints

We now define the following elements:

Cache	C	:	$A \to \hat{V}$
Permission	P	:	$A \to \mathcal{P}$
Var	$\mathsf{\Gamma}$	:	$\mathcal{V} \to \hat{V}$
State	M	:	$\mathcal{L} \times \mathcal{P} \to \hat{V}$
Stack	Υ	:	$\mathcal{N} \times \mathcal{P} \to \mathcal{P} \times \mathcal{P}$
Network	Φ	:	$\mathcal{N} \times \mathcal{P} \to \hat{V}$.

This are the implementation of the elements of the environments and are maps as described above.

A constraint element named E is a pure syntactical element that represent the index of the map described above.

Cache element	$\mathsf{C}(\alpha)$
Permission Element	$\mathsf{P}(\alpha)$
Var element	$\Gamma(x)$
State element	$\mathsf{M}(\ell, \rho)$
Stack element	$\Upsilon(a, \rho)$
Network element	$\Phi(a, \rho)$

As we can see an element is composed by the cache element to which is referred and its index.

Now let us introduce the constraints. We let c range over *constraints*, defined by the following productions:

$$
\begin{array}{rcrcll}
c & ::= & \{\hat{v}\} & \sqsubseteq & \mathsf{E} & \textit{Term inclusion} \\
& | & \mathsf{E} & \sqsubseteq & \mathsf{E} & \textit{Element inclusion} \\
& | & \widehat{Op}(\overrightarrow{\mathsf{E}_i}) & \sqsubseteq & \mathsf{E} & \textit{Operationinclusion} \\
& | & c & \Rightarrow & c & \textit{Implication}
\end{array}
$$

We notice that there are no constraint specific for the three record operations $\widehat{Get}$, $\widehat{Set}$, $\widehat{Del}$, because we treat them as standard operations. The main forms of constraints are:

- $\hat{v} \sqsubseteq \mathsf{C}(\alpha)$ that means that $\hat{v}$ must be in the possible estimate of the node marked with α (e.g., $\{\mathbf{true}\} \sqsubseteq \mathsf{C}(\alpha)$ means that **true** is in the possible estimate of the node marked with α);

- $\hat{v} \sqsubseteq \Gamma(x)$ that means that $\hat{v}$ must be in the possible estimate of the x variable;

- $\mathsf{C}(\alpha_1) \sqsubseteq \mathsf{C}(\alpha)$ that means that the estimate of node α_1 must be contained in the node α (as before the same are with the variables);

- $\widehat{Op}(\overrightarrow{\mathsf{C}(\alpha_i)}) \sqsubseteq \mathsf{C}(\alpha)$ that means that the result of the abstract operation with $\overrightarrow{\mathsf{C}(\alpha_i)}$ as arguments must be contained in the estimate of $\mathsf{C}(\alpha)$ (e.g., $\widehat{Op}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$ means that the result of the abstract operation corresponding to $+$ with $\mathsf{C}(\alpha_1)$ and $\mathsf{C}(\alpha_2)$ as arguments must be contained in the estimate of $\mathsf{C}(\alpha)$);

- $\hat{v} \sqsubseteq \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_1) \sqsubseteq \mathsf{C}(\alpha)$ means that the fact that $\hat{v}$ is contained in $\mathsf{C}(\alpha_0)$ implies that value in $\mathsf{C}(\alpha_1)$ must be contained in the estimate of the node $\mathsf{C}(\alpha)$. For example, $\{\mathbf{true}\} \sqsubseteq \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_1) \sqsubseteq \mathsf{C}(\alpha)$ means that if **true** is contained in the estimate of $\mathsf{C}(\alpha_0)$ than the value of $\mathsf{C}(\alpha_1)$ must be contained in the estimate of the node $\mathsf{C}(\alpha)$ (this form is used in the if construct); and $\{\lambda x.e_0{}^{\alpha_0}\} \sqsubseteq \mathsf{C}(\alpha_1) \Rightarrow \mathsf{C}(\alpha_0) \sqsubseteq \mathsf{C}(\alpha)$ that means that if $\lambda x.e_0{}^{\alpha_0}$ is contained in the estimate of $\mathsf{C}(\alpha_1)$ then the value of the lambda (contained in $\mathsf{C}(\alpha_0)$) must be contained in the estimate of the node α (this form is used in the application);

In order to transform the $\forall$ construct of the compositional rules in our constraints (e.g., in the app expression or in the `deref` expression), since both lambdas and references are finite sets in the program, we generate one implication constraint for each element in the program in this way: let be Ref_* the set of all reference labels of the program $\forall \ell \in v_1 : \mathcal{CV}_{\hat{\mu}}(\ell, \rho_s) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$ is transformed in the set $\{\ell \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{M}(\ell, \rho_s) \sqsubseteq \mathsf{C}(\alpha) | \ell \in Ref_*\}$.

In this way we define these finite sets:

Ref_*	is the set of all references of the program;
$lambda_*$	is the set of all lambdas of the program;
$Names_*$	is the set of all names in the program;
$NamePerms_*$	is the set of all permission associated with channel in the program.

4.2.2 Generation

To obtain the set of constraint the AST of the program (composed only by expression since there are no statements) is given to an algorithm that traverse it and, for each subtree of it, returns the set of constraint generated. The algorithm in tables 4.3 and 4.4 $\mathcal{C}_{*\rho_s}[\![(e)^\alpha]\!]$ explore the tree and produces the set of constraints.

As simple example the program $(((\lambda x.x^1)^2(\lambda y.False^3)^4)^5True^6)^7$, ignoring the permission check, yields this set of constraints:

$$
\begin{array}{l}
\{ \\
\quad \{\mathbf{true}\} \quad \sqsubseteq \mathsf{C}(6), \\
\quad \{\mathbf{false}\} \quad \sqsubseteq \mathsf{C}(3), \\
\quad \{\lambda x.x^1\} \quad \sqsubseteq \mathsf{C}(2), \\
\quad \{\lambda y.\mathbf{true}^6\} \quad \sqsubseteq \mathsf{C}(4), \\
\quad \Gamma(x) \quad \sqsubseteq \mathsf{C}(1), \\
\quad \{\lambda x.x^1\} \quad \sqsubseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(1) \sqsubseteq \mathsf{C}(5), \\
\quad \{\lambda x.x^1\} \quad \sqsubseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \sqsubseteq \Gamma(x), \\
\quad \{\lambda x.x^1\} \quad \sqsubseteq \mathsf{C}(5) \Rightarrow \mathsf{C}(1) \sqsubseteq \mathsf{C}(7), \\
\quad \{\lambda x.x^1\} \quad \sqsubseteq \mathsf{C}(5) \Rightarrow \mathsf{C}(6) \sqsubseteq \Gamma(x), \\
\quad \{\lambda y.\mathbf{true}^6\} \quad \sqsubseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(3) \sqsubseteq \mathsf{C}(5), \\
\quad \{\lambda y.\mathbf{true}^6\} \quad \sqsubseteq \mathsf{C}(2) \Rightarrow \mathsf{C}(4) \sqsubseteq \Gamma(y), \\
\quad \{\lambda y.\mathbf{true}^6\} \quad \sqsubseteq \mathsf{C}(5) \Rightarrow \mathsf{C}(3) \sqsubseteq \mathsf{C}(7), \\
\quad \{\lambda y.\mathbf{true}^6\} \quad \sqsubseteq \mathsf{C}(5) \Rightarrow \mathsf{C}(6) \sqsubseteq \Gamma(y) \\
\}
\end{array}
$$

4.3 Constraint solving

To compute efficiently an analysis estimate from the constraint set we used an algorithm derived from the worklist algorithm presented in [21, 12]. It has in tables 4.5 and 4.6 is shown a simplification of the algorithm that solves only constraint for data, permission and network constraint are removed for sake of readability.

The algorithm works on a graph where each constraint element E is represented by a node and each constraint (except to the term inclusion) is represented by either standard

or conditional edges according to the kind of the constraint. Each node `p` contains a data field `D[p]`. The algorithm also have a worklist `W` of nodes to be processed.

Here we show which edges are generated by a constraint:

- $\{\mathbf{true}\} \sqsubseteq \mathsf{E}$ do not generate any edge; it insert **true** in $\mathtt{D}[\mathsf{E}]$ and add E in `W`;
- $\mathsf{E}_1 \sqsubseteq \mathsf{E}_2$ generates an edge from E_1 to E_2;
- $\widehat{Op}(\overrightarrow{\mathsf{E}_i}) \sqsubseteq \mathsf{E}$ generate an edge from each E_i to E;
- $t \sqsubseteq \mathsf{E} \Rightarrow \mathsf{E}_1 \sqsubseteq \mathsf{E}_2$ give rise to a conditional edge in E and in E_1 both from E_1 to E_2;

When a conditional edge is been processed there is a check on the precondition (the part before the arrow) and if the check succeed the edge is traversed, otherwise no.

The initialization of the algorithm is done in steps 1: it just creates the structure for storing the graph and the worklist. In step 2 the graph is build as shown in the list above according to the constraint in input. Finally the graph is traversed in step 3. The traversal is done removing an element from `W` and processing all his edges. When a node is processed its data are propagated to his neighbors and in case of conditional constraint if the precondition is respected the data is propagated, otherwise no. If in a propagation the data of the destination is modified, the destination node is added to `W`.

Notice that the propagation of a value do not replace the old value with the new one, but it joins the values in a lattice. Since this the analysis is sound because the partial ordering is maintained.

When `W` is empty the system is in a fix point and step 3 terminates. In step 4 all data contained in each node of the graph is copied in the cache, environment and memory of the analysis and form an acceptable program estimate.

4.4 Abstract domains choice

In this section we introduce all abstract domains used in the analysis. In this purpose we used the classical approach of abstract interpretations of [10, 21].

4.4.1 Abstract Value

In the implementation of the analysis an abstract value is a set containing the abstraction of each atomic domain defined in the calculus in section 3.1.

Definition 7 (Partial order on abstract values)**.** *Given two abstract values $\hat{v}, \hat{v}'$ we say that $\hat{v} \sqsubseteq \hat{v}'$ if $\forall \hat{u} \in \hat{v} : \exists \hat{u'} \mid \hat{u} \sqsubseteq \hat{u'}$*

This means that abstract values are partially ordered according with prevalues of which they are composed.

In the definition 7 we assumed a pre-order relation on abstract pre-values. In the following part we introduce all abstract pre-values domains, and so even the pre-order relation used here.

Finite domains

As required in 1 the abstraction of finite domains coincide with their concrete themselves. So, both abstraction and concretization function are just the identity function. We use this abstraction for boolean (**true**, **false**) and **undefined** and **unit**. The pre-order relation in this case is the simple $\subseteq$ between abstract values.

Lambdas

Since all lambdas defined in a program are a finite set, their abstraction coincide with their concrete representation. So, as finite domains, the abstraction and the concretization function, are the identity function. Again the partial order relation in this case is the simple $\subseteq$ relation between abstract value.

References

Since labels ℓ associated to references are finite (the count is the number of $\mathbf{ref}_\ell\ e$ expressions) we abstract as finite domains and lambdas. Even in this case the partial order relation between references is the $\subseteq$ relation between abstract values.

Numbers

Numbers are abstracted with sign domain as in [21] with this abstraction function:

$$\sigma(n) = \begin{cases} + & \text{if } i > 0 \\ - & \text{if } i < 0 \\ 0 & \text{if } i = 0 \end{cases}$$

Of course an abstract values can contain more than one abstract number (e.g., $\hat{v} = +, 0, \ldots$ means that $\hat{v}$ can contains positive numbers or zero, or other).

Even in this case the pre-order relation is the $\subseteq$ between abstract values.

Strings

Since we need to track the form of the messages we need an abstraction of strings that is enough precise to perform the analysis. Aimed by this purpose we adopt, as abstraction of string values, a domain derived from the prefix notation defined in [10].

A string can be either: exact or a prefix. An exact string represent itself in the concrete domain (e.g., "tag" stand for "tag") while a prefix represent all strings with them as prefix (e.g., "abc*" stand for "abc", "abcde", "abc123", but not "bdf"). Prefix strings are pre-ordered by the following relation:

$$\hat{S} \sqsubseteq_{\hat{PR}} \hat{T} \begin{cases} exact(\hat{S}) \wedge exact(\hat{T}) \wedge \hat{T} = \hat{S} \\ S = \bot_{\hat{PR}} \\ \neg exact(\hat{T}) \wedge (\forall i \in [0, len(\hat{T}) - 1] : len(\hat{T}) \leq len(\hat{S}) \wedge \hat{T}[i] = \hat{S}[i]) \end{cases}$$

We say that a string $\hat{S}$ is smaller than $\hat{T}$ if $\hat{S}$ is $\bot_{\hat{PR}}$ or if $\hat{T}$ is not exact and is a prefix of $\hat{S}$ or if both are exact strings and are equals. Where the function *exact* states if a string is exact. Notice that and $\top_{\hat{PR}} =$ " $*$ ".

The $\sqcup$ operation between two abstract strings is an prefix strings containing the longest common prefix between them.

An abstract string represent or itself if it is exact, or all the strings that starts with it. This grants the prerequisites given in 3.6.

Records

Records are abstracted using a field sensitive approach since the analysis must track them with high precision. The abstraction of a record r is a map $\hat{r}$ from abstract strings to abstract values. In order to maintain simplicity in the implementation, an abstract record contains various exact field, but just one prefix field with value $\top$ where all prefix strings are collapsed. This can seem weird, but do not alter the soundness of the analysis; moreover we found that in real JavaScript code accessing a record with a string that is computed, and not literal, is a rare practice. This helps a lot with the $\widehat{Set}$ operation.

Abstract records has this property:
let be $r = \{\overrightarrow{str_i : v_i}\}$ and $\hat{r} = \{\overrightarrow{\widehat{str_j} : \hat{v_j}}\}$. $\sigma(r) = \hat{r}$ if all the following conditions hold true:

- $\forall i : \exists j \mid \sigma(str_i) \sqsubseteq str_j$;
- $\forall i : \exists j \mid \sigma(str_i) \sqsubseteq \widehat{str_j} \Rightarrow \sigma(v_i) \sqsubseteq \hat{v}_j$.

This means that the abstraction of a record must contain an abstract field bigger than the abstraction of the concrete field, and the value associated with it is greater than the abstraction of the concrete value. For example $\sigma(\{a : 12\})$ can be $\{a : +\}$, but even $\{a : \{+, \textbf{false}\}\}$, and $\{a : +, b : \textbf{true}\}$, but not $\{b : \textbf{true}\}$.

Note that $\sigma(\{\}) = \textit{any record}$ because since the empty record has no field, all possible abstract record contains all its field.

To collect more information on records we state the $\sqcup$ as the union of the two maps.

4.4.2 Abstract operations

All operations on abstract values are just operations on the Cartesian product on each component. For example the abstract plus operation $\hat{+}(\hat{v}_1, \hat{v}_2)$ is the $\hat{+}$ on each element of $\hat{v}_1$ and $\hat{v}_2$ such that they are numbers.

$$\hat{+}(\hat{v}_1, \hat{v}_2) = \bigsqcup(\hat{+}(\hat{u_1}, \hat{u_2} \mid \hat{u_1} \in numbers(\hat{v}_1) \wedge \hat{u_2} \in numbers(\hat{v}_2)))$$

Where numbers is a function that returns the set of all abstract numbers in an abstract value.

This example show how abstract operations are total as assumed in 5, indeed, if one of the two arguments $\hat{v}_1, \hat{v}_2$ does not contains any abstract number, the operation returns $\bot$ that is a valid abstract value. The same are with other operations.

In this section we present the main abstract operations of the abstract pre-value domains.

Numbers

The arithmetic operation on sign domain are the following:

$\bar{-}$	$+$	0	$-$	$\top$	$\bot$
	$-$	0	$+$	$\top$	$\bot$

$\bar{+}$	$+$	0	$-$	$\top$
$+$	$+$	$+$	$\top$	$\top$
0	$+$	0	$-$	$\top$
$-$	$\top$	$-$	$-$	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\bar{*}$	$+$	0	$-$	$\top$
$+$	$+$	0	$-$	$\top$
0	0	0	0	0
$-$	$-$	0	$+$	$\top$
$\top$	$\top$	0	$\top$	$\top$

$/$	$+$	0	$-$	$\top$	$\bot$
$+$	$+$	0	$-$	$\top$	$\bot$
0	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
$-$	$-$	0	$+$	$\top$	$\bot$
$\top$	$\top$	0	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

Strings

String operation are interesting, especially the `concat` function.

$$concat(\widehat{str}_1, \widehat{str}_2) = \widehat{str}_1$$

It simply return the first string as a prefix. Indeed, the concatenation of two strings are for definition contained in the abstraction of the first followed by any character.

Another important function on string is the comparison operation $\widehat{==}$

$$\widehat{str}_1 \widehat{==} \widehat{str}_2 \begin{cases} \textbf{true} & \text{if } exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_1 == \widehat{str}_2 \\ \textbf{false} & \text{if } exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_1 \neq \widehat{str}_2 \\ \top_b & \text{if } exact(\widehat{str}_1) \wedge \neg exact(\widehat{str}_2) \wedge \widehat{str}_1 \sqsubseteq \widehat{str}_2 \\ \textbf{false} & \text{if } exact(\widehat{str}_1) \wedge \neg exact(\widehat{str}_2) \wedge \widehat{str}_1 \not\sqsubseteq \widehat{str}_2 \\ \top_b & \text{if } \neg exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_2 \sqsubseteq \widehat{str}_1 \\ \textbf{false} & \text{if } \neg exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_2 \not\sqsubseteq \widehat{str}_1 \\ \top_b & \text{if } \neg exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_1 \sqsubseteq \widehat{str}_2 \vee \widehat{str}_2 \sqsubseteq \widehat{str}_1 \\ \textbf{false} & \text{if } \neg exact(\widehat{str}_1) \wedge exact(\widehat{str}_2) \wedge \widehat{str}_1 \not\sqsubseteq \widehat{str}_2 \vee \widehat{str}_2 \not\sqsubseteq \widehat{str}_1 \end{cases}$$

We finally say that a string $\widehat{str}_1$ is compatible with $\widehat{str}_2$ if $\textbf{true} \in (\widehat{str}_1 \widehat{==} \widehat{str}_2)$.

Records

Record operations are three: $\widehat{Get}, \widehat{Set}, \widehat{Del}$.

Get operation returns the join of all the values contained by all fields compatible with a string.

$$\widehat{Get}(\{\widehat{str}_i : \hat{v}_i\}, \widehat{str}) = \textbf{undefined} \sqcup (\bigsqcup \{\hat{v}_i \mid \widehat{str} \sqsubseteq \widehat{str}_i\})$$

We notice that **undefined** is added to the result. This is required since an abstract record can contains more field of the concrete one. Moreover we have to remember that two concrete strings can have an abstract representation that can be compatible.

Set operation adds the value in the corresponding field or adds a property with the value. Its abstract corresponding function it adds the value to all the compatible fields.

$$\widehat{Set}(\{\overrightarrow{\widehat{str}_i : \hat{v}_i}\}, \widehat{str}, \hat{v}) =$$
$$\begin{cases} \{\overrightarrow{\widehat{str}_i : \hat{v}_i}, \widehat{str}_j : \hat{v}_j \sqcup \hat{v}\} & \text{if } exact(\widehat{str}) \wedge \exists j \mid exact(\widehat{str}_i) \wedge \widehat{str} == \widehat{str}_j \\ \{\overrightarrow{\widehat{str}_i : \hat{v}_i}, \widehat{str}_\top : \hat{v}_\top \sqcup \hat{v}\} & \text{if } \neg exact(\widehat{str}) \\ \{\overrightarrow{\widehat{str}_i : \hat{v}_i}, \widehat{str} : \hat{v}\} & \text{otherwise} \end{cases}$$

Del operation removes a field from an object. Its abstract representation has to remove a field from a record, but abstracted records can contain more field than the concrete ones, hence the $\widehat{Del}$ return the same object in input.

$$\widehat{Del}(\{\widehat{str}_i : \hat{v}_i\}, \widehat{str}) = \{\widehat{str}_i : \hat{v}_i\}$$

4.5 Implementation-specific details

As stated before our tool, written in F# implements the analysis of 3 for a real chrome extension, and is able to detect which privileges ca be obtained by an attacker that infect a content script.

In order to analyze the JavaScript code we desugar the JavaScript code λ_{JS} using the desugarer prototype. This tool is written in Haskell and described in [13]. It also include in the desugared file a prelude containing JavaScript standard API (e.g., Arrays) and non standard operation of it.

Then we parse the desugared λ_{JS} using a `yacc` parser.

Since we are using the 0-CFA approach for the analysis, we do not have any context, so to enhance the precision of the analysis (soundness is kept) we alpha-rename the bound variables (all because a desugared file is closed) in order to distinguish them through the analysis. We also mark the nodes of the abstract syntax tree with unambiguous labels $\alpha \in A$ and we annotate all references with ℓ.

Then the abstract syntax tree representing the program is passed to the algorithm that generates the constraints producing a set of constraints.

Finally, given an abstract value representation, the constraint are resolved using the worklist algorithm and producing an acceptable estimate for the initial program.

Finally we analyze the estimate and we show the permission that an attacker can escalate if it infect a content script.

Table 4.1 Compositional Verbose part 1

[*CV-Val*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (v)^{\alpha}$ iff $\{\hat{v}\} \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-Lambda*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\lambda x.{e_0}^{\alpha_0})^{\alpha}$ iff
	$\{\lambda x.{e_0}^{\alpha_0}\} \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_0}^{\alpha_0}$
[*CV-Let*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{let } x = {e_1}^{\alpha_1} \textbf{ in } {e'}^{\alpha'})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e'}^{\alpha'} \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha') \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{C}}(\alpha') \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge$
	$\mathcal{CV}_{\hat{C}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{\Gamma}}(x_1) \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)$
[*CV-App*]	$\mathcal{CV} \Vdash_{cv,\rho_s} ({e_1}^{\alpha_1}\, {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)$
	$\forall(\lambda x.{e_0}^{\alpha_0}) \in \mathcal{CV}_{\hat{C}}(\alpha_1) :$
	$\mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{\Gamma}}(x) \wedge \mathcal{CV}_{\hat{C}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)$
[*CV-Seq*]	$\mathcal{CV} \Vdash_{cv,\rho_s} ({e_1}^{\alpha_1}; {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{C}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-Op*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (op(\overrightarrow{{e_i}^{\alpha_i}}))^{\alpha}$ iff
	$\widehat{op}(\mathcal{CV}_{\hat{C}}(\alpha_i)) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\forall i : \mathcal{CV} \Vdash_{cv,\rho_s} {e_i}^{\alpha_i} \wedge \mathcal{CV}_{\hat{P}}(\alpha_i) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)$
[*CV-Cond*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{if } ({e_0}^{\alpha_0}) \ \{\ {e_1}^{\alpha_1}\ \} \textbf{ else } \{\ {e_2}^{\alpha_2}\ \})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_0}^{\alpha_0} \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{\textbf{true}} \in \mathcal{CV}_{\hat{C}}(\alpha_0) \Rightarrow$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{C}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{\textbf{false}} \in \mathcal{CV}_{\hat{C}}(\alpha_0) \Rightarrow$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)$
[*CV-While*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{while } ({e_1}^{\alpha_1}) \ \{\ {e_2}^{\alpha_2}\ \})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{\textbf{true}} \in \mathcal{CV}_{\hat{C}}(\alpha_1) \Rightarrow$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{\textbf{false}} \in \mathcal{CV}_{\hat{C}}(\alpha_1) \Rightarrow \widehat{\textbf{undefined}} \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$

Table 4.2 Compositional Verbose part 2

[*CV-GetField*]	$\mathcal{CV} \Vdash_{cv,\rho_s} ({e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{get}(\mathcal{CV}_{\hat{C}}(\alpha_1), \mathcal{CV}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-SetField*]	$\mathcal{CV} \Vdash_{cv,\rho_s} ({e_0}^{\alpha_0}[{e_1}^{\alpha_1}] = {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_0}^{\alpha_0} \wedge \mathcal{CV}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{set}(\mathcal{CV}_{\hat{C}}(\alpha_0), \mathcal{CV}_{\hat{C}}(\alpha_1), \mathcal{CV}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-DelField*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{delete } {e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\widehat{del}(\mathcal{CV}_{\hat{C}}(\alpha_1), \mathcal{CV}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-Ref*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{ref}_{\ell}\ {e_1}^{\alpha_1})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\ell \in \mathcal{CV}_{\hat{C}}(\alpha) \wedge \mathcal{CV}_{\hat{C}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{\mu}}(\ell, \rho_s)$
[*CV-DeRef*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{deref } {e_1}^{\alpha_1})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\forall \ell \in \mathcal{CV}_{\hat{C}}(\alpha_1) : \mathcal{CV}_{\hat{\mu}}(\ell, \rho_s) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-SetRef*]	$\mathcal{CV} \Vdash_{cv,\rho_s} ({e_1}^{\alpha_1} = {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{C}}(\alpha) \wedge$
	$\forall \ell \in \mathcal{CV}_{\hat{C}}(\alpha_1) : \mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{\mu}}(\ell, \rho_s)$
[*CV-Send*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\overline{e_1}\langle e_2 \triangleright \rho \rangle)^{\alpha}$ iff
	$\mathcal{CV} \Vdash_{cv,\rho_s} e_1 \wedge \mathcal{CV}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV} \Vdash_{cv,\rho_s} e_2 \wedge \mathcal{CV}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\forall m \in \mathcal{CV}_{\hat{C}}(\alpha_1) : \forall \rho_m \sqsupseteq \mathcal{CV}_{\hat{P}}(\alpha) :$
	$\mathcal{CV}_{\hat{\Upsilon}}(m, \rho_m) = (\rho_r, \rho_e) \wedge$
	$\rho_r \sqsubseteq \rho_s \Rightarrow \rho_e \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{CV}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{CV}_{\hat{\Phi}}(m, \rho_m) \wedge$
	$\textbf{unit} \in \mathcal{CV}_{\hat{C}}(\alpha)$
[*CV-Exercise*]	$\mathcal{CV} \Vdash_{cv,\rho_s} (\textbf{exercise}(\rho))^{\alpha}$ iff
	$\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha) \wedge$
	$\textbf{unit} \in \mathcal{CV}_{\hat{C}}(\alpha)$

Table 4.3 Constraint generation part 1

[*CG-Val*]	$\mathcal{C}_{*\rho_s}[\![(v)^\alpha]\!] = \hat{v} \sqsubseteq \mathsf{C}(\alpha)$
[*CG-Var*]	$\mathcal{C}_{*\rho_s}[\![(x)^\alpha]\!] = \Gamma(x) \sqsubseteq \mathsf{C}(\alpha)$
[*CG-Lambda*]	$\mathcal{C}_{*\rho_s}[\![(\lambda x.{e_0}^{\alpha_0})^\alpha]\!] =$
	$\{\{\lambda x.{e_0}^{\alpha_0}\} \sqsubseteq \mathsf{C}(\alpha)\} \cup$
	$\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0})]\!]$
[*CG-Let*]	$\mathcal{C}_{*\rho_s}[\![(\textbf{let } x_1 = {e_1}^{\alpha_1} \textbf{ in } {e'}^{\alpha'})^\alpha]\!] =$
	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e'}^{\alpha'})]\!] \cup$
	$\{\mathsf{C}(\alpha_1) \sqsubseteq \Gamma(x_1)\} \cup \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\}) \cup$
	$\{\mathsf{P}(\alpha') \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{C}(\alpha') \sqsubseteq \mathsf{C}(\alpha)\}$
[*CG-App*]	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1}\ {e_2}^{\alpha_2})^\alpha]\!] =$
	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\{\{t\} \sqsubseteq \mathsf{C}(\alpha_1) \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \Gamma(x)$
	$\quad \mid t = (\lambda x.{e_0}^{\alpha_0}) \in lambda_*\} \cup$
	$\{\{t\} \sqsubseteq \mathsf{C}(\alpha_1) \Rightarrow \mathsf{C}(\alpha_0) \sqsubseteq \mathsf{C}(\alpha)$
	$\quad \mid t = (\lambda x.{e_0}^{\alpha_0}) \in lambda_*\} \cup$
	$\{\{t\} \sqsubseteq \mathsf{C}(\alpha_1) \Rightarrow \mathsf{P}(\alpha_0) \sqsubseteq \mathsf{P}(\alpha)$
	$\quad \mid t = (\lambda x.{e_0}^{\alpha_0}) \in lambda_*\} \cup$
[*CG-Op*]	$\mathcal{C}_{*\rho_s}[\![(op(\overrightarrow{{e_i}^{\alpha_i}}))^\alpha]\!] =$
	$\bigcup_i (\mathcal{C}_{*\rho_s}[\![({e_i}^{\alpha_i})]\!] \cup \{\mathsf{P}(\alpha_i) \sqsubseteq \mathsf{P}(\alpha)\}) \cup$
	$\{\widehat{op}(\mathsf{C}(\alpha_i)) \sqsubseteq \mathsf{C}(\alpha)\}$
[*CG-Cond*]	$\mathcal{C}_{*\rho_s}[\![(\textbf{if } ({e_0}^{\alpha_0}) \{\ {e_1}^{\alpha_1}\ \} \textbf{ else } \{\ {e_2}^{\alpha_2}\ \})^\alpha]\!] =$
	$\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\{\mathcal{CV}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{CV}_{\hat{P}}(\alpha)\} \cup$
	$\{\widehat{\textbf{true}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_1) \sqsubseteq \mathsf{C}(\alpha)\} \cup$
	$\{\widehat{\textbf{true}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\{\widehat{\textbf{false}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \mathsf{C}(\alpha)\} \cup$
	$\{\widehat{\textbf{false}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\}$
[*CG-While*]	$\mathcal{C}_{*\rho_s}[\![(\textbf{while } ({e_1}^{\alpha_1}) \{\ {e_2}^{\alpha_2}\ \})^\alpha]\!] =$
	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\{\textbf{true} \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\{\textbf{false} \in \mathsf{C}(\alpha_1) \Rightarrow \widehat{\textbf{undefined}} \sqsubseteq \mathsf{C}(\alpha)\}$

Table 4.4 Constraint generation part 2

[*CG-GetField*] $\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\widehat{get}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$

[*CG-SetField*] $\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0}[{e_1}^{\alpha_1}] = {e_2}^{\alpha_2})]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})^{\alpha}]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_3) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\widehat{set}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$

[*CG-DelField*] $\mathcal{C}_{*\rho_s}[\![(\mathbf{delete}\ {e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\widehat{del}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$

[*CG-Ref*] $\mathcal{C}_{*\rho_s}[\![(\mathbf{ref}_{\ell}\ {e_1}^{\alpha_1})^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\{\mathsf{C}(\alpha_1) \sqsubseteq \mathsf{M}(\ell, \rho_s)\} \cup \{\{\ell\} \sqsubseteq \mathsf{C}(\alpha)\}$

[*CG-DeRef*] $\mathcal{C}_{*\rho_s}[\![(\mathbf{deref}\ {e_1}^{\alpha_1})^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\{\ell \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{M}(\ell, \rho_s) \sqsubseteq \mathsf{C}(\alpha)$
$\mid \ell \in \mathit{Ref}_*\}$

[*CG-SetRef*] $\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1} = {e_2}^{\alpha_2})^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\{\ell \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \mathsf{M}(\ell, \rho_s)$
$\mid \ell \in \mathit{Ref}_*\} \cup$
$\{\mathsf{C}(\alpha_2) \sqsubseteq \mathsf{C}(\alpha)\}$

[*CG-Send*] $\mathcal{C}_{*\rho_s}[\![(\overline{e_1}\langle e_2 \triangleright \rho\rangle)^{\alpha}]\!] =$
$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
$\{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\{\{\mathbf{unit}\} \sqsubseteq \mathsf{C}(\alpha)\} \cup$
$\{\{m\} \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{P}(\alpha) \sqsubseteq \rho_m \Rightarrow \Upsilon(m, \rho_m) = (\rho_r, \rho_e)$
$\mid m \in \mathit{Names}_*, \rho_m \in \mathit{NamePerms}_*\} \cup$
$\{\{m\} \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{P}(\alpha) \sqsubseteq \rho_m \Rightarrow \rho_r \sqsubseteq \rho_s \Rightarrow \rho_e \sqsubseteq \mathsf{P}(\alpha)$
$\mid m \in \mathit{Names}_*, \rho_m \in \mathit{NamePerms}_*\} \cup$
$\{\{m\} \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{P}(\alpha) \sqsubseteq \rho_m \Rightarrow \rho_r \sqsubseteq \rho_s \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \Phi(m, \rho_m)$
$\mid m \in \mathit{Names}_*, \rho_m \in \mathit{NamePerms}_*\}$

[*CG-Exercise*] $\mathcal{C}_{*\rho_s}[\![(\mathbf{exercise}(\rho))^{\alpha}]\!]$
$\{\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \mathsf{P}(\alpha)\} \cup$
$\mathbf{unit} \in \mathsf{C}(\alpha)$

Table 4.5 Worklist Algorithm part 1.

INPUT: $\mathcal{C}_{*\rho_s}[\![(e_*)^\alpha]\!]$
OUTPUT: $(\mathsf{C}, \Gamma, \mathsf{M})$
METHOD: Step 1: Initialization

```
W := [] : Queue(CElem)
D := [] : Map(CElem -> V̂)
E := [] : Map(CElem -> Constraint)
for a in cache do
    Add (C a, ⊥) D
    Add (C a, []) E
for x in vars do
    Add (R x, ⊥) D
    Add (R x, []) E
for r in refs do
    Add (Mu (⊥, ℓ), ⊥) D
    Add (Mu (⊥, ℓ), []) E
```

Step 2: Building the graph

```
for cc in lst do
    case cc of
    | {t} ⊑ p ->
      propagate p {t}
    | p1  ⊑ p2 ->
      Add cc E[p1]
    | {t} ⊑ p ⇒ p1 ⊑ p2 ->
      Add cc E[p]
      Add cc E[p1]
    | {t} ⊑ p ⇒ {t1} ⊑ p2 ->
      Add cc E[p]
    | ôp(p̂s) ⊑ p1 ->
      for p in ps do
        Add cc E[p]
    | Get^(p1, p2) ⊑ p3 ->
      Add cc E[p1]
      Add cc E[p2]
    | Del^(p1, p2) ⊑ p3 ->
      Add cc E[p1]
      Add cc E[p2]
    | Set^(p1, p2, p3) ⊑ p4 ->
      Add cc E[p1]
      Add cc E[p2]
      Add cc E[p3]
```

Table 4.6 Worklist Algorithm part 2.

Step 3: Iteration

```
while W ≠ [] do
    q = dequeue W
    for cc in E[q] do
        case cc of
        | p1  ⊑ p2 ->
            propagate p2 D[p1]
        | {t} ⊑ p ⇒ p1 ⊑ p2 ->
            if t ∈ D[p] then
                propagate p2 D[p1]
        | {t} ⊑ p ⇒ {t1} ⊑ p2 ->
            if t ∈ D[p] then
                propagate p2 {t1}
        | op^(ps→) ⊑ p1 ->
            args = [D[p] | p ∈ ps→]
            res = op^ args
            propagate p1 res
        | Get^(p1, p2) ⊑ p3 ->
            propagate p3
                [D[C α] | α ∈ Get^ (D[p1], D[p2])]
        | Del^(p1, p2) ⊑ p3 ->
            propagate p3 Del^ (D[p1], D[p2])
        | Set^(p1, p2, p3) ⊑ p4 ->
            propagate p4 Set^ (D[p1], D[p2]. D[p3])
```

Step 4: Recording the solution

```
for ℓ in Ref* do μ̂(ℓ) = D[Mu ℓ]
for x in Var* do Γ̂(x) = D[R x]
for α in Cache* do Ĉ(α) = D[C α]
```

USING:

```
propagate q d =
    if d ⋢ D[q] then
        D[q] = D[q] ⊔ d
        Enqueue q W
```

Chapter 5

Experiments

5.1 Findings

Unfortunately the tool is not yet ready to perform an whole analysis. Instead we can approximate a program using the approach of chapter 4.

The performance are critical since for one analysis (a desugared JavaScript file of 100 rows of code correspond to 10000 λ_{JS} rows) takes from 6 to 8 hours. We plan to enhance performance using lazy approach stated in [19, 18, 20].

Chapter 6

Conclusion

6.1 Analysis results

We proposed a sound flow logic based analysis technique targeted at the static detection of privilege escalations attacks on Google Chrome extension, and we developed a prototype of a tool that implements our analysis.

The analysis proposed is sound and can validate real Chrome extension against privilege escalation. Indeed it gives an upper bound of the privilege that an extension leak to an attacker according to the power of such attacker. Since the analysis is sound we can truly predict if a permission is not leaked at all.

The tool that has been developed is not yet ready, but the main goal of it are almost ready. Indeed it can analyze an extension determining all possible values of each node, and from this analysis it is able to extract all possible messages sent by any content script and by the background. With this information we can perform the analysis on systems, finding the pairs privileges of the attacker, privileges leaked to it. The program unfortunately is very slow since the nature of a desugared λ_{JS} code. It takes about six or eight hours to perform an analysis, but we are going to enhance its performance. This enhancement will be done starting from [19, 18, 20] and modifying our constraint solver algorithm using a faster approach like the lazy one.

6.2 Future works

As part of our future work, we would like to enhance the performances of our actual tool and to expand its functionality in order to obtain a full implementation of the analysis.

We also want to expand this work in order to be able not only to check bundled extensions, but also to automatically unbundle bundled extensions inserting in their code stronger security checks.

References

[1] Banshee constraint solver
http://banshee.sourceforge.net/, May 2014.

[2] Chrome extension match pattern specification
https://developer.chrome.com/extensions/match_patterns, May 2014.

[3] Chrome extension overview
https://developer.chrome.com/extensions/overview, May 2014.

[4] Chrome extension runtime specification
https://developer.chrome.com/extensions/runtime, May 2014.

[5] Share me not extension
http://sharemenot.cs.washington.edu/, May 2014.

[6] Devdatta Akhawe, Prateek Saxena, and Dawn Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 23–23, Berkeley, CA, USA, 2012. USENIX Association.

[7] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. Technical Report UCB/EECS-2009-185, EECS Department, University of California, Berkeley, Dec 2009.

[8] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Lintent: Towards security type-checking of android applications. In Dirk Beyer and Michele Boreale, editors, *Formal Techniques for Distributed Systems*, volume 7892 of *Lecture Notes in Computer Science*, pages 289–304. Springer Berlin Heidelberg, 2013.

[9] Nicholas Carlini, Adrienne Porter Felt, and David Wagner. An evaluation of the google chrome extension security architecture. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[10] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. Static analysis of string values. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering*, ICFEM'11, pages 505–521, Berlin, Heidelberg, 2011. Springer-Verlag.

[11] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Kirsten Lackner Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for cml. In *ICFP*, pages 38–51, 1997.

[13] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 126–150, Berlin, Heidelberg, 2010. Springer-Verlag.

[14] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 256–275, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] René Rydhof Hansen. Flow logic for carmel. Technical report, Citeseer, 2002.

[16] René Rydhof Hansen. Implementing the flow logic for carmel. Technical report, SECSAFE-IMM-004-1.0, 2002.

[17] David Van Horn and Matthew Might. An analytic framework for javascript. *CoRR*, abs/1109.4467, 2011.

[18] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the html dom and browser api in static analysis of javascript web applications. In *SIGSOFT FSE*, pages 59–69, 2011.

[19] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 238–255, Berlin, Heidelberg, 2009. Springer-Verlag.

[20] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *SAS*, pages 320–339, 2010.

[21] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[22] Flemming Nielson, Hanne Riis Nielson, Hongyan Sun, Mikael Buchholtz, René Rydhof Hansen, Henrik Pilegaard, and Helmut Seidl. The succinct solver suite. In *TACAS*, pages 251–265, 2004.

[23] Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation*, pages 223–244, 2002.