Fine-Grained Detection of Privilege Escalation Attacks on Browser Extensions

Stefano Calzavara[1], Michele Bugliesi[1], Silvia Crafa[2], and Enrico Steffinlongo[1]

[1] Università Ca' Foscari Venezia
[2] University of Padova

Abstract. Even though their architecture relies on robust security principles, it is well-known that poor programming practices may expose browser extensions to serious security flaws, leading to privilege escalations by untrusted web pages or compromised extension components. We propose a formal security analysis of browser extensions in terms of a fine-grained characterization of the privileges that an active opponent may escalate through the message passing interface and we discuss to which extent current programming practices take this threat into account. Our theory builds on a formal language that embodies the essential features of JavaScript, together with few additional constructs dealing with the security aspects specific to the browser extension architecture. We then present a flow logic specification estimating the safety of browser extensions modelled in our language against the threats of privilege escalation and we prove its soundness. Finally, we show the feasibility of our approach by means of Chen, a prototype static analyser for Google Chrome extensions based on our flow logic specification.

1 Introduction

Browser extensions customize and enhance the functionalities of standard web browsers by intercepting and reacting to a number of events triggered by navigation, page rendering or updates to specific browser data structures. While many extensions are simple and just installed to customize the navigation experience, other extensions serve security-critical tasks and have access to powerful APIs, providing access to the download manager, the cookie jar, or the navigation history of the user. Hence, the security of the web browser (and the assets stored therein) ultimately hinges on the security of the installed browser extensions. Just like browsers, extensions typically interact with untrusted and potentially malicious web pages: thus, all modern browser extension architectures rely on robust security principles, such as *privilege separation* [31].

Browser Extension Architecture. Privilege separated software architectures require programmers to structure their code in separated modules, running with different privileges. In the realm of browser extensions, privilege separation is implemented by structuring the extension in two different types of components: a privileged *background page*, which has access to the browser APIs and runs

J. Vitek (Ed.): ESOP 2015, LNCS 9032, pp. 510–534, 2015.
DOI: 10.1007/978-3-662-46669-8_21

isolated from web pages; and a set of unprivileged *content scripts*, which are injected into specific web pages, interact with them and are at a higher risk of attacks [4,10]. The permissions available to the background page are defined at installation time in a manifest file, to limit the dangers connected to the compromise of the background page. Content scripts interacting with different web pages are isolated one from each other by the same-origin policy of the browser, while process isolation protects the background page. The message passing interface available to extensions only allows the exchange of serialized JSON objects[1] between different components, hence pointers cannot cross trust boundaries.

Language Support for Privilege Separation. We are interested here in understanding to which extent current browser extension development frameworks, such as the Google Chrome extension APIs, naturally support privilege separation and comply with the underlying security architecture. Worryingly, we notice that in these frameworks a single privileged module typically offers a unified entry point to security-sensitive functionalities to all the other extension components, even though not all the components need to access the same functionalities and different trust relationships exist between different components.

To make matters worse, current programming patterns adopted in browser extensions do not safeguard the programmer against *compromised* components, even though the underlying privilege separated architecture was designed with compromise in mind. Compromise adds another layer of complexity to security-aware extension development, since corrupted extension components may get access to surprisingly powerful privileges.

1.1 Motivating Example

We illustrate our argument with a simple, but realistic example, inspired by one of the many cookie managers available in the Chrome Web Store (e.g., `EditThisCookie`). Consider an extension which allows users to add, delete or modify any cookie stored in the browser through an intuitive user interface. Additionally, it allows web pages to specify a set of security policies for the cookies they register: these client-side security policies are enforced by the extension and can be used to significantly strengthen web authentication [6,7].

The extension is composed of three components: two content scripts C and O, and a background page B. The background page is given the `cookies` permission, which grants it access to the browser cookie jar. The content script O is injected in the `options.html` page packaged with the extension and it provides facilities for cookie editing; when the user is done with his changes, O sends B a message and instructs it to update the cookie jar. The content script C, instead, is injected in the DOM of any HTTPS web page P opened by the browser: it is essentially a proxy, which forwards to B the security policies specified by P using the message passing interface. The messages sent by P are extended by C with an additional information: the website which specified the security policy.

[1] `http://json.org`

A possible run involving the described components is the following, where the last message is triggered by a user click:

$P \to C$: {tag: "policy", spec: "read-only"}
$C \to B$: {tag: "policy", site: "paypal.com", spec: "read-only"}
$O \to B$: {tag: "upd", ck: {dom: "a.com", name: "res", value: "1440x900"}}

Using the Google Chrome extension API, the components are programmed in JavaScript, typically by registering appropriate listeners for incoming messages. For instance, the content script C can be programmed as follows:

```
window.addEventListener("message", function(event) {
  /* Accept only internal messages */
  if (event.source != window) { return; }
  /* Get the payload of the message */
  var obj = event.data;
  /* Extend the message with the site and forward it */
  obj.site = window.location.hostname;
  chrome.runtime.sendMessage (obj);
}, false);
```

Web pages can communicate with C by using the `window.postMessage` method available in JavaScript, thus opting-in to custom client-side protection.

The background page B, instead, is typically programmed as follows:

```
chrome.runtime.onMessage.addListener(
  function (msg, sender, sendResp) {
    /* Handle the reception of new policies */
    if (msg.tag == "policy") {
      /* Store a new (valid) policy for the site */
      if (is_valid (msg.spec))
        localStorage.setItem (msg.site, msg.spec);
      else console.log ("Invalid policy");
    }
    /* Handle requests for cookie updates */
    else if (msg.tag == "upd") {
      chrome.cookies.set (msg.ck);
    }
    else console.log ("Invalid message");
});
```

This tag-based coding style featuring a single entry point to the background page is very popular, since it is easy to grasp and allows for fast prototyping, but it also fools programmers into underestimating the attack surface against the extensions they write. In this example, a malicious web page can compromise the integrity of the cookie jar by exploiting the poorly programmed content script C through the following method invocation:

```
window.postMessage ({tag: "upd", ck: {dom: "google.com",
                     name: "SID", value: "aQe73ajs..."}});
```

This allows the web page to carry out dangerous attacks, like session fixation or login CSRF on arbitrary websites [7]. The issue can be rectified by including a *sanitization* in the code of C and by ensuring that only messages with the `"policy"` tag are delivered to the background page.

The revised code is more robust than the original one and it safeguards the extension against the threats posed by malicious (or compromised) web pages. Unfortunately, it does not yet protect the background page against a compromised content script: if an attacker is able to exploit a code injection vulnerability in C, he may force the content script into deviating from the intended communication protocol. Specifically, an attacker with scripting capabilities in C may forge arbitrary messages to the background page and taint the cookie jar.

A much more robust solution then consists in introducing two distinct communication ports for C and O, and dedicating these ports to the reception of the two different message types (see Section 5). This is relatively easy to do in this simple example, but, in general, decoupling the functionalities available to the background page to shield it against privilege escalation is complex, since n different content scripts or extensions may require access to m different, possibly overlapping sets of privileged functionalities.

1.2 Contributions

Our contributions can be summarized as follows:

1. we model browser extensions in a formal language that embodies the essential features of JavaScript, together with a few additional constructs dealing with the security aspects specific to the browser extension architecture;
2. we formalize a fine-grained characterization of the privileges which can be escalated by an active opponent through the message passing interface, assuming the compromise of some untrusted extension components;
3. we propose a flow logic specification estimating the safety of browser extensions against the threats of privilege escalation and we prove its soundness, despite the best efforts of an active opponent. We show how the static analysis works on the example above and supports its secure refactoring;
4. we present CHEN (CHrome Extension aNalyser), a prototype tool that implements our flow logic specification, providing an automated security analysis of existing Google Chrome extensions. The tool opens the way to an automatic security-oriented refactoring of existing extensions. We show CHEN at work on ShareMeNot [30], a real extension for Google Chrome, and we discuss how the tool spots potentially dangerous programming practices.

2 Related Work

Browser Extension Security. Carlini *et al.* performed a security evaluation of the Google Chrome extension architecture by means of a manual review of 100 popular extensions [10]. Liu *et al.* further analysed the Google Chrome extension architecture, highlighting that it is inadequate to provide protection against

malicious extensions [21]. Guha *et al.* [15] proposed a methodology to write provably secure browser extensions, based on refinement typing; the approach requires extensions to be coded in Fine, a dependently-typed ML dialect. Karim *et al.* developed Beacon, a static detector of capability leaks for Firefox extensions [20]. A capability leak happens when a component exports a pointer to a privileged piece of code. These leaks violate the desired modularity of Firefox extensions, but they cannot be directly exploited by content scripts, since the message passing interface prevents the exchange of pointers. Finally, information flow control frameworks have been proposed for browser extensions [13,3].

Privilege Escalation Attacks. Privilege escalation attacks have been extensively studied in the context of Android applications, starting with [12,29]. Fragkaki *et al.* formalized protection against privilege escalation in Android applications as a noninterference property, which is then enforced by a dynamic reference monitor [14]. Bugliesi *et al.* presented a stronger security notion and discussed a static type system for Android applications, which provably enforces protection against privilege escalation [8]. The present paper generalizes both these proposals, by providing a fine-grained view of the privileges leaked to an arbitrarily powerful opponent. Akhawe *et al.* [2] pointed out severe limitations in how privilege separation is implemented in browser extension architectures. Their work has been very inspiring for the present paper, which provides a formal counterpart to many interesting observations contained therein. For instance, [2] defines *bundling* as the collection of disjoint functionalities inside a single module running with the union of the privileges required by each functionality. Our formal notion of privilege leak captures the real dangers of permission bundling.

Formal Analysis of JavaScript. Maffeis *et al.* formalized the first detailed operational semantics for JavaScript [22] and used it to verify the (in)security of restricted JavaScript subsets [23]. Jensen *et al.* proposed an abstract interpretation framework for JavaScript in the realm of type analysis [18]. Guha *et al.* defined λ_{JS} as a relatively small core calculus based on a few well-understood constructs, where the numerous quirks of JavaScript can be encoded with a reasonable effort [16]. The adequacy of the semantics has been assessed by extensive automatic testing. The calculus has been used to support static analyses to detect type errors in JavaScript [17] and to verify the correctness of JavaScript sandboxing [28]. We also develop our flow analysis on top of λ_{JS}, extending it to reason about browser extension security. An alternate solution would have been to base our work on S5 [27]. This approach would have allowed to analyse browser extensions using ECMA5-specific features, but at the cost of significantly complicating the formal development.

3 Modelling Browser Extensions

Our language embodies the essential features of JavaScript, formalized as in λ_{JS} [16], up to a number of changes needed to deal with the security aspects

specific to the browser extension architecture. In our model, several expressions run in parallel with different permissions and are isolated from each other: communication is based on asynchronous message exchanges.

3.1 Syntax

We assume disjoint sets of channel names $\mathcal{N}$ (a, b, m, n) and variables $\mathcal{V}$ (x, y, z). We let r range over a set of references $\mathcal{R}$, and we assume a lattice of permissions $(\mathcal{P}, \sqsubseteq)$, letting ρ range over $\mathcal{P}$. The syntax of the language is given below:

$$
\begin{array}{lll}
\textit{Constants} & c ::= \textit{num} \mid \textit{str} \mid \textit{bool} \mid \textbf{unit} \mid \textbf{undefined}, \\
\textit{Values} & v ::= n \mid x \mid c \mid r_\ell \mid \lambda x.e \mid \{\overrightarrow{str_i : v_i}\} \\
\textit{Expressions} & e ::= v \mid \textbf{let}\ x = e\ \textbf{in}\ e \mid e\,e \mid \mathit{op}(\overrightarrow{e_i}) \mid \textbf{while}\ (e)\ \{\ e\ \} \\
 & \quad \mid \textbf{if}\ (e)\ \{\ e\ \}\ \textbf{else}\ \{\ e\ \} \mid e; e \mid e[e] \mid e[e] = e \\
 & \quad \mid \textbf{delete}\ e[e] \mid \textbf{ref}_\ell\ e \mid \textbf{deref}\ e \mid e := e \\
 & \quad \mid \overline{e}\langle e \triangleright \rho \rangle \mid \textbf{exercise}(\rho) \\
\textit{Systems} & s ::= \mu; h; i & \textit{Memories} \quad \mu ::= \emptyset \mid \mu, r_\ell \overset{\rho}{\mapsto} v \\
\textit{Handlers} & h ::= \emptyset \mid h, a(x \triangleleft \rho : \rho').e & \textit{Instances} \quad i ::= \emptyset \mid i, a\{\!|e|\!\}_\rho
\end{array}
$$

All the value forms are standard, we just note that references r_ℓ bear a label ℓ, taken from a set of labels $\mathcal{L}$. Labels identify the program point where references are created: this is needed for the static analysis and plays no role in the semantics. As usual, the lambda abstraction $\lambda x.e$ binds x in e.

As to expressions, the first three lines correspond to standard constructs inherited from λ_{JS}, including function applications, basic control-flow operators, and the usual operations on records (field selection, field update/creation, field deletion) and references (allocation, dereference and update). As anticipated, reference allocation comes with an annotation ℓ. We leave unspecified the precise set of primitive operations op. The expression $\textbf{let}\ x = e\ \textbf{in}\ e'$ binds x in e'.

The last line of the productions includes the new constructs added to λ_{JS}. The expression $\overline{a}\langle v \triangleright \rho \rangle$ sends the value v on channel a. In order for the sender to protect the message, the expression specifies that the value can be received by any *handler* with at least permission ρ that is listening on a. The expression $\textbf{exercise}(\rho)$ exercises the privilege ρ. This construct uniformly abstracts any security-sensitive operation, such as the call to a privileged API, which requires the permission ρ to successfully complete the task.

We let h range over multisets of *handlers* of the form $a(x \triangleleft \rho : \rho').e$. The handler $a(x \triangleleft \rho : \rho').e$ listens for messages on the channel a. When a value v is sent over a, a new *instance* of the handler is spawned to run the expression e with permission ρ', with the bound variable x replaced by v. The handler protects its body against untrusted senders by specifying that only instances with permission ρ can be granted access. Intuitively, the body of a handler corresponds to the function passed as a parameter to the `addListener` method of `chrome.runtime.onMessage`. Different handlers can listen on the same channel: in this case, only one handler is non-deterministically dispatched. We often refer to a handler with the name of the channel where it is registered.

Table 1. Small-step operational semantics of systems ($s \xrightarrow{\alpha} s'$)

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

$$\text{(R-Set)}\quad \frac{\mu; h; i \xrightarrow{\alpha} \mu'; h'; i'}{\mu; h; i, i'' \xrightarrow{\alpha} \mu'; h'; i', i''}$$

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

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

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

We let i range over multisets of running *instances* of the form $a\{\!|e|\!\}_\rho$. The instance $a\{\!|e|\!\}_\rho$ is a running expression e, which is granted permission ρ. The instance is annotated with the channel name a corresponding to the handler which spawned it.

We let μ range on *memories*, i.e., sets of bindings of the form $r_\ell \overset{\rho}{\mapsto} v$. A memory is a partial map from (labelled) references to values. The annotation ρ on the arrow records the permission of the instance that created the reference, and at the same time tracks the permissions required to have read/write access on the reference. Given a memory μ, we let $dom(\mu) = \{r \mid r_\ell \overset{\rho}{\mapsto} v \in \mu\}$.

Finally, a *system* is defined as a triple $s = \mu; h; i$. Intuitively, a system evolves by letting running instances (i) communicate through the memory μ when they are granted exactly the same permissions, (*ii*) spawn new instances by sending messages to handlers in h, and (*iii*) perform internal computations.

3.2 Semantics

The small-step operational semantics of the calculus is defined in terms of a labelled reduction relation between systems $s \xrightarrow{\alpha} s'$. Labels play no role in the semantics of systems: they are just used to track useful information that is needed in the proofs. 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 label $a{:}\rho_a \gg \rho$ records the exercise of the privilege ρ by an instance a running with permissions ρ_a. The send label $\langle a{:}\rho_a, b{:}\rho_b\rangle$ records that an instance a with permissions ρ_a is sending a message to a handler b with permissions ρ_b. Finally, the empty label $\cdot$ tracks no information. We denote traces by $\overrightarrow{\alpha}$ and we write $\overset{\overrightarrow{\alpha}}{\Longrightarrow}$ for the reflexive-transitive closure of $\xrightarrow{\alpha}$. Table 1 collects the reduction rules for systems and the definition of evaluation contexts. We write $E\langle e\rangle$ when the hole $\bullet$ in E is filled with the expression e.

Table 2. Small-step operational semantics of expressions $(\mu; e \hookrightarrow_\rho \mu'; e')$

$$\frac{e_1 \hookrightarrow e_2}{\mu; e_1 \hookrightarrow_\rho \mu; e_2}\ (\text{JS-Expr}) \qquad \frac{r \notin dom(\mu) \qquad \mu' = \mu, r_\ell \overset{\rho}{\mapsto} v}{\mu; \mathbf{ref}_\ell\ v \hookrightarrow_\rho \mu'; r_\ell}\ (\text{JS-Ref}) \qquad \frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v}{\mu; \mathbf{deref}\ r_\ell \hookrightarrow_\rho \mu; v}\ (\text{JS-Deref})$$

$$\frac{\mu = \mu', r_\ell \overset{\rho}{\mapsto} v'}{\mu; r_\ell := v \hookrightarrow_\rho \mu', r_\ell \overset{\rho}{\mapsto} v; v}\ (\text{JS-SetRef}) \qquad \frac{\mu; e_1 \hookrightarrow_\rho \mu'; e_2}{\mu; E\langle e_1 \rangle \hookrightarrow_\rho \mu'; E\langle e_2 \rangle}\ (\text{JS-Context})$$

Rule (R-Sync) implements a security cross-check between the sender a and the receiver b: by specifying a permission ρ_r on the send expression, the instance a requires the handler b to have at least ρ_r, while by specifying a permission ρ_s in its definition, the handler b requires the instance a to have at least ρ_s. If the security check succeeds, a new instance of b is created and the sent value v is substituted to the bound variable x in the body of the handler. Communication is restricted to *serializable* values, according to the following definition.

Definition 1 (Serializable Value). *A value v is* serializable *iff either (1) v is a name n or a constant c; or (2) $v = \{\overrightarrow{str_i : v_i}\}$ and each v_i is serializable.*

This restriction is consistent with the browser extension security architecture, which prevents the exchange of pointers between different components [10].

Rule (R-Exercise) reduces the expression $\mathbf{exercise}(\rho)$. Reduction takes place only when the expression runs in an instance a which is granted permission $\rho_a \sqsupseteq \rho$. Rule (R-Set) allows for reducing any of the parallel instances running in a system, while rule (R-Internal) performs an internal reduction step based on the auxiliary transition relation $\mu; e \hookrightarrow_\rho \mu'; e'$, annotated with the permission ρ granted to the instance. The internal reduction relation is defined in Table 2; it relies on a basic reduction $e \hookrightarrow e'$, which is directly inherited from λ_{JS} and lifted to the internal reduction by rule (JS-Expr). The definition of the basic reduction is standard and given in the full version [9].

A reference is allocated by means of rule (JS-Ref). According to this rule, two references may have the same label (e.g., when reference allocation occurs inside a program loop) but each reference is guaranteed to have a distinct name. Since read/write operations on memory ultimately depend on the reference name, this ensures that labels on references do not play any role at runtime.

Finally, rules (JS-SetRef) and (JS-Deref) deal with reference update and dereference. Observe that, according to these rules, both read and write access to memory requires *exactly* the permission ρ annotated on the reference. In other words, instances with different privileges cannot communicate through the memory. This corresponds to the heap separation policy implemented in modern browser extension architectures.

3.3 Privilege Leak

We now define the notion of *privilege leak*, which dictates an upper bound to the privileges which can be escalated by an opponent when interacting with the system. We start by defining when a system exercises a given permission.

Definition 2 (Exercise). *Given a system s, we say that s* exercises *ρ iff there exist s' and $\overrightarrow{\alpha}$ such that $s \stackrel{\overrightarrow{\alpha}}{\Longrightarrow} s'$ and $a{:}\rho_a \gg \rho \in \{\overrightarrow{\alpha}\}$.*

In our threat model, an opponent can mount an attack against the system by registering new handlers, which may intercept messages sent to trusted components, and/or by spawning new instances, which may tamper with the system by writing in shared memory cells and by using the message passing interface.

Formally, an opponent is defined as a pair (h, i), with an upper bound ρ for the permissions granted to h and i. For technical reasons, we assume that the set of variables $\mathcal{V}$ is partitioned into the sets $\mathcal{V}_t$ and $\mathcal{V}_u$ (trusted and untrusted variables). We stipulate that all the variables occurring in the system are drawn from $\mathcal{V}_t$, while all the variables occurring in the opponent code belong to $\mathcal{V}_u$.

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

- *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 $x \in \mathcal{V}_u$.*

Definition 4 (Privilege Leak). *A (initial) system $s = \mu; h; \emptyset$* leaks *ρ against ρ' (with $\rho \not\sqsubseteq \rho'$) iff, for any ρ'-opponent (h_o, i_o), the system $s' = \mu; h, h_o; i_o$ exercises at most ρ.*

Our security property is given over *initial* systems, that is systems with no running instances, since we are interested in understanding the interplay between the exercised permissions and the communication interface exposed by the handlers in the system. Intuitively, a system s is "more secure" than another system s' if it leaks fewer privileges than s' against any possible ρ.

3.4 Encoding the Example

To illustrate, we encode in our formal language the example in Section 1.1. Consider the system $s = \mu; h_c, h_o, h_b; \emptyset$, where the handlers h_c, h_o and h_b encode the two content scripts and the background page. The memory μ encodes the private memory of the background page, and it is used to store library functions. We grant the background page two different permissions: MemB to access the references under its control and $\mathsf{Cookies}$ to access the cookie jar.

Let $\mathsf{B} = \mathsf{MemB} \sqcup \mathsf{Cookies}$, we let $\mu = lib_\ell \stackrel{\mathsf{B}}{\mapsto} obj$, where:

$$
\begin{aligned}
obj = \{\ & \textit{“set”} : \lambda x.\mathbf{exercise}(\mathsf{Cookies}); \textit{set/update the cookie } x,\\
& \textit{“is\_valid”} : \lambda x.\textit{check validity of policy } x,\\
& \textit{“store”} : \lambda x.\lambda y.\mathbf{exercise}(\mathsf{MemB}); \textit{bind policy } y \textit{ to site } x,\\
& \textit{“log”} : \lambda x.\textit{print message } x\}
\end{aligned}
$$

We omit the internal logic of the functions, we just observe that we put in place the exercise expressions corresponding to the usage of the required privileges. The definition of the handler h_b modelling the background page is given below, where C and O are the permissions granted to the two content scripts in order to let them contact B through the message passing interface.

$$
\begin{array}{l}
h_b \triangleq b(x \triangleleft \mathsf{C} \sqcap \mathsf{O} : \mathsf{B}). \\
\quad \textbf{let } \mathit{mylib} = \textbf{deref } \mathit{lib}_\ell \textbf{ in} \\
\quad \textbf{if } (x[\,\text{“}\mathit{tag}\text{”}\,] == \text{“}\mathit{policy}\text{”}) \; \{ \\
\quad\quad \textbf{if } (\mathit{mylib}[\,\text{“}\mathit{is\_valid}\text{”}\,]\,(x[\,\text{“}\mathit{spec}\text{”}\,])) \; \{ \\
\quad\quad\quad (\mathit{mylib}[\,\text{“}\mathit{store}\text{”}\,]\,(x[\,\text{“}\mathit{site}\text{”}\,]))\,(x[\,\text{“}\mathit{spec}\text{”}\,]) \\
\quad\quad \} \\
\quad\quad \textbf{else } \{\; \mathit{mylib}[\,\text{“}\mathit{log}\text{”}\,]\; \text{“}\mathit{invalid\ policy}\text{”}\; \} \\
\quad \} \\
\quad \textbf{else } \{ \\
\quad\quad \textbf{if } (x[\,\text{“}\mathit{tag}\text{”}\,] == \text{“}\mathit{upd}\text{”}) \; \{\; (\mathit{mylib}[\,\text{“}\mathit{set}\text{”}\,])\,(x[\,\text{“}\mathit{ck}\text{”}\,]) \;\} \\
\quad\quad \textbf{else } \{\; \mathit{mylib}[\,\text{“}\mathit{log}\text{”}\,]\; \text{“}\mathit{invalid\ message}\text{”}\; \} \\
\quad \}
\end{array}
$$

The handler can be accessed by both C and O, as modelled by the guard $\mathsf{C} \sqcap \mathsf{O}$.

A simplified encoding of the content scripts, corresponding to the handlers h_c and h_o respectively, is given below. This simple encoding will be enough to explain the most important aspects of the flow analysis in Section 4.3.

$$
\begin{array}{l}
h_c \triangleq c(y \triangleleft \mathsf{P} : \mathsf{C}).\textbf{let } y' = (y[\,\text{“}\mathit{site}\text{”}\,] = \ldots) \textbf{ in } \overline{b}\langle y' \triangleright \mathsf{B}\rangle \\
h_o \triangleq o(z \triangleleft \top : \mathsf{O}).\textbf{let } z' = \{\; \text{“}\mathit{tag}\text{”} : \text{“}\mathit{upd}\text{”},\;\; \text{“}\mathit{ck}\text{”} : \ldots\} \textbf{ in } \overline{b}\langle z' \triangleright \mathsf{B}\rangle
\end{array}
$$

The only notable point here is that h_o is protected with permission $\top$, since it is injected in the trusted options page of the extension, while h_c is protected with permission P, modelling access to the `window.postMessage` method used to communicate with C from a web page. As a consequence, any P-opponent has the ability to activate h_c through the message passing interface.

Based on the encoding, we estimate the robustness against privilege escalation attacks. It turns out that the system s leaks B against P, since a P-opponent can force h_c into forwarding an arbitrary (up to the choice of the *“site”* field) message to h_b, hence all the privileges available to h_b may be escalated.

Assume then that h_c is replaced by a new handler h'_c, defined as follows:

$$
\begin{array}{l}
h'_c \triangleq c(y \triangleleft \mathsf{P} : \mathsf{C}).\; \textbf{let } y_{new} = \{\; \text{“}\mathit{tag}\text{”} : \text{“}\mathit{policy}\text{”},\; \text{“}\mathit{site}\text{”} : \ldots\} \textbf{ in} \\
\qquad\qquad\qquad\qquad \textbf{let } y' = (y_{new}[\,\text{“}\mathit{spec}\text{”}\,] = y[\,\text{“}\mathit{spec}\text{”}\,]) \textbf{ in } \overline{b}\langle y' \triangleright \mathsf{B}\rangle
\end{array}
$$

The new system $s_{tag} = \mu; h'_c, h_o, h_b; \emptyset$ leaks MemB against P, since a P-opponent can only communicate with h_b through the proxy h'_c, which ensures that only messages tagged with *“policy”* are delivered to the background page and the integrity of the cookie jar is preserved. However, s_{tag} leaks B against C, since a C-opponent can send arbitrary messages to h_b and thus escalate all the available privileges.

3.5 Fixing the Example

The key observation here is that there is no good reason to let C and O share the same entry point to B, since they request distinct functionalities. We can then split the logic of h_b into two different handlers: h_{b_1} protected by permission C, and h_{b_2} protected by permission O.

$b_1(x \triangleleft \mathsf{C} : \mathsf{B})$.
let $mylib =$ **deref** lib_ℓ **in**
if $(x[\text{“tag”}] == \text{“policy”})$ { ... }
else $\{mylib[\text{“log”}]\ \text{“invalid policy”}\}$

$b_2(x \triangleleft \mathsf{O} : \mathsf{B})$.
let $mylib =$ **deref** lib_ℓ **in**
if $(x[\text{“tag”}] == \text{“upd”})$ { ... }
else $\{mylib[\text{“log”}]\ \text{“invalid message”}\}$

Clearly, the code of h_c and h_o must also be changed to communicate on the new channels b_1 and b_2 respectively: call these new handlers $\hat{h}_c$ and $\hat{h}_o$. Now the handler h_{b_1} is only accessible by $\hat{h}_c$, while the handler h_{b_2} can only be accessed by $\hat{h}_o$, hence, if O is not compromised, the integrity of the cookie jar is preserved.

Unfortunately, the current extension architecture does not support a fine-grained assignment of permissions to different portions of the background page [2], hence we are forced to violate the principle of least privilege and assign to both h_{b_1} and h_{b_2} the full set of permissions $\mathsf{B} = \mathsf{MemB} \sqcup \mathsf{Cookies}$ available to the original h_b, even though h_{b_1} and h_{b_2} only require a subset of these permissions. Still, the system $s_{chan} = \mu; \hat{h}_c, \hat{h}_o, h_{b_1}, h_{b_2}; \emptyset$ only leaks MemB against C.

Notice that this refactoring can be performed on existing Google Chrome extensions by using the `chrome.runtime.connect` API for the dynamic creation of communication ports towards the background page.

4 Security Analysis: Flow Logic

To precisely reason about privilege escalation, it is crucial to statically capture the interplay between the format of the exchanged messages and the exercised privileges: we then resort to the flow logic framework [24]. The main judgement of our flow analysis is $\mathcal{E} \Vdash s$ **despite** ρ, meaning that the environment $\mathcal{E}$ represents an acceptable analysis estimate for s, even when s interacts with a ρ-opponent. This implies that any ρ-opponent will at most escalate privileges up to an upper bound which can be immediately computed from $\mathcal{E}$ (see Theorem 1).

4.1 Analysis Specification

Abstract Values. We let $\hat{V}$ stand for the set of abstract values $\hat{v}$, defined as sets of abstract pre-values (we often omit brackets around singletons):

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

Channel names n are abstracted into themselves. The abstract pre-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 at least **true**, **false**, **unit** and **undefined**.

A reference r_ℓ is abstracted into the label ℓ. A function $\lambda x.e$ is abstracted into the simpler representation λx^ρ, keeping track of the privileges ρ exercised by the expression e. The abstract pre-value $\langle\!|\overrightarrow{str_i : v_i}|\!\rangle_{\mathcal{E},\rho}$ is the abstract representation of the concrete record $\{\overrightarrow{str_i : v_i}\}$ in the environment $\mathcal{E}$, assuming that the record is created in a context with permission ρ. We do not fix any apriori abstract representation for records, e.g., both field-sensitive and field-insensitive representations are admissible.

We associate to each concrete operation *op* an abstract counterpart $\widehat{op}$ 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. Finally, we assume that abstract values are ordered by a pre-order $\sqsubseteq$ containing set inclusion, with the intuition that smaller abstract values are more precise (we overload the symbol used to order permissions, to keep the notation lighter). All the abstract operations and the abstract value pre-order can be chosen arbitrarily, as long as they satisfy some relatively mild and well-established conditions needed in the proofs. For instance, we require abstract operations to be monotonic and to soundly over-approximate their concrete counterparts (see the full version [9] for details).

Abstract Environments. The judgements of the analysis are specified relative to an abstract environment $\mathcal{E} = \hat{\Upsilon}; \hat{\Phi}; \hat{\Gamma}; \hat{\mu}$, consisting of the following four components, where $\Lambda = \{\lambda x \mid x \in \mathcal{V}\}$ is used to store the abstract return value for lambdas:

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

Abstract variable environments are standard: they associate abstract values to variables and to functions, corresponding to the abstraction of their return value. Abstract memories are also standard: they associate abstract values to labels denoting references. Specifically, if $\hat{\mu}(\ell, \rho) = \hat{v}$, then $\hat{v}$ is a sound abstraction of any value stored in a reference labelled with ℓ and protected with permission ρ.

Abstract stacks are novel and are central to the privilege escalation analysis. This part of the environment is used to keep track of the permissions required to get access to each handler and the privileges which are exercised (also *transitively*, i.e., by communicating with other components) by the handlers themselves. Specifically, if $\hat{\Upsilon}(a, \rho_a) = (\rho_s, \rho_e)$, then the handler a with permission ρ_a can be accessed by any component with permission ρ_s and it will be able to exercise privileges up to ρ_e, possibly by calling other handlers in the system.

Finally, abstract networks are adapted from flow logic specifications for process calculi [26] and they are used to keep track of the messages sent to the handlers in the system. For instance, if we have $\hat{\Phi}(a, \rho_a) = \hat{v}$, then $\hat{v}$ is a sound

Table 3. Flow analysis for values

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

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

abstraction of any message received by the handler a with permission ρ_a. Given an abstract environment $\mathcal{E}$, we denote by $\mathcal{E}_{\hat{\Gamma}}, \mathcal{E}_{\hat{\mu}}, \mathcal{E}_{\hat{\Upsilon}}, \mathcal{E}_{\hat{\Phi}}$ its four components.

Flow Analysis for Values and Expressions. The flow analysis for values and expressions consists of two mutually inductive judgements: $\mathcal{E} \Vdash_\rho v \rightsquigarrow \hat{v}$ and $\mathcal{E} \Vdash_\rho e : \hat{v} \gg \rho'$. The first judgement means that, assuming permission ρ, the concrete value v is mapped to the abstract value $\hat{v}$ in the abstract environment $\mathcal{E}$. The judgement $\mathcal{E} \Vdash_\rho e : \hat{v} \gg \rho'$ means that in the context of a handler (or an instance) with permission ρ, and under the abstract environment $\mathcal{E}$, the expression e may evaluate to a value abstracted by $\hat{v}$ and exercise at most ρ'.

The rules to derive $\mathcal{E} \Vdash_\rho v \rightsquigarrow \hat{v}$ are collected in Table 3. Most of these rules are straightforward. The only rule worth commenting on here is (PV-Fun), which can be explained as follows: to abstract $\lambda x.e$ into $\hat{v}$, we first analyse the function body e to compute an approximation $\hat{v}'$ of the value it may evaluate to and an upper bound ρ' for the exercised privileges. Then, we check that $\lambda x^{\rho_e} \in \hat{v}$ for some $\rho_e \sqsupseteq \rho'$, i.e., we ensure that the exercised privileges are over-approximated in $\hat{v}$. Finally, we check that $\hat{v}' \sqsubseteq \mathcal{E}_{\hat{\Gamma}}(\lambda x)$, i.e., we guarantee that the abstract variable environment correctly over-approximates the return value of the function.

The analysis rules for expressions are collected in Table 4. We comment on some representative rules below. Rule (PE-Let) can be explained as follows: to analyse **let** $x = e_1$ **in** e_2, we first analyse e_1 to compute an approximation $\hat{v}_1$ of the value it may evaluate to and an upper bound ρ_1 for the exercised privileges. We then ensure that the abstract variable environment $\mathcal{E}_{\hat{\Gamma}}(x)$ contains an over-approximation of $\hat{v}_1$ for the bound variable x, and we analyse e_2 to approximate its value as $\hat{v}_2$ and the exercised privileges as ρ_2. The analysis is acceptable if the abstract value $\hat{v}$ given to the let expression is an over-approximation of $\hat{v}_2$ and the estimated exercised privileges ρ are an upper bound for $\rho_1 \sqcup \rho_2$.

Rule (PE-App) deals with function applications: it states that, to analyse $e_1\, e_2$, we first analyse the e_i's to compute the approximations $\hat{v}_i$ of the value they may evaluate to and the upper bounds ρ_i for the exercised privileges. We then focus on each λx^{ρ_e} contained in $\hat{v}_1$ and we check that: (1) the abstract variable environment binds x to an over-approximation of the abstraction of the actual argument of the function, (2) the abstract value $\hat{v}$ given to the application

Table 4. Flow analysis for expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is an over-approximation of the abstract return value of the function $\mathcal{E}_{\hat{\Gamma}}(\lambda x)$, and (3) the exercised privileges $\rho_1 \sqcup \rho_2 \sqcup \rho_e$ are bounded above by the privileges ρ assigned to the application.

The rules in the central portion of the table should be relatively easy to understand. Notice that the rules for control flow operators, i.e., (PE-Cond) and (PE-While), allow for excluding from the static analysis some program branches which are never reached at runtime. The rules for references use the information ρ_s annotated on the turnstile, corresponding to the privileges granted to the handler/instance that is accessing the reference. These rules ensure that any value stored in a reference is correctly over-approximated by the abstract memory; and dually, that any value retrieved from a reference is abstracted with an over-approximation of the content of the abstract memory. This ensures that any value which is first stored in a reference and then retrieved from it is over-approximated correctly by the flow logic.

Rule (PE-Send) first analyses e_1 and e_2 to compute the approximations of the recipient ($\hat{v}_1$) and the sent message ($\hat{v}_2$). Then, the last premise enforces two invariants: (1) the privileges ρ_e escalated by communicating with other handlers in the system are bounded above by the privileges ρ' assigned to the send expression, and (2) the abstraction of the sent message $\hat{v}_2$ is over-approximated by the information in the abstract network for each possible recipient. We also check that **unit** is included in the abstract value assigned to the expression, accordingly to the operational semantics of the send construct. Finally, rule (PE-Exercise) ensures that, whenever an instance with permission ρ_s exercises $\rho \sqsubseteq \rho_s$, then ρ is bounded above by the privileges ρ' assigned to the expression.

Flow Analysis for Systems. Finally, we extend the flow analysis to systems by defining the main judgement $\mathcal{E} \Vdash s$ **despite** ρ, which follows from similar judgements for memories, handlers and instances. The definition is given in Table 5.

In the rules for memories we just need to ensure (cf. rule (PM-Single)) that, whenever a value v is stored in a reference r_ℓ protected with permission ρ_r, then v can be abstracted to some $\hat{v}$ over-approximated by the abstract memory entry $\mathcal{E}_{\hat{\mu}}(\ell, \rho_r)$. As for instances, rule (PI-Single) computes an approximation of the privileges ρ_e exercised by the running expression. Then, if the instance is granted permission $\rho_a \not\sqsubseteq \rho$, i.e., if it is not compromised, we check that the abstract stack correctly approximates with ρ_e the privileges exercised by the instance body. This check is not enforced for instances that might be under the control of the opponent, according to the idea that any opponent must be accepted by a sufficiently weak abstract environment. This is needed to prove an *opponent acceptability* result (Lemma 2), which allows for a convenient soundness proof technique for the analysis [1,5].

Handlers are accepted by rule (PH-Single), which states that, to analyse $a(x \triangleleft \rho_s : \rho_a).e$ despite ρ-opponents, we first lookup the abstract stack $\hat{\Upsilon}$: let $\hat{\Upsilon}(a, \rho_a) = (\rho'_s, \rho'_e)$. If we are not analysing a (possibly) compromised handler, i.e., if $\rho_a \not\sqsubseteq \rho$, we ensure that the permission ρ'_s in the abstract stack matches the permission ρ_s guarding access to the handler. We then lookup the abstract network $\hat{\Phi}$: if $\hat{\Phi}(a, \rho_a) = \emptyset$, no instance of the system will ever communicate

Table 5. Flow analysis for systems

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

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

$$\text{(PM-Many)}\quad \frac{\mathcal{E} \Vdash \mu_1 \textbf{ despite } \rho \qquad \mathcal{E} \Vdash \mu_2 \textbf{ despite } \rho}{\mathcal{E} \Vdash \mu_1, \mu_2 \textbf{ despite } \rho}$$

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

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

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

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

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

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

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

with the handler and we can skip the analysis of its body. Otherwise, we ensure that the abstract variable environment maps the bound variable x to an over-approximation of the incoming message, abstracted by $\hat{\Phi}(a, \rho_a)$, and we analyse the body of the handler, to detect the exercised privileges ρ_e. If we are not analysing the opponent, we further ensure that ρ_e matches the permissions ρ'_e annotated in the abstract stack, i.e., we guarantee that the abstract stack contains reliable information.

Finally, rule (PS-Sys) states that a system $s = \mu; h; i$ is acceptable for $\mathcal{E}$ only whenever μ, h and i are all acceptable for $\mathcal{E}$, and $\mathcal{E}$ is a *ρ-conservative* abstract environment. This notion corresponds to the informal idea of "sufficiently weak abstract environment" needed to prove the opponent acceptability result. In order to define ρ-conservativeness, we first define the notion of *static leak* for an abstract environment.

Definition 5 (Static Leak). *We define the* static leak *of $\mathcal{E}$ against ρ as: $SLeak_\rho(\mathcal{E}) = \bigsqcup_{\rho_e \in L} \rho_e$, where $L = \{\rho_e \mid \exists a, \rho_a, \rho_s.\, \mathcal{E}_{\hat{\Upsilon}}(a, \rho_a) = (\rho_s, \rho_e) \wedge \rho_s \sqsubseteq \rho\}$.*

Intuitively, $SLeak_\rho(\mathcal{E})$ is the upper bound of all the permissions ρ_e that can be (transitively) exercised by any handler that can be called by a ρ-opponent. We then define the set $\mathcal{V}_\rho(\mathcal{E})$ of the opponent-controlled variables as:

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

The set contains all the variables $\mathcal{V}_u$ occurring in the opponent code, together with all the variables bound in lambda abstractions stored in references under the control of the opponent. All these variables can be instantiated at runtime with values chosen by the opponent. We use this set of variables also to define a sound abstraction of any value which can be generated by/flow to the opponent.

Definition 6 (Canonical Disclosed Abstract Value). *Given an abstract environment $\mathcal{E}$ and a permission ρ, the* canonical disclosed abstract value *is defined as:* $\hat{v}_\rho(\mathcal{E}) = \{\hat{u} \mid vars(\hat{u}) \subseteq \mathcal{V}_\rho(\mathcal{E})\}$.

The canonical disclosed abstract value is a canonical representation of any abstract value under the control of a ρ-opponent in a system accepted by $\mathcal{E}$. It is the set of all the pre-values which contain only opponent-controlled variables.

Based on the notions above, we define ρ-conservativeness.

Definition 7 (ρ-Conservative Abstract Environment). *An abstract environment $\mathcal{E}$ is* ρ-conservative *if and only if all the following conditions hold true:*

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

In words, an abstract environment is ρ-conservative whenever: (1) any handler that can be under the control of the opponent is in fact assumed to be accessible by the opponent and to escalate up to the static leak; (2) any handler that can be under the control of the opponent, or (3) that can be contacted by the opponent, is assumed to receive the canonical disclosed abstract value $\hat{v}_\rho(\mathcal{E})$; (4) any reference possibly under the control of the opponent is assumed to contain $\hat{v}_\rho(\mathcal{E})$; and (5) the argument of any function which can be called by the opponent is assumed to contain the canonical disclosed abstract value $\hat{v}_\rho(\mathcal{E})$ and similarly these functions are assumed to return $\hat{v}_\rho(\mathcal{E})$.

4.2 Formal Results

Our main formal result defines an upper bound for the privileges which can be escalated by the opponent in a system accepted by the flow analysis. Complete proofs are in the full version [9]; here, we start proving the soundness of the flow logic specification by means of a subject reduction result, which ensures that the acceptability of the analysis is preserved upon reduction.

Lemma 1 (Subject Reduction). *If $\mathcal{E} \Vdash s$* **despite** *ρ and $s \xrightarrow{\alpha} s'$, then $\mathcal{E} \Vdash s'$* **despite** *ρ.*

The next lemma states that any ρ-opponent is accepted by a ρ-conservative abstract environment. Intuitively, the combination of this result with subject reduction ensures that the acceptability of the analysis is preserved at runtime, even when the analysed system interacts with the opponent.

Lemma 2 (Opponent Acceptability). *If (h, i) is a ρ-opponent and $\mathcal{E}$ is ρ-conservative, then $\mathcal{E} \Vdash h$* **despite** *ρ and $\mathcal{E} \Vdash i$* **despite** *ρ.*

Moreover, proving the safety theorem requires to explicitly track the call chains carried out by the system reduction, to collect the privileges transitively exercised by system components. The next lemma then relies on the following definition of call chain to prove that the abstract stack contains a static approximation of the privileges which are exercised by each system component either directly or by communicating with other components.

Definition 8 (Call Chain). *A* call chain $(\overrightarrow{\alpha}, a{:}\rho_a \gg \rho')$ *is a trace of length* $(n+1)$ *such that:*

1. *the trace $\overrightarrow{\alpha} = \langle a_1{:}\rho_{a_1}, b_1{:}\rho_{b_1}\rangle, \ldots, \langle a_n{:}\rho_{a_n}, b_n{:}\rho_{b_n}\rangle$ is a sequence of send labels where the sender occurring in each label is the receiver occurring in the previous label, i.e., $\forall i \in [1, n{-}1].\, a_{i+1} = b_i \wedge \rho_{a_{i+1}} = \rho_{b_i}$, and*
2. *the component exercising the privilege ρ' at the end of the call chain corresponds to the last receiver, i.e., $b_n = a \wedge \rho_{b_n} = \rho_a$.*

A trace $\overrightarrow{\beta}$ includes *a call chain $\overrightarrow{\alpha}$ iff $\overrightarrow{\alpha}$ is a sub-trace of $\overrightarrow{\beta}$.*

According to the intuition given above, proving the soundness of the abstract stack amounts to showing that, given a call chain leading to the exercise of some privilege ρ' not available to the opponent, the abstract stack $\mathcal{E}_{\hat{\Upsilon}}$ approximates the privileges exercised by any component involved in the chain with a permission greater than or equal to ρ'. The proof uses the subject reduction result.

Lemma 3 (Soundness of the Abstract Stack). *If $\mathcal{E} \Vdash s$* **despite** *ρ and $s \overset{\overrightarrow{\beta}}{\Longrightarrow} s'$ for a trace $\overrightarrow{\beta}$ including the call chain $(\overrightarrow{\alpha}, a{:}\rho_a \gg \rho')$ for some $\rho' \not\sqsubseteq \rho$, then for each label $\alpha_j = \langle a_j{:}\rho_{a_j}, b_j{:}\rho_{b_j}\rangle \in \{\overrightarrow{\alpha}\}$ we have $\mathcal{E}_{\hat{\Upsilon}}(b_j, \rho_{b_j}) = (\rho_{s_{b_j}}, \rho_{e_{b_j}})$ with $\rho' \sqsubseteq \rho_{e_{b_j}}$ and $\mathcal{E}_{\hat{\Upsilon}}(a_j, \rho_{a_j}) = (\rho_{s_{a_j}}, \rho_{e_{a_j}})$ with $\rho' \sqsubseteq \rho_{e_{a_j}}$.*

Theorem 1 (Flow Safety). *Let $s = \mu; h; \emptyset$. If $\mathcal{E} \Vdash s$* **despite** *ρ, then s leaks $SLeak_\rho(\mathcal{E})$ against ρ.*

Proof. By contradiction. Let $\hat{s}$ be the system obtained by composing s with a ρ-opponent and assume that $\hat{s}$ eventually reaches a state s' such that s' exercises privileges ρ_{bad}, with $\rho_{bad} \not\sqsubseteq \rho$ and $\rho_{bad} \not\sqsubseteq SLeak_\rho(\mathcal{E})$.

By inverting rule (PS-Sys) on the hypothesis $\mathcal{E} \Vdash s$ **despite** ρ, we have that $\mathcal{E}$ is ρ-conservative. Using Lemma 2 (Opponent Acceptability), we show that $\mathcal{E} \Vdash \hat{s}$ **despite** ρ. Given that $\rho_{bad} \not\sqsubseteq \rho$, the privileges ρ_{bad} cannot be directly exercised by the opponent, hence there must exist a call chain leading to ρ_{bad} from $\hat{s}$. Let a_i range over the components in the call chain and ρ_i range over

their corresponding permissions. Consider now the first sender a_1 in the call chain: given that the original system s does not have running instances, it turns out that a_1 must be the opponent, hence $\rho_1 \sqsubseteq \rho$. Since $\mathcal{E}$ is ρ-conservative and $\rho_1 \sqsubseteq \rho$, we have $\mathcal{E}_{\hat{\Upsilon}}(a_1, \rho_1) = (\bot, \mathit{SLeak}_\rho(\mathcal{E}))$. By Lemma 3 (Soundness of the Abstract Stack), for each component a_i with permissions ρ_i occurring in the call chain we must have $\mathcal{E}_{\hat{\Upsilon}}(a_i, \rho_i) = (\rho_{s_i}, \rho_{e_i})$ for some ρ_{s_i} and some $\rho_{e_i} \sqsupseteq \rho_{bad}$. But then we get $\rho_{bad} \sqsubseteq \mathit{SLeak}_\rho(\mathcal{E})$, which is contradictory.

4.3 Analysing the Example

We now show the analysis at work on our running example in its three variants, namely the systems s, s_{tag} and s_{chan} introduced in Section 3. We assume that the abstract domain for strings includes all the string literals syntactically occurring in the program code, plus the distinguished symbol * to represent all the other strings (or any string which we cannot statically reconstruct). We let $\widehat{str}$ range over elements of this abstract domain and we assume that $\widehat{str} \sqsubseteq *$ for any $\widehat{str}$. As to records, we choose the field-sensitive representation $\{\!|\overrightarrow{\widehat{str}_i : \hat{v}_i}|\!\}$ where both the field names and contents are inductively abstracted. In the following we mostly focus on the intuitions behind the analysis: additional details, including the formal definitions of the expected abstract record operations and the abstract value pre-order, are given in the full version [9].

The Original System. We start by studying the robustness of the original system s against a P-opponent, i.e., an opponent with the only ability to dispatch the content script C attached to untrusted web pages. We have that $\mathcal{E} \Vdash s$ **despite** P, where $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ satisfies the following assumptions:

$$\hat{\Phi}(c, \mathsf{C}) = \hat{v}_\mathsf{P}(\mathcal{E}) \qquad \hat{\Phi}(o, \mathsf{O}) = \emptyset \qquad \hat{\Phi}(b, \mathsf{B}) = \{\{\!|\, \textit{“site”} : \hat{v}_\mathsf{P}(\mathcal{E}), * : \hat{v}_\mathsf{P}(\mathcal{E})|\!\}\}$$
$$\hat{\Upsilon}(c, \mathsf{C}) = (\mathsf{P}, \mathsf{B}) \qquad \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) \qquad \hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B})$$

Since C can be accessed by the opponent, the value of $\hat{\Phi}(c, \mathsf{C})$ must be equal to $\hat{v}_\mathsf{P}(\mathcal{E})$ to ensure the P-conservativeness of $\mathcal{E}$. Conversely, O can never be accessed by the opponent or by any other component in the system, hence $\hat{\Phi}(o, \mathsf{O}) = \emptyset$. By rule (PH-Single), this implies that there is no need to analyse the body of O, which allows for ignoring the format of the messages sent by O: this explains why the value of $\hat{\Phi}(b, \mathsf{B})$ includes just one element, corresponding to the message sent by C. Indeed, observe that $\widehat{\mathit{set}}(\hat{v}_\mathsf{P}(\mathcal{E}), \textit{“site”}, \mathit{str}) \sqsubseteq \{\{\!|\, \textit{“site”} : \hat{v}_\mathsf{P}(\mathcal{E}), * : \hat{v}_\mathsf{P}(\mathcal{E})|\!\}\}$ for any str to accept the send expression in the body of C.

Now observe that $\{\textit{“policy”}, \textit{“upd”}\} \sqsubseteq \widehat{\mathit{get}}(\{\!|\, \textit{“site”} : \hat{v}_\mathsf{P}(\mathcal{E}), * : \hat{v}_\mathsf{P}(\mathcal{E})|\!\}, \textit{“tag”})$, hence both branches of the conditional in the body of B are reachable and the conditional expression may exercise B; we then let $\hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B})$ by rule (PH-Single). Given that C communicates with B, the privileges exercised by C must be greater or equal than B by rule (PE-Send), and propagated into $\hat{\Upsilon}(c, \mathsf{C})$ by rule (PH-Single). Since $\mathit{SLeak}_\mathsf{P}(\mathcal{E}) = \mathsf{B}$, we know that the system s leaks B against P by Theorem 1.

The System with Tags. Let us focus now on the system s_{tag} and a P-opponent. We have that $\mathcal{E} \Vdash s_{tag}$ **despite** P, where $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ is such that:

$$\hat{\Phi}(c, \mathsf{C}) = \hat{v}_{\mathsf{P}}(\mathcal{E}) \qquad \hat{\Phi}(o, \mathsf{O}) = \emptyset$$
$$\hat{\Phi}(b, \mathsf{B}) = \{\!|\, \text{“tag”} : \text{“policy”}, \text{“site”} : *, \text{“spec”} : \hat{v}_{\mathsf{P}}(\mathcal{E}) \,|\!\}$$
$$\hat{\Upsilon}(c, \mathsf{C}) = (\mathsf{P}, \mathsf{MemB}) \qquad \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) \qquad \hat{\Upsilon}(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{MemB})$$

Based on this information, rule (PE-Cond) allows for analysing only the program branch of B corresponding to the processing of a message with tag *“policy”*, which only exercises the privilege MemB: this motivates the precise choice of $\hat{\Upsilon}(b, \mathsf{B})$. Since $SLeak_{\mathsf{P}}(\mathcal{E}) = \mathsf{MemB}$, the system leaks MemB against P.

Assume now an opponent with permission C, then we have $\mathcal{E}' \Vdash s_{tag}$ **despite** C, where $\mathcal{E}' = \hat{\Gamma}'; \hat{\mu}'; \hat{\Upsilon}'; \hat{\Phi}'$ is such that:

$$\hat{\Phi}'(c, \mathsf{C}) = \hat{v}_{\mathsf{C}}(\mathcal{E}') \qquad \hat{\Phi}'(o, \mathsf{O}) = \emptyset \qquad \hat{\Phi}'(b, \mathsf{B}) = \hat{v}_{\mathsf{C}}(\mathcal{E}')$$
$$\hat{\Upsilon}'(c, \mathsf{C}) = (\bot, \mathsf{B}) \qquad \hat{\Upsilon}'(o, \mathsf{O}) = (\top, \bot) \qquad \hat{\Upsilon}'(b, \mathsf{B}) = (\mathsf{C} \sqcap \mathsf{O}, \mathsf{B})$$

With respect to the previous scenario, the abstract network entry for B contains $\hat{v}_{\mathsf{C}}(\mathcal{E}')$, abstracting all the values which may be generated by a C-opponent: this is needed for C-conservativeness. The consequence is that all the program branches of B are reachable, hence B may exercise its full set of privileges B. Since $SLeak_{\mathsf{C}}(\mathcal{E}') = \mathsf{B}$, the system leaks B against C by Theorem 1.

The System with Channels. We are able to prove $\mathcal{E} \Vdash s_{chan}$ **despite** C for an abstract environment $\mathcal{E} = \hat{\Gamma}; \hat{\mu}; \hat{\Upsilon}; \hat{\Phi}$ such that:

$$\hat{\Phi}(c, \mathsf{C}) = \hat{v}_{\mathsf{C}}(\mathcal{E}) \qquad \hat{\Phi}(o, \mathsf{O}) = \emptyset \qquad \hat{\Phi}(b_1, \mathsf{B}) = \hat{v}_{\mathsf{C}}(\mathcal{E}) \qquad \hat{\Phi}(b_2, \mathsf{B}) = \emptyset$$
$$\hat{\Upsilon}(c, \mathsf{C}) = (\bot, \mathsf{MemB}) \quad \hat{\Upsilon}(o, \mathsf{O}) = (\top, \bot) \quad \hat{\Upsilon}(b_1, \mathsf{B}) = (\mathsf{C}, \mathsf{MemB}) \quad \hat{\Upsilon}(b_2, \mathsf{B}) = (\mathsf{O}, \bot)$$

For the new abstract environment $\mathcal{E}$ we have $SLeak_{\mathsf{C}}(\mathcal{E}) = \mathsf{MemB}$, which ensures that the new system only leaks MemB against C. Since the privilege $\mathsf{Cookies}$ cannot be escalated by a compromised C anymore, there is no way to corrupt the cookie jar without compromising the background page B itself (or the options page O). Interestingly, this is a formal characterization of the dangers connected to the development of *bundled* browser extensions in a realistic setting [2].

5 Implementation: Chen

Chen is a prototype Google Chrome extension analyser written in F#. Given a Chrome extension, Chen translates it into a corresponding system in our formalism and computes an acceptable flow analysis estimate by constraint solving. Chen can be used by programmers to evaluate the robustness of their extensions against privilege escalation attacks and to support their security refactoring.

5.1 Flow Logic Implementation

Implementing the flow logic specification amounts to defining an algorithm that, given a system s and a permission ρ characterizing the power of the opponent, computes an abstract environment $\mathcal{E}$ such that $\mathcal{E} \Vdash s$ **despite** ρ. Following a standard approach [25], we first define a verbose variant of the flow logic, which associates an analysis estimate to each sub-expression of s, and then we devise a constraint-based formulation of the analysis. Any solution of the constraints is an abstract environment $\mathcal{E}$ which accepts s.

We initially implemented in Chen a simple worklist algorithm for constraint solving. However, consistently with what has been reported by Jensen *et al.* in the context of JavaScript analysis [19], we observed that this solution does not scale, taking hours to perform the analysis even on small examples. Therefore, in our implementation we use a variant of the worklist algorithm where most of the constraint generation is performed *on demand* during the solving process. Even though this approach does not allow us to reuse existing solvers, it leads to a dramatic improvement in the performances of the analysis.

The current prototype implements a context-insensitive analysis, which is enough to capture the privileges escalated by the content scripts, provided that some specific library functions introduced by the desugaring process from JavaScript to λ_{JS} (see below) are inlined. The choice of the abstract pre-values for constants is standard: in the current implementation, we represent numbers with their sign and we approximate strings with finite prefixes [11]. The representation of records is field-sensitive, but we collapse into a single label * all the entries bound to approximate labels (string prefixes). As to the ordering, we consider a standard pre-order $\sqsubseteq_p$ on abstract pre-values, and we lift it to abstract values using a lower powerset construction, i.e., we let $\hat{v} \sqsubseteq \hat{v}'$ if and only if $\forall \hat{u} \in \hat{v}.\, \exists \hat{u}' \in \hat{v}'.\, \hat{u} \sqsubseteq_p \hat{u}'$.

5.2 Using Chen to Assess Google Chrome Extensions

Given an extension, Chen takes as an input a sequence of *component* names, along with the JavaScript files corresponding to their implementation. Components represent isolation domains, in that different components must be able to communicate only using the message passing interface. Different content scripts which may injected in the same web page should be put inside the same component, since Google Chrome does not separate their heaps. The background page should be put in a separate component, since it runs in an isolated process[2].

From JavaScript to the Model. Let c be a component name and $f_1, \ldots, f_n$ the corresponding JavaScript files: our tool concatenates $f_1, \ldots, f_n$ into a single file f, which is desugared into a closed λ_{JS} expression using an existing tool [16]. The adequacy of the translation from JavaScript to λ_{JS} has been assessed by extensive automatic testing, hence safety guarantees for JavaScript programs can be provided just by analysing their λ_{JS} translation; see [16] for further details.

[2] An appropriate mapping of JavaScript files to components can be derived from the manifest file of the extension, but the current prototype does not support this feature.

The obtained λ_{JS} expression is then transformed into a set of handlers: more precisely, for any function $\lambda x.e'$ passed as an argument to the `addListener` method of `chrome.runtime.onMessage`, we introduce a new handler on a channel with the same name of the component, whose body is obtained by closing e' with the introduction of all the bindings defined before the registration of the listener. For each component we introduce a unique permission for memory access, granted to each handler in the component; handlers corresponding to the background page are also given the permissions specified in the manifest of the extension. Any invocation of `chrome.runtime.sendMessage` in the definition of a content script is translated to a send expression over a channel with the name of the component corresponding to the background page.

Notice that CHEN exploits an existing tool to translate JavaScript to λ_{JS}, but our target language has two new constructs: message sending and privilege exercise. In JavaScript, both operations correspond to function calls to the Chrome extension API, hence, to introduce the syntactic forms corresponding to them in the translation to our formalism, we extend the JavaScript code to redefine the functions of interest in the Chrome API with *stubs*. For instance, `chrome.cookies.set` is redefined to a function including the special tag `"#Cookies#"`, which is preserved when desugaring JavaScript to λ_{JS}: we then post-process the λ_{JS} expression to replace this tag with **exercise**(Cookies).

Running the Analysis. The tool supports two analyses. The option `-compromise` instructs CHEN to analyse the privileges which may be escalated by an opponent assuming the full compromise of an arbitrary content script, i.e., it estimates the safety of the system despite the permission that protects the background page. If the background page requests some permission ρ intended for internal use, but ρ is available to some content script according to the results of the analysis, then the developer is recommended to review the communication interface.

Alternatively, the option `-target` n allows to get an approximation of the privileges available to the content scripts in the component n in absence of compromise. We model absence of compromise by considering a $\bot$-opponent as the threat model, since this opponent cannot directly communicate with the background page: if the option `-target` n is specified, CHEN transforms the system by protecting with permission $\bot$ all the handlers included in n, and computes a permission ρ such that the system is ρ-safe despite $\bot$. This allows to estimate which privileges are enabled by messages sent from n, so as to identify potential room for a security refactoring, as we discuss below.

Both the analyses additionally support the option `-flag` p, which allows to define a dummy permission p assigned to the background page. The programmer may then annotate specific program points with the tag `"#p#`, corresponding to the exercise of this dummy permission; by checking the presence of the flag among the escalated privileges, CHEN can be used to implement an opponent-aware reachability analysis on the extension code.

Supporting a Security Refactoring. To exemplify, we analyse with CHEN our motivating example. By first specifying the option `-target` O, the tool detects that the options page O is only accessing the privilege Cookies as part of its standard

functionalities, even though the background page B is given the permissions $\mathsf{MemB} \sqcup \mathsf{Cookies}$. To support least privilege, the developer is thus recommended to introduce a distinct communication port for B. Notably, the permission gap arises from the presence in the code of B of program branches which are never triggered by messages sent by O in absence of compromise: in principle, Chen could then automatically introduce the new port, replicate the code from the handler of the background page, and improve its security against compromise by eliminating the dead branches, even though the current prototype does not implement this feature.

Then, by using the option `-target` C, the tool outputs that the privilege $\mathsf{MemB} \sqcup \mathsf{Cookies}$ can be escalated by the content script C. Hence, no automated refactoring is possible, but the output of the analysis is still helpful for a careful developer, who realizes that C should not be able to access the Cookies privilege. Based on the output of the analysis, the developer may opt for a manual reviewing and refactoring of the extension.

Current Limitations. Being a proof-of-concept implementation, the current version of Chen lacks a full coverage of the Chrome extension APIs. Moreover, Chen cannot analyse extensions which use ports to communicate: in our model, ports are just channels and do not pose any significant problem to the analysis. Unfortunately, the current Chrome API makes it difficult to support the analysis of extensions using ports, since the underlying programming patterns make massive usage of callbacks. Based on our experience and a preliminary investigation, however, ports are not widely used in practice, hence many extensions can still be analysed by Chen.

5.3 Case Study: ShareMeNot

ShareMeNot [30] is a popular privacy-enhancing extension developed at the University of Washington. The extension looks for social sharing buttons in the web pages and replaces them with dummy buttons: only when the user clicks one of these buttons, its original version is loaded and the cookies registered by the corresponding social networks are sent. This means that the social network can track the user only when the user is willing to share something.

ShareMeNot consists of four components: a content script, a background page, an option page and a popup, for a total of approximately 1,500 lines of JavaScript code. The background page offers a unique entry point to all the other extension components and handles seven different message types. Interestingly, one of these messages allows to unblock all the trackers in an arbitrary tab, by invoking the `unblockAllTrackersOnTab` function: this message should only be sent by the popup page. We then put a flag in the body of the function and we performed the analysis of ShareMeNot with the `-compromise` option, observing that the flag is reachable: hence, a compromised content script could entirely deactivate the extension. The analysis took around 150 seconds on a standard commercial machine.

We then ran the analysis with the `-target` C option, where C is the name of the component including only the content script, and we observed that the

flag was not reachable. This means that C does not need to access the function `unblockAllTrackersOnTab` as part of its standard functionalities, hence the code should be refactored to comply with the principle of the least privilege and prevent a potential security risk. The analysis took around 210 seconds on the same machine.

6 Conclusions

We presented a core calculus to reason about browser extensions security and we proposed a flow analysis aimed at detecting which privileges may be leaked to an opponent which compromises some (arbitrarily chosen) untrusted extension components. The analysis has been proved sound and it has been implemented in Chen, a prototype static analyser for Google Chrome extensions. We discussed how Chen can assist developers in writing more robust extensions.

As future work, we plan to further engineer Chen, to make it support more sophisticated communication patterns used in Google Chrome extensions. We ultimately plan to evolve Chen into a compiler, which automatically refactors the extension code to make it more secure, by unbundling functionalities based on their exercised permissions. Based on a preliminary investigation, this will require a non-trivial programming effort.

Acknowledgements. We would like to thank Arjun Guha for insightful discussions about the λ_{JS} semantics. Alvise Spanò provided useful F# libraries and advices for the development of Chen. This work was partially supported by the MIUR projects ADAPT and CINA, and by the University of Padova under the PRAT project BECOM.

References

1. Abadi, M.: Secrecy by typing in security protocols. J. ACM 46, 749–786 (1999)
2. Akhawe, D., Saxena, P., Song, D.: Privilege separation in HTML5 applications. In: USENIX Security Symposium, pp. 429–444 (2012)
3. Bandhakavi, S., Tiku, N., Pittman, W., King, S.T., Madhusudan, P., Winslett, M.: Vetting browser extensions for security vulnerabilities with VEX. Communications of the ACM 54(9), 91–99 (2011)
4. Barth, A., Porter Felt, A., Saxena, P., Boodman, A.: Protecting browsers from extension vulnerabilities. In: NDSS (2010)
5. Bodei, C., Buchholtz, M., Degano, P., Nielson, F., Nielson, H.R.: Static validation of security protocols. Journal of Computer Security 13(3), 347–390 (2005)
6. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W.: Automatic and robust client-side protection for cookie-based sessions. In: Jürjens, J., Piessens, F., Bielova, N. (eds.) ESSoS. LNCS, vol. 8364, pp. 161–178. Springer, Heidelberg (2014)
7. Bugliesi, M., Calzavara, S., Focardi, R., Khan, W., Tempesta, M.: Provably sound browser-based enforcement of web session integrity. In: CSF, pp. 366–380 (2014)
8. Bugliesi, M., Calzavara, S., Spanò, A.: Lintent: Towards security type-checking of android applications. In: Beyer, D., Boreale, M. (eds.) FORTE 2013 and FMOODS 2013. LNCS, vol. 7892, pp. 289–304. Springer, Heidelberg (2013)

9. Calzavara, S., Bugliesi, M., Crafa, S., Steffinlongo, E.: Fine-grained detection of privilege escalation attacks on browser extensions (full version), http://www.dais.unive.it/textasciitildecalzavara/papers/esop15-full.pdf
10. Carlini, N., Porter Felt, A., Wagner, D.: An evaluation of the Google Chrome extension security architecture. In: USENIX Security Symposium, pp. 97–111 (2012)
11. Costantini, G., Ferrara, P., Cortesi, A.: Static analysis of string values. In: ICFEM, pp. 505–521 (2011)
12. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011)
13. Dhawan, M., Ganapathy, V.: Analyzing information flow in JavaScript-based browser extensions. In: ACSAC, pp. 382–391 (2009)
14. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing Android's permission system. In: ESORICS, pp. 1–18 (2012)
15. Guha, A., Fredrikson, M., Livshits, B., Swamy, N.: Verified security for browser extensions. In: 32nd IEEE Symposium on Security and Privacy, pp. 115–130 (2011)
16. Guha, A., Saftoiu, C., Krishnamurthi, S.: The essence of javaScript. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 126–150. Springer, Heidelberg (2010)
17. Guha, A., Saftoiu, C., Krishnamurthi, S.: Typing local control and state using flow analysis. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 256–275. Springer, Heidelberg (2011)
18. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
19. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 320–339. Springer, Heidelberg (2010)
20. Karim, R., Dhawan, M., Ganapathy, V., Shan, C.-c.: An analysis of the mozilla jetpack extension framework. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 333–355. Springer, Heidelberg (2012)
21. Liu, L., Zhang, X., Yan, G., Chen, S.: Chrome extensions: Threat analysis and countermeasures. In: NDSS (2012)
22. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for JavaScript. In: APLAS, pp. 307–325 (2008)
23. Maffeis, S., Taly, A.: Language-based isolation of untrusted JavaScript. In: CSF, pp. 77–91 (2009)
24. Nielson, F., Nielson, H.R.: Flow logic and operational semantics. Electronic Notes on Theoretical Computer Science 10, 150–169 (1997)
25. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)
26. Nielson, H.R., Nielson, F., Pilegaard, H.: Flow logic for process calculi. ACM Computing Surveys 44(1), 1–39 (2012)
27. Politz, J.G., Carroll, M.J., Lerner, B.S., Pombrio, J., Krishnamurthi, S.: A tested semantics for getters, setters, and eval in javascript. In: DLS, pp. 1–16 (2012)
28. Politz, J.G., Eliopoulos, S.A., Guha, A., Krishnamurthi, S.: Adsafety: Type-based verification of JavaScript sandboxing. In: USENIX Security Symposium (2011)
29. Porter Felt, A., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E.: Permission redelegation: Attacks and defenses. In: USENIX Security Symposium (2011)
30. Roesner, F., Kohno, T., Wetherall, D.: Detecting and defending against third-party tracking on the web. In: NSDI, pp. 155–168 (2012)
31. Saltzer, J.H., Schroeder, M.D.: The protection of information in computer systems. Proceedings of the IEEE 63(9), 1278–1308 (1975)