Frontespizio.pdf

List of Tables

Todo list

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

Chapter 1

Introduction

1.1 Privilege separation

1.2 Privilege escalation attacks

1.3 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 lot of extensions with very various behaviors like security enhancers, theme changers, organizers or other utilities, multimedia visualizer, games and others. For example, AdBlock (one of the top downloaded) is an extension made to block all ads on websites; ShareMeNot "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 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 [6], the extension framework is built to force programmers to adopt privilege separation, least privilege and strong isolation. Privilege separation, as explained before in 1.1, force the developer to split the application in components providing for the communication a message passing interface; least privilege gives to the app the least set of permission needed through the execution of the extension and the strong isolation separate the heaps of the various components of the extension running them in different processes in order to block any possible escalation and direct delegation.

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, instead, have 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, but they can communicate with the content scripts only via message passing.

1.4 Proposal

titolare meglio... tipo our work...

In this work do a study on Chrome Extensions identifying a possible weakness. We write a calcolus

Chapter 2

Background

2.1 Chrome extension architecture

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 contains 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 containing the specification of the extension. It is the entry point of the extension and 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 is used the `script` field, the scripts are injected in a empty extension core page, while if it is used `page` the HTML document with all his 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;
- `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

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>
```

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"]
      }
    ]
}
```

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 is injected the file `background.js`. It also has permissions tabs and download, and can execute XHR to all path contained 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 require `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 to access and modify only standard fields of the DOM object, while other changes are kept locally[6]. This strong isolation mitigate the risk of code injection since it blocks almost completely pointer exchange.

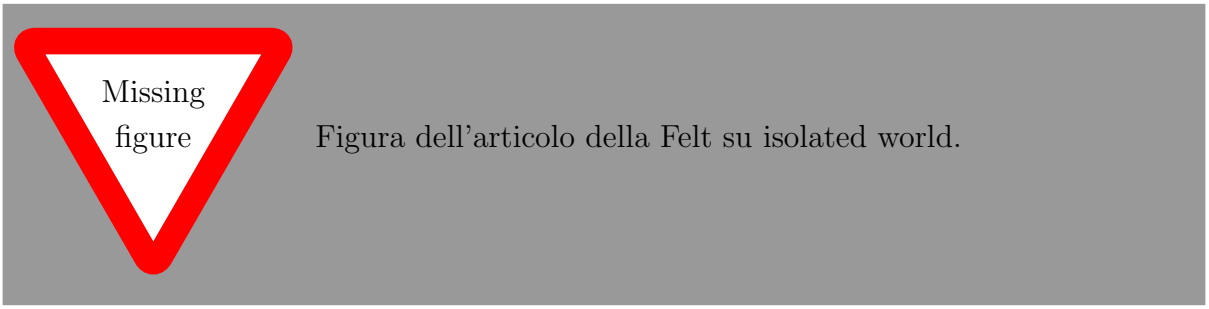

Figura dell'articolo della Felt su isolated world.

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 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 object, they can also request to the web such resources, but this can be very dangerous. In fact if the resources are on simple HTTP connections these can be altered by an attacker. In [8] is described how to enforce the security policy in order to avoid such possible weakness. Background pages can interact with content scripts via message passing.

Maybe figura schema Chrome Extensions comm & elements.

2.1.4 Message passing API

Every content script of the extension can access `chrome.runtime` object that contains the implementation of the message passing interface [4].

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. This prevents exchange of pointers or of functions, but limits the expressiveness of the prototype-based object-oriented feature of Javascript. It also fails in presence of recursive objects.

In order to listen to inbound 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 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. In it 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 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` is triggered 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 grant 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 Bundling

esempio;

scriviere molt bene questa parte

As seen in table 2.3 the choice taken by a 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 is called 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 escalating a permission that it does not have in the original setting.

To avoid such weakness 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 table 2.6 is showed an unbundled code.

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)
```

2.3 Flow logic

Flow logic, introduced in [19], is a static analysis approach that derive from state of the art in program verification and has been successfully used in research projects [13, 12]. It has its root in classical approaches of static program analysis [17] like control flow analysis [9], 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 semantic, 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 more closer to standard semantic 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 [17] or with a more sophisticated ones like the succinct solver [18] or the BANSHEE solver [1], is possible to compute the estimate for a program.

quale dei due

In this work, indeed, 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.

traballante...

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 Two unbundled code.

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 explain 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}. $\lambda_{JS}$is a Scheme dialect described in [10] 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 [11] and [14]. 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 [7].

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 remove the complexity to deal with a scoping different to the statical one. In [10] is shown how the desugaring function transform 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. In other words the three construct over, bind a value to a variable and in the expression that follows the bind the variable lookup gives the value of its in the bind.

ile?

Values.

Let us now introduce the possible native value that are found in the calculus. We let u, v range over *values*, defined by the following productions:

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

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.

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

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

The operations $op(\overrightarrow{e_i})$ is used for all arithmetic, boolean and string operations. It includes even string equality, denoted by ==. The creation of new references comes with an annotation: $\textbf{ref}_\ell\ e$ 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 **exercise**(ρ) 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 **exercise**(ρ) 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 world 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 $\rho \sqsubseteq \rho_s$ are 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 more or less as in table 3.1.

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 [10], 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 $\xRightarrow{\vec{\alpha}}$ for the reflexive-transitive closure of $\xrightarrow{\alpha}$.

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.

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
```

Evaluation contexts are defined by the following productions:

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

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.

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

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

$$\text{(R-Exercise)}\quad \frac{\rho \sqsubseteq \rho_a}{\mu; h; a\{\!|E\langle \mathbf{exercise}(\rho)\rangle|\!\}_{\rho_a} \xrightarrow{a:\rho_a \gg \rho} \mu; h; a\{\!|E\langle \mathbf{unit}\rangle|\!\}_{\rho_a}} \qquad \text{(R-Set)}\quad \frac{\mu; h; i \xrightarrow{\alpha} \mu'; h'; i'}{\mu; h; i, i'' \xrightarrow{\alpha} \mu'; h'; i', i''}$$

$$\text{(R-Basic)}\quad \frac{\mu; e \hookrightarrow_\rho \mu'; e'}{\mu; h; a\{\!|e|\!\}_\rho \xrightarrow{\cdot} \mu'; h; a\{\!|e'|\!\}_\rho}$$

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 $\lambda_{JS}$with no impact on the, but 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

nome sezione!!

Definition 2 (Exercise)**.**

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

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

- *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.

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, 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. This means that a system is safe if the opponent cannot access privileges that the system initially did not have. For example in the chrome extension, given a clean component that executes at most permission `tabs`, an attacker that compromise it cannot access privilege higher than `tabs`.

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:

```
cs1(x <| CS1: SEND).send(b,{tag: "Message1"} |> BACK)
cs2(x <| CS2: SEND).send(b,{tag: "Message2"} |> BACK)
b(x <| SEND: BACK).
        if (x[tag] == "Message1") then exercise(rho)
        else exercise(rho')
```

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.

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 ---> rho join rho'
cs2: CS2 ---> rho join rho'
b:   SEND ---> rho join rho'
```

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). lasciato ugual

commented work here! maybe remove or move to future work.

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,

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

the abstract environment is a four-tuple made of the following components:

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

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 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

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

$$\frac{\lambda x^{\rho_e} \in \hat{v} \qquad \mathcal{C} \Vdash_\rho e : \hat{v}' \gg \rho' \qquad \hat{v}' \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(\lambda x) \qquad \rho' \sqsubseteq \rho_e}{\mathcal{C} \Vdash_\rho \lambda x.e \rightsquigarrow \hat{v}}\ (\text{PV-Fun})$$

$$\frac{\{\overrightarrow{\langle\!| str_i : v_i |\!\rangle_{\mathcal{C},\rho}}\} \sqsubseteq \hat{v}}{\mathcal{C} \Vdash_\rho \{\overrightarrow{str_i : v_i}\} \rightsquigarrow \hat{v}}\ (\text{PV-Rec})$$

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.

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).

Running Example. For our running example, we are able to analyse the code with respect to the abstract stack $\hat{\Upsilon}$ such that: $\hat{\Upsilon}(cs1, CS1) = (\top, \rho \sqcup \rho')$ and $\hat{\Upsilon}(cs2, CS2) = (\top, \rho \sqcup \rho')$ and $\hat{\Upsilon}(b, B) = (CS1 \sqcap CS2, \rho \sqcup \rho')$.

iorare

Table 3.5 Flow analysis for expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 3.6 Flow analysis for systems

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

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

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

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

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

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

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

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

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

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

3.5 Theorem

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* $\rho' = Leak_\rho(\mathcal{C})$*.*

3.6 Requirements for correctness

In this part 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 $\bot$

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* $\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho_a} = \langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho_b}$.

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

4.1 Analysis specification

specifica dell'analisi

4.1.1 Compositional Verbose

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

To lighten the notation, we denote by $\mathcal{C}$ all $\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}, \hat{\Upsilon}, \hat{\Phi}$.

4.2 Constraint generation

Constraint elements: E.

Cache element	$\mathsf{C}(\ell)$	:	$\mathcal{L} \to \hat{V}$
Var element	$\mathsf{\Gamma}(x)$	:	$\mathcal{V} \to \hat{V}$
State element	$\mathsf{M}(\mathcal{P}, ref)$	:	$\mathcal{L} \times \mathcal{P} \to \hat{V}$

Permission Element: $\mathsf{P}(\ell) : \mathcal{L} \to \mathcal{P}$
Constraint form.

Term inclusion	$\{\hat{v}\}$	$\sqsubseteq$	E
Element inclusion	E	$\sqsubseteq$	E
Permission inclusion	$\mathsf{P}(\ell)$	$\sqsubseteq$	$\mathsf{P}(\ell')$
Operation	$\widehat{Op}(\overrightarrow{\mathsf{E}_i})$	$\sqsubseteq$	E
Implication	$\{\hat{v}\} \sqsubseteq \mathsf{E}$	$\Rightarrow$	$\mathsf{E} \sqsubseteq \mathsf{E}$

Table 4.1 Compositional Verbose part 1

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

Misc:

r_*	is the set of all references of the program;
$lambda_*$	is the set of all lambdas of the program;

4.3 Constraint solving

4.4 Abstract domains choice

$R_1 = \{\overrightarrow{\widehat{str_i} : \widehat{v_i}}\} \sqsubseteq \{\overrightarrow{\widehat{str_j} : \widehat{v_j}}\} = R_2$ sse:

1. R_1 ha meno campi di R_2
2. ogni campo di R_1 e' piu' preciso del **corrispondente** campo di R_2

$\forall i, \exists j : \widehat{str_i} \sqsubseteq \widehat{str_j}$
$\forall i, \exists j : \widehat{str_i} \sqsubseteq \widehat{str_j} \Rightarrow \widehat{v_i} \sqsubseteq \widehat{v_j}$
Set:

- Exact
  - $\exists \rightarrow Union$
  - $\nexists \rightarrow addinprefix$
- Prefix
  - aggiungo in $*$

$\hat{v} \sqsubseteq \hat{v}'$ iff $\forall \widehat{u}_i \in \hat{v}, \exists \widehat{u}_j \in \hat{v}' : \widehat{u}_i \sqsubseteq \widehat{u}_j$.
If Galois connection then
$\hat{v} \sqsubseteq \hat{v}'$ iff $\gamma(\hat{v}) \subseteq \gamma(\hat{v}')$
where $\gamma : \widehat{V} \rightarrow P(V)$ is the concretisation function.
$\gamma_p : \widehat{PV} \rightarrow P(V)$
$\gamma(\hat{v}) = \bigcup_{\widehat{u}_i \in \hat{v}} \gamma_p(\widehat{u}_i)$

$\widehat{pre_{bool}} = \widehat{true} | \widehat{false}$

$\widehat{u_{bool}} = \{\widehat{pre_{bool}}\}^{\rightarrow}$ with $\sqsubseteq = \subseteq$

$\widehat{pre_{int}} = \oplus | 0 | \ominus$

$\widehat{u_{int}} = \{\widehat{pre_{int}}\}^{\rightarrow}$ with $\sqsubseteq = \subseteq$

$\widehat{pre_{string}} = s | s*$

$\widehat{u_{string}} = \{\widehat{pre_{string}}\}^{\rightarrow}$ with $\sqsubseteq = \subseteq$
— Giulia's spec. is more tricky than $\subseteq$

$\widehat{pre_{ref}} = r$

$\widehat{u_{ref}} = \{\widehat{pre_{ref}}\}^{\rightarrow}$ with $\sqsubseteq = \subseteq$

$\widehat{pre_{\lambda}} = \lambda$

$\widehat{u_{\lambda}} = \{\widehat{pre_{\lambda}}\}^{\rightarrow}$ with $\sqsubseteq = \subseteq$

$\widehat{pre_{rec}} = \{\widehat{str_i} : \hat{v}_i\}^{\rightarrow}$

$\widehat{u_{rec}} = \widehat{pre_{rec}}$ with $\sqsubseteq = \widehat{u_{rec}}_{\sqsubseteq}$

$\hat{v} = (\widehat{u_{bool}}, \widehat{u_{int}}, \widehat{u_{string}}, \widehat{u_{ref}}, \widehat{u_{\lambda}}, \widehat{u_{rec}}, \{\widehat{Null}\}, \{\widehat{Undef}\})$

with $\hat{v} \sqsubseteq \hat{v}'$ iff
$\widehat{u_{bool}} \sqsubseteq \widehat{u_{bool}}' \wedge$
$\widehat{u_{int}} \sqsubseteq \widehat{u_{int}}' \wedge$
$\widehat{u_{string}} \sqsubseteq \widehat{u_{string}}' \wedge$
$\widehat{u_{ref}} \sqsubseteq \widehat{u_{ref}}' \wedge$
$\widehat{u_{\lambda}} \sqsubseteq \widehat{u_{\lambda}}' \wedge$
$\widehat{u_{rec}} \sqsubseteq \widehat{u_{rec}}' \wedge$
$\widehat{Null} \notin \hat{v}' \vee \widehat{Null} \in \hat{v} \wedge \widehat{Null} \in \hat{v}' \wedge$
$\widehat{Undef} \notin \hat{v}' \vee \widehat{Undef} \in \hat{v} \wedge \widehat{Undef} \in \hat{v}'$

4.5 Abstract operations

4.6 Requirements verification

4.7 Implementation-specific details

Table 4.2 Compositional Verbose part 2

[*CV-GetField*]	$\mathcal{C} \models_{c\rho_s} ({e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{C}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\widehat{get}(\mathcal{C}_{\hat{C}}(\alpha_1), \mathcal{C}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{C}_{\hat{C}}(\alpha)$
[*CV-SetField*]	$\mathcal{C} \models_{c\rho_s} ({e_0}^{\alpha_0}[{e_1}^{\alpha_1}] = {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_0}^{\alpha_0} \wedge \mathcal{C}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{C}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\widehat{set}(\mathcal{C}_{\hat{C}}(\alpha_0), \mathcal{C}_{\hat{C}}(\alpha_1), \mathcal{C}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{C}_{\hat{C}}(\alpha)$
[*CV-DelField*]	$\mathcal{C} \models_{c\rho_s} (\mathbf{delete}\ {e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{C}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\widehat{del}(\mathcal{C}_{\hat{C}}(\alpha_1), \mathcal{C}_{\hat{C}}(\alpha_2)) \sqsubseteq \mathcal{C}_{\hat{C}}(\alpha)$
[*CV-Ref*]	$\mathcal{C} \models_{c\rho_s} (\mathbf{ref}_{\ell}\ {e_1}^{\alpha_1})^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\ell \in \mathcal{C}_{\hat{C}}(\alpha) \wedge \mathcal{C}_{\hat{C}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{\mu}}(\ell, \rho_s)$
[*CV-DeRef*]	$\mathcal{C} \models_{c\rho_s} (\mathbf{deref}\ {e_1}^{\alpha_1})^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\forall \ell \in \mathcal{C}_{\hat{C}}(\alpha_1) : \mathcal{C}_{\hat{\mu}}(\ell, \rho_s) \sqsubseteq \mathcal{C}_{\hat{C}}(\alpha)$
[*CV-SetRef*]	$\mathcal{C} \models_{c\rho_s} ({e_1}^{\alpha_1} = {e_2}^{\alpha_2})^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} {e_1}^{\alpha_1} \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} {e_2}^{\alpha_2} \wedge \mathcal{C}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{C}}(\alpha) \wedge$
	$\forall \ell \in \mathcal{C}_{\hat{C}}(\alpha_1) : \mathcal{C}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{\mu}}(\ell, \rho_s)$
[*PE-Send*]	$\mathcal{C} \models_{c\rho_s} (\overline{e_1}\langle e_2 \triangleright \rho \rangle)^{\alpha}$ iff
	$\mathcal{C} \models_{c\rho_s} e1 \wedge \mathcal{C}_{\hat{P}}(\alpha_1) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C} \models_{c\rho_s} e2 \wedge \mathcal{C}_{\hat{P}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\forall m \in \mathcal{C}_{\hat{C}}(\alpha_1) : \forall \rho_m \sqsupseteq \mathcal{C}_{\hat{P}}(\alpha) :$
	$\mathcal{C}_{\hat{\Upsilon}}(m, \rho_m) = (\rho_r, \rho_e) \wedge$
	$\rho_r \sqsubseteq \rho_s \Rightarrow \rho_e \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathcal{C}_{\hat{C}}(\alpha_2) \sqsubseteq \mathcal{C}_{\hat{\Phi}}(m, \rho_m) \wedge$
	$\mathbf{unit} \in \mathcal{C}_{\hat{C}}(\alpha)$
[*CV-Exercise*]	$\mathcal{C} \models_{c\rho_s} (\mathbf{exercise}(\rho))^{\alpha}$ iff
	$\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha) \wedge$
	$\mathbf{unit} \in \mathcal{C}_{\hat{C}}(\alpha)$

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*]	$\mathcal{C}_{*\rho_s}[\![(\textbf{if } ({e_0}^{\alpha_0}) \{\ {e_1}^{\alpha_1}\ \} \textbf{ else } \{\ {e_2}^{\alpha_2}\ \})^\alpha]\!] =$
	$\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\{\mathcal{C}_{\hat{P}}(\alpha_0) \sqsubseteq \mathcal{C}_{\hat{P}}(\alpha)\} \cup$
	$\{\widehat{\textbf{true}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_1) \sqsubseteq \mathsf{C}(\alpha)\} \cup$
	$\{\widehat{\textbf{true}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\{\widehat{\textbf{false}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \mathsf{C}(\alpha)\} \cup$
	$\{\widehat{\textbf{false}} \in \mathsf{C}(\alpha_0) \Rightarrow \mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\}$
[*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*]	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\quad \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \widehat{get}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$
[*CG-SetField*]	$\mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0}[{e_1}^{\alpha_1}] = {e_2}^{\alpha_2})]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_0}^{\alpha_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})^{\alpha}]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\quad \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_3) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \widehat{set}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$
[*CG-DelField*]	$\mathcal{C}_{*\rho_s}[\![(\mathbf{delete}\ {e_1}^{\alpha_1}[{e_2}^{\alpha_2}])^{\alpha}]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\quad \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \widehat{del}(\mathsf{C}(\alpha_1), \mathsf{C}(\alpha_2)) \sqsubseteq \mathsf{C}(\alpha)$
[*CG-Ref*]	$\mathcal{C}_{*\rho_s}[\![(\mathbf{ref}_{\ell}\ {e_1}^{\alpha_1})^{\alpha}]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \{\mathsf{C}(\alpha_1) \sqsubseteq \mathsf{M}(\ell, \rho_s)\} \cup \{\{\ell\} \sqsubseteq \mathsf{C}(\alpha)\}$
[*CG-DeRef*]	$\mathcal{C}_{*\rho_s}[\![(\mathbf{deref}\ {e_1}^{\alpha_1})^{\alpha}]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \{\ell \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{M}(\ell, \rho_s) \sqsubseteq \mathsf{C}(\alpha)$
	$\qquad \mid \ell \in Ref_*\}$
[*CG-SetRef*]	$\mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1} = {e_2}^{\alpha_2})^{\alpha}]\!] =$
	$\quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\alpha_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\alpha_2})]\!] \cup$
	$\quad \{\mathsf{P}(\alpha_1) \sqsubseteq \mathsf{P}(\alpha)\} \cup \{\mathsf{P}(\alpha_2) \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \{\ell \in \mathsf{C}(\alpha_1) \Rightarrow \mathsf{C}(\alpha_2) \sqsubseteq \mathsf{M}(\ell, \rho_s)$
	$\qquad \mid \ell \in Ref_*\} \cup$
	$\quad \{\mathsf{C}(\alpha_2) \sqsubseteq \mathsf{C}(\alpha)\}$
[*CG-Send*]	...
[*CG-Exercise*]	$\mathcal{C}_{*\rho_s}[\![(\mathbf{exercise}(\rho))^{\alpha}]\!]$
	$\quad \{\rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \mathsf{P}(\alpha)\} \cup$
	$\quad \mathbf{unit} \in \mathsf{C}(\alpha)$

Table 4.5 Worklist Algorithm part 1.

INPUT: $C_*[e_*]$
OUTPUT: (cat, ro)
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

share me not

5.2 Performance

SLOW... Very SLOW!!! =¿ Lazy [15, 16]

Chapter 6

Conclusion

6.1 Conclusions

6.2 Future works

[19]

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] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] 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.

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