Privilege separation in browser architectures

Michele Bugliesi   Stefano Calzavara   Enrico Steffinlongo

June 4, 2014

List of Tables

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"[4]. 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 [5], 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 [2] 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 Chrome extension architecture weaknesses

1.5 Proposal

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 details

As showed in [2] a Chrome Extension is an archive containing files of various kind like JavaScript, HTML, JSON, pages, images and other 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 that with all the specification of the extension. It is the entry point of the extension and contain two mandatory fields: `name` and `version` containing the name and the version of the extension. Other important field are `background`, `content_scripts`, `permissions` and we will explain here.

- `background`: has or a `script` field containing the source of the content script or a `page` field containing the source of an HTML page. If the `script` field is used the scripts are injected in a empty extension core page, while if is used `page` the HTML document with all his elements, including scripts, compose the extension core.

- `content_scripts`: contains a list of content script objects. A content script object can contains field `js` that contain the list of Javascript files to be injected and other, and must contain field `matches`: a list of match patterns. Match patterns are explained below.

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

Another possible field is `optional_permissions`. It contains the list of optional permission that the extension could require and are used to restrict the privilege granted to the app. To use one of this permissions the background page has to require explicitly them and to release after use. Using the optional permission is possible to reduce the possible privileges escalated by an attacker, **but are used rarely and are not in our interest.**

Togliere?

Table 2.1 Url pattern syntax. Table taken from [1]

```
<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"]
      }
    ]
}
```

A match pattern is a string composed of three parts: `scheme`, `host` and `path`. A part can contains a value or `"*"` that means all possible values. In table 2.1 is shown the syntax of the URL patterns. For more details refer to [1]. As we can see we can decide to inject some content scripts on pages derived from a given match. This is used when 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 pages with google as host and with all path (e.g., `mail.google.com`, `www.google.com`, `docs.google.com/mine`).

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` an that has tabs, downloads permission and that 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 can be are automatically injected to the web page if this match with the pattern defined in the manifest or that 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 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 the page. 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 same extension are run together in their own address space, and the only way they have to interact with the page is via DOM API. As explained in [5] browser provide one common DOM element accessed via its API and all scripts both on the page or in the extension can modify it, but only changes of the standard DOM properties are shared, while other changes are kept locally.

mettere figura dell'articolo?

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 requested. 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 because if the resources are on simple HTTP connections them can be altered by an attacker. In [6] is described how to enforce the security policy in order to avoid such possible weakness. As already said background pages can interact with content scripts via message passing.

2.1.4 Message passing API

Every content script of the extension can access `chrome.runtime` that is the object on wich the message passing interface is implemented (for more details refer to [3]).

The main method 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. The message, before sending is marshaled using a JSON serializer. This prevent 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.

An element, to listen to inbound messages, has to register a function on the `chrome.runtime.onMess` 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 that message. The sender field is very important because is the only warranty

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);
```

about 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 proper of the `tab` object to which the message has to be sent. Its behavior is the same of the `chrome.runtime.sendMessage` method.

In table 2.3 we can see how to use the simple message passing interface. A component simply sends the message and wait for a response. The other register `onMessage` function as event listener for messages. When it is triggered by an incoming message `onMessage` check the message and decide to compute something according to the request or to refuse the message.

Another way to communicate, that is more secure, is done using a channel as in table 2.4. In the message passing API there is a method called `connect` that takes as optional arguments a message to deliver when the corresponding event `onConnect` is triggered and that returns a port. Such object is a bidirectional channel that can be used to communicate and 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 the port endpoints can communicate. This grants the sender of the message.

2.1.5 Bundling

iere molto
e questa
e

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.

Table 2.4 Port creation.

Port opening active	Port opening passive

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

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 it, and according to this decide 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 title escalating a permission that he 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, 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.

2.2 Flow logic

Flow logic, introduced in [12], is a static analysis approach that derive from state of the art in program verification and has been successfully used in research projects [9, 8]. It has its root in classical approaches of static program analysis [?] like control flow analysis [7], 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. Another is that, like structural operational semantic that shares some properties with it, 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 and an analysis can be translated easily 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 control flow analysis.

Usually the formal analysis is done starting an abstract-succinct specification stating when an estimate is acceptable. Such specification is turned in a compositional-verbose analysis. and given to an algorithm that from a program and the analysis produces a set of classical constraints on its internal elements. Finally this set of constraint, and an abstract representation of the concrete values, are given to a worklist algorithm that computes an acceptable estimate for the program.

The modularity fit 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 to a compositional verbose specification and from this, using a simple constraint solver like the worklist algorithm [?] or a more sophisticated one like the succinct solver [?], is possible to compute the estimate for a program.

e dei due?

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 developed together with Stefano Calzavara.

3.1 Calculus

In this section we introduce the language used to study privilege escalation. The core of the calculus models Javascript essential features and is a subset of λ_{JS}. λ_{JS} is a Scheme dialect described in [?] and studied in [?] and [?] used to desugar Javascript in order to simplify it in few construct with easy semantic behavior. 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, specific constructs are added 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.

modificato un po'

3.1.1 Syntax

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 "undefined"; 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.

lasciato ugual

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

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

All the forms above are standard. Notice that references bear a label ℓ, identifying the program point where they are created (see below). This is just needed for our static analysis and plays no role in the semantics. For simplicity, we consider only *closed* record values: this does not involve any loss of expressiveness.

restriction losed record es is useful he substitu- lemma and onent accept- ty. the next nition of seri- able value is l to prevent tions from g communi- d through message ing interface R-Sync), ch either ks the sub- ution lemma pponent ac- ability: that fies also the e restrictive tment of ref- ces.

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.*

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 $op(\overrightarrow{e_i})$ include arithmetic operations, boolean operations and string operations including 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. We observe that ℓ plays no role in the semantics: annotating the reference with the program point where it has been created will be useful for our static analysis (actually to deal with reference creation within loops).

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.

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

Instances. We let i range over pools of running *instances*, i.e., the active part of a system which is spawned when a message is received by an handler. An instance is a running expression, which is granted a set of permissions. 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 $s = \mu; h; i$.

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:

$$\begin{array}{l} A \to B : \{tag : \text{"}init\text{"}, val : x\} \\ B \to A : y \\ A \to B : \{tag : \text{"}okay\text{"}, other : z\} \end{array}$$

In Chrome, and in λ_{JS}, the component B is programmed more or less as in table 3.1. OK?

3.1.2 Semantics

The small-step operational semantics is defined in terms of a labelled reduction relation between systems, i.e., $s \xrightarrow{\alpha} s'$, and an auxiliary reduction relation between expressions that is directly inherited from λ_{JS}, i.e., $\mu; e \hookrightarrow_\rho \mu'; e'$. We associate labels to reduction steps just to easily state our security property 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 $\overset{\vec{\alpha}}{\Rightarrow}$ 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;
}
```

λ_{JS}

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

Evaluation contexts are defined by the following productions:

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

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.

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.

simplicity
exclude the
s for proto-
e inheritance
ambdaJS:
an include
n if needed,
sure we con-
r them in the

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

(R-Sync)

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

(R-Exercise)

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

(R-Set)

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

(R-Basic)

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

Definition 2 (Exercise)**.**

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

- *A system s* exercises at most ρ *iff $\forall s', \overrightarrow{\alpha}$ such that $s \overset{\overrightarrow{\alpha}}{\Longrightarrow} s'$, if $a : \rho_a \gg \rho' \in \{\overrightarrow{\alpha}\}$ then $\rho' \sqsubseteq \rho$.*

We say that a phrase of syntax ϕ is *closed* iff $fv(\phi) = \emptyset$.

We now introduce our threat model. We assume that the set of variables $\mathcal{V}$ is partitioned into two sets $\mathcal{V}_t$ (trusted variables) and $\mathcal{V}_u$ (untrusted variables). We stipulate 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 $\mathcal{V}_u$.

important fo
opponent typ
bility

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

- *for any handler $a(x \triangleleft \rho : \rho').e \in h$, we have $\rho' \sqsubseteq \rho$;*

- *for any instance $a\{\!|e|\!\}_{\rho'} \in i$, we have $\rho' \sqsubseteq \rho$;*

- *for any $x \in vars(h) \cup vars(i)$, we have $x \in \mathcal{V}_u$.*

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.

the security
property is n
given only ov
initial system
the old versio
with arbitrar
systems looke
too weak.

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.*

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

3.2 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 formal encoding of the Google Chrome extension 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.

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.

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).

lated exam- still useful next project.

Permission bundling analysis. Identifying dangerous permission bundling is challenging, since it depends on the structure of the exchanged messages. For instance, if both $CS1$ and $CS2$ employ the same tags, then the handler B is not bundled at all and must be accepted. We associate to each handler a set of incoming messages and a set of outgoing messages, as follows:

```
cs1: (emptyset, b <- {tag: "Message1"})
cs2: (emptyset, b <- {tag: "Message2"})
b:   ({{tag: "Message1"}, {tag: "Message2"}}, emptyset)
```

This is what is represented in the *abstract networks* which we introduce below, even though the real structure of the abstract networks is slightly more complicated than this to make the static analysis more precise.

Based on this information, we understand that we can refactor the code as follows:

```
cs1(x <| top).send(b1,{tag: "Message1"} |> B) with CS1
cs2(x <| top).send(b2,{tag: "Message2"} |> B) with CS2
b1(x <| CS1).if (x[tag] == "Message1") then >> rho else >> rho' with B
b2(x <| CS2).if (x[tag] == "Message1") then >> rho else >> rho' with B
```

For the opponent-aware analysis, this code is exactly as dangerous as the old one, since compromised components can just ignore tags (which indeed do not provide any security guarantee). Still, we can reuse our flow analysis to eliminate dead code:

```
cs1(x <| top).send(b1,{tag: "Message1"} |> B) with CS1
cs2(x <| top).send(b,{tag: "Message2"} |> B) with CS2
b1(x <| CS1).>> rho with B
b2(x <| CS2).>> rho' with B
```

Now the opponent-aware analysis shows a different surface for privilege escalation. Specifically, a caller with $CS1$ can escalate to ρ, while a caller with $CS2$ can escalate to ρ', which is much better than before.

3.3 Safety properties

The aim of the analysis is to statically predict which privileges are granted to the opponent through the message-passing interface: for this purpose, it is helpful to approximate also the values an expression may evaluate to. The analysis works with abstract representations of the concrete values. The flow logic specification then 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 $\mathcal{C}$.

Abstract Values and Abstract Operations. 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]:

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

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. Similarly to constants, 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. All these abstract operations can be chosen arbitrarily, as long as they satisfy the conditions needed for the proofs. We assume that abstract values are ordered by a pre-order $\sqsubseteq$: we require this pre-order to satisfy some relatively mild conditions (see the proofs section).

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

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

Abstract variable environments are standard: they associate abstract values to variables and to abstract functions. Abstract memories are also somewhat standard: they 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

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

references labelled with ℓ contain the abstract value $\hat{v}$, provided that they are protected with permission ρ.

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

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

Table 3.4 Flow analysis for values

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

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

The judgements of the flow analysis 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:*

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

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)
$$\frac{\begin{array}{c}\mathcal{C} \Vdash_{\rho_s} e_1 : \hat{v}_1 \sqsubseteq \mathcal{C}_{\hat{\Gamma}}(x) \gg \rho_1 \sqsubseteq \rho \\ \mathcal{C} \Vdash_{\rho_s} e_2 : \hat{v}_2 \sqsubseteq \hat{v} \gg \rho_2 \sqsubseteq \rho\end{array}}{\mathcal{C} \Vdash_{\rho_s} \textbf{let } x = e_1 \textbf{ in } e_2 : \hat{v} \gg \rho}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $\hat{\Upsilon}(b, B) = (CS1 \sqcap CS2, \rho \sqcup \rho')$.

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

3.5 Requirements for correctness

dizioni neces-
e per corret-
a

o parte 2
t

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

Table 3.6 Flow analysis for systems

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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. $\forall c \in \{\mathbf{true}, \mathbf{false}, \mathbf{unit}, \mathbf{undefined}\} : \forall \hat{v} : \{\hat{c}\} \sqsubseteq \hat{v} \Rightarrow \hat{c} \in \hat{v}$.

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

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. $\forall \hat{v}_1, \hat{v}_2 : vars(\widehat{del}(\hat{v}_1, \hat{v}_2)) \subseteq vars(\hat{v}_1)$.

Chapter 4

Implementation

4.1 Analysis specification

specifica dell'analisi

4.1.1 Abstract succinct

$$
\begin{array}{lcll}
\textit{Abstract cache} & \hat{C} & : & \mathcal{L} \to \hat{V} \\
\textit{Abstract variable environment} & \hat{\Gamma} & : & \mathcal{V} \to \hat{V} \\
\textit{Abstract memory} & \hat{\mu} & : & \mathcal{L} \times \mathcal{P} \to \hat{V} \\
\textit{Abstract permission cache} & \hat{P} & : & \mathcal{L} \to \mathcal{P}
\end{array}
$$

$$
\begin{array}{ll}
[\textit{PV-Name}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} n : \hat{v} \text{ iff } n \in \hat{v} \\
[\textit{PV-Var}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} x : \hat{v} \text{ iff } \hat{\Gamma}(x) \subseteq \hat{v} \\
[\textit{PV-Cons}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} c : \hat{v} \text{ iff } \{\hat{c}\} \subseteq \hat{v} \\
[\textit{PV-Ref}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \ell : \hat{v} \text{ iff } \ell \in \hat{v} \\
[\textit{PV-Lambda}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \lambda x.e : \hat{v} \text{ iff } \lambda_x^{\rho_e} \in \hat{v} \wedge (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}' \gg \rho' \\
 & \quad \hat{v}' \sqsubseteq \hat{\Gamma}(\lambda x) \wedge \rho' \sqsubseteq \rho_e \\
[\textit{PV-Ref}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \{\overrightarrow{str_i : v_i}\} : \hat{v} \text{ iff } \{\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{C},\rho}\} \sqsubseteq \hat{v}
\end{array}
$$

$$
\begin{array}{ll}
[\mathit{PE\text{-}Val}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} v : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} v : \hat{v} \\
[\mathit{PE\text{-}Let}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \hat{v}_1 \sqsubseteq \hat{\Gamma}(x) \wedge \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \hat{v}_2 \sqsubseteq \hat{v} \wedge \rho_2 \sqsubseteq \rho \\
[\mathit{PE\text{-}App}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1\, e_2 : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad \rho_2 \sqsubseteq \rho \wedge \\
& \quad \forall \lambda x^{\rho_e} \in \hat{v}_1 : \\
& \quad\quad \hat{v}_2 \subseteq \hat{\Gamma}(x) \wedge \\
& \quad\quad \hat{\Gamma}(\lambda x) \sqsubseteq \hat{v} \wedge \\
& \quad\quad \rho_e \sqsubseteq \rho \\
[\mathit{PE\text{-}Seq}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1; e_2 : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_2 \sqsubseteq \rho \wedge \\
& \quad \hat{v}_2 \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}Op}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} op(\overrightarrow{e_i}) : \hat{v} \gg \rho \text{ iff} \\
& \quad \forall i : \\
& \quad\quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_i : \hat{v}_i \gg \rho_i \wedge \\
& \quad\quad \rho_i \sqsubseteq \rho \wedge \\
& \quad \widehat{op}(\overrightarrow{\hat{v}_i}) \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}Cond}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{if}\ (e_0)\ \{\ e_1\ \}\ \mathbf{else}\ \{\ e_2\ \} : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \wedge \\
& \quad \rho_0 \sqsubseteq \rho \wedge \\
& \quad \mathbf{true} \in \hat{v}_0 \Rightarrow \\
& \quad\quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \hat{v}_1 \sqsubseteq \hat{v} \wedge \rho_1 \sqsubseteq \rho \wedge \\
& \quad \mathbf{false} \in \hat{v}_0 \Rightarrow \\
& \quad\quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \hat{v}_2 \sqsubseteq \hat{v} \wedge \rho_2 \sqsubseteq \rho \\
[\mathit{PE\text{-}While}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{while}\ (e_1)\ \{\ e_2\ \} : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad \mathbf{true} \in \hat{v}_1 \Rightarrow \\
& \quad\quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \rho_2 \sqsubseteq \rho \wedge \\
& \quad \widehat{\mathbf{false}} \in \hat{v}_1 \Rightarrow \\
& \quad\quad \widehat{\mathbf{undefined}} \sqsubseteq \hat{v}
\end{array}
$$

$$
\begin{array}{ll}
[\mathit{PE\text{-}GetField}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1[e_2] : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_2 \sqsubseteq \rho \wedge \\
& \quad \widehat{get}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}SetField}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_0[e_1] = e_2 : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_0 : \hat{v}_0 \gg \rho_0 \wedge \\
& \quad \rho_0 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_2 \sqsubseteq \rho \wedge \\
& \quad \widehat{set}(\hat{v}_0, \hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}DelField}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{delete}\ e_1[e_2] : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_2 \sqsubseteq \rho \wedge \\
& \quad \widehat{del}(\hat{v}_1, \hat{v}_2) \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}Ref}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{ref}_r\ e : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad \hat{v}_1 \sqsubseteq \hat{\mu}(r, \rho_s) \wedge \\
& \quad r \in \hat{v} \\
[\mathit{PE\text{-}DeRef}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{deref}\ e : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad \forall r \in \hat{v}_1 : \hat{\mu}(r, \rho_s) \sqsubseteq \hat{v} \\
[\mathit{PE\text{-}SetRef}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_1 = e_2 : \hat{v} \gg \rho \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e_2 : \hat{v}_2 \gg \rho_2 \wedge \rho_2 \sqsubseteq \rho \wedge \\
& \quad \hat{v}_2 \sqsubseteq \hat{v} \wedge \\
& \quad \forall r \in \hat{v}_1 : \hat{v}_2 \sqsubseteq \hat{\mu}(r, \rho_s) \\
[\mathit{PE\text{-}Send}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \overline{e_1}\langle e_2 \triangleright \rho \rangle : \hat{v}_0 \gg \rho_0 \text{ iff} \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}_1 \gg \rho_1 \wedge \\
& \quad \rho_1 \sqsubseteq \rho_0 \wedge \\
& \quad (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} e : \hat{v}_2 \gg \rho_2 \wedge \\
& \quad \rho_2 \sqsubseteq \rho_0 \wedge \\
& \quad \forall m \in \hat{v}_1 : \forall \rho_m \sqsupseteq \rho : \\
& \quad\quad \hat{\Upsilon}(m, \rho_m) = (\rho_r, \rho_e) \wedge \\
& \quad\quad \rho_r \sqsubseteq \rho_s \Rightarrow \rho_e \sqsubseteq \rho_0 \wedge \\
& \quad\quad \hat{v}_2 \sqsubseteq \hat{\Psi}(m, \rho_m) \wedge \\
& \quad\quad \mathbf{unit} \in \hat{v}
\end{array}
$$

$$
\begin{array}{ll}
[\mathit{PE\text{-}Exercise}] & (\hat{\Gamma}, \hat{\mu}) \models_{\rho_s} \mathbf{exercise}(\rho) : \hat{v}_1 \\
& \quad \rho \sqsubseteq \rho_s \Rightarrow \rho \sqsubseteq \rho_1 \wedge \\
& \quad \mathbf{unit} \in \hat{v}
\end{array}
$$

4.1.2 Compositional Verbose

$$
\begin{array}{ll}
[\mathit{CV\text{-}Val}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (c)^{\ell} \text{ iff } \{d_c\} \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}Var}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (x)^{\ell} \text{ iff } \hat{\Gamma}(x) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}Lambda}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (\lambda x.{e_0}^{\ell_0})^{\ell} \text{ iff} \\
& \quad \{\lambda x.{e_0}^{\ell_0}\} \subseteq \hat{C}(\ell) \wedge \\
& \quad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_0}^{\ell_0} \\
[\mathit{CV\text{-}Obj}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (\{\overrightarrow{str_i : {e_i}^{\ell_i}}\})^{\ell} \text{ iff} \\
& \quad \forall i : \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_i}^{\ell_i} \wedge \\
& \qquad \hat{P}(\ell_i) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \{\overrightarrow{str_i : \hat{C}(\ell_i)}\} \subseteq \hat{C}_{\ell} \\
[\mathit{CV\text{-}Let}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (\textbf{let } \overrightarrow{x_i = {e_i}^{\ell_i}} \textbf{ in } {e'}^{\ell'})^{\ell} \text{ iff} \\
& \quad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e'}^{\ell'} \wedge \\
& \quad \hat{P}(\ell') \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \hat{C}(\ell') \subseteq \hat{C}(\ell) \wedge \\
& \quad \forall i : \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_i}^{\ell_i} \wedge \\
& \qquad \hat{C}(\ell_i) \subseteq \hat{\Gamma}(x_i) \wedge \\
& \qquad \hat{P}(\ell_i) \sqsubseteq \hat{P}(\ell) \\
[\mathit{CV\text{-}App}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} ({e_1}^{\ell_1}\, {e_2}^{\ell_2})^{\ell} \text{ iff} \\
& \quad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \\
& \quad \forall (\lambda x.{e_0}^{\ell_0}) \in \hat{C}(\ell_1) : \\
& \qquad \hat{C}(\ell_2) \subseteq \hat{\Gamma}(x) \wedge \hat{C}(\ell_0) \subseteq \hat{C}(\ell) \wedge \\
& \qquad \hat{P}(\ell_0) \sqsubseteq \hat{P}(\ell) \\
[\mathit{CV\text{-}Op}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (op(\overrightarrow{{e_i}^{\ell_i}}))^{\ell} \text{ iff} \\
& \quad \forall i : \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_i}^{\ell_i} \wedge \\
& \qquad \hat{P}(\ell_i) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{op}(\hat{C}(\ell_i)) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}Cond}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (\textbf{if } ({e_0}^{\ell_0}) \; \{ \; {e_1}^{\ell_1} \; \} \textbf{ else } \{ \; {e_2}^{\ell_2} \; \})^{\ell} \text{ iff} \\
& \quad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_0}^{\ell_0} \wedge \\
& \quad \hat{P}(\ell_0) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{\textbf{true}} \in \hat{C}(\ell_0) \Rightarrow \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \hat{C}(\ell_1) \subseteq \hat{C}(\ell) \wedge \\
& \qquad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{\textbf{false}} \in \hat{C}(\ell_0) \Rightarrow \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \hat{C}(\ell_2) \subseteq \hat{C}(\ell) \wedge \\
& \qquad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \\
[\mathit{CV\text{-}While}] & (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} (\textbf{while } ({e_1}^{\ell_1}) \; \{ \; {e_2}^{\ell_2} \; \})^{\ell} \text{ iff} \\
& \quad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{\textbf{true}} \in \hat{C}(\ell_1) \Rightarrow \\
& \qquad (\hat{C}, \hat{\Gamma}, \hat{\mu}, \hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \hat{C}(\ell_2) \subseteq \hat{C}(\ell) \wedge \\
& \qquad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{\textbf{false}} \in \hat{C}(\ell_1) \Rightarrow \widehat{\textbf{undefined}} \subseteq \hat{C}(\ell)
\end{array}
$$

$$
\begin{array}{ll}
[\mathit{CV\text{-}GetField}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} ({e_1}^{\ell_1}[{e_2}^{\ell_2}])^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \\
& \quad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{get}(\hat{C}(\ell_1),\hat{C}(\ell_2)) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}SetField}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} ({e_0}^{\ell_0}[{e_1}^{\ell_1}] = {e_2}^{\ell_2})^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_0}^{\ell_0} \wedge \\
& \quad \hat{P}(\ell_0) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \\
& \quad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{set}(\hat{C}(\ell_0),\hat{C}(\ell_1),\hat{C}(\ell_2)) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}DelField}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} (\mathbf{delete}\ {e_1}^{\ell_1}[{e_2}^{\ell_2}])^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \\
& \quad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \widehat{del}(\hat{C}(\ell_1),\hat{C}(\ell_2)) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}Ref}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} (\mathbf{ref}_{r,\rho_r}\ {e_1}^{\ell_1})^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \{r\} \subseteq \hat{C}(\ell) \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \rho_r \sqsubseteq \rho_s \Rightarrow \hat{C}(\ell_1) \subseteq \hat{\mu}(r,\rho_r) \\
[\mathit{CV\text{-}DeRef}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} (\mathbf{deref}\ {e_1}^{\ell_1})^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \forall r \in \hat{C}(\ell_1) : \forall \rho_r \sqsubseteq \rho_s : \\
& \quad\quad \hat{\mu}(r,\rho_r) \subseteq \hat{C}(\ell) \\
[\mathit{CV\text{-}SetRef}] & (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} ({e_1}^{\ell_1} = {e_2}^{\ell_2})^{\ell} \text{ iff} \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_1}^{\ell_1} \wedge \\
& \quad \hat{P}(\ell_1) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad (\hat{C},\hat{\Gamma},\hat{\mu},\hat{P}) \models_{c\rho_s} {e_2}^{\ell_2} \wedge \\
& \quad \hat{P}(\ell_2) \sqsubseteq \hat{P}(\ell) \wedge \\
& \quad \forall r \in \hat{C}(\ell_1) : \forall \rho_r \sqsubseteq \rho_s : \\
& \quad\quad \hat{C}(\ell_2) \subseteq \hat{\mu}(r,\rho_r) \wedge \\
& \quad \hat{C}(\ell_2) \subseteq \hat{C}(\ell) \\
[\mathit{PE\text{-}Send}] & \ldots \\
[\mathit{PE\text{-}Err}] & \ldots \\
[\mathit{PE\text{-}Exercise}] & \ldots
\end{array}
$$

4.2 Constraint generation

Constraint elements: E.

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

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

Constraint form.

$$
\begin{array}{lrcl}
\textit{Term inclusion} & \{\hat{v}\} & \subseteq & \mathsf{E} \\
\textit{Element inclusion} & \mathsf{E} & \subseteq & \mathsf{E} \\
\textit{Permission inclusion} & \mathsf{P}(\ell) & \sqsubseteq & \mathsf{P}(\ell') \\
\textit{Operation} & \widehat{Op}(\overrightarrow{\mathsf{E}_i}) & \subseteq & \mathsf{E} \\
\textit{Implication} & \{\hat{v}\} \subseteq \mathsf{E} & \Rightarrow & \mathsf{E} \subseteq \mathsf{E}
\end{array}
$$

Misc:

$$
\begin{array}{ll}
r_* & \text{is the set of all references of the program;} \\
lambda_* & \text{is the set of all lambdas of the program;}
\end{array}
$$

$$
\begin{array}{ll}
[CG\text{-}Val] & \mathcal{C}_{*\rho_s}[\![(c)^{\ell}]\!] = \{d_c\} \subseteq \mathsf{C}(\ell) \\
[CG\text{-}Var] & \mathcal{C}_{*\rho_s}[\![(x)^{\ell}]\!] = \Gamma(x) \subseteq \mathsf{C}(\ell) \\
[CG\text{-}Lambda] & \mathcal{C}_{*\rho_s}[\![(\lambda x.{e_0}^{\ell_0})^{\ell}]\!] = \\
& \quad \{\{\lambda x.{e_0}^{\ell_0}\} \subseteq \mathsf{C}(\ell)\} \cup \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_0}^{\ell_0})]\!] \\
[CG\text{-}Obj] & \mathcal{C}_{*\rho_s}[\![(\{\overrightarrow{str_i : {e_i}^{\ell_i}}\})^{\ell}]\!] = \\
& \quad \bigcup_i (\mathcal{C}_{*\rho_s}[\![({e_i}^{\ell_i})]\!] \cup \\
& \quad\quad \{\mathsf{P}(\ell_i) \sqsubseteq \mathsf{P}(\ell)\}) \cup \\
& \quad \{\{\overrightarrow{str_i : \mathsf{C}(\ell_i)}\} \subseteq \mathsf{C}(\ell)\} \\
[CG\text{-}Let] & \mathcal{C}_{*\rho_s}[\![(\textbf{let } \overrightarrow{x_i = {e_i}^{\ell_i}} \textbf{ in } {e'}^{\ell'})^{\ell}]\!] = \\
& \quad \bigcup_i (\mathcal{C}_{*\rho_s}[\![({e_i}^{\ell_i})]\!] \cup \\
& \quad\quad \{\mathsf{C}(\ell_i) \subseteq \Gamma(x_i)\} \cup \\
& \quad\quad \{\mathsf{P}(\ell_i) \subseteq \mathsf{P}(\ell)\}) \cup \\
& \quad \mathcal{C}_{*\rho_s}[\![({e'}^{\ell'})]\!] \cup \\
& \quad \{\mathsf{P}(\ell') \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{C}(\ell') \subseteq \mathsf{C}(\ell)\} \\
[CG\text{-}App] & \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1}\, {e_2}^{\ell_2})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \{\mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_2) \subseteq \Gamma(x) \\
& \quad\quad | t = (\lambda x.{e_0}^{\ell_0}) \in lambda_*\} \cup \\
& \quad \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_0) \subseteq \mathsf{C}(\ell) \\
& \quad\quad | t = (\lambda x.{e_0}^{\ell_0}) \in lambda_*\} \cup \\
& \quad \{\{t\} \subseteq \mathsf{C}(\ell_1) \Rightarrow \mathsf{P}(\ell_0) \sqsubseteq \mathsf{P}(\ell) \\
& \quad\quad | t = (\lambda x.{e_0}^{\ell_0}) \in lambda_*\} \cup \\
[CG\text{-}Op] & \mathcal{C}_{*\rho_s}[\![(op(\overrightarrow{{e_i}^{\ell_i}}))^{\ell}]\!] = \\
& \quad \bigcup_i (\mathcal{C}_{*\rho_s}[\![({e_i}^{\ell_i})]\!] \cup \{\mathsf{P}(\ell_i) \sqsubseteq \mathsf{P}(\ell)\}) \cup \\
& \quad \{\widehat{op}(\mathsf{C}(\ell_i)) \subseteq \mathsf{C}(\ell)\} \\
[CG\text{-}Cond] & \mathcal{C}_{*\rho_s}[\![(\textbf{if } ({e_0}^{\ell_0}) \ \{\ {e_1}^{\ell_1}\ \} \textbf{ else } \{\ {e_2}^{\ell_2}\ \})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_0}^{\ell_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\hat{P}(\ell_0) \sqsubseteq \hat{P}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{true}} \in \mathsf{C}(\ell_0) \Rightarrow \mathsf{C}(\ell_1) \subseteq \mathsf{C}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{true}} \in \mathsf{C}(\ell_0) \Rightarrow \mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{false}} \in \mathsf{C}(\ell_0) \Rightarrow \mathsf{C}(\ell_2) \subseteq \mathsf{C}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{false}} \in \mathsf{C}(\ell_0) \Rightarrow \mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \\
[CG\text{-}While] & \mathcal{C}_{*\rho_s}[\![(\textbf{while } ({e_1}^{\ell_1}) \ \{\ {e_2}^{\ell_2}\ \})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{true}} \in \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_2) \subseteq \mathsf{C}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{true}} \in \mathsf{C}(\ell_1) \Rightarrow \mathsf{P}(\ell_2) \subseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\widehat{\textbf{false}} \in \mathsf{C}(\ell_1) \Rightarrow \widehat{\textbf{undefined}} \subseteq \mathsf{C}(\ell)\}
\end{array}
$$

$$
\begin{array}{ll}
[\mathit{CG\text{-}GetField}] & \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1}[{e_2}^{\ell_2}])^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \widehat{get}(\mathsf{C}(\ell_1), \mathsf{C}(\ell_2)) \subseteq \mathsf{C}(\ell) \\
[\mathit{CG\text{-}SetField}] & \mathcal{C}_{*\rho_s}[\![({e_0}^{\ell_0}[{e_1}^{\ell_1}] = {e_2}^{\ell_2})]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_0}^{\ell_0})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})^{\ell}]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_3) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \widehat{set}(\mathsf{C}(\ell_1), \mathsf{C}(\ell_2), \mathsf{C}(\ell_2)) \subseteq \mathsf{C}(\ell) \\
[\mathit{CG\text{-}DelField}] & \mathcal{C}_{*\rho_s}[\![(\mathbf{delete}\ {e_1}^{\ell_1}[{e_2}^{\ell_2}])^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \widehat{del}(\mathsf{C}(\ell_1), \mathsf{C}(\ell_2)) \subseteq \mathsf{C}(\ell) \\
[\mathit{CG\text{-}Ref}] & \mathcal{C}_{*\rho_s}[\![(\mathbf{ref}_{r,\rho_r}\ {e_1}^{\ell_1})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \\
& \quad \{\{r\} \subseteq \mathsf{C}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\rho_r \sqsubseteq \rho_s \Rightarrow \mathsf{C}(\ell_1) \subseteq \mathsf{M}(r, \rho_r)\} \\
[\mathit{CG\text{-}DeRef}] & \mathcal{C}_{*\rho_s}[\![(\mathbf{deref}\ {e_1}^{\ell_1})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{r \in \mathsf{C}(\ell_1) \Rightarrow \mathsf{M}(r, \rho_r) \subseteq \mathsf{C}(\ell) \\
& \quad\quad \mid r \in r_*, \rho_r \sqsubseteq \rho_s\} \\
[\mathit{CG\text{-}SetRef}] & \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1} = {e_2}^{\ell_2})^{\ell}]\!] = \\
& \quad \mathcal{C}_{*\rho_s}[\![({e_1}^{\ell_1})]\!] \cup \mathcal{C}_{*\rho_s}[\![({e_2}^{\ell_2})]\!] \cup \\
& \quad \{\mathsf{P}(\ell_1) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{\mathsf{P}(\ell_2) \sqsubseteq \mathsf{P}(\ell)\} \cup \\
& \quad \{r \in \mathsf{C}(\ell_1) \Rightarrow \mathsf{C}(\ell_2) \subseteq \mathsf{M}(r, \rho_r) \\
& \quad\quad \mid r \in r_*, \rho_r \sqsubseteq \rho_s\} \cup \\
& \quad \{\mathsf{C}(\ell_2) \subseteq \mathsf{C}(\ell)\} \\
[\mathit{PE\text{-}Send}] & \ldots \\
[\mathit{PE\text{-}Err}] & \ldots \\
[\mathit{PE\text{-}Exercise}] & \ldots
\end{array}
$$

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.
$\gamma_p : \widehat{PV} \rightarrow P(V)$
$\gamma(\hat{v}) = \bigcup_{\widehat{u}_i \in \hat{v}} \gamma_p(\widehat{u}_i)$

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

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

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

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

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

$\widehat{u_{string}} = \{\overrightarrow{\widehat{pre_{string}}}\}$ with $\sqsubseteq = \subseteq$

— Giulia's spec. is more tricky than $\subseteq$

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

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

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

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

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

$\widehat{u_{rec}} = \widehat{pre_{rec}}$ 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

Chapter 5

Experiments

5.1 Findings

share me not

5.2 Performance

SLOW... Very SLOW!!! =¿ Lazy [10, 11]

Chapter 6

Conclusion

6.1 Conclusions

6.2 Future works (unbundling)

[12]

References

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

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

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

[11] 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.

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