Projet Ajacs

Deliverable WP3

Formal definitions for main security goals for JavaScript applications

June 2016

This deliverable includes the following articles describing work done on WP3 for the first 18 months of the project.

- **Mashic Compiler: Mashup Sandboxing using Inter-frame Communication**, by Zhengqin Luo, Jose Fragoso Santos, Ana Almeida Matos, and Tamara Rezk
- **BrowserAudit: Automated Testing of Browser Security Features**, by Charlie Hothersall-Thomas, Sergio Maffeis, and Chris Novakovic
- **Hybrid Typing of Secure Information Flow in a JavaScript-like Language**, by Jos Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt
- **Modular Monitor Extensions for Information Flow Security in JavaScript**, by Jos Fragoso Santos, Tamara Rezk, Ana Almeida Matos
- **A Taxonomy of Information Flow Monitors**, by Nataliia Bielova and Tamara Rezk
- **On access control, capabilities, their equivalence, and confused deputy attacks**, by Vineet Rajani, Deepak Garg, and Tamara Rezk
- **Hybrid Monitoring of Attacker Knowledge**, by Frdric Besson, Nataliia Bielova and Thomas Jensen

Journal of Computer Security 1 (2014) 1
IOS Press

Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication

Zhengqin Luo [a] José Fragoso Santos [c] Ana Almeida Matos [b] Tamara Rezk [c]
[a] *Google*
[b] *IST*
[c] *INRIA*

Abstract. Mashups are a prevailing kind of web applications integrating external gadget APIs often written in the JavaScript programming language. Writing secure mashups is a challenging task due to the heterogeneity of existing gadget APIs, the privileges granted to gadgets during mashup executions, and JavaScript's highly dynamic environment. We propose a new compiler, called Mashic, for the automatic generation of secure JavaScript-based mashups from existing mashup code. The Mashic compiler can effortlessly be applied to existing mashups based on a wide-range of gadget APIs. It offers security and correctness guarantees. Security is achieved via the Same Origin Policy. Correctness is ensured in the presence of benign gadgets, that satisfy confidentiality and integrity constrains with regard to the integrator code. The compiler has been successfully applied to real world mashups based on Google maps, Bing maps, YouTube, and Zwibbler APIs.

1. Introduction

Mixing existing online libraries and data into new online applications in a rapid, inexpensive manner, often referred to as mashups, has captured the way of designing web applications. ProgrammableWeb mashup graphs currently report that over 6000 mashup-based web applications and over 11000 gadget APIs currently exist (`http://www.programmableweb.com/`). Since the release of the first major example, `HousingMaps.com` in early 2005, mashups are the de-facto applications in the web today.

In a mashup, the *integrator* code integrates *gadgets* from external code providers. Typically, code is written in JavaScript (JS) and executes on the browser as embedded script nodes in the Document Object Model (DOM) [17]. External gadget code in a mashup can be included in two ways:

- either by using the script tag and granting access to all the resources of the integrator;
- or by using the iframe tag, in which case the Same Origin Policy (SOP) applies. The SOP isolates untrusted JavaScript external code, limiting the interaction of gadget and integrator to message sending [3].

This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008 and the EPSRC Grant Reference EP/H008373/1.

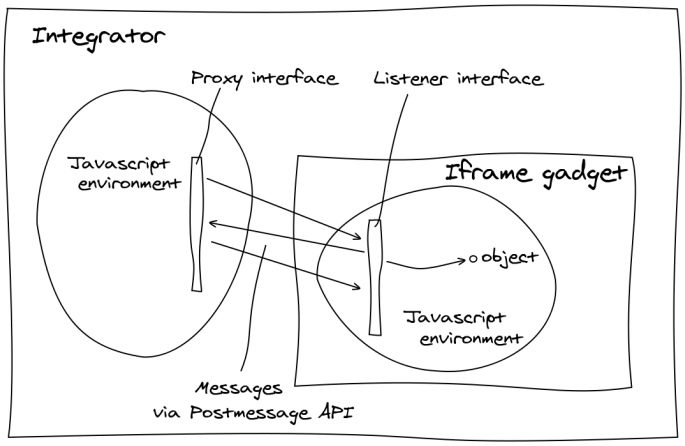


Fig. 1. Target Architecture Automatically Generated by Mashic

Static analysis to confine JavaScript programs is feasible for large-scale code consumers such as Facebook.com or Google.com, since they can restrict the JavaScript subset in which developers can write gadget code. Furthermore the size of those gadgets are relatively small. However, when it comes to small code consumers and large gadget providers, such as Google Maps API, full-fledged static analysis is usually infeasible since code providers do not confine them to a certain subset of JavaScript, and the gadget code size is usually large and difficult to be analyzed. Moreover, gadget code is subject to change from time to time by the provider. Mashup programmers are challenged to provide flexible functionality even if the code consumer is not willing to trust the gadgets that mashups utilize. Unfortunately, programmers often choose to include gadgets using the script tag and resign to security in the name of functionality.

Recently, Smash [23], AdJail [25], and Postmash [2] proposed to use inter-frame communication between integrator and gadgets. Smash proposes a secure component model for mashups that generalizes the security policies imposed by the SOP. The model is implemented via inter-frame communication and offered as JavaScript libraries. However, integrators and gadgets code have to be adapted to this specific way of communication. AdJail focuses on advertisement scripts by delegating limited DOM interfaces from the integrator. PostMash targets interfaces to operate on gadgets and proposes an architecture for mashups depicted in Figure 1. In the PostMash design there are stub libraries on both the integrator and the gadget. On the integrator side, the stub library must provide an interface similar to the original gadget's interface. The stubbed interface sends corresponding messages by means of the PostMessage API in HTML5. On the gadget side, there is another stub library, listening and decoding incoming messages. Barth et al. [2] evaluate the feasibility of the PostMash design via a case study using a version of a Google Maps gadget by creating a stub library that mimicked GMap2 API. Regarding the libraries, the authors argue that the stub library can either be provided by the integrator (one for each untrusted gadget), or by the gadget in which case the library must be audited for security by the integrator.

In this work, we address the following questions about the PostMash design:

1. Can the stub libraries be made general (the same libraries for every gadget and integrator)?
2. Can PostMash mashups be automatically generated starting from potentially insecure mashups and preserving only the good behavior of the original mashup?
3. Is it possible to precisely define the security guarantees offered by the architecture?

We have positively answered these questions.

We address questions 1 and 2 with a novel compiler called Mashic which inputs existing mashup code, JavaScript code integrated to HTML, to generate reliable mashups using gadget isolation as shown in Figure 1. In addition, for question 2, we formalize the notion of "benign gadget" that is useful to prove precisely in which cases the generated mashup behaves as the original one. Notably, the answer to question 3 corresponds to the first formalization as an observational semantics equivalence of the security guarantees offered by the Same Origin Policy in a browser, that, we conjecture, coincides with a form of declassification policy known as delimited release [34]. The Mashic compiler [30] offers the following features:

Automation and generality: Inter-frame communication and sandboxing code is fully generated by the compiler and can be used with any untrusted gadget without rewriting the gadget's code. After sandboxing, gadget objects are not directly reached by the integrator when the SOP applies. Instead the integrator uses opaque handles [36] to interact with the gadget. Due to the asynchronous nature of the PostMessage API, integrator's code is transformed into Continuation Passing Style (CPS).

Correctness guarantees: We prove a correctness theorem that states that the behavior of the Mashic compiled code is equivalent to the original mashup behavior under the hypotheses that the gadget is *benign* and correctness of marshaling/unmarshaling for objects that are sent via postMessage.

The correctness notion of marshaling/unmarshaling allows us to identify, for example, that the implementation of a secure mashup is not correct as soon as the integrator sends an object with a cyclic structure to the gadget (if the implementation uses the json stringify for marshaling).

Precisely defining a benign gadget turned out to be a technical challenge in itself. For that, we instrument the JavaScript semantics extended with HTML constructs by a generalization of colored brackets [14] and resort to equivalences used in information flow security [33].

Security guarantees: We prove a security theorem that guarantees a delimited form of integrity and confidentiality for the compiled mashup. Information sent from the integrator to the gadget, corresponds to a declassification. We prove that the gadget cannot learn more than what the integrator sends. Analogously, the influence that the gadget can have on the integrator is delimited to the actions that the integrator performs with the messages that the gadget sends to the integrator. These guarantees are essential for the success of the compiler since the programmer can rely on this precise notion of security for compiled mashups using untrusted gadgets without further hypotheses. Indeed, if the gadget is not benign in the original mashup, malicious behavior is neutralized in the compiled mashup. This proof relies on the browsers' SOP, that we formalize by means of iframe DOM elements.

The proposed compiler is directly applicable to real world and widespread mashups. We present evidence that our compiler is effective. We have compiled several mashups based on Google and Bing maps, YouTube, and Zwibbler APIs.

In summary our contributions are:

1. The Mashic compiler, its design and implementation, that can effortlessly be applied to existing mashups. The Mashic prototype and proofs are available online [30].
2. Security and correctness guarantees for Mashic compiled code and hence direct guarantees for the mashup end consumer. To our knowledge, this is the first work to formalize and prove guarantees of correctness and security for real world mashups.
3. A formalization of the SOP in browsers, related to frames and script tags, in a standard small-step semantics style.
4. A decorated semantics for JS extended with HTML constructs. This semantics provides clarity, in a highly dynamic language as JS, regarding ownership of properties in the heap and we prove it useful by specifying confidentiality and integrity policies in our theorems.

5. Case studies based on existing widespread mashups that demonstrate the effectiveness of the compiler.
6. An optimization for reducing the cost of cross-domain operations between gadgets and integrator in which an automatic batching mechanism groups together cross-domain operations in straight-line code, supporting loops and branching instructions.

Limitations The current implementation of the Mashic compiler suffers from the following limitations:

- Unsupported Constructs: Our integrator transformer currently supports the full JavaScript language [11] except for a few programming constructs. Specifically, the *for-in* construct and *exception* construct are not supported. Some JavaScript features considered "dangerous" such as *eval* are not supported either.
- Multiple gadgets inter-communication: The compiler is completely independent of gadget code, and does not support inter-gadget communication (communication is always done via the integrator), since this would imply transforming gadgets that want to use others' interfaces. For simplicity, the formal presentation of the Mashic compiler applies to one gadget but its implementation supports multiple gadgets by generating unique ids for each iframe and using them in the proxy interface. Authentication is ensured by the PostMessage mechanism [3].
- Symmetric Interface: The current Mashic architecture is restricted to mashups that employ only one-way communication, i.e. only the integrator will invoke interfaces provided by the gadget. (Although the gadget is not allowed to send requests to the integrator, it can certainly reply to those that it receives as Mashic supports callbacks.) Certain types of mashups do not fall into this category, notably mashups containing advertisement scripts. Louw et al. [25] addresses two-way communication in AdJail where a subset of the DOM interface from the integrator is also provided to the gadget by dynamically modifying the DOM interface in the sandboxed gadget. In Mashic, in order to enable general interfaces to be exposed to gadgets, the gadget has to be CPS-transformed. At the cost of losing gadget-code independence, it is straightforward to use the Mashic compiler (transformations applied to integrator) for gadgets code, without loosing any of the correctness guarantees.

Remarks. This paper extends an earlier conference version [26]. We extend the conference version by adding explanations, examples, and studying the performance of the Mashic compiler. We propose and implement an optimization based on future batches by Bogle and Liskov [5] and on batching remote procedure calls by Ibrahim et al. [18].

Related Work The closest works to Mashic are AdJail [25], Smash [23], and Postmash [2], and are described above. We focus now in other related work. Nikiforakis et al. [31] crawl more than three million pages over the top 10000 Alexa sites and show that many sites trust gadgets that could be successfully compromised by determined attackers. Moreover, a study over 6,805 unique websites [38] reveal that insecure JavaScript practices are common showing that at least 66.4% of mashups include gadgets into the top-level documents of their webpages. Jang et al. [22] study on top of 50000 websites privacy violating information flows in JavaScript based web applications. Their survey shows that top-100 sites present vulnerabilities related to cookie stealing, location hijacking, history sniffing and behavior tracking. Browser implementation vulnerabilities have also been shown to leak JavaScript capabilities between different origins [4]. Many mechanisms to prevent JavaScript based attacks have been deployed. For example the Facebook JavaScript subset (FBJS) [19] was intended to prevent user-written gadgets to attack trusted code but it did not really succeed in its goals [27]. Google Caja [20] is similar to FBJS, transforming JavaScript programs to insert run-time checks to prevent malicious access. Yahoo AD-

safe [10] statically validates JavaScript programs. Maffeis et al. [29] resort to language-based techniques to find out a subset of JavaScript that can be used to prove an isolation property for JavaScript code. For that, they identify a capability-safe subset of JavaScript. They do not formalize the SOP and they focus on pure isolation of gadgets in contrast to our confidentiality and integrity properties.

Static analysis is usually not applicable or not sound for large web applications due to the highly dynamic nature of JavaScript programs and because gadgets in general cannot be restricted to subsets of JavaScript. Static analysis for JavaScript subsets has been proposed by [32], providing a formal guarantee of isolation for ADSafe subset. Relying on type-based invariants and applicative bisimilarity Fournet et al. [13] show full abstraction of a translation from ML programs to JavaScript programs.

As a response to the increasing need to get flexible functionality without resigning to security guarantees, the research community has proposed several communication abstractions [37,9,21,23]. Specifically, Wang et al. [37] create an analogy between operating systems security [39,12] and mashups security to develop communication abstractions. OMASH [8] proposes a refined SOP to enable mashup communication.These abstractions usually require browser modifications and so far have not been adopted in HTML standards [16]. There are other works [6,1] pursuing the direction of formalizing web applications, but none of them formally model the SOP as a security property using observational semantics equivalences.

2. Running Example

In order to provide some background, we illustrate with a mashup different kinds of gadget inclusions and inter-frame communication. We reuse this example throughout the rest of the sections. There are two major types of gadgets in web mashups. The first type *requires* an interface from the integrator to accomplish some tasks. For instance advertisement scripts, which necessarily need to gather information of the integrator page through DOM APIs to implement the advertisement strategy. Another example of the first type of gadgets are user-supplied gadgets in social network platforms such as `facebook.com`. The second type *provides* a set of interfaces to the integrator. For instance the Google maps API, that provides various interfaces to operate a map gadget, is of this kind. We focus on the second type, that is gadget scripts that provide a set of interfaces to enable the integrator to manipulate the gadget.

In the example an integrator at `i.com` wants to include a gadget `gadget.js` provided by `untrusted.com`. The integrator creates an empty `div` element to delegate part of the DOM tree. The integrator includes the gadget by using a script tag:

```
1 <div id=gadget_canvas></div>
2 <script src='http://untrusted.com/gadget.js'></script>
```

Listing 1: Code Snippet of http://i.com/integrator.html

We focus on gadget scripts that provide a set of interfaces to enable the integrator to manipulate the gadget. The integrator calls methods or functions as interfaces to change the state of the gadget. For example, the following is a code snippet (in the integrator) to manipulate the untrusted gadget via interfaces:

```
1 var mydiv = document.getElementById("gadget_canvas")
2 var instance = new gadget.newInstance(
3                  mydiv, gadget.Type.SIMPLE);
```

```
4 instance.setLevel(9);
```

Listing 2: Code Snippet of http://i.com/integrator.html

The gadget defines a global variable `gadget` to provide interfaces to the integrator.

The `gadget.newInstance` is used to create a new gadget instance that binds to the div; and `instance.setLevel` is a method used to change state at the gadget instance. Let us assume that the integrator stores a secret in global variable `secret` and a global variable `price` holding certain information with an important integrity requirement:

```
1 var secret = document.getElementById("secret_input");
2 var price = 42;
```

Listing 3: Code Snippet of http://i.com/integrator.html

The secret flows to an untrusted source, and the price is modified at the gadget's will if the gadget contains the following code:

```
1 var steal;
2 steal = secret;
3 price = 0;
```

Listing 4: Non-benign Gadget

If the gadget is isolated using the iframe tag with a different origin, variables `secret` and `price` cannot be directly accessed by the gadget. We can modify the example in the following way:

```
1 <iframe src='http://u-i.com/gadget.html'></iframe>
```

Listing 5: Code Snippet of http://i.com/integrator-msg.html

```
1 <div id=gadget_canvas></div>
2 <script src='http://untrusted.com/gadget-msg.js'>
3 </script>
```

Listing 6: Code Snippet of http://u-i.com/gadget.html

Instead of directly including the script, the integrator invents a new origin `u-i.com` to be used as an untrusted gadget container, and puts the gadget code in a frame belonging to this origin. By doing this, the JavaScript execution environment between integrator and gadget is isolated, as guaranteed by the browser's SOP. Limited communication between frames and integrator is possible through the PostMessage API in the browser [1] if there is an event listener for the 'message' event. To register a listener one provides a callback function as parameter and treats messages in a waiting queue, asynchronously. Only strings can be sent as messages with PostMessage. However, it is possible to marshal objects without cyclical references (as e.g. the global object) via a marshaling method, such as the stan-

[1] Inter-frame communication is also possible via e.g navigation policies [3] but this kind of communication is now obsolete.

dard JSON stringify. Code in `gadget-msg.js` and `integrator-msg.html` needs to adapt to the asynchronous behavior. Instead of calling methods or functions, the integrator must send messages to manipulate the untrusted gadget as shown in the following example:

```
PostMessage(stringify({action : "newInstance",
          container : "gadget_div",
          type : "SIMPLE"}),
         "http://u-i.com");
PostMessage(stringify({action : "setLevel",
          container : "gadget_div"}),
          "http://u-i.com");
```

Listing 7: PostMessage Example

Compilation with Mashic will not preserve the malicious behavior of Listing 4 but will only preserve behavior that does not represent a confidentiality or integrity violation to the integrator.

The compiler relieves the programmer from rewriting code. Instead of rewriting gadget's code, our compiler inserts a proxy and a listener library that implement a communication protocol for manipulating gadgets independently of untrusted gadgets sandboxed in frames. Instead of rewriting integrator code, our compiler implements a CPS transformation to overcome the asynchronous nature of PostMessage. After compilation of code in Listing 1 and 2 the gadget is modified in the following way:

```
<html>
  <script src="listener.js"></script>
  <script src="untrusted.com/gadget.js"></script>
</html>
```

Listing 8: Compiled Gadget at http://u-i.com/gadget.html

The integrator is modified in the following way:

```
<html>
  <script src="proxy.js"></script>
  <iframe src="http://u-i.com/gadget.html"></iframe>
  <script src="integrator_cps.js"></script>
</html>
```

Listing 9: Compiled Integrator

Notice that the gadget code used in the compiled gadget is not modified from the original one. The proxy library and the listener library provide general ways to encode gadget operations, and the programmer does not need to manually write the stub library to operate on confined gadgets. The integrator code, as we mentioned above, is transformed to CPS code `integrator_cps.js` to perform the same task as in the code shown in Listing 2.

3. Decorated Semantics

We propose a decorated semantics to partition a JavaScript heap at the granularity of object properties. In order to prove security policies in a mashup, it is essential to distinguish at each execution step proper-

ties corresponding to different principals. Note that static decorations assigned to variables, traditionally used in information flow security policies [33], are not enough to specify security in JS programs due to two reasons: the dynamic nature of JS does not always allow us to syntactically determine the set of properties modified by a program (c.f. [27]), and existing native properties in the heap may either be changed by programs or its decoration may depend on the context due to the SOP.

The following example shows some dynamic features of JavaScript.

```
1 // integrator.js
2 var o = {}
3 o.secret = 43
4 o.['more' + 'secret'] = 52
5 ...
6 // gadget.js
7 steal = o.["sec"+"ret"] + o.moresecret
8 o.y = "some additional information"
```

Listing 10: Dynamic features of JS

First, the minimal container in JavaScript is property rather than object (or variable in other languages), so two properties of the same object could belong to different principals. Second, properties can be created dynamically, so static decoration is not possible.

Hence, we have resorted to ideas from colored brackets [14] and adapt them to a semantics modeling the SOP in the browsers. When decorations are erased, our JavaScript decorated rules are compliant with JavaScript semantics (Maffeis et al. [28]). For the sake of simplicity in the presentation, we limit this section to the inclusion of only one gadget as a frame, although the JavaScript semantics (and the Mashic compiler) is not limited in the number of gadgets included in a mashup. Thus, in this presentation, we need to distinguish three different colors. The ♠ color for the *Integrator Principal*, the ♥ color for the *Gadget Principal*, and the ♦ color to denote a neutral principal. We use $\square$ or $\triangle$ to denote any of them. For the sake of brevity, we do not include an origin parameter in the primitive PostMessage since there are only two possibilities: either the integrator communicates with the frame or vice-versa. We also simplify AddListener for the formal presentation: we assume that the only events it listens to is the event "message".

3.1. Decorated Objects

An object o is a tuple $\{i_{1\{\square\}} : v_1, \ldots i_{n\{\triangle\}} : v_n\}$ associating decorated properties $i_{\{\square\}}$ (internal identifiers or strings) to values. We use i instead of $i_{\{\square\}}$ whenever the decoration is not important. We distinguish internal properties that cannot be changed by programs with the symbol "@" in front of an identifier. We do not model attributes of properties that may indicate access controls as for example "do-not-delete" attributes [28]. We present a series of auxiliary definitions used in the operational semantics. For an object o and a property i, we use $i_{\{\square\}} \in o$ to denote that o has property i with decoration $\square$, and use $i \notin o$ to denote that o does not have property i.

3.2. Heaps

Objects are stored in heaps. A heap h is a partial mapping from locations in a set $\mathcal{L}$ to objects. We use the notation $h(\ell) = o$, to retrieve the object o stored in location ℓ; and the notation $o.i_{\{\square\}} = v$ to retrieve the value stored in property $i_{\{\square\}}$. We also use a shortcut $h(\ell).i_{\{\square\}}$ whenever possible. To update

(or create) a property $i_{\{\square\}}$ of an object at location ℓ in the heap, we use the notation $h(\ell.i_{\{\square\}} = v) = h'$, where h' is the updated heap. We also use $\mathrm{Alloc}(h, o) = h', \ell'$, where $\ell' \notin \mathsf{dom}(h)$, for allocating a fresh location for an object in the heap. After adding the location, the new heap is h'. JavaScript heaps contain two important chains of objects. The *scope chain* keeps track of the dynamic chains of function calls via the $@scope$ property. To resolve a scope of a variable name, one starts from the bottom of the chain, until reaching a scope object which contains the searched variable name. The scope look-up process is modeled by the $\mathrm{Scope}(h, \ell, m)$ function[2] It takes 3 parameters: a current heap, a heap location for a scope object (as the bottom of the scope chain), and a variable name as string to look up.

$$\textsc{Scope-null}\quad \mathrm{Scope}(h, null, m) = null$$

$$\textsc{Scope-ref}\quad \frac{m \in h(\ell)}{\mathrm{Scope}(h, \ell, m) = \ell}$$

$$\textsc{Scope-lookup}\quad \frac{m \notin h(\ell) \qquad \mathrm{Scope}(h, h(\ell).@scope, m) = \ell_n}{\mathrm{Scope}(h, \ell, m) = \ell_n}$$

Example 1. To lookup for name x from scope object ℓ in h, we use $\mathrm{Scope}(h, \ell, \text{“}x\text{”})$.

Similarly, the *prototype chain* represents the hierarchy between objects. A property that is not present in the current object, will be searched in the prototype chain, via the $@prototype$ property. The helper function $\mathrm{Prototype}(h, \ell, m)$ looks for the m property of the object $h(\ell)$ via the prototype chain.

$$\textsc{Prototype-null}\quad \mathrm{Prototype}(h, null, m) = null$$

$$\textsc{Prototype-ref}\quad \frac{m \in h(\ell)}{\mathrm{Prototype}(h, \ell, m) = \ell}$$

$$\textsc{Prototype-lookup}\quad \frac{m \notin h(\ell) \qquad \mathrm{Prototype}(h, h(\ell).@prototype, m) = \ell_n}{\mathrm{Prototype}(h, \ell, m) = \ell_n}$$

On top of a scope chain, there is a distinguished object, namely the global object.

Integrator and Gadgets Global Objects We define a (simplified) initial integrator global object below (we use the form $\#addr$ to represent an unique heap location):

$$global_i = \left\{ \begin{array}{ll} @this_{\{\blacklozenge\}} : & \#global_i \\ @scope_{\{\blacklozenge\}} : & null \\ \text{“}Stringify\text{”}_{\{\blacklozenge\}} : & \#stringify_i \\ \text{“}Parse\text{”}_{\{\blacklozenge\}} : & \#parse_i \\ \text{“}PostMessage\text{”}_{\{\blacklozenge\}} : & \#postmessage_i \\ \text{“}Addlistener\text{”}_{\{\blacklozenge\}} : & \#addlistener_i \\ \text{“}window\text{”}_{\{\blacklozenge\}} : & \#global_i \end{array} \right\}$$

Global variables are defined as properties in the global object. For example $window$ is a global variable holding the location $\#global_i$ of the initial global object. Notice that properties in the initial global object are decorated with $\blacklozenge$, which are not considered as heap locations created neither by the integrator nor the gadget.

[2].

Since by SOP the integrator and the frame do not share objects in the heap, we define similarly an initial global object $global_f$ for the frame, in which the properties hold locations $\#global_f$, $\#stringify_f$,..., and $\#addlistener_f$.

$$global_f = \left\{ \begin{array}{ll} @this_{\{\blacklozenge\}} : & \#global_f \\ @scope_{\{\blacklozenge\}} : & null \\ \text{“}Stringify\text{”}_{\{\blacklozenge\}} : & \#stringify_f \\ \text{“}Parse\text{”}_{\{\blacklozenge\}} : & \#parse_f \\ \text{“}PostMessage\text{”}_{\{\blacklozenge\}} : & \#postmessage_f \\ \text{“}Addlistener\text{”}_{\{\blacklozenge\}} : & \#addlistener_f \\ \text{“}window\text{”}_{\{\blacklozenge\}} : & \#global_f \end{array} \right\}$$

Heap locations of the form $\#addr_f$ with a subscript f, as in $\#global_f$, denote native objects that reside in the frame reserved part of the heap, as described by the semantics rules shown later.

Native functions in a heap are represented by locations (e.g. $\#postmessage_i$) as abstract function objects. We use $NativeFuns$ to denote the set of locations of native functions. We give definition for pre-defined native objects existing in an initial heap when initializing an integrator or a frame. These native objects are defined below:

OBJECT PROTOTYPE
$objprot = \{@prototype : null\}$

FUNCTION PROTOTYPE
$funprot = \{@prototype : null\}$

STRINGIFY FUNCTION
$$stringify = \left\{ \begin{array}{ll} @prototype : & \#funprot \\ @call : & true \end{array} \right\}$$

PARSE FUNCTION
$$parse = \left\{ \begin{array}{ll} @prototype : & \#funprot \\ @call : & true \end{array} \right\}$$

POSTMESSAGE FUNCTION
$$postmessage = \left\{ \begin{array}{ll} @prototype : & \#funprot \\ @call : & true \end{array} \right\}$$

ADDLISTENER FUNCTION
$$addlistener = \left\{ \begin{array}{ll} @prototype : & \#funprot \\ @call : & true \end{array} \right\}$$

The prototype objects of object and function are used as default prototypes. We model four native functions defined for marshaling/unmarshaling objects to strings, posting messages, and setting event listeners. We assume that $\mathrm{Alloc}(h, o)$ never allocates those pre-defined heap locations mentioned above. We also use $\oplus$ to denote the union of two disjoint heaps (with non-overlapping addresses).

It is useful to define an initial heap. An initial heap for the integrator (resp. for the frame), denoted by h_{in} (resp. h^f_{in}), is one that contains an element in its domain such that $h_{in}(\#global_i) = global_i$ (for the case of frame $h^f_{in}(\#global_f) = \#global_f$). We give the definition of the initial heaps below:

$$h_{in} = \left\{ \begin{array}{ll} \#global & \mapsto global, \\ \#objprot & \mapsto objprot, \\ \#funprot & \mapsto funprot, \\ \#stringify & \mapsto stringify, \\ \#parse & \mapsto parse, \\ \#postmessage & \mapsto postmessage, \\ \#addlistener & \mapsto addlistener \end{array} \right\}$$

$$h^f_{in} = \left\{ \begin{array}{ll} \#global_f & \mapsto global, \\ \#objprot_f & \mapsto objprot, \\ \#funprot_f & \mapsto funprot, \\ \#stringify_f & \mapsto stringify, \\ \#parse_f & \mapsto parse, \\ \#postmessage_f & \mapsto postmessage, \\ \#addlistener_f & \mapsto addlistener \end{array} \right\}$$

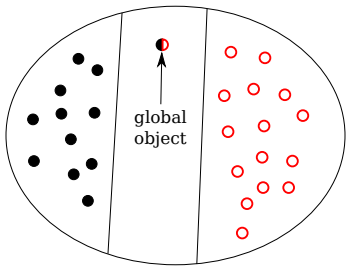


Fig. 2. Example: Uniformly Colored Heap

We say that a decorated object o is single-colored if and only if all properties of o are decorated with the same color. We say that a decorated heap h is uniformly colored if and only if for all $\ell \in \mathsf{dom}(h)$ such that $h(\ell)$ is not a global object, then $h(\ell)$ is a single-colored object (see Figure 2, where solid black dots are ♠-colored objects, hollow red dots are ♥-colored objects). We say that a decorated object o is single-colored if and only if all properties of o are decorated with the same color. The projection $o\lfloor_\square$ for a decorated object o is defined by eliminating non-$\square$ colored properties of o. If there is no property in o with color $\square$ then the projection is undefined and denoted by $\bot$. We define heap projections in order to reason about the portion of the heap owned by a given principal.

Projection $h\lfloor_\square$ is either undefined if there is no property of color $\square$ in h or it is a heap h' such that: $\forall \ell \in \mathsf{dom}(h), h(\ell)\lfloor_\square \neq \bot \Leftrightarrow \ell \in \mathsf{dom}(h')$ & $h'(\ell) = h(\ell)\lfloor_\square$.

Remark 1. If h is a uniformly colored heap, and $h' = h\lfloor_\square$, then for all $\ell \in \mathsf{dom}(h)$ such that $\ell \neq \#global$ or $\ell \neq \#global_f$, $h'(\ell) = h(\ell)$.

We define $h = h'$ as equality on heaps. We denote $h' =_\square h$ for $h'\lfloor_\square = h\lfloor_\square$. We also denote $h' \subseteq_\square h$ for $h'\lfloor_\square \subseteq h\lfloor_\square$.

3.3. Syntax

We present in Figure 3 a simplified syntax of the extension of JavaScript with HTML constructs. We assume that $u \in \mathcal{U}rl$ where $\mathcal{U}rl$ is a set of URLs or origins. A program in the language is an HTML page M with embedded scripts and frames. Frames are important to reason about the SOP and untrusted code. For simplicity, we choose to restrict the language with at most one frame in HTML pages. Inclusion of many frames adds confusion and does not add any insights to the technical results. (This restriction does not apply to the Mashic compiler.) We assume that there is an implicit environment $\mathsf{Web} : \mathcal{U}rl \mapsto J$ that maps URLs to gadgets code. In the frame rule, we model with $\mathsf{Web}(u)$ a gadget from a different origin $u \in \mathcal{U}rl$. In the syntax, scripts are decorated with a color to denote the principal owner of the script. Statements and expressions ranged over by P, s, and e are standard (see e.g. [28]).

Before a JavaScript program in a script node is executed, or before a body of a function is evaluated, all variable declarations are added to the current scope object in the heap. To that end, we use a function VD that returns a heap and takes as parameters a heap h, a location ℓ of the current scope object, a statement s, and a color $\square$ to bind variables declared by $\mathsf{var}\ x$ with proper decorations to the scope object ℓ. Function VD is presented in Figure 4.

M	::=	`<html>` F J `</html>`	HTML page
F	::=	`<iframe src=`u`></iframe>` $\mid \epsilon$	a frame or empty
J	::=	`<script`$\square$`>` s `</script>` $J \mid \epsilon$	sequence of scripts
P, s	::=	e	expression
	$\mid$	$s; s$	block
	$\mid$	var x	variable declaration
	$\mid$	if (e) s else s	conditional
	$\mid$	while (e) s	while loop
	$\mid$	return e	return
e	::=	this	special property
	$\mid$	x	identifier
	$\mid$	f	native functions
	$\mid$	pv	primitive values
	$\mid$	$\{m_0 : e_0, \ldots, m_n : e_n\}$	object literal
	$\mid$	$e_0[e_1]$	member selector
	$\mid$	new $e_0(e_1)$	constructor invocation
	$\mid$	$e_0(e_1)$	function invocation
	$\mid$	function$(\vec{x})\{s\}$	function expressions
	$\mid$	e_0 bin e_1	binary operations
	$\mid$	typeof e	typeof expression
f	::=	$PostMessage \mid AddListener$ $Stringify \mid Parse$	native functions
pv	::=	m	string
	$\mid$	n	number
	$\mid$	b	boolean
	$\mid$	$null$	null
bin	::=	`+` $\mid$ `-` $\mid$ `<` $\mid$ `>` $\mid$ `===` $\mid$ `=`	binary operators

Fig. 3. JavaScript Syntax with Decorations

$$\mathrm{VD}(h, \ell, s, \square) = \begin{cases} h & \text{if } s = e \\ \mathrm{VD}(\mathrm{VD}(h, \ell, s_0, \square), \ell, s_1, \square) & \text{if } s = s_0; s_1 \\ h(\ell.x_{\{\square\}} = \mathit{undefined}) & \text{if } s = \mathsf{var}\ x \\ \mathrm{VD}(h, \ell, s_0; s_1, \square) & \text{if } s = \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1 \\ \mathrm{VD}(h, \ell, s, \square) & \text{if } s = \mathsf{while}\ (e)\ s \\ \mathrm{VD}(h, \ell, s, \square) & \text{if } s = \mathsf{return}\ e \end{cases}$$

Fig. 4. Variable Declaration Function

Finally we define a helper function $\mathrm{GetType}(v)$, to return a string as the type of a primitive value.

$$\mathrm{GetType}(h, v) = \begin{cases} \text{“}\mathit{number}\text{”} & \text{if } v = n \\ \text{“}\mathit{string}\text{”} & \text{if } v = m \\ \text{“}\mathit{boolean}\text{”} & \text{if } v = b \\ \text{“}\mathit{undefined}\text{”} & \text{if } v = \mathit{null} \text{ or } v = \mathit{undefined} \\ \text{“}\mathit{object}\text{”} & \text{if } v = l \text{ and } @\mathit{call} \notin h(\ell) \\ \text{“}\mathit{function}\text{”} & \text{if } v = l \text{ and } @\mathit{call} \in h(\ell) \end{cases}$$

3.4. Configurations

Instrumented global configurations feature a decoration component that denotes the owner principal of the program being executed. Decorations are propagated via semantics rules and, importantly, do not affect the normal semantics of JavaScript programs (they can be erased without further changes in the state). A global configuration is a 5-tuple $\langle \square, h, \ell, R, Q \rangle_x$ that features:

- A subscript x identifying the execution context of current code, I denotes that the current context is the *integrator*, and F denotes that the current context is the *frame*. We use the subscript x to denote a wildcard symbol for both I or F.
- A decoration $\square$ that denotes the principal of the current program in the configuration.
- A heap h.
- A location $\ell \in \mathcal{L}$ pointing to the current scope object (or $null$ only for the initial configuration).
- A run-time program R currently being executed (see Section 3.6).
- A waiting queue Q in order to give semantics to the PostMessage mechanism.

A waiting queue is of the form $\langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle$, where ℓ_i and ℓ_f are locations for event listeners and mq_i and mq_f are message queues for both, the integrator and the frame, respectively. The syntax for defining a message queue is :

$$mq ::= m\ mq \mid \varepsilon$$

where m is a string. We use $mq_1 + mq_2$ to denote the concatenation of two message queues.

An initial configuration is of the form $\langle \square, \varepsilon, null, M, Q_{init} \rangle_I$ where $Q_{init} = \langle null, \varepsilon \rangle \parallel \langle null, \varepsilon \rangle$.

We also define a configuration for the core JavaScript semantics

$$(\square, h, \ell, s)$$

to be a 4-tuple featuring:

- A decoration $\square$ that denotes the principal of the current program in the configuration;
- A heap h;
- A location ℓ of the current scope object;
- A current statement s being evaluated.

Transitions between core JavaScript configurations are featured with a label ℓmq: $(h, \ell, s) \xrightarrow{\ell mq} (h', \ell, s')$. The label ℓmq carries the side-effect of PostMessage and event listeners added by the native function addlistener, and is defined by the following syntax:

$$\ell mq ::= mq \mid \ell + mq$$

where mq is a message queue and ℓ is a mark that denotes that an event listener is added and holds in location ℓ of the heap. Let $\xrightarrow{\ell mq}^*$ be the transitive closure of the transition relation, where ℓmq denotes the accumulated side-effect. A trace of transitions is *valid* only if there is at most one side-effect for *addlistener*.

3.5. Semantics Rules

We use a transition system to define the semantics of our language, via the $\rightarrow$ relation between global configurations. We denote by $\rightarrow^*$ the reflexive and transitive closure of $\rightarrow$. Figure 5 presents rules on the HTML extensions and the SOP property (see frame rules and DSCRIPT). The semantics rules of core JavaScript are defined in a context-redex style in Figure 6,7, and 8.

3.6. DOM Semantics Rules

We extend the syntax with run-time expressions:

$$
\begin{array}{rll}
R ::= & M \mid F\ J \mid F_{RT}\ J \mid J \mid s\ J & \text{run-time programs} \\
F_{RT} ::= & \texttt{<iframe>}\ J\ \texttt{</iframe>} & \text{run-time frames} \\
& \mid \texttt{<iframe>}\ s\ J\ \texttt{</iframe>} & \\
e ::= & \ldots \mid @\mathsf{FunExe}(\ell, s, \Box) \mid @\mathsf{NewExe}(\ell_o, \ell, s, \Box) & \text{run-time expressions} \\
v ::= & pv \mid \ell \mid \mathit{undefined} & \text{run-time values} \\
i ::= & @x \mid m & \text{properties of objects}
\end{array}
$$

In the run-time syntax, R denotes run-time programs being executed, extended by run-time frame F_{RT}. Run-time expressions e are extended with two types of functions $@\mathsf{FunExe}(\ell, s, \Box)$ and $@\mathsf{NewExe}(\ell_o, \ell, s, \Box)$, v denotes run-time values which consist of primitive values pv, heap locations ℓ, and undefined value $\mathit{undefined}$.

We define semantics rules for the DOM, *i.e.* the global transitions, in Figure 5. Now we comment on the semantics rules.

DINIT A mashup execution initializes the heap of the configuration to the initial heap of the integrator h_{in}. The scope object is set to the global object $\#global$.

DSCRIPT A $\triangle$-decorated script starts by $\mathrm{VD}(h, \ell, s, \triangle)$ to initialize variables defined in s to the current scope object ℓ in h. The new configuration has color $\triangle$.

DSCRIPTFINI When an execution of a statement terminates, we continue with the rest of the computation.

DSCRIPT-I-1 This is a contextual rule: if the core JavaScript configuration can advance by 1 step with label mq as the integrator, then the global configuration will accordingly update the message queue mq_f for the frame. If the listener for the frame ℓ_f is not $null$, then we append mq to mq_f, otherwise we do nothing since no listener will respond to incoming messages.

DSCRIPT-I-2 This rule is similar to DSCRIPTFINI except that it sets the listener of the integrator to ℓ' (see the label of core JavaScript transition).

DSCRIPT-F-1, DSCRIPT-F-2 Similar to rule DSCRIPT-I-1 and DSCRIPT-I-2.

DFRAMEINIT, DFRAMEEXEC, DFRAMEFINI These rules are for execution of a frame. A frame fetches the content $\mathsf{Web}(u)$ and joins the initial frame heap h^f_{in} to the current heap. Addresses in h do not overlap with addresses in h^f_{in} by the SOP. Notice that the current scope object is set to the frame's global object.

DCALLBACK-I When no program is executing, we can apply the event listeners to pending messages in the queues (this rule and rule DCALLBACK-F). For example, if the integrator's event listener ℓ_i is not $null$ and the message queue mq_i is not empty, then we can apply the listener to the first message in the queue. Note that the only non-determinism comes from these two rules for event listeners.

DCALLBACK-F See explanation above.

DINIT

$$\langle \square, \varepsilon, null, \texttt{<html>}\ FJ\ \texttt{</html>}, Q_{init} \rangle_I \rightarrow \langle \square, h_{in}, \#global, FJ, Q_{init} \rangle_I$$

DSCRIPT

$$\frac{\mathrm{VD}(h, \ell, s, \triangle) = h'}{\langle \square, h, \ell, \texttt{<script}\triangle\texttt{>}\ s\ \texttt{</script>} J, Q \rangle_x \rightarrow \langle \triangle, h', \ell, s\ J, Q \rangle_x}$$

DSCRIPTFINI

$$\frac{}{\langle \square, h, \ell, v\ J, Q \rangle_x \rightarrow \langle \square, h, \ell, J, Q \rangle_x}$$

DSCRIPT-I-1

$$\frac{(\square, h, \ell, s) \xrightarrow{mq} (\square, h', \ell, s') \qquad mq'_f = \begin{cases} mq_f & \text{if } \ell_f = null \\ mq_f + mq & \text{otherwise} \end{cases}}{\langle \square, h, \ell, s\ J, \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_I \rightarrow \langle \square, h', \ell, s'\ J, \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq'_f \rangle \rangle_I}$$

DSCRIPT-I-2

$$\frac{(\square, h, \ell, s) \xrightarrow{\ell' + mq} (\square, h', \ell, s') \qquad mq'_f = \begin{cases} mq_f & \text{if } \ell_f = null \\ mq_f + mq & \text{otherwise} \end{cases}}{\langle \square, h, \ell, s\ J, \langle null, \varepsilon \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_I \rightarrow \langle \square, h', \ell, s'\ J, \langle \ell', \varepsilon \rangle \parallel \langle \ell_f, mq'_f \rangle \rangle_I}$$

DSCRIPT-F-1

$$\frac{(\square, h, \ell, s) \xrightarrow{mq} (\square, h', \ell, s') \qquad mq'_i = \begin{cases} mq_i & \text{if } \ell_i = null \\ mq_i + mq & \text{otherwise} \end{cases}}{\langle \square, h, \ell, s\ J, \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_F \rightarrow \langle \square, h', \ell, s'\ J, \langle \ell_i, mq'_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_F}$$

DSCRIPT-F-2

$$\frac{(\square, h, \ell, s) \xrightarrow{\ell' + mq} (\square, h', \ell, s') \qquad mq'_i = \begin{cases} mq_i & \text{if } \ell_i = null \\ mq_i + mq & \text{otherwise} \end{cases}}{\langle \square, h, \ell, s\ J, \langle \ell_i, mq_i \rangle \parallel \langle null, \varepsilon \rangle \rangle_F \rightarrow \langle \square, h', \ell, s'\ J, \langle \ell_i, mq'_i \rangle \parallel \langle \ell', \varepsilon \rangle \rangle_F}$$

DFRAMEINIT

$$\frac{\mathsf{Web}(u) = J' \qquad J' \neq \varepsilon}{\langle \square, h, \#global, \texttt{<iframe src=}u\texttt{></iframe>}\ J, Q \rangle_I \rightarrow \langle \square, h \oplus h_f, \#global_f, \texttt{<iframe>}\ J'\ \texttt{</iframe>}\ J, Q \rangle_F}$$

DFRAMEFINI

$$\langle \square, h, \#global_f, \texttt{<iframe>}\ v\ \texttt{</iframe>}\ J, Q \rangle_F \rightarrow \langle \square, h, \#global, J, Q \rangle_I$$

DFRAMEEXEC

$$\frac{\langle \square, h, \#global_f, P, Q \rangle_F \rightarrow \langle \triangle, h', \#global_f, P', Q' \rangle_F}{\langle \square, h, \#global_f, \texttt{<iframe>}\ P\ \texttt{</iframe>}\ J, Q \rangle_F \rightarrow \langle \triangle, h', \#global_f, \texttt{<iframe>}\ P'\ \texttt{</iframe>}\ J, Q' \rangle_F}$$

DCALLBACK-I

$$\frac{\ell_i \neq null}{\langle \square, h, \ell, \varepsilon, \langle \ell_i, m + mq_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_x \rightarrow \langle \square, h, \#global, \ell_i(m), \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_I}$$

DCALLBACK-F

$$\frac{\ell_f \neq null}{\langle \square, h, \ell, \varepsilon, \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, m + mq_f \rangle \rangle_x \rightarrow \langle \square, h, \#global_f, \ell_f(m), \langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle \rangle_F}$$

Fig. 5. Decorated Semantics Rules (DOM)

3.7. Core Semantics Rules

The semantics rules of core JavaScript are defined in a context-redex style in Figure 6,7, and 8. The evaluation contexts of the core JavaScript are defined below, where $op \in \{<, >, +, -, ===\}$:

$$
\begin{array}{rl}
\mathbf{C} & ::= \_ \mid \mathbf{C_i}[\mathbf{C}] \mid \mathbf{C}{=}e \\
\mathbf{C_i} & ::= \mathbf{C_v} \mid \_(e) \\
\mathbf{C_v} & ::= \_[e] \mid l[\_] \mid \mathsf{new}\ \_(e) \mid \mathsf{new}\ l(\_) \\
 & \quad \mid \ l(\_) \mid l[m](\_) \mid \_\ op\ e \mid v\ op\ \_ \\
 & \quad \mid \ \mathsf{typeof}\ \_ \mid x = \_ \mid l[m] = \_
\end{array}
$$

We need to define a special evaluation context $\mathbf{C_i}$ to evaluate redexes that contain an identifier x. For example, in the expression $x = 3$, x should not be evaluated to a value. Therefore $\_{=}e$ is not a $\mathbf{C_i}$ context. Evaluation context $\mathbf{C_v}$ is special for a redex in the form of $\ell[m]$ (a property accessor) to be evaluated to a value. For example, in the expression $\ell[m](e)$, $\ell[m]$ should not be evaluated into a value since it is a method invocation. Therefore $\_(e)$ is not a $\mathbf{C_v}$ context.

Now we explain the transition rules in detail.

DTHIS To resolve the this keyword, we return the $@this$ property of the current scope object ℓ in h.

DOBJ-LITERAL We first allocate a new empty object in h, represented by ℓ_o. Then we evaluate each e_i separately, adding the results v_i as properties m_i of the object in ℓ_o[3]. The result is the location ℓ_o of created object. The properties of the created object is all decorated with $\Box$.

DCALLFUNC To invoke a function, we create a new scope object ℓ_s as current scope object in which the $@scope$ property is set to the function's closure scope $h_1(\ell_1).@fscope$. Furthermore, the $@this$ property of ℓ_s is set to ℓ_g, the global scope object of the current scope chain. $\mathrm{VD}(h_1, \ell_s, s, \triangle)$ initializes local variables defined in the body of the function in ℓ_1 with the decoration of the function object rather than the current decoration in the configuration. The resulting expression $@\mathsf{FunExe}(\ell, s, \Box)$ keeps record of the scope object ℓ to return, and the decoration $\Box$ to recover when the function execution finishes.For simplicity, we present functions with one parameter only. Function GetGlobal look up in the scope chain to get the address of the global object via the window property.

DCALLMETHOD This rule is similar to DCALLFUNC, the different is that we look up thought the prototype chain to obtain the function object ℓ_3 from $\ell_1[m]$, and we set the $@this$ property of ℓ_s is set to ℓ_1, since it is a method call rather than a function call.

DCALLCONTEXT This is a contextual rule for evaluating a body of a function.

DCALLFINI When a function invocation is finished and no value is returned, we restore the scope to ℓ and return $undefined$ as result.

DCALLRET When a function invocation is finished and value v is returned, we restore the scope to ℓ and return v as result.

DNEW The new construct uses a function as a constructor to initialize an object. It behaves method invocation. We first creates an empty object ℓ_o in which the internal $@prototype$ property is set to the "*prototype*" property of the function $h\ell_1$. Then we proceed as in method invocation. The result expression $@\mathsf{NewExe}(\ell_o, \ell, s, \Box)$ keep records of both ℓ_o and ℓ.

DNEWCONTEXT This is a contextual rule for evaluating a body of a function as object initialization.

[3]We do not explicitely mention the heap h when there is no ambiguity.

DTHIS

$$\frac{h(\ell).@this = v}{(\square, h, \ell, \mathbf{C}[\mathsf{this}]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[v])}$$

DOBJ-LITERAL

$$\frac{\begin{array}{c} o = \left\{@prototype_{\{\square\}} : \#objprot\right\} \qquad \mathrm{Alloc}(h, o) = h_1, \ell_o \\ (\square, h_i, \ell, e_i) \xrightarrow{mq_i}{}^{*} (\square, h'_i, \ell, v_1) \qquad h'_i(\ell_o.m_{i\{\square\}} = v_i) = h_{i+1} \qquad mq = mq_1 + mq_2 + \cdots mq_n \end{array}}{(\square, h, \ell, \mathbf{C}[\{m_1 : e_1, \ldots, m_n : e_n\}]) \xrightarrow{mq} (\square, h_{n+1}, \ell, \mathbf{C}[\ell_o])}$$

DCALLFUNC

$$\frac{\begin{array}{c} \ell_1 \notin \mathit{NativeFuns} \qquad h(\ell_1).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \qquad \ell_g = \mathrm{GetGlobal}(h, \ell) \\ o_s = \left\{\begin{array}{ll} @scope_{\{\triangle\}} : & h(l_1).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & \ell_g \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array}\right\} \qquad \mathrm{Alloc}(h, o_s) = h_1, \ell_s \qquad \mathrm{VD}(h_1, \ell_s, s, \triangle) = h_2 \end{array}}{(\square, h, \ell, \mathbf{C}[\ell_1(v)]) \rightarrow (\triangle, h_2, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \square)])}$$

DCALLMETHOD

$$\frac{\begin{array}{c} \mathrm{Prototype}(h, \ell_1, m) = \ell_2 \neq null \\ h(\ell_2).m = \ell_3 \qquad \ell_3 \notin \mathit{NativeFuns} \qquad h(\ell_3).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \\ o_s = \left\{\begin{array}{ll} @scope_{\{\triangle\}} : & h(\ell_3).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & \ell_1 \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array}\right\} \qquad \mathrm{Alloc}(h, o_s) = h_1, \ell_s \qquad \mathrm{VD}(h_1, \ell_s, s, \triangle) = h_2 \end{array}}{(\square, h, \ell, \mathbf{C}[\ell_1[m](v)]) \rightarrow (\triangle, h_2, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \square)])}$$

DCALLCONTEXT

$$\frac{(\square, h, \ell_s, s) \rightarrow (\square, h', \ell'_s, s')}{(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s, \triangle)]) \rightarrow (\square, h', \ell'_s, \mathbf{C}[@\mathsf{FunExe}(\ell, s', \triangle)])}$$

DCALLFINI

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[\mathit{undefined}])$$

DCALLRET

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{FunExe}(\ell, \mathsf{return}\ v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[v])$$

DNEW

$$\frac{\begin{array}{c} o = \left\{@prototype_{\{\triangle\}} : h(\ell_1).\text{“}prototype\text{”}\right\} \\ \mathrm{Alloc}(h, o) = h_1, \ell_o \qquad \ell_1 \notin \mathit{NativeFuns} \qquad h_1(\ell_1).@body_{\{\triangle\}} = \mathsf{function}(x)\{s\} \\ o_s = \left\{\begin{array}{ll} @scope_{\{\triangle\}} : & h_1(\ell_1).@fscope \\ @prototype_{\{\triangle\}} : & null \\ @this_{\{\triangle\}} : & l_o \\ \text{“}x\text{”}_{\{\triangle\}} : & v \end{array}\right\} \qquad \mathrm{Alloc}(h_1, o_s) = h_2, \ell_s \qquad \mathrm{VD}(h_2, \ell_s, s, \triangle) = h_3 \end{array}}{(\square, h, \ell, \mathbf{C}[\mathsf{new}\ \ell_1(v)]) \rightarrow (\triangle, h_3, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s, \square)])}$$

DNEWCONTEXT

$$\frac{(\square, h, \ell_s, s) \rightarrow (\square, h', \ell'_s, s')}{(\square, h, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s, \triangle)]) \rightarrow (\square, h', \ell'_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, s', \triangle)])}$$

DNEWFINI

$$(\square, h, \ell_s, \mathbf{C}[@\mathsf{NewExe}(\ell_o, \ell, v, \triangle)]) \rightarrow (\triangle, h, \ell, \mathbf{C}[\ell_o])$$

Fig. 6. Decorated Semantics Rules (Core JavaScript)

DFUN

$$\frac{p = \{@prototype_{\{\square\}} : \#objprot\} \qquad \mathrm{Alloc}(h, p) = h_1, \ell_1 \qquad o = \left\{\begin{array}{ll} \text{“}prototype\text{”}_{\{\square\}} : & \ell_1 \\ @prototype_{\{\square\}} : & \#funprot \\ @call_{\{\square\}} : & true \\ @fscope_{\{\square\}} : & \ell \\ @body_{\{\square\}} : & \mathsf{function}(x)\{s\} \end{array}\right\} \qquad \mathrm{Alloc}(h_1, o) = h', \ell'}{(\square, h, \ell, \mathbf{C}[\mathsf{function}(x)\{s\}]) \xrightarrow{\varepsilon} (\square, h', \ell, \mathbf{C}[\ell'])}$$

DTYPEOF

$$\frac{\mathrm{GetType}(h, v) = m}{(\square, h, \ell, \mathbf{C}[\mathsf{typeof}\ v]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[m])}$$

DOP

$$\frac{v_1\ op\ v_2 = v}{(\square, h, \ell, \mathbf{C}[v_1\ op\ v_2]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[v])}$$

DASGNIDENT

$$\frac{\mathrm{Scope}(h, \ell, \text{“}x\text{”}) = \ell_n \qquad \ell_g = \mathsf{GetGlobal}(h, \ell) \qquad h_1 = \begin{cases} h(\ell_g.\text{“}x\text{”}_{\{\square\}} = v) & \text{if } \ell_n = null \\ h(\ell_n.\text{“}x\text{”} = v) & \text{otherwise} \end{cases}}{(\square, h, \ell, \mathbf{C}[x = v]) \xrightarrow{\varepsilon} (\square, h_1, \ell, \mathbf{C}[v])}$$

DASGN-NEW-PROPERTY

$$\frac{m \notin h(\ell_1) \qquad h(\ell_1.m_{\{\square\}} = v) = h_1}{(\square, h, \ell, \ell_1[m] = v, Q) \xrightarrow{\varepsilon} (\square, h_1, \ell, v, Q)}$$

DMODIFY-PROPERTY

$$\frac{m_{\{\square\}} \in h(\ell_1) \qquad h(\ell_1.m_{\{\square\}} = v) = h'}{(\triangle, h, \ell, \ell_1[m] = v, Q) \xrightarrow{\varepsilon} (\triangle, h', \ell, v, Q)}$$

DGETVPROP

$$\frac{\mathrm{Prototype}(h, \ell, m) = \ell_2 \qquad v = \begin{cases} undefined & \text{if } \ell_2 = null \\ h(\ell_2).\text{“}x\text{”} & \text{otherwise} \end{cases}}{(\square, h, \ell, \mathbf{C}[\mathbf{C_v}[\ell_1[m]]]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\mathbf{C_v}[v]])}$$

DGETVIDENT

$$\frac{\mathrm{Scope}(h, \ell, \text{“}x\text{”}) = \ell_1 \neq null \qquad v = h(\ell_1).\text{“}x\text{”}}{(\square, h, \ell, \mathbf{C}[\mathbf{C_i}[x]]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\mathbf{C_i}[v]])}$$

DPARSE

$$\frac{\ell_1 = \#Parse \text{ or } \ell_1 = \#Parse_f \qquad o = \mathsf{parse}(m) \qquad \mathrm{Alloc}(h, o) = h_1, \ell_o}{(\square, h, \ell, \mathbf{C}[\ell_1(m)]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[\ell_o])}$$

DSTRINGIFY

$$\frac{\ell_1 = \#Stringify \text{ or } \ell_1 = \#Stringify_f \qquad m = \mathsf{stringify}(h, v)}{(\square, h, \ell, \mathbf{C}[\ell_1(v)]) \xrightarrow{\varepsilon} (\square, h, \ell, \mathbf{C}[m])}$$

DPOSTMSG

$$\frac{\ell_1 = \#Postmessage \text{ or } \ell_1 = \#Postmessage_f}{(\square, h, \ell, \mathbf{C}[\ell_1(m)]) \xrightarrow{m} (\square, h, \ell, \mathbf{C}[undefined])}$$

DADDLISTENER

$$\frac{\ell_1 = \#Addlistener \text{ or } \ell_1 = \#Addlistener_f}{(\square, h, \ell, \mathbf{C}[\ell_1(\ell_i)]) \xrightarrow{\ell_i} (\square, h_1, \ell, \mathbf{C}[undefined])}$$

Fig. 7. Decorated Semantics Rules (Core JavaScript , continued)

DVAR
$(\square, h, \ell, \mathsf{var}\ x) \xrightarrow{\varepsilon} (\square, h, \ell, \mathit{undefined})$

DBLOCKNEXT
$(\square, h, \ell, v\ s*) \xrightarrow{\varepsilon} (\square, h, \ell, s*)$

DBLOCKCONTEXT
$$\frac{(\square, h, \ell, s_0) \xrightarrow{lmq} (\square, h', \ell, s_1)}{(\square, h, \ell, s_0; s) \xrightarrow{lmq} (\square, h, \ell, s_1; s)}$$

DIFTRUE
$$\frac{(\square, h, \ell, e) \xrightarrow{lmq}{}^{*} (\square, h', \ell, \mathit{true})}{(\square, h, \ell, \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1) \xrightarrow{lmq} (\square, h', \ell, s_0)}$$

DIFFALSE
$$\frac{(\square, h, \ell, e) \xrightarrow{lmq}{}^{*} (\square, h', \ell, \mathit{false})}{(\square, h, \ell, \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1) \xrightarrow{lmq}{}^{*} (\square, h, \ell, s_1)}$$

DWHILETRUE
$$\frac{(\square, h, \ell, e) \xrightarrow{lmq} (\square, h', \ell, \mathit{true})}{(\square, h, \ell, \mathsf{while}\ (e)\ s) \xrightarrow{lmq} (\square, h', \ell, s\ \mathsf{while}\ (e)\ s)}$$

DWHILEFALSE
$$\frac{(\square, h, \ell, e) \xrightarrow{lmq} (\square, h', l, \mathit{false})}{(\square, h, \ell, \mathsf{while}\ (e)\ s) \xrightarrow{lmq} (\square, h', \ell, \mathit{undefined})}$$

DRETURN
$$\frac{(\square, h, \ell, e) \xrightarrow{lmq} (\square, h', \ell, v)}{(\square, h, \ell, \mathsf{return}\ e; s) \xrightarrow{\varepsilon} (\square, h', \ell, \mathsf{return}\ v)}$$

Fig. 8. Decorated Semantics Rules (Core JavaScript, continued)

DNEWFINI When an execution of $@\mathsf{NewExe}(\ell_o, \ell, v)$ finished, we restore the scope object to ℓ and return ℓ_o as the result of creating an object.

DFUN To create a new function, we first create an empty prototype object ℓ_p, then we create ℓ' as the function object, where the "*prototype*" property is set to ℓ_p. We keep the current scope object ℓ in the $@fscope$ property of ℓ' as the closure captured by the function definition. We finally returnℓ' as result. The function object is decorated with $\square$, keeping track the owner of the function.

DTYPEOF We use the $GetTypeh, v$ to obtain a string representing the type of v.

DOP We use a conventional interpretation of op.

DASGNIDENT We first look up through the scope chain to check if x is defined in the chain. If it exist in scope object ℓ_n , then we update the property of "x" in ℓ_n, otherwise we create a property "x" in the global object ℓ_g.

DASGN-NEW-PROPERTY To create a property m of ℓ_1, we directly update ℓ_1 in h without following the prototype chain. A decoration is created only when a new property is created and cannot be changed afterward.

Example 2 (Decoration of a new property). Suppose a location ℓ stored in variable x represents the object o in the heap. A program, with decoration ♥, $x["b"] = 3$ results in the decorated object on the right side:

$$o = \left\{ \begin{matrix} a_{\{\heartsuit\}} : 2 \\ c_{\{\spadesuit\}} : 4 \end{matrix} \right\} \Rightarrow \left\{ \begin{matrix} a_{\{\heartsuit\}} : 2 \\ b_{\{\heartsuit\}} : 3 \\ c_{\{\spadesuit\}} : 4 \end{matrix} \right\}$$

DMODIFY-PROPERTY It is similar to DASGN-NEW-PROPERTY. The color of the property in the heap is not changed.

DGETVPROP To access a property of an object, we look up through the prototype chain. The value v could possibly be an location. When the property m does not exist we return $undefined$.
DGETVIDENT To resolve a variable name, we look up through the scope chain.
DPARSE To de-arshal an object we use $\mathsf{parse}(m)$ to reconstruct an object o. Note that it is a special rule for a native function.
DSTRINGIFY To marshal an object we use $\mathsf{stringify}(o)$ to return the string (in JSON format) representation of the object.
DPOSTMSG To post a message m, we use a label m to indicate the side-effect.
DADDLISTENER To set a event listener ℓ_i, we use a label ℓ_i to indicate the side-effect.
DVAR Since var x has already been treated by VD before statement execution, we just skip this statement.
DBLOCKNEXT When a statement evaluates to a value, we continue with the next one.
DBLOCKCONTEXT It is a contextual rule for evaluating sequential composition of statements (block).
DIFTRUE If the condition expression e evaluates to $true$ then we execute the "*then*" branch.
DIFFALSE If the condition expression e evaluates to $false$ then we execute the "*else*" branch.
DWHILETRUE If the condition expression e evaluates to $true$ then we unfold the while body s once.
DWHILEFALSE If the condition expression e evaluates to $true$ then we skip the while body.
DRETURN For return statement, we skip all the rest of the statement s.

Example 3 (Decorated Global Object). Recall variables `secret` and `steal` in Listing 4 and 3 of Section 2. Assuming that the secret input is "yes", by semantics (after execution of the non-benign gadget) the shared global object has the following form:

$$h(\#global_i) = \left\{ \begin{array}{c} \vdots \\ \text{“}price\text{”}_{\{\spadesuit\}} : 0 \\ \text{“}secret\text{”}_{\{\spadesuit\}} : \text{“}yes\text{”} \\ \text{“}steal\text{”}_{\{\heartsuit\}} : \text{“}yes\text{”} \end{array} \right\}$$

If the gadget is sandboxed as in Listing 5, the gadget code gets stuck by the semantics when trying to read "secret" since the variable has not been defined. (In practice, however, the program raises an exception that we do not model in the semantics.)

4. Compilation Overview

In this section we describe in detail how proxy and listener libraries work. For that, we need to define opaque object handles.

4.1. Opaque Object Handle

According to the SOP policy, the integrator and the framed gadget cannot exchange JavaScript references to objects. Our libraries provide a way for the integrator to refer to objects that are defined inside the gadget, called *opaque object handles* [2].

An opaque object handle is essentially an abstract representation of a JavaScript object. In our libraries it is a unique number associated with an object in the frame.

The following code excerpt demonstrates the data type for an opaque object handle:

```
function OHandle(id){
   if (id == undefined) id = handle_id_gen();
   this._id = id;
   this._is_ohandle = true;}
```

In practice, an opaque object handle is an object with a field `_is_ohandle` being true and a field `_id` being the corresponding id. The `handle_id_gen` function generates a unique id. Since the data structure for handles only contains primitive values, they can be exchanged via PostMessage and standard marshaling methods.

On the listener library side, we keep a list for associating handles and objects:

```
var handle_list = {};
function add_handle_obj(ohandle,obj){
 handle_list[ohandle._id] = obj;}
function get_obj_by_handle(ohandle){
 return handle_list[ohandle._id];}
```

Since an object could possibly be an opaque object handle, it is necessary to dynamically check whether the object being operated is an opaque object handle or a local object existing in the integrator. If it is an opaque object handle, we need to proxy the operation to the sandbox; if it is a local object, we can directly operate on this object. We define an `isOpaque` function to do the dynamic check:

```
function isOpaque(obj){
 if ((obj != null) && obj._is_ohandle) return true;
 return false;
}
```

Listing 11: isOpaque Function

Bootstrapping We model the interface provided by a given gadget as a set $\mathcal{V}$ of global variables in the gadget.

Example 4. For instance in our running example, $\mathcal{V} = \{\texttt{gadget}\}$, since `gadget` is the only global variable defined by the gadget. Another example is the interface provided by Google Maps API, that contains only the global variable `google`.

The Mashic compiler inserts bootstrapping scripts on both sides, integrator and gadget. The bootstrapping script for the integrator takes a set of variables $\mathcal{V} = \{x_1, \ldots, x_n\}$ and generates opaque object handles for each of them:

```
var xi = new OHandle(i);
```

Listing 12: Integrator Bootstrapping

The bootstrapping script for the gadget also generates opaque object handles and adds them to a list.

```
add_handle_object(new OHandle(i),xi);
```

Listing 13: Gadget Bootstrapping

In the rest of the paper we let $Bootstrap_i^{\mathcal{V}}$ and $Bootstrap_g^{\mathcal{V}}$ be the bootstrapping scripts for variable set $\mathcal{V}$ for the integrator and the gadget respectively.

Proxy and Listener Interface In the rest of the paper we let P_p denote the proxy library, and P_l the listener library. On the proxy library side, we provide a series of interfaces to obtain an opaque object handle, or operate on it.

To obtain an opaque object handle from a global variable in the gadget, we use the `GET_GLOBAL_REF` interface.

```
1 function GET_GLOBAL_REF(global_name,cont){
2  var m_id = gen_id();
3  var msg = {msg_id : m_id,
4           msg_type : 'GET_GLOBAL_REF',
5           global_name : global_name};
6  PostMessage(stringify(msg));
7  set_cont(m_id,cont);}
```

Listing 14: Code Snippet of the Proxy Library

The `GET_GLOBAL_REF` interface takes two parameters, the `global_name`, and a function `cont` to be used as continuation.

The `GET_GLOBAL_REF` function, upon invocation on the proxy side, composes a message with a fresh message id and sends it to the gadget in iframe. Because of the asynchronous nature of the PostMessage communication, the listener library on the gadget side cannot respond to this message immediately. Hence, we register a continuation `cont` with the message id `m_id`.

There are other interfaces that are supported for operating on opaque objects handles:

- `GET_PROPERTY`: to obtain an opaque object handle or the primitive value of a property of a given object (opaque object handle);
- `OBJ_PROP_ASSIGN`: to assign a primitive value or an object or an opaque object handle to a property of a given object;
- `CALL_FUNCTION`: to call a function (opaque object handle) with all parameters being primitive values, objects or opaque object handles;
- `CALL_METHOD`: to call a method of an object (opaque object handle) with all parameters being primitive values or objects or opaque object handles;
- `NEW_OBJECT`: to instantiate a function object (that is, an opaque object handle) with all parameters being primitive values or objects or opaque object handles.

Example 5. Recall the mashup from Section 2. The interface to obtain an opaque object handle in the integrator is:

```
GET_GLOBAL_REF("gadget", function(val){...});
```

where "gadget" is the interface provided by the gadget and the second parameter is a callback function. Once the integrator obtains an opaque object handle, it can use other interfaces from the integrator to operate on the opaque object handle. If `opq_inst` corresponds to an instance object inside the gadget, to mimic the code of line 4 in Listing 2 we use:

```
CALL_METHOD(opq_inst, "setlevel", function(val){...},9);
```

The interface `CALL_METHOD` sends a message via PostMessage, and waits for a response from the gadget. Once the response arrives, the callback `function(val){...}` is invoked on the returned result. Note that the result might be an opaque object handle as well.

In the listener library, there are interfaces to generate a response as the following function:

```
function GET_GLOBAL_REF_L(recv){
 var obj = window[recv.global_name];
 return make_resp_msg(recv,obj);
}
function make_resp_msg(recv,obj){
 var ohandle, msg;
 if (obj != null &&
    (typeof(obj) == "object" ||
     typeof(obj) == "function"))
     {ohandle = new OHandle();
      add_handle_obj(ohandle,obj);
      msg = {msg_id : recv.msg_id,
           msg_type:'EXE_CONT',
           return_val : ohandle};}
 else {msg = {msg_id: recv.msg_id,
          msg_type:'EXE_CONT',
          return_val : obj};}
 return msg;
}
```

The function `GET_GLOBAL_REF_L` gets the real object by the global name, and generates an opaque object handle if the object is not a primitive value. Then the opaque object handle is sent back to the integrator via PostMessage as a response for the previous sent message. Finally, the associated continuation `cont` will be applied on the response (possibly an opaque object handle).

Here we give details on how interface `CALL_METHOD` works.

1. The integrator invokes `CALL_METHOD(opq_obj,method,cont,args)`, where `opq_obj` stands for the object inside the iframe on which we want to invoke the `method`;`cont` is the continuation; the `args` is possibly a list of arguments.
2. The proxy sends the following message to the listener in iframe:

```
{ msg_id : m_id,
 msg_type: 'CALL_METHOD',
 object : opq_obj,
 method_name: method,
 arguments : args }
```

3. The proxy library associates `m_id` with the continuation `cont`.
4. When the listener receives the message, it first obtains the real `object` corresponding to the handle `obj`; and then it converts the opaque object handles in `arguments` to corresponding objects; and finally it invokes `object[method_name]` on the `arguments`.
5. Once the invocation is finished, it sends back a message:

```
{ msg_id: m_id,
 msg_type:'EXE_CONT',
 return_val : val}
```

   where `val` is either a primitive value or an opaque object handle.
6. Upon receiving the response, the proxy library applies the continuation `cont` with the received result `val`.

4.2. *Integrator Code Transformation*

JavaScript does not support Scheme-style call/cc (Call-with-Current-Continuation) for suspending and resuming an execution. Demanding the programmer to write in CPS would render the proposal impractical.

Example 6. Recall the example in Section 2. In order to obtain the property `gadget.Type.SIMPLE`, the programmer should write the following code (using the proxy interface):

```
1 GET_GLOBAL_REF("gadget",
2   function(opq_gadget){
3     GET_PROPERTY(opq_gadget,"Type",
4       function(opq_Type){
5         GET_PROPERTY(opq_Type,"SIMPLE",
6           function(val_SIMPLE){...});});});
```

This style is similar to CPS, where one needs to explicitly specify continuations for the rest of a computation. In order to reuse the legacy code that operates on a gadget, we propose an automated transformation of legacy code in such a way that programmers *do not need* to rewrite their code. Legacy code in the integrator will be CPS-transformed, and inserted with dynamic checks for opaque object handles when necessary.

We formally define the CPS transformation of an integrator code s, denoted $\mathcal{C}\langle s \rangle$. The function $\mathcal{C} : s \mapsto s$ transforms JavaScript code into CPS. CPS-transformed programs are functions that take as parameter another function as an explicit continuation of the computation. The transformation rules for statements are shown in Figure 9. The CPS transformation rules shown in Figure 10 are standard with respect to the call-by-value lambda calculus. The transformation rules defined in Figure 11 represent interesting and non-standard cases where the proxy library and dynamic checks are inserted into the CPS transformed code. We give explanations for important cases while other rules are similar. For each operation, the compilation inserts dynamic checks to verify whether the object is an opaque object handle or not.

We transform $e_0[e_1]$ to a function taking a parameter $\_k$ as continuation. In the body of this function, we apply the transformed code of e_0 to a continuation where the transformed code of e_1 is applied to an inner-most continuation. In the inner-most continuation $\_x_0$ and $\_x_1$ bind to the results of evaluating e_0 and e_1 respectively. We dynamically check if $\_x_0$ is an opaque object handle to decide whether to use the proxy interface or to apply $\_k$ to $\_x_0[\_x_1]$ directly. Notice that $\_x_1$ can only hold a string, otherwise the execution blocks since in our simplified JavaScript semantics we do not consider type-casting. The transformation of the expression new $e_0(e_1)$ is trickier. This is due to the semantics of new and its return value. Its semantics is similar to calling a function except that the return value is not the result of evaluating the function but the newly created object. Directly supplying the continuation $\_k$ to evaluation $\_x_0$ of e_0, as in the case for $e_0[e_1]$ will not work since the returned value must be a reference and not the result of the function. In the inner-most continuation, we first create a dummy function $\_x_3$ with the same prototype as the object obtained from e_0 in order to simulate the return value of an object reference. Then we create an empty object $\_x_2$ with this dummy function. Next we create a continuation $\_x_4$ with parameter $\_v$ where $\_k$ is always applied to $\_x_2$, no matter what is the parameter $\_v$. Finally, we apply $\_x_0$ to $\_x_1$, to simulate function e_0 execution, using $\_x_4$ as continuation and binding $\_x_2$ to *this* keyword (to initialize properties in $\_x_2$) via $\_x_4$. Notice that the continuation will be always applied to $\_x_2$, and as in new $e_0(e_1)$, will return the created object rather than the result of the function invocation.

$\mathcal{C}\langle s_0; s_1\rangle$:
```
function(_k){
  C⟨s0⟩(function(_v){C⟨s1⟩(_k);});
}
```

$\mathcal{C}\langle$if (e) s_0 else $s_1\rangle$:
```
function(_k){
  C⟨e⟩(function(_b){
        if (_b) C⟨s0⟩(_k)
          else C⟨s1⟩(_k);
      });
}
```

$\mathcal{C}\langle$while (e) $s\rangle$:
```
function(_k){
  var _c;
  _c = function(_v){
        C⟨e⟩(function(_b){
              if (_b) C⟨s⟩(_c)
                else _k(undefined);
            });
      };
  _c(undefined);
}
```

$\mathcal{C}\langle$return $e\rangle$:
```
function(_k){C⟨e⟩(_fun_cont)}
```

Fig. 9. Transformation of Statements

$\mathcal{C}\langle$this$\rangle$:
```
function(_k){_k(_this);}
```

$\mathcal{C}\langle pv\rangle$:
```
function(_k){_k(pv);}
```

$\mathcal{C}\langle x\rangle$:
```
function(_k){_k(x);}
```

$x = e$
```
function(_k){
  C⟨e0⟩(function(_x0){
    _k(x = _x0);
  });
}
```

e_0 *op* e_1
```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      _k(_x0 op _x1);
    });
  });
}
```

typeof e
```
function(_k){
  C⟨e0⟩(function(_x0){
    _k(typeof _x0);
  });
}
```

$\{m_0 : e_0, m_1 : e_1\}$
```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      _k({m0 : _x0, m1 : _x1});
    });
  });
}
```

function$(x)\{s\}$
```
function(_k){
  _k(function(_fun_cont, x){
    var _this;
    _this = this;
    C⟨s⟩(_fun_cont);
  });
}
```

Fig. 10. Transformation of Expressions, Non-Message-Passing part

4.3. Overall Picture

In order to state the theorem, we define decorations for original and compiled mashups. In the original mashup we decorate the integrator as ♠ and the gadget as ♥.

Definition 1 (Decorated Original Mashup)**.** Let P_i be an integrator script and P_g be a gadget script. We define the original mashup $\tilde{M}(P_i, P_g)$ to be:

```
<html>
    <script♥> Pg </script>
    <script♠> Pi </script>
</html>
```

In the compiled mashup we decorate the run-time libraries as ♦. The run-time libraries are marked as neutral color ♦ since we show with the correctness theorem that the integrator's heap is preserved in the original and compiled version. The runtime libraries do not appear in the original heap.

$\mathcal{C}\langle e_0[e_1]\rangle$:

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      if (isOpaque(_x0)){
        GET_PROPERTY(_x0, _x1, _k);
      } else {
        _k(_x0[_x1]);
      }
    }); }); }
```

$\mathcal{C}\langle \text{new } e_0(e_1)\rangle$:

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      if (isOpaque(_x0)){
        NEW_OBJECT(_x0, _x1, _k);
      } else {
        var _x2, _x3, _x4;
        _x3 = function(x){};
        _x3["prototype"] = _x0["prototype"];
        _x2 = new _x3();
        _x4 = function(_v){_k(_x2); };
        _x2["_fun"] = _x0;
        _x2["_fun"](_x4, _x1);
      }
    }); }); }
```

$\mathcal{C}\langle e_0(e_1)\rangle$:

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      if (isOpaque(_x0)){
        CALL_FUNCTION(_x0, _x1, _k);
      } else {
        _x0(_k, _x1);
      }
    }); }); }
```

$\mathcal{C}\langle e_0[e_1](e_2)\rangle$:

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      C⟨e2⟩(function(_x2){
        if (isOpaque(_x0[_x1])){
          CALL_METHOD(_x0, _x1, _x2, _k);
        } else {
          _x0[_x1](_k, _x2);
        }
      }); }); }); }
```

$e_0[e_1] = e_2$

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      C⟨e2⟩(function(_x2){
        if (isOpaque(_x0[_x1])){
          PROPERTY_ASSIGN(_x0, _x1, _x2, _k);
        } else {
          _k(_x0[_x1] = _x2);
        }
      }); }); }); }
```

Fig. 11. Transformation of Expressions, Message-Passing Part

Definition 2 (Mashic Compilation)**.** Let P_i be an integrator script, P_g be a gadget script, $\mathcal{V}$ be a set of variables denoting global names exported by the gadget script. We define the Mashic compilation $\tilde{M}_c(P_i, P_g, \mathcal{V})$ to be:

```
<html>
    <iframe src=u></iframe>
    <script♦> P_p; Bootstrap_i^V </script>
    <script♠> C⟨P_i⟩(function(_x){_x}) </script>
</html>
```

where

$$\mathsf{Web}(u) = \begin{array}{l} \texttt{<script♦>}\ P_l\ \texttt{</script>} \\ \texttt{<script♥>}\ P_g\ \texttt{</script>} \\ \texttt{<script♦>}\ \mathit{Bootstrap}_g^{\mathcal{V}}\ \texttt{</script>} \end{array}$$

We formally define the bootstrapping script that appears in Section 2.

Definition 3 (Bootstrapping Script)**.** Given a set of variables $\mathcal{V} = \{x_0, \ldots, x_n\}$, the Mashic bootstrapping script for an integrator $Bootstrap_i^{\mathcal{V}}$ is defined, for $i \in \{0 \ldots n\}$, as

$$\mathsf{var}\ x_i; x_i = \mathsf{new}\ OHandle(i);$$

and the bootstrapping script for a gadget $Bootstrap_g^{\mathcal{V}}$ is:

$$add\_handle\_obj(\mathsf{new}\ OHandle(i), x_i);$$

5. Correctness Theorem

In this section we formally present the correctness theorem and its assumptions.

5.1. Preliminary definitions

Correct Marshaling We define the notion of correct marshal and unmarshal functions w.r.t. to a set of objects S. Intuitively this definition states that the process of marshaling and then unmarshaling an object preserves the structure of the object in the heap and preserves values that are not locations.

Definition 4 (Correct marshal/unmarshal for S)**.** Let $\sim$ be defined as $v \sim v'$ in h iff there exists a bijection β such that $v, v' \notin \mathcal{L}$ and $v = v'$ or $v, v' \in \mathcal{L}$ and $\beta(v) = v'$ and for every property p in $h(v)$, $h(v).p \sim h(v').p$. Given two functions f and f^{-1}, we say that they are correct for a set of objects S if for all $o \in S$, heap h, and $f^{-1}(f(o)) = o'$ we have o' satisfies: for every property p in o, $o.p \sim o'.p$ in h.

Definition 4 is useful for the correctness theorem of the compiler. It captures the weakest hypothesis possible for the correctness theorem to hold. Following this hypothesis, implementation of marshaling/unmarshaling functions may vary. In the current prototype of the Mashic compiler we implement these functions with JSON stringify and parse, which do not preserve the structure of the objects if the structure contains a cycle. Thus, these functions are considered correct only if the set S of objects to be marshaled does not contain objects with cyclic structures. We have chosen JSON stringify/parse for efficiency reasons. However, it is straightforward to write correct marshaling/unmarshaling functions for a set of objects that also contain cycles in their structures.

Benign Gadget Intuitively, a benign gadget P_g does not rely on the integrator's portion (marked by ♠) and the neutral portion (marked with ♦) of the heap. Furthermore the evaluation of P_g does not depend on any part of the heap except for the initial heap.

In order to state the definition we first define a benign gadget heap as a heap that contains gadget functions with confidentiality and integrity properties.

Definition 5 (Benign Gadget Heap)**.** A heap h_g is benign if and only if for any heaps h_0, h_1 such that $h_j \lfloor_{\heartsuit} = h_g$ ($j \in \{0, 1\}$), for any function located in $\ell \in \mathsf{dom}(h_g)$, for any ℓ' such that $h_0(\ell') = h_1(\ell')$ is an object, and $(\spadesuit, h_j, \ell_i, \ell(\ell')) \rightarrow^* (\spadesuit, h'_j, \ell'_j, v'_j)$, the following conditions holds:

1. $v'_0 = v'_1$;
2. (integrity) $h_j =_{\spadesuit} h'_j$ and $h_j =_{\blacklozenge} h'_j$;
3. (confidentiality) $h'_0 \lfloor_{\heartsuit} = h'_1 \lfloor_{\heartsuit}$;

4. (preservation of benignity) $h_1' \lfloor_{\heartsuit}$ is benign

Example 7 (Benign Heap). Recall the integrator's code in Listing 3 in Section 2. If the gadget contains the following code, then the gadget will not produce a benign gadget heap:

```
var rungadget;
rungadget = function(x){
            var steal;
            steal = secret;
            price = 0;
};
```

Listing 15: Non-benign Gadget Heap

The gadget defines a function in the heap which tries to read from the global variable `secret` and tries to write into the global variable `price`. Calling the function from the integrator will violate the integrity and confidentiality requirement.

Definition 6 (Benign Gadget)**.** Program P_g is benign if and only if for any heaps h_i ($i \in \{0, 1\}$) such that $h_i \lfloor_{\heartsuit} = \emptyset$ and $(\heartsuit, h_i, \ell, P_g) \rightarrow^* (\heartsuit, h_i', \ell, v_i)$, the following conditions hold:

1. (integrity) $h_i =_{\spadesuit} h_i'$ and $h_i =_{\blacklozenge} h_i'$;
2. (confidentiality) $h_0' =_{\heartsuit} h_1'$;
3. $h_0' \lfloor_{\heartsuit}$ is benign.

Example 8 (Benign Gadget Example). Recall the example in Section 2, Listing 4. The gadget is not benign since it tries to read from the global variable `secret` and tries to write into the global variable `price`.

In the benign gadget definition we explicitly require that the initialization phase (adding functions to the heap) and execution of all functions (that are defined in the heap) always terminate.

It is possible to relax this definition by not requiring termination of benign gadgets (by using indistinguishability invariants for intermediate running expressions) but we consider more appropriate to see non-terminating behavior in gadgets as non-benign behavior since the gadget will never let the integrator execute. Hence if the gadget is non-terminating we do not offer any correctness guarantees (security guarantees still apply).

Notice that the termination requirement on gadgets does not imply termination of the mashup. The mashup might never terminate if gadget and integrator continuously run listener continuations and this is independent of termination of functions in gadgets (see e.g. fair termination [7]).

Correct Integrator For correctness, we impose some reasonable restrictions on the integrator's code. Intuitively, a correct integrator does not modify directly a non-$\blacklozenge$-colored property; and does not use objects defined by gadgets in the prototype chain. This restriction is not limiting in practice since an integrator usually operates on gadgets via the interfaces provided by it and not by directly modifying its properties. Given marshal/unmarshal functions, we also require that a correct integrator only sends to gadgets objects for which these functions are correct.

First, we give a notion of reachability of a location from a global variable in a given heap h.

Definition 7 (Reachability)**.** A location l is reachable from a variable x in h if and only if either:

– $h(@global).x = l$; or

– $\exists p$ such that l is reachable from $h(h(@global).x).p$.

Now we give the definition for correct integrator.

Definition 8 (Correct Integrator for f,f^{-1},$\mathcal{V}$)**.** Program P_i is a correct integrator, if and only if, for any benign heap h_g such that $(\spadesuit, h_{in} \oplus h_g, \#global, P_i)$ reaches a redex e and a heap h, then the following conditions hold:

1. If e is of the form $x = v$ and $\text{Scope}(h, \ell, \text{“}x\text{”}) = \ell_n$, then either $\ell_n = null$ or “x” is a ♠-colored property of $h(\ell_n)$.
2. If e is of the form $\ell'[m] = v$, then $h(\ell')$ is a ♠-single-colored object.
3. For any ℓ such that $\ell \in \mathsf{dom}(h|_{\spadesuit})$ and $h(\ell).@prototype = \ell_n$, either $\ell_n = null$ or $h(\ell_n)$ is a ♠-colored object.
4. If e is of the form $\ell_f(\ell')$ and $h(\ell_f)$ is a ♥-colored function, then $h(\ell')$ is an object correct for f and f^{-1} and ℓ_f is reachable from $\mathcal{V}$ in h.

Example 9 (Correct integrator prototype chain). We illustrate why an integrator's object cannot have a gadget's object as its prototype object (bullet 3). Assume that in the heap of the original mashup, $h(\ell_i)$ is a ♠-colored object, and $h(\ell_g)$ is a ♥-colored object such that

$$h(\ell_i) = \{@prototype : \ell_g\} \qquad\qquad h(\ell_i) = \{a : 3\}$$

By reading the property “a” of ℓ_i in the original mashup we get 3. The heap of the compiled code will contain a pointer to a handle:

$$h(\ell_i) = \{@prototype : \ell_o\} \qquad\qquad h(\ell_o) = \left\{ \begin{array}{ll} \text{“}\_id\text{”} & n \\ \text{“}\_is\_ohandle\text{”} & true \end{array} \right\}$$

Hence, by reading the property “a” of ℓ_i in the compiled mashup we do not get value 3.

5.2. Indistinguishability and Correctness

To define indistinguishability between the original heap and compiled heap, the structure of the scope chain in the heap must be preserved. We start by defining the notion of scope object and scope chain. We use $\#global$ as the address of the original global object.

Definition 9 (Scope Object)**.** Let h be a heap, and ℓ be a location for a scope object. We say ℓ is a scope object in h if one of the following conditions is satisfied:

1. $\ell = \#global$, and $h(\ell).@scope = null$;
2. $\ell \neq \#global$, $h(\ell).@scope = \ell' \neq null$, and ℓ' is also a scope object in h.

Definition 10 (Scope Chain)**.** Let h be a heap, and ℓ_1 be a scope object in h. We say that $\ell_1\ell_2 \ldots \ell_n$ is the scope chain of ℓ_1 in h, if

1. For $i < n$, $h(\ell_i).@scope = \ell_{i+1}$;
2. $h(l_n).@scope = null$

We use $\ell \in \ell_1\ell_2 \ldots \ell_n$ to denote that scope object ℓ is included in the scope chain $\ell_1\ell_2 \ldots \ell_n$ w.r.t some heap h.

We define the β-indistinguishability $\sim_\beta$ on values, objects, and scope chains, where $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ is a partial injective function between heap locations.

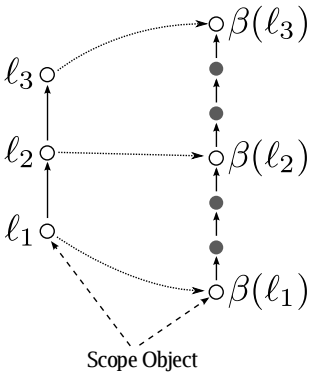


Fig. 12. Example: Scope Indistinguishability

Definition 11 (Scope Chain Indistinguishability)**.** Let ℓ_1 be a scope object in h and ℓ'_1 be a scope object in h', and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. Let $\ell_1\ell_2 \ldots \ell_n$ be the scope chain of ℓ_1 in h, let $\ell'_1\ell'_2 \ldots \ell'_m$ be the scope chain of ℓ'_1 in h'. We say that the two scope chains are indistinguishable, denoted $(h, \ell_1) \approx_\beta (h', \ell'_1)$ if and only if:

1. $\beta(\ell_1)\beta(\ell_2) \ldots \beta(\ell_n)$ is a sub-sequence of $\ell'_1\ell'_2 \ldots \ell'_m$;
2. for $\ell \notin \beta(\ell_1)\beta(\ell_2) \ldots \beta(\ell_n)$, and $\ell \in \ell'_1\ell'_2 \ldots \ell'_m$, $\forall i \in \mathsf{dom}(h'(\ell))$, $i \in \{@scope, @prototype, @this, \text{“\_}k\text{”}, \text{“\_}l\text{”}, \text{“\_}m\text{”}, \text{“\_}x_i\text{”}\}$

The intuition of scope indistinguishability is that the structure of scope chains is preserved by the integrator transformation (even if scope chains do not have a one to one correspondence), as illustrated in Fig. 12. In the figure, scope objects are represented by round points, and the solid arrows represent the scope chain. The scope chain on the left is obtained by a normal execution of integrator code. The scope chain on the right is obtained by execution of the corresponding transformed code, where there are more CPS-administrative scope objects (gray-colored in the figure). The scope indistinguishability does not take into consideration those CPS-administrative scope objects.

Two values are indistinguishable either if they are equal or if they are both locations related by β. Even assuming a deterministic allocator, we need β to relate two heaps because objects created in the original mashup and compiled mashup will be necessarily different. In particular, the compiled heap will contain more objects.

Definition 12 (Value Indistinguishability)**.** Let v_1 and v_2 be two values, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. Value indistinguishability is defined as follows:

$$\frac{v \notin \mathcal{L}}{v \sim_\beta v} \qquad\qquad \frac{v_1, v_2 \in \mathcal{L} \qquad \beta(v_1) = v_2}{v_1 \sim_\beta v_2}$$

Objects are related if they have the same properties with the same values. Exceptions to this are properties $\{@scope, @fscope, @this\}$ and function objects. Properties $\{@scope, @fscope\}$ are related via the scope chain indistinguishability as explained above. Function objects are indistinguishable if the $@body$ property contains the same code in its original and compiled form.

Definition 13 (Object indistinguishability)**.** Let o_1 and o_2 be two objects, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. We say $o_1 \simeq_\beta o_2$, if for every $i \in \mathsf{dom}(o_1)$ one of the following holds:

1. $i \in \{@scope, @fscope, @this\}$;
2. $i \notin \{@body, @scope, @fscope, @this\}$ and if $o_1.i \in \mathsf{dom}(\beta)$ then $o_1.i \sim_\beta o_2.i$;
3. $i = @this$ then $o_1.@this \sim_\beta o_2.\text{“}\_this\text{”}$;
4. $i = @body$ then $o_1.@body = \mathsf{function}(x)\{s\}$, then $@body \in \mathsf{dom}(o_2)$ and $o_2.@body = \mathsf{function}(\_fun\_cont, x)\{s\}$, where

$$
\begin{array}{l}
s = \mathsf{var}\ \_this; \\
\quad \_this = \mathsf{this}; \\
\quad (\mathcal{C}'\langle s \rangle)(\_fun\_cont)
\end{array}
$$

We give an example illustrating object indistinguishability.

Example 10 (Object indistinguishability). Let o_1, o_2 o_3 be:

$$
o_1 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \ell_1 \\ @scope: & \ell_2 \\ @this: & \ell_3 \end{array} \right\} \qquad
o_2 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \beta(\ell_1) \\ @scope: & \ell'_2 \\ \text{“}\_this\text{”}: & \beta(\ell_3) \end{array} \right\} \qquad
o_3 = \left\{ \begin{array}{ll} a: & 2 \\ b: & \ell'_1 \\ @scope: & \ell'_2 \\ \text{“}\_this\text{”}: & \beta(\ell_3) \end{array} \right\}
$$

If $\ell'_1 \neq \beta(\ell_1)$ and $\ell_2 \neq \ell'_2$, then we have $o_1 \simeq_\beta o_2$ and $o_1 \not\simeq_\beta o_3$. We do not compare the $@scope$ property between o_1 and o_2; but we do compare property b between o_2 and o_3.

Finally, heaps are indistinguishable if all objects are indistinguishable and respective scope chains are indistinguishable.

Definition 14 (Heap indistinguishability)**.** We say that (h_1, ℓ_1) and (h_2, ℓ_2), are indistinguishable with respect to $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $\mathsf{dom}(\beta) = \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$, denoted $(h_1, \ell_1) \simeq_\beta (h_2, \ell_2)$, if and only if:

1. $h_1(\ell) \simeq_\beta h_2(\beta(\ell))$ for every $\ell \in \mathsf{dom}(\beta)$
2. if $\ell \in \mathsf{dom}(\beta)$ and $h_1(\ell)$ has the $@body$ property, then $(h_1, h_1(\ell).@fscope) \approx_\beta (h_2, h_2(\beta(\ell)).@fscope)$
3. $(h_0, \ell_0) \approx_\beta (h_1, \ell_1)$.

The correctness theorem gives strong guarantees if the gadget is benign: behavior of original and compiled mashup are equivalent in terms of the integrator's heap. If the gadget is not benign there are no correctness guarantees but only security guarantees described in the following section. We use in the hypothesis that integrator and gadget do not declare the same variables $\mathsf{var}(P_i) \cap \mathsf{var}(P_g) = \emptyset$, where var is defined by:

$$
\mathsf{var}(s) = \begin{cases} \emptyset & \text{if } s = e \text{ or } s = \mathsf{return}\ e \\ \mathsf{var}(s_0) \cup \mathsf{var}(s_1) & \text{if } s = s_0; s_1 \text{ or } s = \mathsf{if}\ (e)\ s_0\ \mathsf{else}\ s_1 \\ \mathsf{var}(s) & \text{if } s = \mathsf{while}\ (e)\ s \\ \{x\} & \text{if } s = \mathsf{var}\ x \end{cases}
$$

Notice that this definition of var refers only to declared variables, and the hypothesis does not assume that integrator and gadget do not share variables.

In the following, let $\tilde{M}_c$ be the Mashic compiler using f and f^{-1} for marshaling and unmarshaling. Let $\mathcal{V}$ be a set of names used by the integrator as the gadget interface.

Theorem 1 (Correctness)**.** *Let P_i be a correct integrator for $f, f^{-1}, \mathcal{V}$ and P_g be a benign gadget such that* $\mathsf{var}(P_i) \cap \mathsf{var}(P_g) = \emptyset$*. If* $\langle \spadesuit, \varepsilon, null, \tilde{M}(P_i, P_g), Q_{init} \rangle_I \rightarrow^* \langle \square, h_0, \ell_0, \varepsilon, Q_{init} \rangle_x$ *then,*

$$\langle \spadesuit, \varepsilon, null, \tilde{M}_c(P_i, P_g, \mathcal{V}), Q_{init} \rangle_I \rightarrow^* \langle \square, h_1, \ell_1, \varepsilon, Q_1 \rangle_x$$

where Q_1 has no message waiting, and there exists β such that

$$(h_1 \lfloor_{\spadesuit}, \ell_1) \simeq_\beta (h_0 \lfloor_{\spadesuit}, \ell_0)$$

The proof proceeds in two stages by means of an intermediate compilation and by structural induction on programs. The behavior of the original mashup is indistinguishable from that of the intermediate compilation; and the intermediate compilation behaves indistinguishably from the Mashic compilation.

5.3. Auxiliary Definitions and Lemmas

In this section we give some auxiliary definitions and some useful lemmas to prove the main theorem. First we define the notion of intermediate compilation in which the gadget is not sandboxed by an iframe and the integrator is compiled to CPS-only code without using the proxy and listener interface. The intermediate compilation will be used in the proofs.

Definition 15 (Decorated Intermediate Compilation)**.** We define decorated intermediate compilation $\tilde{M}_i(P_i, P_g)$ as the follows:

```
<html>
    <script♥> P_g </script>
    <script♠> C'⟨P_i⟩(function(_x){_x}) </script>
</html>
```

where $\mathcal{C}'\langle\rangle$ is an intermediate CPS-transformation.

The intermediate CPS transformation is identical to the Mashic CPS transformation except for the rules shown in Figure 13, where the proxy and the listener library are not used, since the gadget is not sandboxed.

The following lemma shows that the Mashic compilation and the intermediate compilation preserve correctness of the integrator, since they will not introduce more behavior.

Lemma 2. *If P_i is a correct integrator, then $\mathcal{C}\langle P_i \rangle$ and $\mathcal{C}'\langle P_i \rangle$ are both correct integrators.*

Proof. Straightforward by structural induction on definition of P_i. □

We define the notion of *strong object indistinguishability*, denoted $\sim_\beta$, where we have a stronger condition in item 2 when comparing to object indistinguishability. The technical intuition for strong object indistinguishability is that the execution of an intermediate compilation and a mashic compilation of an integrator P do have a strong similarity due to the same structure in CPS transformed code, allowing us to prove a stronger lemma for the intermediate compilation.

Definition 16 (Strong Object Indistinguishability)**.** Let o_1 and o_2 be two objects, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. We say $o_1 \sim_\beta o_2$, if for every $i \in \mathsf{dom}(o_1)$ one of the following holds:

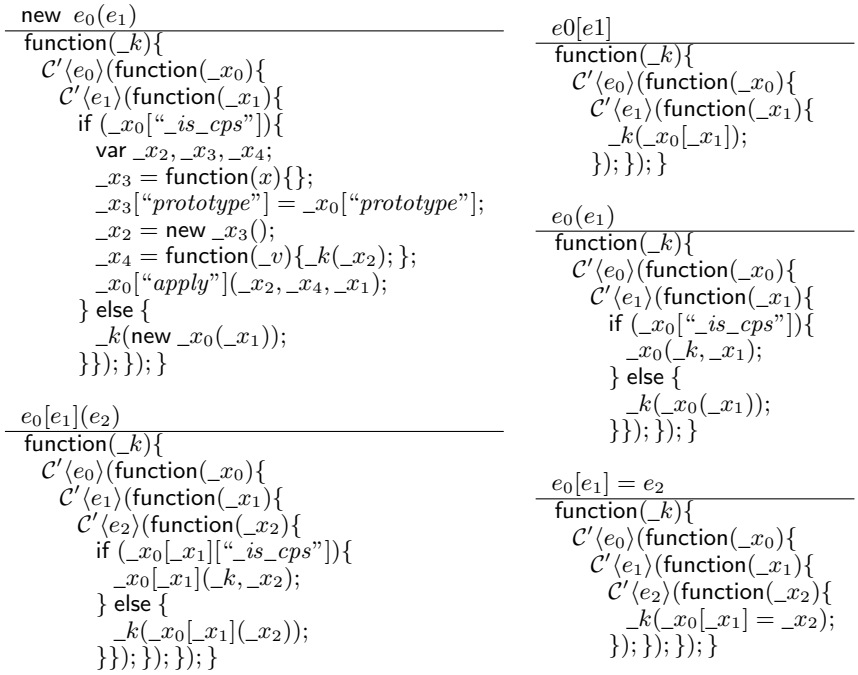


Fig. 13. CPS Transformation (Intermediate)

1. $i \in \{@scope, @fscope, @this\}$;
2. $i \notin \{@body, @scope, @fscope, @this\}$ and if $o_1.i \in \mathsf{dom}(\beta)$ then $o_1.i \sim_\beta o_2.i$ otherwise $o_1.i = o_2.i$.
3. $i = @this$ then $o_1.@this \sim_\beta o_2.$"_this";
4. $i = @body$ then $o_1.@body = \mathsf{function}(x)\{s\}$, then $@body \in \mathsf{dom}(o_2)$ and $o_2.@body = \mathsf{function}(\_fun\_cont, x)\{s_{cps}\}$, where

$$
\begin{aligned}
s_{cps} = {} & \mathsf{var}\ \_this; \\
& \_this = \mathsf{this}; \\
& (\mathcal{C}'\langle s\rangle)(\_fun\_cont)
\end{aligned}
$$

Accordingly, we update the definition of *strong heap indistinguishability*.

Definition 17 (Strong Heap indistinguishability)**.** Two pairs of heap and scope object (h_1, ℓ_1) and (h_2, ℓ_2), are indistinguishable with respect to a partial injective function $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ such that $\mathsf{dom}(\beta) = \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$, denoted $(h_1, \ell_1) \sim_\beta (h_2, \ell_2)$, if and only if:

1. for every $\ell \in \mathsf{dom}(\beta)$ with $o_1 = h_1(\ell)$ and $o_2 = h_2(\beta(\ell))$:
   (a) $o_1 \sim_\beta o_2$
   (b) if o_1 has the $@body$ property, then $(h_1, o_1.@fscope) \approx_\beta (h_2, o_2.@fscope)$
2. $(h_1, \ell_1) \approx_\beta (h_2, \ell_2)$.

The following lemma shows that given two indistinguishable scope chains, the scope looking-up process will return indistinguishable heap locations for any normal variable. By normal variable we mean variables that do not start with a "_". Special case applies to resolving the $@this$ identifier.

Lemma 3 (Scope look-up)**.** *Let h, g be two heaps, and ℓ, j be two locations for scope objects. If $h \sim_\beta g$ and $(h, \ell) \sim_\beta (g, j)$, then the following holds:*

1. *if $i \neq$ @this,* $\text{Scope}(h, \ell, i) = \ell_1$ *then* $\text{Scope}(g, j, i) = j_1$ *and* $\beta(\ell_1) = j_1$*;*
2. *if* $\text{Scope}(h, \ell, @this) = \ell_1$ *then* $\text{Scope}(g, j, \text{“\_this”}) = j_1$ *and* $\beta(\ell_1) = j_1$*;*

Proof. Straightforward by Definition 11 and Definition of $\text{Scope}(\_, \_, \_)$ in semantics rules. □

We formally define the shape of an opaque object handle.

Definition 18 (Opaque Object Handle)**.** Let o be an object, o is an opaque object handle with id n if and only if

$$o = \left\{ \begin{array}{ll} \text{“\_id”} & n \\ \text{“\_is\_ohandle”} & true \end{array} \right\}$$

We define a relation between two heaps up to a mapping from id of opaque object handles to heap locations. The intuition is that if in one heap, a property points to an opaque object handle, then in the other heap, it must point to a location corresponding to the opaque object handle by the mapping.

Definition 19. Let $f : \mathcal{N} \mapsto \mathcal{L}$ be a partial injective function from numbers to locations. We say that $h_c \overset{f}{=} h_i$ if

1. $h_c =_{\heartsuit} h_i$;
2. $\text{dom}(h_c \lfloor_{\spadesuit}) = \text{dom}(h_i \lfloor_{\spadesuit})$;
3. $\forall \ell \in \text{dom}(h_c \lfloor_{\spadesuit}), o_c = h_c(\ell)$ and $o_i = h_i(\ell)$, such that
   (a) if $o_c.i_{\{\spadesuit\}} = \ell_o$, and $h_c(\ell_o)$ is an opaque object handle with id n, then $o_i.i = f(n)$;
   (b) otherwise $o_c.i_{\{\spadesuit\}} = o_i.i_{\{\spadesuit\}}$.

6. Security Theorem

In this section we present the security theorem. In Mashic compiled code, the integrator has complete access to gadget resources but the gadget only has access to resources offered by the integrator in the proxy library. After Mashic compilation, the malicious gadget cannot scan properties of the integrator, as e.g. in Listing 4, because the SOP policy prevents the framed gadget from accessing the JavaScript execution environment of the integrator as shown in the DFRAMEINIT rule in Figure 5.

Example 11 (Gadget modifies native functions). A native function that can commonly appear in the integrator code is the `setTimeout` function. This function takes two parameters. The first one is a function that will be executed when the time (in milliseconds) specified in the second parameter has passed:

```
1 setTimeout("alert(timeout!!)",5);
```

In this example, after 5ms a pop-up window with caption “timeout!” appears.

This function, as all native JavaScript functions, is associated as a property of the global object. As many native functions the code associated to the `setTimeout` function can be changed at execution time, changing in this way the assumed behavior for `setTimeout`.

Suppose the untrusted gadget owned by the attacker writes a function of its own into the `setTimeout` property:

```
1 setTimeout=function (x,y) { evil code here} ;
```

Then every call to `setTimeout` in the integrator's code will be calling the attacker's code with the integrator privileges.

If instead the gadget is enclosed in a frame, the same code trying to affect the `setTimeout` property of the global object will only affect the property of the global object of the frame, that is in a disjoint part of the heap according to the SOP.

In order to state the security guarantee, we consider that all code coming from origin u is part of the gadget principal ♥. In contrast to the decorations used for correctness, we now consider the listener library and bootstrapping as gadget's code. This should not be surprising since the gadget can modify this code and the security theorem must be valid also in this case. We decorate all code residing in the integrator with ♠. This is also different from the correctness theorem. Essentially, we are now interested in asserting that the gadget cannot change the proxy library or bootstrapping in the integrator, whereas for the correctness theorem we were interested in heap indistinguishability only for the integrator heap in original and compiled mashups. Furthermore, we assume h_{in} is decorated with ♠, and h^f_{in} is decorated with ♥. (Notice that decorations do not affect the compiler or semantics of JavaScript code and are only used as technical instrumentation for the theorems and their proofs.)

Definition 20 (Decorated Mashic Compilation (for security theorem))**.** Given an integrator script P_i, a gadget script P_g, and a set of variable $\mathcal{V}$ denoting global names exported by the gadget script, we define the Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$ to be:

```
<html>
    <iframe src=u></iframe>
    <script♠>
      P_p; Bootstrap_i^V; C⟨P_i⟩(function(_x){_x})
    </script>
</html>
```

where

$$\mathsf{Web}(u) = \texttt{<script♥>}\ P_l; P_g; \mathit{Bootstrap}^{\mathcal{V}}_g\ \texttt{</script>}$$

Example 12 (Integrity violation). In the example referred just above, the initial heap contains the native function *setTimeout*. Since the initial heap is decorated with ♠, the "*timeout*" property of the global object is a property of the integrator.

$$h(\#global) = \left\{ \begin{array}{c} \vdots \\ \text{“}timeout\text{”}_{\{\spadesuit\}} : \ell \\ \vdots \end{array} \right\}$$

By using decoration of Definition 20 and semantics rules, we get that the projection $h\lfloor_{\spadesuit}$ of the integrator heap before execution of the gadget and projection $h'\lfloor_{\spadesuit}$ after execution of the gadget do not coincide. The `setTimeout` property of the integrator's global object has been changed by the gadget execution. This represents an integrity violation.

Example 13 (Confidentiality violation). Recall variable `secret` in the example of Section 2. Let us assume that the gadget's heap is $h\lfloor_{\heartsuit}$.

After execution of the non-benign gadget in Listing 4 with an integrator's global object containing $\text{“}secret\text{”}_{\{\spadesuit\}} : \text{“}yes\text{”}$

$$h(\#global) = \left\{ \begin{array}{c} \vdots \\ \text{“}secret\text{”}_{\{\spadesuit\}} : \text{“}yes\text{”} \\ \vdots \end{array} \right\}$$

the gadget heap has $h\lfloor_{\heartsuit}(\#global_f).\text{“}steal\text{”} = \text{“}yes\text{”}$. But starting with integrator's global object containing $\text{“}secret\text{”}_{\{\spadesuit\}} : \text{“}no\text{”}$

$$h(\#global) = \left\{ \begin{array}{c} \vdots \\ \text{“}secret\text{”}_{\{\spadesuit\}} : \text{“}no\text{”} \\ \vdots \end{array} \right\}$$

the gadget heap is $h\lfloor_{\heartsuit}(\#global_f).\text{“}steal\text{”} = \text{“}no\text{”}$. This difference depends on the integrator's heap and represents a confidentiality violation.

We show that for any gadget code P_g, and any integrator code P_i, the Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$ provides integrity and confidentiality guarantees. Notice that even if iframes provide strict heap separation, the theorem shows that this does not imply that the SOP provides strict isolation. Security provided by the SOP is not equivalent to a noninterference property or strict isolation but rather equivalent to a declassification property (the queue component in the configuration is set to be the same in the two executions). This is mainly due to inter-frame communication.

Theorem 2 (Security Guarantee of Integrator)**.** *Let P_g and P_i be gadget and integrator code respectively, and let $\mathcal{V}$ be a set of variables. For any configuration reachable from a Mashic compilation $\dot{M}_c(P_i, P_g, \mathcal{V})$:*

$$\langle \spadesuit, \varepsilon, null, \dot{M}_c(P_i, P_g, \mathcal{V}), Q_{init} \rangle_I \rightarrow^* \langle \heartsuit, h, \ell, s, Q \rangle_F$$

if

$$\langle \heartsuit, h, \ell, s, Q \rangle_F \rightarrow \langle \heartsuit, h', \ell', s', Q' \rangle_F$$

then we have

1. *(integrity.)* $h\lfloor_{\spadesuit} = h'\lfloor_{\spadesuit}$ *;*
2. *(confidentiality.) For any* h_0 *such that* $h_0\lfloor_{\heartsuit} = h\lfloor_{\heartsuit}$*, we have* $\langle \heartsuit, h_0, \ell, s, Q \rangle_F \rightarrow \langle \heartsuit, h'_0, \ell', s', Q' \rangle_F$*, and* $h'_0\lfloor_{\heartsuit} = h'\lfloor_{\heartsuit}$ *.*

The proof of security proceeds by induction on the length of the execution and is simpler than the one of the correctness theorem.

7. Implementation and Case Studies

The Mashic compiler is written in Bigloo [4] (a dialect of Scheme) and JavaScript. It has 3.3k lines of Bigloo code and 0.8k lines of JavaScript code. We now turn to discuss practical issues as well as an optimization that we have designed and implemented based on batched futures. We also report on case studies.

CPS in JavaScript Since JavaScript does not support any tail-recursive call optimization, CPS-transformed code can easily run out of call stacks. In order to deal with this, we implement a trampoline mechanism as proposed by Loitsch [24]. We define a global variable `counter` to count the depth of current call stacks. If the counter exceeds a certain limit (in the following example it is 30) a tail call will return a trampoline object instead of invoking the function.

This is shown in Listing 16.

```
1 if (counter > 30)
2    return new Trampoline(fun, arg);
3    return fun(arg);
```

Listing 16: Trampoline of Tail Call

A guard loop, on the top level, detects if a trampoline object is returned, as shown in Listing 17. If a trampoline object is detected, the loop restarts the execution of the tail call.

```
1 res_or_tramp=fun(arg);
2 while (res_or_tramp instanceof Trampoline)
3    res_or_tramp = res_or_tramp.restart();
```

Listing 17: Guard Loop of Trampoline Execution

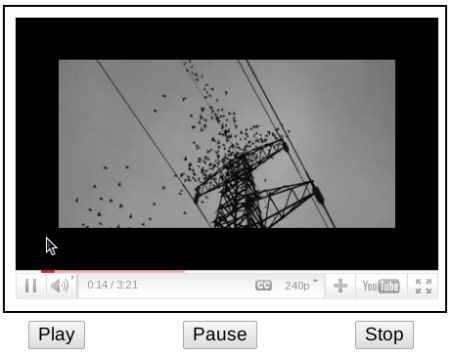
Event Handler In mashups, we also find demands for registering integrator-defined functions as event handlers of gadgets' DOM objects. For example, the Google Maps API provides an interface to set an integrator's function as a handler of the event of clicking on the map. Every time the map is clicked, the corresponding function will be invoked, to notify the integrator of the event. By the SOP, the integrator and the gadget in a Mashic compilation cannot exchange function references. Hence we design and implement a mechanism called *Opaque Function Handle* to achieve the same functionality of an event handler. Similar to the opaque object handle, we associate opaque function handles with function objects on the integrator side. When an iframe-sandboxed gadget receives a function handle, it creates a wrapper function by using the function shown in Listing 18.

```
1 function wrap_fun(fhandle){
2    return function(arg){
3       var msg = { fun : fhandle,
4                   msg_type : 'CALLBACK',
5                   argument : arg};
6       PostMessage(stringify(msg));
7       return;};}
```

Listing 18: Wraping Function Handle

[4] http://www-sop.inria.fr/mimosa/fp/Bigloo/

(a) Map Directions



(b) Youtube Player Controls

Fig. 14. Case Studies of Applying Mashic Compiler

Mashup	Gadget API	Description
Polyline Drawing (P)	Google Maps	Integrator uses the APIs to draw several random lines on the displayed map.
Marker Drawing (P)	Google Maps	Integrator uses the APIs to place several random markers on the displayed map.
Map Controls (O)	Bing Maps	Integrator implements several controls over the map such as zooming, relocating, etc.
Player Controls (O)	Youtube	Integrator implements several controls over the player such as forwarding, stop, etc.
Translator (O)	Bing Translator	Integrator uses the provided translating API to do translation.
Polyline and Marker (O)	Google Maps	A mashup that contains multiple gadgets.

Fig. 15. Selected Case Studies

The wrapped function, upon each invocation, sends a message to notify the integrator to invoke the function associated with the function handle.

Case Studies We have successfully applied our compiler to mashups using well-known gadget APIs, such as Google Maps API, Bing Maps API, and Youtube API. Those examples involve non-trivial interactions between the integrator and the gadget.

In Figure 14 we show two concrete examples. The first example is a mashup using the Google Maps API to calculate driving directions between two cities. The map gadget is sandboxed by the Mashic compiler in an iframe, as indicated by a black box in the figure. The compiled integrator, as in the original integrator, permits to choose a starting point and an ending point to display a route in the map. The gadget's response displayed by the integrator, is the distance between the two points. The latter example shows a sandboxed Youtube player, where one can control the behavior of the player through buttons in the integrator.

We report a selected list of mashups in Fig. 15. In the first column of the figure, the mark 'P' means that the integrator's code was obtained from publicly available code in the web, whereas mark 'O' means that the code is ours.

For the 25 examples of Google Maps API we have studied, we have successfully compiled 23 of them. The other 2 examples are not supported by the Mashic compiler. They are `overlay-remove` and `overlay-simple`. The example of `overlay-remove` uses the for-in construct which is not currently supported by our compiler. In the `overlay-simple` example, the integrator uses some gadget's object as the prototype of an integrator's object, which is not allowed by Definition 8 (correct integrator).

Discussion on Performance and Optimization The Mashic compiler prototype does have a running overhead on a compiled mashup compared to the original mashup. (This penalty is not perceptible for the final consumer of the mashup, if the interaction with the gadget is not inside a loop, for example.) The performance penalty in the Mashic compiler without optimizations [26] mainly comes from the unoptimized CPS-transformation and message-passing. We have implemented an optimized version of the compiler based on batched futures [5], [18] that we discuss here.

Batching Optimization Message-passing is the main cause of performance penalty, especially inside a loop. For example in the *marker drawing* mashup we show in Figure 15, a loop inserts markers into the map. For each marker, it requires two round-trips of messages. The total message-passing overhead is proportional to the number of loop iterations. Although in practice, as in the above example, it is often the case that the loop can be parallelized, parallelism is not yet available in JS. Another alternative is to "batch" these messages to reduce the total message-passing overhead to constant time.

The key idea of our optimization is to transform programs in such a way that messages are only exchanged when the result produced by the gadget is actually required for determining the control flow in the integrator code. Consider, for instance, the program below in which the gadget is assumed to implement three methods g1, g2, and g3, while the integrator is assumed to implement two functions i1 and i2:

```
1 x = gadget.g1(); gadget.g2(); y = gadget.g3();
2 if (x == y) {
3   i1();
4 } else {
5   i2();
6 }
```

During the execution of the program generated by the original mashic, three messages are exchanged between the integrator and the gadget, each of them triggered by a single call to one of the gadget's methods. In contrast, the program generated by the optimised mashic only exchanges one message with the gadget, just after the evaluation of the guard of the condition, as the outputs of previous calls are required for determining the control flow in the integrator code.

The code generated by the original mashic compiler is such that every time the integrator code interacts with an opaque object handle, the interface of the proxy library responsible for creating the corresponding message and sending it to the gadget is invoked. For instance, when using an opaque object handle as a function, the interface `CALL_FUNCTION` of the proxy library is invoked. This interface receives as parameters the opaque object handle corresponding to the gadget's function as well as the corresponding arguments and the current continuation. It then creates a message that encapsulates the function call request, sends it to the gadget along with the arguments, and registers the current continuation. Once the gadget's response arrives, the current continuation is invoked using the response value as its argument. In contrast, the optimised proxy interfaces do not send any message to the gadget but rather create a *batched future* that represents the future value returned by the gadget. Once the value of a batched future is needed for determining the control flow in the integrator's code (for instance, for deciding which branch of a

conditional to take), all the pending requests are batched together and sent to the gadget. To this end, the proxy provides an interface `GET_REAL_VALUE`, whose code is given below, that receives as input a value and a continuation.

```
function GET_REAL_VALUE(v, cont) {
    if (isMashicObject(v)) {
        _cont = function() { cont.call(null, v._value);};
        FLUSH()
    } else { cont.call(null, v); }
}
```

This interface checks whether its first argument is a mashic internal object (either a batched future or a value envelope, which is explained later in this section) in which case it registers the current continuation and dispatches all the pending requests to the gadget using a special proxy function called `FLUSH`.

Every time a batched future is created, it is registered in a special array bound to the global variable `_batched_futures`. We distinguish two types of batched futures: those that represent a value to be returned by the gadget – that we call *simple batched futures* – and those that represent a value to be computed by the integrator using a value returned by the gadget – that we call *complex batched futures*. Function `FLUSH`, whose code is given below, batches together and sends to the gadget all the requests corresponding to the registered batched futures up to the first complex batched future. Additionally, the global variable `_current_batch_index` is set to the index of the first complex batched future in the array of registered batched futures.

```
function FLUSH() {
    var i, requests;

    requests = [];
    for (i=0; i<_batched_futures.length; i++) {
        if (isComplexBatchedFuture(batched_futures[i]) break;
        requests[i] = createMsg(batched_futures[i]);
    }

    postMessage(requests);
    _current_batch_index = i;
}
```

When the gadget's message arrives with the responses for all pending requests, the function `_message_handler` is invoked. This function starts by transforming all the batched futures whose value is sent by the gadget into *value envelopes* that encapsulate their corresponding values. A *value envelope* is an object with a field `_value` that holds its corresponding value. After transforming the batched futures into value envelopes, the complex batched futures that depended on these simple batched futures are resolved, that is, their values are determined and they are transformed into value envelopes. If, after this process, there are still registered batched futures to determine, the function `FLUSH` is invoked again. Otherwise, the registered continuation is invoked.

```
function _message_handler(m) {
    var responses;

    responses = parse(m);
    for (i=0; i<responses.length; i++) {
        _build_simple_envelope(batched_futures[i], responses[i]);
```

```
7     }
8
9     for (i=responses.length; i<_batched_futures.length; i++) {
10        if (!isComplexBatchedFuture(batched_futures[i])) break;
11        _resolve_complex_batched_future(batched_futures[i]);
12    }
13
14    _batched_futures = _batched_futures.slice(i);
15    if (i < _batched_futures.length) {
16       FLUSH();
17       return;
18    } else { _cont() }
19 }
```

The proxy interfaces in the integrator's side must be changed in order to handle the two kind of batched futures and the value envelopes. In the original mashic runtimes the role of the proxy interfaces `GET_PROPERTY`, `OBJ_PROP_ASSIGN`, `CALL_FUNCTION`, `CALL_METHOD`, and `NEW_OBJECT` is to:

- Create the message containing the request to the gadget;
- Register the current continuation;
- Send the message to the gadget.

In the optimised version of mashic, the role of these interfaces is to determine whether the output of the corresponding operations should yield a "real" value, a simple batched future, or a complex batched future, generate the appropriate result, and immediately call the current continuation using it as its argument. The execution of GET_PROPERTY(g, prop, cont) calls the continuation `cont` with:

- a simple batched future, if `g` is bound to an opaque object handle or simple batched future and `prop` to a string or a simple batched future;
- a complex batched future, if `g` is bound to an object belonging to the integrator or a complex batched future and `prop` to a simple or complex batched future; ;
- a "real" value, if `g` is bound to an object belonging to the integrator and `prop` to a string.

Note that these proxy interfaces must also take into account value envelopes. Namely, they have to unnest the real value they contain before applying the corresponding operation. Since binary operations may be performed on both value envelopes and batched futures, we introduce an additional proxy interface `BINARY_OPERATION` that takes care of all the possibly different cases.

Since messages are only dispatched to the gadget when the value it returns is required for determining the control flow on the integrator's side, the optimised version of mashic can use a partial-CPS transformation. Hence, continuations are only generated in program points where the control flow is at stake, such as the guards of conditionals and loops, function calls, and method calls. The partial-CPS transformation has to combine CPS terms with non-CPS terms. Therefore, it has to consider several cases for each type of expression. However, to avoid cluttering the presentation, we assume in the rest of this section a full CPS transformation.

In order for the batching mechanism to work, the CPS transformation performed by the mashic compiler must be slightly modified. We illustrate the difference between the original and the optimized transformations using the examples of the rules for the conditional statement, the member selector, and the binary operation given in Figure 16.

$\mathcal{C}\langle e_0[e_1]\rangle$:

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
        GET_PROPERTY(_x0, _x1, _k);
  }); }); }
```

$\mathcal{C}\langle e_0\ op\ e_1\rangle$

```
function(_k){
  C⟨e0⟩(function(_x0){
    C⟨e1⟩(function(_x1){
      BINARY_OPERATION("op", _x0, _x1, _k);
    });
  });
}
```

$\mathcal{C}\langle$if $(e)\ s_0$ else $s_1\rangle$:

```
function(_k){
  C⟨e⟩(function(_b){
        GET_REAL_VALUE(_b,
                function(_b){if (_b) C⟨s0⟩(_k)  else C⟨s1⟩(_k); });
      });
}
```

Fig. 16. Optimised Mashic-CPS Transformation

Given a member selector expression, the original mashic CPS transformation generates a conditional expression that checks whether the inspected object is an opaque object handle, in which case `GET_PROPERTY` is invoked in order to dispatch the corresponding request to the gadget. Contrastingly, the code generated by the optimised mashic compiler always invokes `GET_PROPERTY` whose role is to handle the property look-up, possibly generating a batched future. The compilation of a binary operation generates a call to the proxy library function – `BINARY_OPERATION`. Observe that one can invoke a binary operation on different types of batched futures, thus generating different types of batched futures. For instance, while invoking a binary operator on two simple batched futures yields a simple batched future, invoking a binary operator on a simple batched future and a complex batched future yields a complex batched future. In order to determine which branch to take, the compilation of a conditional statement must invoke `GET_REAL_VALUE` on the value to which the guard evaluates (since it can be a batched future).

The greater the number of messages that can be batched together, the greater the impact on performance of the optimised mashic compiler. In order to measure this impact, we wrote a simple mashup using Google Maps that randomly generates map markers inside a loop and then adds them to a map as shown in Figure 17. While the mashup compiled using the original mashic sends a message to the gadget for each marker that is randomly generated, the mashup that is compiled using the optimised version sends to the gadget a single message containing the requests for the creation of all markers. The chart given in Figure 18 illustrates the impact on performance of the batching transformation depending on the number of generated markers. Experimental results were obtained on Firefox 30.0 on a 2.4 GHz Intel Core i5 running OS X Version 10.9.3.

8. Conclusion

We have proposed the Mashic compiler as an automatic process to secure existing real world mashups. The Mashic compiler can offer a significant practical advantage to developers in order to effortlessly

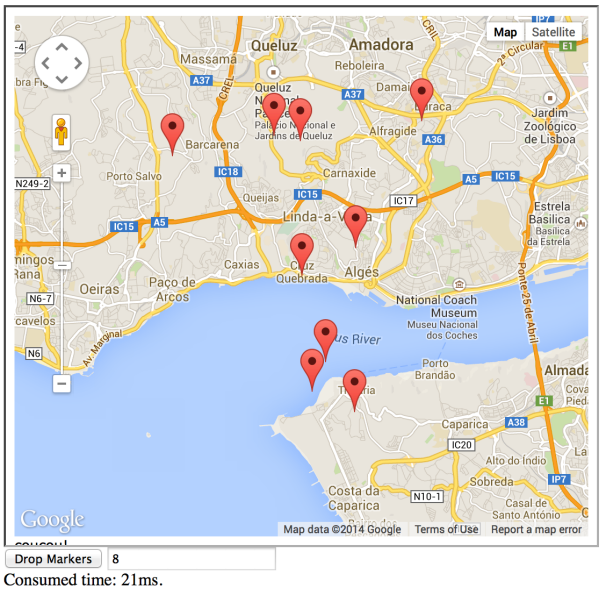


Fig. 17. Map Markers

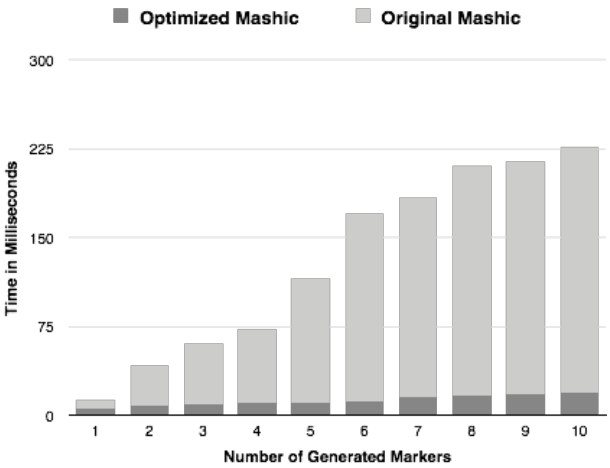


Fig. 18. Comparison between mashic optimised version and original version

write secure mashups without giving up on functionality. Compiled code is formally guaranteed to satisfy precisely defined integrity and confidentiality properties of integrator's sensitive resources.

We do not address in this paper analysis to prevent security vulnerabilities introduced by the integrator's code. Consider the following silly code:

```
1 CALL_METHOD(eval,opq_obj,"foo",{});
```

This integrator will `eval` the result from calling the `foo` method of the opaque object handle `opq_obj`.

The gadget might return some string representing a malicious JavaScript program. Then the integrator will execute the malicious code with its own privilege. To avoid this kind of vulnerabilities, analysis of

the integrator's code is required. This is orthogonal to the current Mashic compilation: information flow analyses for JavaScript can be found for example in [15,35].

Mashic offers correctness guarantees *only if* untrusted gadgets are benign. This is a goal of the compiler and not a disadvantage: mashup behavior should *not* be the same if a gadget is malicious. If the gadget is malicious the programmer does not get any alert that the compiled secured mashup does not behave as the original mashup: an interesting future direction will be to provide JavaScript code analysis that will conservatively detect non-benign gadgets in order to alert the programmer.

Acknowledgement We acknowledge anonymous reviewers for their useful comments on this article.

References

[1] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of web security. In *CSF*, pages 290–304, 2010.
[2] Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *W2SP2009*, 2009.
[3] Adam Barth, Collin Jackson, and John C. Mitchell. Securing Frame Communication in Browsers. *Commun. ACM*, 52(6): 83–91, 2009.
[4] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin Javascript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX security symposium*, pages 187–198, 2009.
[5] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *OOPSLA*, 1994.
[6] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: Formalizing the core of a web browser. In *Usenix Conference on Web Application Development (WebApps)*, June 2010.
[7] Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208:716–736, 2010.
[8] Steven Crites, Francis Hsu, and Hao Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In *CCS*, pages 99–108, 2008.
[9] Douglas Crockford. The `<module>` Tag , 2010. http://www.json.org.
[10] Douglas Crockford. ADsafe, 2011. http://www.adsafe.org/.
[11] ECMA. ECMAScript Language Specification. Technical report, ECMA, 2009. http://www.ecma-international.org/.
[12] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5), October 2005.
[13] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Évariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, 2013.
[14] Dan Grossman, J. Gregory Morrisett, and Steve Zdancewic. Syntactic type abstraction. *TOPLAS*, 22:1037–1080, 2000.
[15] Daniel Hedin and Andrei Sabelfeld. Information-flow security for a core of javascript. In *IEEE Computer Security Foundations Symposium, CSF 2012*, 2012.
[16] Ian Hickson. HTML5. Technical report, W3C, May 2011.
[17] Arnaud Le Hors, Philippe Le Hegaret, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne. Document Object Model (DOM) level 2 Core Specification. Technical report, W3C, November 2000.
[18] Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *ECOOP*, 2009.
[19] Facebook Inc. Facebook Javascript Subset, 2011. https://developers.facebook.com/docs/fbjs/.
[20] Google Inc. Google Caja Project, 2011. http://code.google.com/p/google-caja/.
[21] Collin Jackson and Helen J. Wang. Subspace: Secure Cross-domain Communication for Web Mashups. In *WWW*, 2007.
[22] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *CCS*, 2010.
[23] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *WWW*, 2008.
[24] Florian Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice - Sophia Antipolis, March 2009.
[25] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical Enforcement of Confidentiality and Integrity Policies on Web Advertisements. In *USENIX Security Symposium*, 2010.
[26] Zhengqin Luo and Tamara Rezk. Mashic compiler: Sandboxing using inter-frame communication. In *IEEE Computer Security Foundations Symposium, CSF 2012*, 2012.
[27] S. Maffeis and A. Taly. Language-based Isolation of Untrusted Javascript. In *CSF*, pages 77–91. IEEE, 2009.

[28] S. Maffeis, J.C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, volume 5356 of *LNCS*, pages 307–325, 2008.

[29] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE Security and Privacy*, 2010.

[30] The Mashic Compiler Website. http://www-sop.inria.fr/indes/mashic/.

[31] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[32] Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Typed-based verification of web sandboxes. *Journal of Computer Security*, 22(4):511–565, 2014. doi: 10.3233/JCS-140504. URL `http://dx.doi.org/10.3233/JCS-140504`.

[33] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003.

[34] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*, Lecture Notes in Computer Science, pages 174–191, 2004.

[35] José Fragoso Santos and Tamara Rezk. An information flow monitor-inlining compiler for securing a core of javascript. In Nora Cuppens-Boulahia, Frédéric Cuppens, Sushil Jajodia, Anas Abou El Kalam, and Thierry Sans, editors, *SEC*, volume 428 of *IFIP Advances in Information and Communication Technology*, pages 278–292. Springer, 2014. ISBN 978-3-642-55414-8.

[36] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[37] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *SOSP '07*, pages 1–16, 2007. ISBN 978-1-59593-591-5.

[38] Chuan Yue and Haining Wang. A measurement study of insecure javascript practices on the web. *ACM Trans. Web*, 7(2), May 2013.

[39] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, 2006.

BrowserAudit: Automated Testing of Browser Security Features

Charlie Hothersall-Thomas
Netcraft Ltd, UK
me@charlie.ht

Sergio Maffeis
Department of Computing
Imperial College London, UK
maffeis@doc.ic.ac.uk

Chris Novakovic
Department of Computing
Imperial College London, UK
c.novakovic@imperial.ac.uk

ABSTRACT

The security of the client side of a web application relies on browser features such as cookies, the same-origin policy and HTTPS. As the client side grows increasingly powerful and sophisticated, browser vendors have stepped up their offering of security mechanisms which can be leveraged to protect it. These are often introduced experimentally and informally and, as adoption increases, gradually become standardised (e.g., CSP, CORS and HSTS). Considering the diverse landscape of browser vendors, releases, and customised versions for mobile and embedded devices, there is a compelling need for a systematic assessment of browser security.

We present BrowserAudit, a tool for testing that a deployed browser enforces the guarantees implied by the main standardised and experimental security mechanisms. It includes more than 400 fully-automated tests that exercise a broad range of security features, helping web users, application developers and security researchers to make an informed security assessment of a deployed browser. We validate BrowserAudit by discovering both fresh and known security-related bugs in major browsers.

Categories and Subject Descriptors

D.4.6 [**Operating systems**]: Security and Protection

Keywords

Web security, web browser testing, same-origin policy, Content Security Policy, Cross-Origin Resource Sharing, clickjacking, cookies

1. INTRODUCTION

Personal data, business transactions, critical infrastructure and even cars, refrigerators and lightbulbs are exposed through web interfaces to a wide variety of web browsers. Hence, the browser plays a key role in the modern information infrastructure, as the main gateway to access the information and capabilities made available online.

ISSTA'15, July 12–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771789

As such, browsers need to offer a variety of standardised security mechanisms which can be relied upon uniformly by the client side of web applications, in order to deliver security guarantees to their users. For example, the same-origin policy (SOP) [34] is effective at preventing a range of cross-site scripting (XSS) attacks [38] against users' web browsers and is an integral aspect of modern web-based security. On the other hand, it is sometimes excessively strict; for instance, it forbids the sharing of information between different subdomains, a common requirement of large web sites. It is also coarse-grained, and several attempts have been made to enforce finer-grained access control [41, 39] and origins [22, 23, 29] in the browser. A variety of contemporary web browsers implement the Cross-Origin Resource Sharing (CORS) [46] standard, which may be used to control the flow of information between server-side resources and client-side scripts that attempt to access those resources via APIs. However, even fully-compliant implementations of the SOP and CORS mechanisms in some cases do not regulate access to other resources, such as images, embedded objects and web fonts, that can leave web applications vulnerable to cross-site request forgery (CSRF) attacks [20], clickjacking [36], framebusting [43] and CSS-based attacks [33]. The Content Security Policy (CSP) standard [45] enables much finer-grained control over the loading of arbitrary resources on a web page, mitigating several of these issues. These are just some examples of established and emerging security mechanisms offered by modern browsers.

Such mechanisms are often introduced experimentally and informally. As adoption increases, they gradually become standardised, and after numerous security reviews and bug reports they can eventually be relied upon consistently across browsers [19, 37, 20]. Reaching that stage is not easy. For example, correctly implementing the CSP specification is non-trivial: it is a lengthy document with many cross-references to other standards and RFCs, many of which have been superseded by newer (and conflicting) standards and RFCs. It is possible that a browser vendor could incorrectly implement some part of the CSP and thus fail to provide some of its security guarantees to their users. There is therefore a need for an automated tool that enables browser developers to complement low-level unit tests targeted at isolated source code modules with high-level testing of the effectiveness of the implementation of the security features once the browser is deployed.

In this paper we introduce BrowserAudit, a framework for testing whether a deployed browser correctly enforces the security guarantees implied by the main standardised

security mechanisms. For practical purposes, we present BrowserAudit as a standalone web application that automatically tests the browser used to access it. BrowserAudit has been designed with different sets of users in mind. A casual web user can run the tests to gain a simple security assessment of their browser: critically vulnerable, non-critically vulnerable, or okay. With the recent surge of security breaches reported in the news, people are becoming increasingly security-conscious and we believe there is an increasing demand for tools that inform the public about security. A security researcher can benefit even more, viewing a detailed breakdown of each test result, and seeing which security features passed our tests and which had problems. We display textual descriptions for each category of tests and the client-side source code of the tests. Browser developers can use BrowserAudit to debug their security features and web developers can use it as a way to ascertain the security capability of users' browsers (Section 2). We chose to implement a careful selection of tests that covers both the most important browser security mechanisms that should be implemented in any browser, and some of the most promising experimental ones that are not yet widely implemented. Starting from the code of individual test cases, we identified and generalised common patterns in order to automatically generate hundreds of tests. BrowserAudit automatically tests over 400 behaviours where a certain action should either be allowed or blocked according to an implied browser security policy (Section 3).

We designed BrowserAudit to be efficient and scalable, and we evaluated its performance and its accuracy extensively by running it on a number of browsers and platforms. Using BrowserAudit, we have discovered several previously unknown security bugs in recent versions of Mozilla Firefox, which we have reported to the developers (Section 4.4).

Whilst there are well-understood methodologies for generating unit tests for a given code base, there is no general solution to the problem of testing the end-to-end security behaviour of a family of applications (in our case web browsers) that must respect precise interoperability constraints (web standards) but can widely differ in implementation architectures, languages and design. Hence, we faced a significant challenge when developing our tests, carrying out a substantial amount of practical experimentation, guided by the official RFCs, our formal and informal models of web security, and a substantial body of academic and practical research on browser and web security (surveyed in Section 5.1). Although we believe that BrowserAudit is unique in its focus and breadth, we were inspired by a number of related web applications described in Section 5.2.

Contributions. Summarising, our main contributions are:

- We analysed the specifications of HTML5, CSP, CORS and HTTP Strict Transport Security (HSTS), identifying the concrete security guarantees implied by the proposed mechanisms. This allowed us to formulate precise goals for security test cases.
- We built a suite of more than 400 browser security tests, which brings together a wealth of explicit and implicit knowledge of the guarantees afforded by modern browser security mechanisms. We made the tests available to the community by open-sourcing the BrowserAudit code base [32].
- We implemented the first fully-automated web application that comprehensively tests browser security features and provides detailed information to a variety of user bases.
- We used BrowserAudit to discover previously unknown vulnerabilities in a major web browser.

2. DESIGN OVERVIEW

The goals underlying the design of BrowserAudit are the following:

- *Wide coverage*: BrowserAudit should demonstrate that a wide range of browser security mechanisms can be tested automatically, reliably and efficiently. Complete test coverage of any such mechanism is not practically feasible, and beyond the scope of this project.[1]
- *Extensibility*: By its very nature, BrowserAudit will always be a work in progress. As the browser threat landscape evolves, more tests will be needed to cover new security mechanisms, or to extend the coverage of existing ones. Our design should ease the task of creating, debugging and integrating additional test cases.
- *Ease of use*: BrowserAudit should be easily accessible on any modern browser connected to the Internet, without the need to install additional software. It should require no interaction from the user, otherwise running hundreds of tests would be impractical. Moreover, relying on user interaction would prevent the desired aim of running the tests transparently in the background.
- *Broad audience*: Our design should support a diverse range of users. A report on the security effectively offered by a deployed browser should benefit browser developers, penetration testers, security researchers and web users.
- *Scalability*: Our design should be scalable on the server side. Several users may be testing their browser at the same time, and many security tests concern features that involve communicating with the server.

We now sketch the architecture of BrowserAudit and highlight the main design choices. We defer further implementation details to Sections 3 and 4.1.

2.1 User Experience

BrowserAudit is accessible by simply pointing the browser to be tested to `https://browseraudit.com/`. This is a landing page that briefly describes the aims of the project and contains a "Test me" button to move the user to the actual test page, hosted at `https://browseraudit.com/test`. This intermediate step avoids surprising users by actively requiring their consent to begin the testing phase. Once the user clicks to start the tests, the main testing loop initiates.[2]

BrowserAudit is completely automated, and the user does not need to interact with the browser whilst it is being tested.

[1] For example, an exhaustive test of the same-origin policy would also need to demonstrate that, for any domains A and B, a page from domain A cannot access certain properties of a page from an incompatible domain B.

[2] Unless JavaScript is disabled, in which case we display a warning to the user. Automated tests cannot be run without JavaScript, and some security features need JavaScript in order to be exercised.

As the tests are running, the user can see a progress bar advancing, and four test counters being incremented, as shown in Figure 1. For the benefit of typical web users, test runs

Figure 1: The test summary box part-way through the execution of our tests.

are categorised using a simple Okay/Warning/Critical/Skipped traffic light indicator. Okay denotes passed tests, Warning and Critical denote failed tests, and Skipped denotes tests that are skipped because the feature being tested is not supported by the browser. Failures regarding SOP, cookies, and the `Referer` header, which we consider the most crucial security features, are reported as Critical; failures regarding CSP, CORS, HSTS and the `X-Frame-Options` header are reported as Warnings. This distinction is somewhat arbitrary, and will change as these features become more broadly supported and new ones are introduced.

After the test suite has finished running, the grey background of the summary box assumes the colour of the worst failed test, or green if all tests passed. This traffic light indicator provides a basic level of information about the current level of security offered by the browser.

More sophisticated users, such as security researchers or browser developers, need more information on the tests performed and on their outcomes. Clicking on the "Show/Hide Details" button displays a summary box that shows the various categories of tests (reflecting the security mechanisms that have been tested), and the number of failed tests for each of them, as shown in Figure 2.

Figure 2: BrowserAudit summary box.

Each category can be expanded and collapsed to show a description of the corresponding security mechanism, and a list of sub-headers that in turn can be expanded to reveal individual tests for a specific feature, as illustrated in Figure 3. For each individual test we show a descriptive title that can be clicked to show the client-side source code of the test itself. Our design uses the Bootstrap front-end framework [1], which

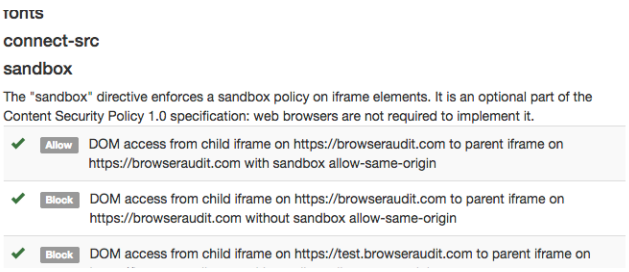


Figure 3: Some sub-categories of CSP tests, with expandable test titles and result indicators.

makes it easy to produce a layout that works consistently across browsers and devices.

2.2 Architecture

The client side and server side of BrowserAudit work together in order to run tests in the browser: the server side exercises browser security features, and the client side tests that these features are implemented as expected.

When multiple concurrent users access BrowserAudit, we need to avoid congestion on the server side, as testing each browser causes a bursty interaction with the BrowserAudit server in the form of hundreds of requests per user per minute. For this reason, we adopt a standard three-tier server architecture, consisting of a public-facing Nginx [11] web server, a Go [16] application server and a PostgreSQL [13] database backend. The Nginx server is running as a *reverse proxy* in front of the Go server, which is not publicly accessible. When the Nginx server receives HTTP(S) requests for static resources, such as our JavaScript tests, it responds by directly fetching the resource from the local `static/` directory. Dynamic requests are instead proxied to the Go server, and the responses are forwarded back to the client. Nginx also handles SSL termination, caching, gzip compression, URL rewriting, and keeps access and error logs. This architecture reduces the load on the Go server, which can focus on serving only dynamic requests that depend on the user's session, and limits security risks because the Go server can run as a non-privileged user.

Certificates. In order to ensure good coverage of various security features that involve the use of HSTS and cross-origin testing, BrowserAudit makes use of four domains: `browseraudit.com`, `test.browseraudit.com`, `browseraudit.org` and `test.browseraudit.org`. The server presents a single SSL certificate that is valid for all of these domains.

Sessions. We use sessions to keep track of intermediate test results and other test-related data for each user whilst their tests are in progress. Sessions are needed because in many of our security tests, it is the *server* that makes the decision as to whether or not the browser passed the test, not the test framework running in the browser. In these cases, the client must send an additional request asking the server what the test result was, so that it can be displayed to the user.

Caching. In our tests, there are many cases in which a request is first made to store a default result on the server, and then a second request *may* be sent to overwrite this result, depending on whether or not the browser correctly

implements a given security feature. If a user runs the tests twice in short succession, and this second result was cached and therefore did not reach our server, our application would report an incorrect test result. We ensure that this cannot happen by preventing HTTP responses from being cached.

2.3 Tests

A typical test of a security feature involves making multiple AJAX or image requests to the server and checking if the actual responses match the expected responses.

JavaScript and libraries. Our tests are written directly in JavaScript, using the jQuery library [9] for convenience. We deploy our tests using the Mocha framework for browser-based JavaScript unit testing [10], with some custom modifications to improve the output layout.

```
1 $.get("/del_httponly_cookie", function() {
2   expect($.cookie("httpOnlyCookie")).to.be.undefined;
3   $.get("/set_httponly_cookie", function() {
4     expect($.cookie("httpOnlyCookie")).to.be.undefined;
5     done();
6   });
7 });
```

Figure 4: The client side of a proof-of-concept `HttpOnly` cookie test.

Figure 4 shows a proof-of-concept test to check that the browser correctly implements `HttpOnly` cookies (see Section 3.4). Line 1 loads a page to clear any leftover cookies from previous test runs, line 2 checks that the cookie is not defined, line 3 loads a second page that sets the cookie, and line 4 checks that we are unable to read it via JavaScript. The call to `done()` on line 5 informs Mocha that the asynchronous test is complete. In order to make the source code of the tests easier to understand and maintain, we also leverage the Chai assertion library [7].

```
1 function ajaxSopTest(globalTestId, shouldBeBlocked,
       sourcePrefix, destPrefix) {
2   // omitted code:  variable initialisation
3
4   var test_template = function(done) {
5     $.get("/sop/"+defaultResult+"/"+id,
6       function() {$("<iframe>", { src: iframeSrc })
7         .css("visibility", "hidden")
8         .appendTo("body").load(function() {
9           $.get("/sop/result/"+id,function(result) {
10            expect(result).to.equal("pass");
11            done();
12          });
13        });
14      });
15    };
16
17  // omitted code:  save source code for display
18
19  browserAuditTest(globalTestId, test_template);
20 }
```

Figure 5: Code to generate SOP tests for AJAX calls.

Tests. In most cases, we automatically generate the JavaScript code for tests that have a similar structure but depend on different parameters. For example, in Figure 5 we show the most interesting parts of the `ajaxSopTest` function, which generates Mocha code for testing AJAX calls with respect to the SOP. The choice of the right parameters for the resources to load (`defaultResults`, `iframeSrc`) are crucial to the correctness of each test instance. To favour modularity and coverage, we instantiate a separate Mocha test for each case to be tested, rather than bundling a large number of cases in the same test. To ensure maximum portability, we implement as much as possible on the client side using standard, browser-independent features.

Whenever possible, we write asynchronous tests using callback patterns rather than timeouts. We annotate the titles of tests whose results depend on timeouts with a small clock icon. We try to avoid using timeouts because, when a timeout expires, it is not possible to distinguish a true test failure from an anomalous delay in a browser event or network connection. Moreover, it is difficult to estimate appropriate timeout values for many events. For certain tests, however, we cannot avoid using timeouts.

For example, to detect whether a CSP policy that denies the use of JavaScript but allows the loading of fonts in an iframe is enforced correctly, the BrowserAudit test framework needs to give time for the iframe to try to load the font, and then ask the server if the font was requested. We are not allowed to run JavaScript in the iframe to inspect the page and detect whether the font was loaded; likewise, we cannot ask the user for confirmation, because our tests must run without user interaction.

3. BROWSER SECURITY MECHANISMS

In this section, we describe the range of security mechanisms currently exercised by BrowserAudit. Each mechanism induces — sometimes implicitly — a security policy. Our emphasis is on testing representative instances of behaviours that should be allowed or blocked according to the corresponding security policy.

3.1 Same-Origin Policy

In the early days of the web, there was little incentive to control the resources that could be included in a web page: most web pages were static, and web developers were free to include resources (e.g., images) from any source in their web pages. As web sites became dynamic and interactive, thus allowing web developers to include user-supplied content in their pages, and requiring web browsers to execute scripts supplied by the web server, browser vendors became more security-aware: they recognised that permitting the execution of arbitrary code (e.g., JavaScript) from untrustworthy sources was potentially dangerous, and began to impose restrictions on the execution of scripts from "foreign" locations. In particular, "foreign" scripts were forbidden from accessing the Document Object Model (DOM) — the browser's internal hierarchical representation — of the web page in which the script was included. These are the foundations of the *same-origin policy* (SOP) [34], still implemented in contemporary web browsers: a script executing in the context of a web page is only permitted to access the DOM of another web page if the schemes, hostnames and port numbers in the URIs of the two pages — their *origins* — match.

There are mechanisms for relaxing the SOP so that information can be shared between DOMs with differing origins; the easiest method of doing so is to set the same `document.domain` property in each DOM, so that the web browser considers the DOMs to have the same origin.

BrowserAudit comprehensively exercises a web browser's implementation of the SOP and the mechanisms for relaxing it to ensure that inter-DOM access is permitted when both DOMs are deemed to have the same origin, and is otherwise forbidden. Our DOM SOP tests have a common structure: scripts running on web pages loaded in nested iframes manipulate the DOM's `document.domain` property, and the script from one iframe attempts to access the DOM of the other iframe. Each test exercises a particular combination of the following parameters:

- The domain from which the web page loaded by the parent iframe is served (one of `browseraudit.{com/org}` or `test.browseraudit.{com/org}`);
- The domain from which the web page loaded by the child iframe is served (also selected from the list above, and potentially the same domain used by the parent iframe's web page);
- The value of `document.domain` to be set by a script running in the parent iframe;
- The value of `document.domain` to be set by a script running in the child iframe; and
- The direction in which the DOM access is attempted (parent iframe to child, or child iframe to parent).

The client-side test framework checks whether the web browser satisfies the SOP by selecting combinations of these parameters that should be allowed or blocked by the SOP and verifying that the correct behaviour is observed.

For example, Figure 6 shows a diagram for a test in which a parent iframe tries to access the DOM of its child iframe. The parent is loaded from `https://browseraudit.org` whereas the child is loaded from `https://test.browseraudit.org`. We expect this access to be blocked since we are not setting any `document.domain` values in this test, and the hostnames are not the same. To communicate test results from the server to the client whilst avoiding the restrictions imposed by the SOP itself, we use the established technique of loading images from specially-crafted addresses (i.e., `https://browseraudit.com/sop/[pass|fail]/TEST_ID`).

In general, if a script running in either iframe is able to access the DOM of the other, the script notifies the BrowserAudit server that access to the other iframe's DOM was granted; the test framework then queries the server for whether this notification was sent. If the notification was sent and DOM access was expected given the chosen test parameters, or if the notification was *not* sent and DOM access was *not* expected given the chosen test parameters, the test framework considers the browser to have passed that particular test; otherwise, the browser permitted insecure DOM access and is considered to have failed the test.

The SOP applies not only to DOM access, but also to cookies with differing paths and HTTP requests made to other domains via the XMLHttpRequest API; BrowserAudit also tests a browser's implementation of the SOP for all of these features, providing a total of 84 SOP tests, generated by four JavaScript templates.

3.2 Cross-Origin Resource Sharing

Cross-Origin Resource Sharing (CORS) [46] is a flexible standard for relaxing the SOP that selectively permits resources to be shared across origins; it is implemented in APIs capable of initiating cross-origin resource requests (e.g., XMLHttpRequest) in a range of modern web browsers. It allows a client to include a resource from a server with a different origin only if the resource request is explicitly authorised by the server. This is achieved via two additional HTTP headers: an `Origin` header is sent by the client as part of the request and specifies the origin of the resource attempting to use the cross-origin resource, and an `Access-Control-Allow-Origin` header is sent by the server as part of the response and specifies the origins from which this resource may be used, effectively ordering the client to uphold or relax the SOP for this resource request.

The majority of cross-origin requests made using CORS are "simple", defined in the CORS specification [46] as an HTTP request with one of `GET`, `POST` or `HEAD` as the request method and headers from a narrowly-defined whitelist (`Accept`, language-related headers and a small number of acceptable `Content-Type`s). Other requests are deemed "non-simple"; the CORS specification requires that the client precedes such requests with a "preflight" request that includes further detail so that the server can more accurately decide whether or not to allow the cross-origin request (although, in reality, some browsers misclassify simple and non-simple requests). In response to the preflight request, the server sends additional headers: `Access-Control-Allow-Methods`, a comma-delimited list of HTTP methods permitted to be used to access the resource; `Access-Control-Allow-Headers`, a comma-delimited list of headers that may be sent with the main CORS request; and `Access-Control-Expose-Headers`, a list of headers that should be exposed to the requester (e.g., a script accessing a resource using XMLHttpRequest). If the main CORS request violates either of the restrictions imposed by the `Access-Control-Allow` headers, the main request is considered a violation of the SOP and is aborted.

BrowserAudit exercises the browser's implementation of CORS by sending a series of cross-origin XMLHttpRequest requests from the browser and verifying that the client exhibits CORS-compliant behaviour when the BrowserAudit server sends a response containing a range of CORS HTTP headers. The testing methodology is similar to that for the SOP, described in Section 3.1: the client attempts to retrieve a file from the BrowserAudit server, and sends a notification to the BrowserAudit server if this retrieval was successful. The BrowserAudit test framework then queries the server for whether the notification was sent. If the notification was sent for CORS-compliant requests and *not* sent for CORS-violating requests, the browser is deemed to correctly implement the CORS standard; if a notification was sent for CORS-violating requests, or if one was *not* sent for CORS-compliant requests, the browser is considered to lack full compliance.

We currently test 54 different CORS scenarios, automatically generated by four JavaScript test templates.

3.3 Content Security Policy

The *Content Security Policy* (CSP) standard[3] [45] enables much finer-grained control over the loading of arbitrary re-

[3]We concern ourselves only with version 1.0 of the Content Security Policy standard, as its successor (version 1.1) is still in Working Draft status at the time of writing; however, the two versions are similar, and the latter can be viewed as an extension of the former.

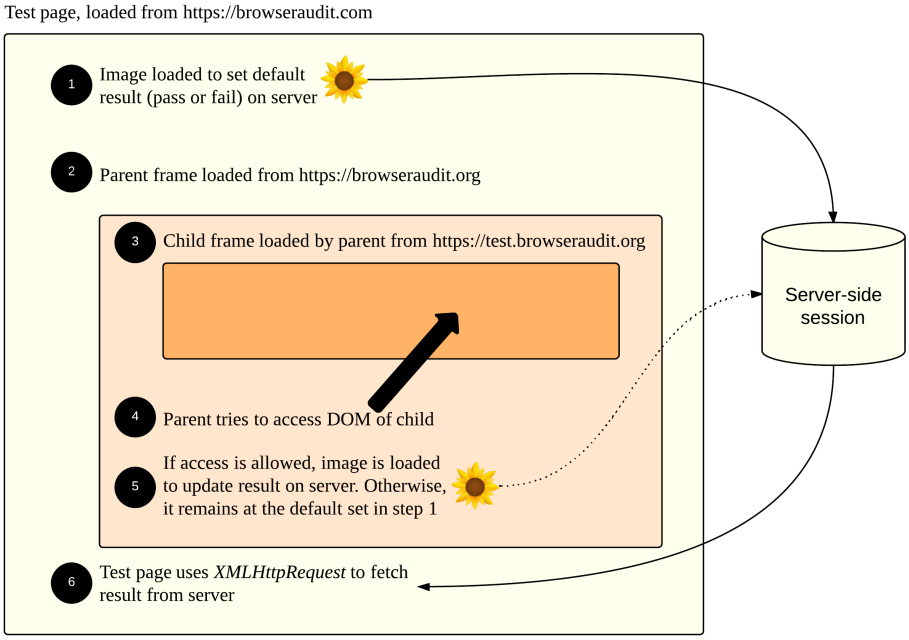


Figure 6: An example of an SOP test in which the parent frame tries to access the DOM of its child.

sources on a web page than the SOP and CORS. As with CORS, a content security policy is delivered via an HTTP header (or via a `<meta>` element in the HTML header); the CSP specification states that the `Content-Security-Policy` header should be used for this purpose.

The header allows servers to declare to CSP-compliant clients the permitted origins of a range of resources: images, stylesheets, scripts, web fonts, embedded objects and other types of resource may all be controlled by a single policy. *Directives* may be used to restrict the origins of these different types of resource independently of each other, and a "default" directive may be used to restrict the origins of all resources that are not explicitly controlled elsewhere in the policy. For example, a server at `example.com` serving web pages to CSP-compliant browsers could restrict the loading of images to those hosted on the same server and the loading of embedded objects (such as Java applets) to those hosted on a trusted server at `applets.example.com` (and thus forbid embedded objects and images from being loaded from other origins) by specifying the following value for the `Content-Security-Policy` header:

```
image-src 'self'; object-src http://applets.example.com
```

When served alongside a web page to a CSP-compliant web browser, such policies can preempt many common web attacks; e.g., using the `script-src` directive to control the permissible origins of scripts mitigates the effects of CSRF, clickjacking and framebusting (since they rely primarily on successful JavaScript injection), and using the `style-src` directive to control the permissible origins of stylesheets defeats CSS-based attacks. Note that one cannot specify which specific resources may be loaded from these other origins: permitting a particular Java applet to be loaded from `applets.example.com` also permits *any other* embeddable object to be loaded from `applets.example.com`, so whitelisted origins should be trustworthy (particularly those granting the power to execute arbitrary code, such as `script-src`).

The CSP standard also includes a mechanism for reporting violations of a given policy via a special `report-uri` directive; this directive defines a URL to which a *violation report* should be sent.

BrowserAudit exercises a browser's CSP implementation by performing a battery of tests on each directive defined in the CSP specification, as well as the violation-reporting capabilities of the `report-uri` directive. Similarly to the SOP tests (described in Section 3.1), each CSP test attempts to load a resource inside an iframe using a particular combination of the following parameters:

- The domain from which the web page loaded by the iframe is served (one of `browseraudit.com` or `test.browseraudit.com`);
- The domain from which the desired resource is requested (also selected from the list above, and potentially the same domain used by the iframe's web page); and
- The CSP imposed on the iframe by the BrowserAudit server via the `Content-Security-Policy` header.

We run 226 CSP tests, generated by three JavaScript templates, that in turn load approximately 280 iframes representing particular behaviours to be tested. In each test, the browser is expected to either allow or block access to the given resource, and the act of requesting the resource from the BrowserAudit server allows it to track violations of the given policy. On the client side, the BrowserAudit test framework queries the server after the iframe has loaded to find whether the browser accessed the resource and therefore determine whether the browser exhibited the behaviour

expected of a CSP-compliant browser: allowing a request permitted by the given policy or blocking a request restricted by the policy is regarded as a correct implementation of the CSP standard and thus a test success, whilst an attempt to access the resource when given a restrictive policy or a failure to request the resource when given a permissive policy is regarded as an erroneous implementation of the standard and thus a test failure.

```
$("<iframe>", { src: "/csp/serve/206/param-html?policy='
    sandbox allow-same-origin allow-scripts'&defaultResult=
    pass" })
  .css("visibility", "hidden").appendTo("body")
  .load(function() {
    $.get("/csp/result/206", function(result) {
      expect(result).to.equal("pass");
      done();
    });
  });
```

Figure 7: A CSP test exercising the browser's implementation of the `sandbox` directive.

```
<html><body>
  <iframe src="/csp/serve/206/param-htmlb?sessid=
      sessionCookie"></iframe>
</body></html>
```

Figure 8: The HTML for the outer iframe loaded by the test script shown in Figure 7.

Figure 7 shows the client-side code for a CSP test. The code runs on the main BrowserAudit page and loads an outer iframe from `browseraudit.com` with the CSP header `sandbox allow-same-origin allow-scripts`. This outer iframe is very simple (Figure 8), and its role is simply to load an inner iframe from `browseraudit.com` that is subject to the given policy: scripts can run, and have same-origin permissions. The inner frame, whose code is shown in Figure 9, tries to perform an XMLHttpRequest to `test.browseraudit.com`, which should be blocked. Note that since we cannot rely on user credentials to be sent with synchronous XMLHttpRequests, we pass the session cookie (abstracted for readability in Figure 9 as `sessionCookie`) as a parameter of the request. All of this information is also visible to the BrowserAudit user by clicking on the corresponding test title in the user interface.

```
<html><body>
  <script>
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "https://test.browseraudit.com/csp/serve
        /206/oktext?sessid=sessionCookie&corsOrigin=
        browseraudit.com&corsMethod=GET", false);
    xhr.send(null);
    if (xhr.status == 200) {
      var img = document.createElement("img")
      img.setAttribute("src", "/csp/fail/206/png");
      document.body.appendChild(img);
    }
  </script>
</body></html>
```

Figure 9: The HTML for the inner iframe corresponding to the outer iframe shown in Figure 8.

3.4 Cookies

In our SOP tests (Section 3.1) we explore the security implications of setting the cookie scope through the `Domain` and `Path` attributes. There are two other important aspects of cookie security: the `HttpOnly` and `Secure` attributes. We test the browser's treatment of these attributes, expecting the behaviour defined in RFC 6265 [17].

The `HttpOnly` attribute of a cookie instructs the browser to reveal that cookie only through an HTTP request; i.e., it should not be made available to client-side scripts. The benefit of this is that, even if an XSS vulnerability is exploited, the cookie cannot be stolen. `HttpOnly` cookies are supported by all major browsers, with the notable exception of Android 2.3's stock browser. BrowserAudit includes tests that check that an `HttpOnly` cookie sent from the server cannot then be accessed by JavaScript, and that `HttpOnly` cookies cannot be created by JavaScript.

When a cookie has the `Secure` attribute set, a compliant browser will include the cookie in an HTTP request only if the request is transmitted over a secure channel (i.e., in an HTTPS request). This keeps the cookie confidential: an attacker would not be able to read it even if he were able to intercept the connection between the victim and the destination server. The `Secure` attribute is supported by all major browsers. BrowserAudit includes tests checking the browser's treatment of the `Secure` attribute both when the cookies are set by the server and set by JavaScript.

3.5 Referer Header

The `Referer` header should not be included in a non-secure request if the referring page was served via a secure protocol; this behaviour is defined in RFC 2616 [31]. This requirement exists because the referrer might disclose an otherwise private information source. In BrowserAudit, we test this behaviour by loading a web page over HTTPS containing an image loaded over HTTP and checking that the `Referer` header was not sent to the server with the request for the image.

3.6 Response Headers

3.6.1 X-Frame-Options

`X-Frame-Options`, defined in RFC 7034 [42], is a server-side technique that can be used to prevent clickjacking attacks. `X-Frame-Options` is a response header that specifies whether or not the document being served is allowed to be rendered in a frame; more specifically, the header specifies the origin (scheme, hostname and port number) that is allowed to render the document in a frame. BrowserAudit tests for correct treatment of the `DENY`, `SAMEORIGIN` and `ALLOW-FROM` directives. The tests try to load iframes served with different headers; each iframe that loads reports its success to the server, which assesses whether the browser behaved as expected. Our tests currently only cover the `<iframe>` element, although the header also applies to `<frame>`, `<object>`, `<applet>` and `<embed>` elements.

`X-Frame-Options` is supported in all modern browsers, although the implementations across browsers differ. Some browsers behave differently when dealing with nested frames, so we do not test these cases as there is no defined correct behaviour. Note also that not all browsers support the `ALLOW-FROM` directive.

3.6.2 Strict-Transport-Security

HTTP Strict Transport Security (HSTS) is a security mechanism that allows a server to instruct browsers only to communicate with it over a secure (HTTPS) connection for the given domain. It exists primarily to defend against man-in-the-middle attacks in which an attacker is able to intercept his victim's network connection [37]. The server sends this instruction via the `Strict-Transport-Security` header, as defined in RFC 6797 [35].

When HSTS is enabled on a domain, a compliant browser must rewrite any plain HTTP requests to that domain to use HTTPS. This includes both URLs entered into the navigation bar by the user, and resources included on a web page. The `Strict-Transport-Security` header should only be sent in an HTTPS response. If the browser receives the header in a response sent over plain HTTP, it should be ignored.

In BrowserAudit, we test the basic behaviour of HSTS and its `includeSubDomains` directive. We also ensure that the header is ignored when transferred via an insecure protocol, and that the HSTS state correctly expires based on the `max-age` value set in the header. All of these tests work by testing whether a request for an image at `http://browseraudit.com/set_protocol` is rewritten to use HTTPS or not.

Almost all current browsers support HSTS, with the notable exception of Internet Explorer 11 (the latest available version at the time of writing).

4. EVALUATION

4.1 Performance

A primary concern of BrowserAudit is scalability, given that a single invocation of the full test suite invokes approximately 1,500 requests and transfers around 3MB of data between the client and server. The server must handle all of these requests quickly (ideally in under 300ms), given the large number of tests in the BrowserAudit test suite and the reliance of some of the tests on timeouts (see Section 2.3).

The BrowserAudit web and database servers are currently hosted on a single virtualised server with two CPU cores and 2GB of memory, running Ubuntu 14.04. We evaluated BrowserAudit's server-side performance by running the BrowserAudit test suite in 15 web browsers repeatedly and concurrently for 15 minutes. Over this period, the BrowserAudit server handled around 225,000 requests and served a total of 450MB of data. The 1- and 5-minute load averages on the BrowserAudit server are shown in Figure 10; the peak load averages over the 15-minute duration of the performance test are 1.2 and 0.7 respectively, where a load average of 1 indicates that a single CPU core is operating at capacity. Based on these performance figures, we estimate that a single BrowserAudit application server using this configuration could comfortably support up to 25 concurrent test suite executions.

As described in Section 2.2, our design is ready to be scaled up as the BrowserAudit user base grows. Nginx can be configured as a load balancer, passing requests to one of many application servers. Deploying Go application server instances is trivial thanks to Go's ability to compile a program to a single statically-linked binary, so there is no dependency chain. In order to maintain session persistence, Nginx's `ip_hash` directive can be used to ensure that all requests from the same IP address reach the same application server, maintaining the integrity of a single suite execution.

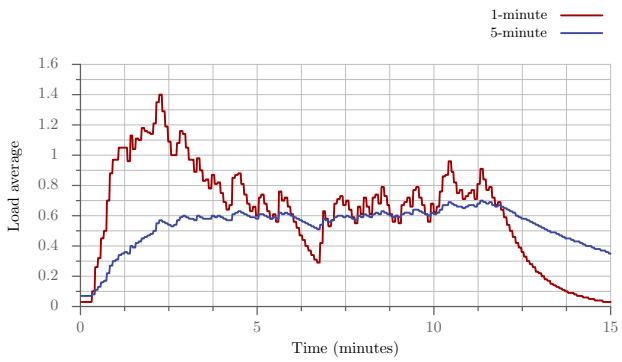


Figure 10: The 1- and 5-minute load averages on the BrowserAudit server during the performance evaluation.

Most client-side tests contain components that are loaded synchronously inside dynamically-created iframes, which become redundant as soon as the test result is reported in the browser; over time, the DOM of the main BrowserAudit window would therefore amass an overwhelming number of iframes, slowing down the execution of tests as the browser struggles to create and append additional iframes. We avoid this problem by dynamically removing any iframes appended to the DOM during each test's tear-down phase (via Mocha's `afterEach()` routine). We ran 15 repetitions of 10 concurrent executions of the whole test suite on a 64-bit Windows 7 machine with a 6-core Intel i7 4930K CPU and 64GB of memory, and Chromium 40.0.2205.0. Under these conditions, the average execution time for the test suite is just over a minute. By contrast, a single execution in Safari 8.0 on an iPhone 5 with iOS 8.1 takes on average 1.35 minutes, skipping 24 tests. The execution time varies broadly across browsers and platforms, but we consider this an acceptable cost for performing an in-depth browser security scan.

4.2 Correctness

Verifying the correctness of our tests is challenging, as they need to convey in a final pass or fail result a whole security-sensitive behaviour: a test containing a small bug could still pass, which is generally the expected result for browsers correctly implementing a given security mechanism.

Of course, no web browsers contain intentional security flaws that would allow us to verify the correctness of tests. Modifying the source code of existing open-source browsers to break their security features in order to ensure that tests fail when expected is possible but challenging given the complexity of modern web browser code bases.

However, it is a matter of public record that some web browsers either do not implement some of the security mechanisms tested by BrowserAudit, or only implement subsets of those security mechanisms. We leverage the results of browser-profiling projects such as *Browserscope* [3] and *Can I Use...* [6] to broadly identify the security features implemented by each web browser, and for those features we manually verify that the BrowserAudit test suite results are accurate.

Using *BrowserStack* [5], a web-based browser testing service, we have evaluated BrowserAudit in a range of browsers on a number of different operating systems, across both

desktop and mobile platforms. The full BrowserAudit test suite runs reliably in Safari 6, Firefox 13 and Chrome 25 or more recent versions, automatically skipping tests where a feature is not supported. BrowserAudit also runs correctly on Internet Explorer 11, but due to problems relating to Mocha and IE's limited call stack, it cannot execute the whole test suite. In older versions of these browsers, it is instead possible to run a subset of the test suite.

4.3 Test Coverage

We noted in Section 2 that full coverage for browser security feature tests is unattainable. Here we discuss a number of security features not covered by BrowserAudit, but that we believe can be added to our framework.

We imply in Section 3 that there is no single same-origin policy but rather a collection of related security mechanisms. We currently test the same-origin policy for DOM access, XMLHttpRequest and cookies. This could be expanded to test the same-origin policies for Flash, Java, Silverlight, and HTML5 web storage.

The postMessage API is used by many developers to communicate across origins [19]. Since the API allows the sender of a message to specify the origins of the recipients that may receive the message, there are lots of origin-related tests that we could write for this feature in BrowserAudit.

Another security feature that could be tested is the `X-Content-Type-Options` response header first introduced in Internet Explorer 8 [40]. It is now also supported by Chromium and Safari; the Firefox team is still debating its implementation [26]. It is designed to prevent browser-sniffing attacks where a resource (e.g., a HTML document) is sent with an inappropriate MIME type (e.g., `text/plain`) but is nonetheless erroneously rendered by the browser as if the correct MIME type had been sent [18].

In Sections 3.1–3.4 we discussed how to extend coverage of features for which we already have some tests. Summarising, the main limitations are that: in many tests involving origin mismatches, we only test origins that differ by hostname rather than by scheme or port number; we do not test CSP directives where a resource is loaded from a URL that redirects; we do not test that cookies cannot be set for top-level domains that include a country code, such as `co.uk` (whereas, for example, they should be settable for `example.uk`). We also do not test the `Report-Only` header defined by the CSP standard, but this is not due to a limitation of the BrowserAudit framework and a suitable test could be added to the test suite.

Finally, cryptographic APIs such as the W3C WebCrypto API and the OpenSSL library are important aspects of browser security, but cryptographic testing is beyond the scope of BrowserAudit and better left to dedicated projects such as *How's My SSL?* [8].

4.4 Uncovering Security Bugs

BrowserAudit's test suite has uncovered two previously-unknown bugs in Firefox's implementation of the CSP standard; these bugs are present in all versions of Firefox that implement the CSP standard up to version 32.0.3. The first bug [24] allows the loading of same-origin stylesheets with the policy

```
default-src 'none'; style-src 'unsafe-inline';
```

similarly, the second bug [25] allows the loading of same-origin Worker and SharedWorker objects in scripts with the policy

```
default-src 'none'; script-src 'unsafe-inline'.
```

In both cases, the `'unsafe-inline'` declaration in the policy states that only inline stylesheets and scripts must be permitted: external resources, even those from the same origin, must be blocked. We reported both of these bugs to Mozilla during the version 29 release cycle, and they were fixed in version 33 of Firefox.

Firefox does not currently implement the `sandbox` CSP directive; this optional feature of the CSP 1.0 specification directs browsers to relax the given security controls on iframes embedded in the page, as if they had been supplied in the `sandbox` attribute of each `<iframe>` element. The `sandbox` attribute is in fact a feature of the HTML5 specification [34] and states that an iframe containing a `sandbox` attribute should have *all* security controls enabled unless specifically disabled by values inside the `sandbox` attribute. Development work on the implementation of this directive in Firefox is currently underway [27]. However, the current implementation does not correctly handle the case where an empty value is given for the `sandbox` CSP directive; the CSP 1.0 specification implies that the browser should apply a `sandbox` attribute with an empty value (and thus enforce a highly-restrictive sandboxing policy — a view also taken by developers of other browsers, such as Chromium), but Firefox's implementation does not apply a `sandbox` attribute at all in this scenario (thus failing to enforce *any* sandboxing policy). This flaw was uncovered by the current set of CSP tests in BrowserAudit, and we are in discussions with Firefox developers to address it before their `sandbox` implementation lands in a stable version of the browser.

5. RELATED WORK

In this section we discuss some related work on browser security, which influenced the design of our tests, and review some web applications that perform security-relevant tests, which served as a source of inspiration for BrowserAudit.

5.1 Browser Security

The authoritative sources of information on upcoming browser security mechanisms are of course the W3C RFCs and Drafts such as [34, 45, 21, 46, 35]. Most security measures are the result of a lot of practical experimentation and academic research that led to proposals that gradually gained adoption and became more robust through security reviews and public scrutiny. Paradigmatic examples are the early contributions of Barth, Jackson et al. to postMessage, the `Origin` header and HTTPS [19, 37, 20].

The standards themselves provide a lot of detail about the intended security behaviour, but additional research is needed to interpret the consequences for deployed web applications. For example, De Ryck et al. perform a security analysis of some of the upcoming standards in [28], finding them to be be of high quality but also highlighting potential security risks. Singh et al. [44] discover potentially dangerous incoherencies amongst different browser access control policies.

A broad, in-depth analysis of browser security can be found in Zalewski's *Browser Security Handbook* [47] and the companion book *The Tangled Web* [48]; they gather a wealth of information on browser security features, their shortcomings and the peculiar differences in browser support.

5.2 Web Sites

Panopticlick [12] is an experiment to investigate how unique — and therefore trackable — modern web browsers are, by fingerprinting their version and configuration information. Some of this information can be gleaned directly from browser requests, whereas other information is made available by the presence of JavaScript and browser plugins. Visitors click a "Test Me" button and are then provided with their browser's uniqueness score and a breakdown of the measurements used to obtain the result. These data are then anonymously stored in the project database to make future uniqueness scores more accurate, and to allow for analysis of the data, as discussed in [30]. Although focussed on privacy rather than security, *Panopticlick* was the main inspiration for BrowserAudit.

BrowserSpy [4] is another web site that reports how much information can be retrieved from a browser by visiting a test page. Its focus is on privacy, yet some of its tests are security-related, although not presented as such; for example, one test checks that JavaScript cannot read `HttpOnly` cookies. Each of *BrowserSpy*'s 75 current tests has to be run individually, since the output is rather verbose, and the output does not show implementation details that could be useful for a technical audience. In contrast, our 400+ tests run automatically, and advanced users can view the client-side code driving each individual test.

How's My SSL? [8] is a recent project that advises the user on the security of their TLS client (web browsers act as TLS clients when engaged in HTTPS communication). It works by running a TLS server that has been modified so that the client-server handshake is exposed to the web application, allowing it to inspect the cipher suites that the client supports and perform a security assessment. The results are reported clearly, with "Learn More" links for more technical background which also inspired our design. The test results can be accessed via a JSON API, and could be potentially integrated into BrowserAudit to complement our tests. *Qualys SSL Labs* [14] also offers browser-based tests for SSL clients that display a concise report of their TLS capabilities, intended for the expert user. In BrowserAudit we instead strived to produce reports that can be interpreted by users at different levels of technical competence.

The *Can I Use...* test suite [15] gathers browser compatibility data for a wide variety of browser features such as support for HTML5 and CSS3. Some of these tests are automatic and others require visual confirmation or interaction from the user. A few tests check for support for security features; for example, one (interactive) test detects support for the CSP. In contrast, BrowserAudit runs 226 automated tests to assess the security of the CSP implementation.

The *Browser DOM access checker* [2] is a web page also included in the Chromium browser source code that uses JavaScript to test the enforcement of some domain-related security policies such as cross-domain DOM access, JavaScript cookie access, `XMLHttpRequest` calls, and event and transition handling; for example, it runs hundreds of tests to ensure that read or write attempts to the visible properties of the `document` object are blocked cross-domain. In contrast, we are satisfied with testing cross-domain access for one representative property of the `document` object: if such access is blocked, we conclude that the policy is effective. We could programmatically extend our tests to try accessing all properties, but that goes beyond the scope of BrowserAudit: DOM-based cross-domain access is only one of the hundreds of qualitatively different behaviours that we consider.

Finally, *Browserscope* [3] is a community-driven project for profiling web browsers; it detects the browser version and runs tests that cover a broad range of features such as network performance, CSS support, and JavaScript optimisations. Test results are aggregated and made publicly available, making it easy for web developers to keep track of functionality across all browsers that have been tested.

Currently, *Browserscope* also includes 17 tests which automatically check whether the browser supports a number of standard features relevant to security and displays a list of which tests passed or failed. In contrast, BrowserAudit is engineered to run hundreds of tests that ascertain whether security features are implemented correctly, and provides an interface that allows different types of users to access detailed descriptions of each test case, including client-side test code.

6. CONCLUSIONS

We introduced BrowserAudit, a web application to test the implementation of browser security features. It complements the unit testing used by browser vendors to debug their implementations by checking that deployed browsers effectively deliver the security behaviours entailed by the specifications of browser security mechanisms.

All of our tests run automatically without interaction from the user, and provide detailed information for each test category, including the source code of each individual test. This makes BrowserAudit useful for a broad audience, from the casual user to the web developer and the security researcher. No other publicly-accessible web application tests such a breadth of browser security mechanisms as ours, either established or experimental.

In Section 4.3 we highlighted aspects of browser security mechanisms that are currently not covered by our tests. BrowserAudit is designed to be modular and extensible; adding variants of existing tests with different combinations of parameters, or new client-side-only tests (e.g., to test different features of the SOP) is straightforward. We are currently investigating the more challenging problem of allowing similar extensibility of the server-side components of tests.

BrowserAudit is an open-source project [32], and we hope that the web security community will help us extend it with even more test cases.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers of ISSTA'15 for their comments and suggestions. This work began as Hothersall-Thomas's final year project at Imperial College London. Maffeis is supported by EPSRC grant EP/I004246/1 and Novakovic is supported by EPSRC grant EP/K032089/1.

8. REFERENCES

[1] Bootstrap. `http://getbootstrap.com/`.
[2] Browser DOM access checker. `http://lcamtuf.coredump.cx/dom_checker/`.
[3] Browserscope. `http://www.browserscope.org/`.
[4] BrowserSpy. `http://browserspy.dk/`.
[5] BrowserStack. `http://www.browserstack.com/`.
[6] Can I Use.... `http://caniuse.com/`.
[7] Chai. `http://chaijs.com/`.

[8] How's My SSL? https://www.howsmyssl.com/.
[9] jQuery. http://jquery.com/.
[10] Mocha. http://mochajs.org/.
[11] Nginx. http://nginx.org/.
[12] Panopticlick. https://panopticlick.eff.org/.
[13] PostgreSQL. http://www.postgresql.org/.
[14] Qualys SSL Labs. https://www.ssllabs.com/.
[15] The Can I Use... test suite. http://tests.caniuse.com/.
[16] The Go Programming Language. https://golang.org/.
[17] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), Apr. 2011.
[18] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of S&P 2009*, pages 360–371, 2009.
[19] A. Barth, C. Jackson, and J. Mitchell. Securing Frame Communication in Browsers. In *Proceedings of USENIX Security 2008*, pages 17–30, 2008.
[20] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-site Request Forgery. In *Proceedings of CCS'08*, pages 75–88, 2008.
[21] A. Barth and M. West. Content Security Policy 1.1, June 2013. W3C Working Draft WD-CSP11-20130604.
[22] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis. Language-Based Defenses Against Untrusted Browser Origins. In *Proceedings of USENIX Security 2013*, pages 653–670, 2013.
[23] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang. You Can't Be Me: Enabling Trusted Paths and User Sub-origins in Web Browsers. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Proceedings of RAID 2014*, volume 8688 of *Lecture Notes in Computer Science*, pages 150–171. Springer, 2014.
[24] Bugzilla. Bug 1007205 — CSP allows local CSS @import with only 'unsafe-inline' set. https://bugzilla.mozilla.org/show_bug.cgi?id=1007205.
[25] Bugzilla. Bug 1007634 — CSP allows local Worker construction with only 'unsafe-inline' set. https://bugzilla.mozilla.org/show_bug.cgi?id=1007634.
[26] Bugzilla. Bug 471020 — Add X-Content-Type-Options: nosniff support to Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=471020.
[27] Bugzilla. Bug 671389 — Implement CSP sandbox directive. https://bugzilla.mozilla.org/show_bug.cgi?id=671389.
[28] P. De Ryck, L. Desmet, P. Philippaerts, and F. Piessens. A security analysis of next generation web standards. Technical report, ENISA, July 2011.
[29] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with CRYPTONS. In *Proceedings of CCS'13*, pages 1311–1324, 2013.
[30] P. Eckersley. How unique is your web browser? In *Proceedings of PETS'10*, pages 1–18, 2010.
[31] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.
[32] GitHub. BrowserAudit project. https://github.com/browseraudit/.
[33] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless Attacks: Stealing the Pie Without Touching the Sill. In *Proceedings of CCS'12*, pages 760–771, 2012.
[34] I. Hickson and D. Hyatt. HTML5: A vocabulary and associated APIs for HTML and XHTML. W3C Candidate Recommendation CR-HTML5-20140429, Apr. 2014.
[35] J. Hodges, C. Jackson, and A. Barth. HTTP Strict Transport Security (HSTS). RFC 6797 (Proposed Standard), Nov. 2012.
[36] L.-S. Huang, A. Moshchuk, H. J. Wang, S. Schechter, and C. Jackson. Clickjacking: Attacks and Defenses. In *Proceedings of USENIX Security 2012*, pages 22–22, 2012.
[37] C. Jackson and A. Barth. Forcehttps: Protecting High-security Web Sites from Network Attacks. In *Proceedings of WWW'08*, pages 525–534, 2008.
[38] E. Kirda. Cross Site Scripting Attacks. In *Encyclopedia of Cryptography and Security*, pages 275–277. 2011.
[39] S. Maffeis, J. C. Mitchell, and A. Taly. Object Capabilities and Isolation of Untrusted Web Applications. In *Proceedings of S&P 2010*, pages 125–140, 2010.
[40] MSDN Blogs. IE8 Security Part VI: Beta 2 Update. http://blogs.msdn.com/b/ie/archive/2008/09/02/ie8-security-part-vi-beta-2-update.aspx.
[41] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang. Towards Fine-Grained Access Control in JavaScript Contexts. In *Proceedings of ICDCS'11*, pages 720–729, 2011.
[42] D. Ross and T. Gondrom. HTTP Header Field X-Frame-Options. RFC 7034 (Informational), Oct. 2013.
[43] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting Framebusting: a Study of Clickjacking Vulnerabilities at Popular Sites. In *Proceedings of W2SP 2010*, 2010.
[44] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the Incoherencies in Web Browser Access Control Policies. In *Proceedings of S&P 2010*, pages 463–478, 2010.
[45] B. Sterne and A. Barth. Content Security Policy 1.0. Nov. 2012. W3C Candidate Recommendation CR-CSP-20121115.
[46] A. Van Kesteren. Cross-origin Resource Sharing. W3C Recommendation REC-cors-20140116, Jan. 2014.
[47] M. Zalewski. Browser Security Handbook, 2010.
[48] M. Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. No Starch Press, 2012.

Hybrid Typing of Secure Information Flow in a JavaScript-like Language

José Fragoso Santos, Thomas Jensen, Tamara Rezk, Alan Schmitt

Hybrid Typing of Secure Information Flow in a JavaScript-like Language

José Fragoso Santos[1], Thomas Jensen[2], Tamara Rezk[2], Alan Schmitt[2]

[1] Imperial College London, jfaustin@imperial.ac.uk
[2] Inria, firstname.lastname@inria.fr

Abstract. As JavaScript is highly dynamic by nature, static information flow analyses are often too coarse to deal with the dynamic constructs of the language. To cope with this challenge, we present and prove the soundness of a new hybrid typing analysis for securing information flow in a JavaScript-like language. Our analysis combines static and dynamic typing in order to avoid rejecting programs due to imprecise typing information. Program regions that cannot be precisely typed at static time are wrapped inside an internal *boundary* statement used by the semantics to interleave the execution of statically verified code with the execution of code that must be dynamically checked.

1 Introduction

The dynamic aspects of JavaScript make the analysis of JavaScript programs very challenging. On one hand, one may use a purely static analysis, but either restrict the language to exclude these dynamic aspects or over-approximate them; this is too coarse to be applicable in practice. On the other hand, one may use purely dynamic mechanisms, such as monitoring or secure multi-executions [1, 6, 8, 16]; but the gained precision comes at the cost of a much lower performance compared to the original code [7].

We propose a general hybrid analysis to statically verify secure information flow in a core of JavaScript. Following the hybrid typing motto "static analysis where possible with dynamic checks where necessary"[5], we are able to reduce the runtime overhead introduced by purely dynamic analyses without excluding dynamic field operations. In fact, our analysis can handle some of the most challenging JavaScript features, such as prototype-based inheritance, extensible objects, and constructs for checking the existence of object properties. Its key ingredient is an internal boundary statement inspired by recent work in inter-language interoperability [10]. The static component of our analysis wraps program regions that cannot be precisely verified inside an internal *boundary* statement instead of rejecting the whole program. This boundary statement identifies the regions of the program that must be verified at runtime—which may be as small as a single statement—and enables the initial set up required by the dynamic analysis. In summary, the proposed boundary statement allows the semantics to effortlessly interleave the execution of statically verified code with the execution of code that must be verified at runtime.

Although our work is generally motivated by the verification of dynamic features of JavaScript, we choose to focus on the particular case of constructs that rely on dynamic computation of object field names, which we call *dynamic field operations*. In JavaScript, one can access a field `f` of an object `o` either by writing `o.f` or `o[e]`, where `e` is an expression that dynamically evaluates to the string `f`. Dynamic computation of field names is one of the major sources of imprecision of static analyses for JavaScript [9].

Example 1 (Running example: the challenge of typing dynamic field operations). Below we present a program that creates an object `o` with a secret field `secret1` and two public fields `public1` and `public2`.

```
o = {}; o.secret1 = secret_input();
o.public1 = public_input(); o.public2 = public_input(); public = o[g()]
```

The secret field `secret1` gets a secret input via function `secret_input`, while the two public fields `public1` and `public2` each get a public input via function `public_input`. The program then assigns the value of one of the three fields to the public variable `public`, as determined by the return value of function `g`. Concretely, when `g` returns the string `"secret1"`, the program assigns a secret value to `public` and the execution is insecure. On the other hand, when `g` returns either `"public1"` or `"public2"`, the program assigns a public value to `public` and the execution is secure. However, in order to make sure that `g` never returns `"secret1"`, a static analysis needs to predict the dynamic behaviour of `g`, which is, in general, undecidable.

The loss of precision introduced by the dynamic computation of field names is not exclusive to field projections. It also occurs in method calls, field deletions, and membership checks. We account for the use of these operations by verifying them at runtime. When verifying a statement containing a dynamic field operation, the static component of the analysis wraps it inside a boundary statement. In the case of the running example, all statements except the last one are statically typed. In contrast, the last assignment is re-written as `@monitor(@type_env, @pc, @ret, public = o[g()])`, where the first three arguments of the monitor statement are used for the setup of the runtime analysis. Hence, when the program is executed the only overhead introduced by the dynamic component of our hybrid analysis regards the security checks for validating or rejecting the statement `public = o[g()]`.

Contributions. The main contribution of the paper is the design of a new hybrid analysis for verifying secure information flow in a JavaScript-like language. To achieve this, we introduce: **(1)** a type language specifically designed to control information flow in a subset of JavaScript, **(2)** a static type system for verifying statements not containing dynamic field operations, **(3)** a dynamic typing analysis for verifying statements containing dynamic field operations, and **(4)** a novel boundary operator for interleaving the execution of statically verified regions with dynamically verified ones. Finally, we have implemented a prototype as well as a case study, available online at [15].

$$
\begin{array}{rcl}
v \in \mathcal{V}al & ::= & lit \mid \underline{lit} \mid l \mid \lambda x : \dot{\tau}.s \\
e \in \mathcal{E}xpr & ::= & v \mid \mathsf{this} \mid x \mid x = e \mid \{\,\}\,[\dot{\tau}] \mid e.f \mid e_1[e_2] \mid e_1.f = e_2 \\
 & & \mid e_1[e_2] = e_3 \mid f \,\mathsf{in}\, e \mid [e_1] \,\mathsf{in}\, e_2 \mid \mathsf{delete}\, e.f \mid \mathsf{delete}\, e_1[e_2] \\
 & & \mid \mathsf{function}\,(x)[\dot{\tau}]\{s\} \mid e_1(e_2) \mid e_1.x(e) \mid e_1[e_2](e_3) \\
s \in \mathcal{S}tmt & ::= & e \mid \mathsf{var}\, x\,[\dot{\tau}] \mid s_1;\, s_2 \mid \mathsf{if}(e)\,\{s_1\}\,\mathsf{else}\,\{s_2\} \mid \mathsf{return}\, e
\end{array}
$$

Table 1. Core JS Syntax - Expressions and Statements

2 Core JS

Syntax. The syntax of Core JS is given in Table 1. Expressions include values, the keyword this, variables, variable assignments, object literals, static and dynamic field projections, static and dynamic field assignments, static and dynamic membership checks, static and dynamic field deletions, function literals, function calls, and static and dynamic method calls. Statements include expression statements, variable declarations, sequences, conditional statements, and return statements. We distinguish two types of *values*: literal values and runtime values. Literal values include numbers, booleans, strings, and undefined. Runtime values, ranged over by $\underline{v}$, include parsed literal values, locations, and parsed function literals. Object literals, function literals, and variable declarations are annotated with their respective *security types* (which are explained in Section 3). In the following, we use $\mathcal{E}xpr_{\dot{\iota}}$ for the set of Core JS dynamic field operations.

Memory Model. A heap $H \in \mathcal{H}eap : \mathcal{L}oc \times \mathcal{X} \rightharpoonup \mathcal{V}al$ is a partial mapping from locations in $\mathcal{L}oc$ and field names in $\mathcal{X}$ to values in $\mathcal{V}al$. We denote a heap cell by $(l, f) \mapsto v$, the union of two disjoint heaps by $H_1 \uplus H_2$, a read operation by $H(l, f)$, and a heap update operation by $H[l.f \mapsto v]$. An object can be seen as a set of heaps cells addressed by the same location but with different field names. We use $l \mapsto \{f_1 : v_1, \ldots, f_n : v_n\}$ as an abbreviation for the object $(l, f_1) \mapsto v_1 \uplus \ldots \uplus (l, f_n) \mapsto v_n$.

Every object has a prototype, whose location is stored in a special field *_proto_*. In order to determine the value of a field f of an object o, the semantics first checks whether f is one of the fields of o. If that is the case, the field look-up yields that value. Otherwise, the semantics checks whether f belongs to the fields of the prototype of o and so forth. The sequence of objects that can be accessed from a given object through the inspection of the respective prototypes is called a prototype chain. The prototype chain inspection procedure is modelled by the semantic function π given in appendix. Informally, the expression $\pi(H, l, f)$ denotes the location of the first object that defines f in the prototype chain of the object pointed to by l (if no such object exists, π returns null). Given that most implementations of JavaScript allow for explicit prototype mutation, we include this feature in Core JS. For instance, $x.\_proto\_$ evaluates to the the prototype of the object bound to x and $x.\_proto\_ = y$ sets the prototype of the object bound to x to the object bound to y.

Scope is modelled using *environment records*. An environment record is simply an internal object that maps variable names to their respective values.

$$
\begin{array}{rcl}
\hat{E} & ::= & \Box \mid x = \hat{E} \mid \hat{E}.f \mid \hat{E}[e] \mid l[\hat{E}] \mid \hat{E}.f = e \mid \hat{E}[e_1] = e_2 \\
& & \mid l[\hat{E}] = e \mid l[f] = \hat{E} \mid [\hat{E}]\,\mathsf{in}\,e \mid [f]\,\mathsf{in}\,\hat{E} \mid \mathsf{delete}\,\hat{E}.f \mid \mathsf{delete}\,\hat{E}[e] \\
& & \mid \mathsf{delete}\,l[\hat{E}] \mid \hat{E}(e) \mid l(\hat{E}) \mid \hat{E}.f(e) \mid \hat{E}[e](e) \mid l[\hat{E}](e) \mid l[f](\hat{E}) \\
E & ::= & \hat{E} \mid E;\, s \mid \mathsf{if}(\hat{E})\,\{s_1\}\,\mathsf{else}\,\{s_2\} \mid \mathsf{return}\,\hat{E}
\end{array}
$$

Table 2. Evaluation Contexts

An environment record is created for every function or method call. We use $\mathsf{act}(l, x, v, s, l')$ to denote the environment record that: **(1)** is identified by location l where it is stored, **(2)** maps x to v, **(3)** maps all the variables declared in s to undefined, and **(4)** maps the field @*this* to the location l'. (Note that environment records map a single variable because functions have a single argument. Moreover, in the execution of a method call, the field @*this* is used to store the location of the object on which the method was invoked.) Variables are resolved with respect to a list of environment record locations, called *scope chain*. The variable inspection procedure is modelled by the semantic function σ given in appendix. We let $\sigma(H, L, x)$ denote the location of the first environment record that defines x in the scope chain L. The global object, assumed to be pointed to by a fixed location l_g, is the environment record that binds global variables.

Since functions are first-class citizens, the evaluation of a function literal triggers the creation of a special type of object, called *function object*. Every function object has two fields: @*body* and @*scope*, which respectively store the corresponding parsed function literal and the scope chain that was active when the function literal was evaluated. Functions execute in the scope in which the they were evaluated.

Semantics. Figure 1 presents a fragment of the semantics of Core JS in the style of Wright and Felleisen [19] (the full semantics is given in appendix). A configuration Ψ has the form $\langle H, L, s\rangle$ where H is the current heap, L the current scope chain, and s the statement to execute. Transitions are labelled with an internal event α for the use of the dynamic analysis. The evaluation order is specified with the help of evaluation contexts, whose syntax is given in Table 2. In the following, we use $l :: L$ for the list obtained by prepending l to L and $\mathsf{head}(L)$ for the first element of L.

Rule Variable uses σ to determine the location l' of the environment record that defines x and reads its value from the heap. Rule Dyn Field Projection uses π to determine the location l'' of the object that defines f in the prototype chain of the object pointed to by l' and then reads its value from the heap. Rule Dyn Field Assignment updates the current heap with a mapping from l and f to $\underline{v}$. Rule Membership Check - True checks if f is defined in the prototype chain of the object pointed to by l and evaluates to true. Rule Function Literal adds a new function object to the heap. Rule Function Call extends the heap with a new environment record for the evaluation of the function pointed to by l. The current scope chain L is replaced with the scope chain L' that was active when the corresponding function literal was evaluated extended with the location l'' of the newly created environment record. The se-

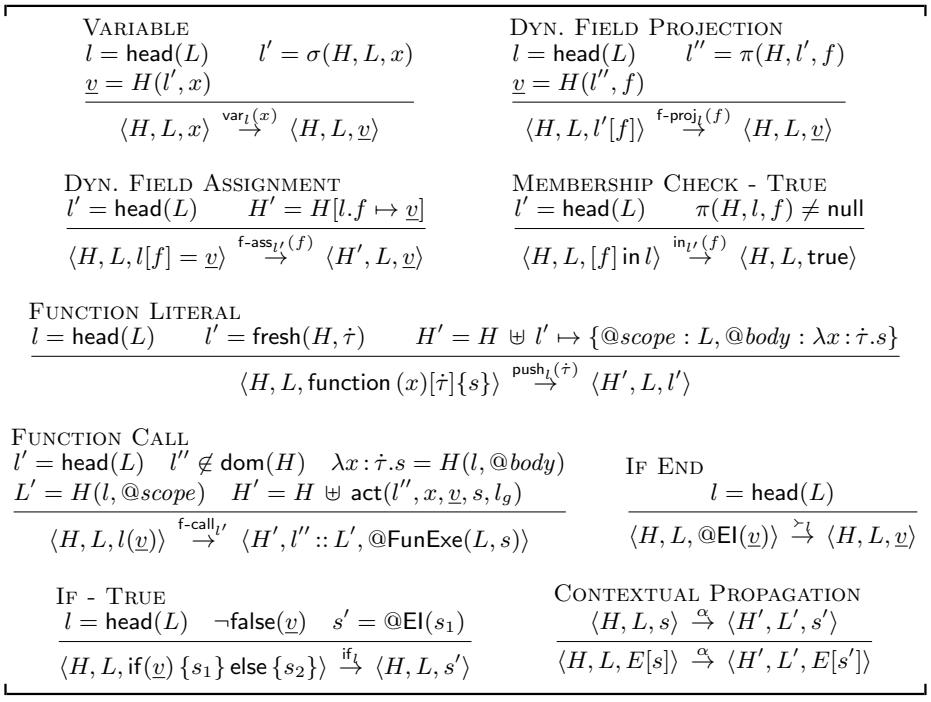


Fig. 1. Fragment of the Small-Step Semantics of Core JS

mantics makes use of an internal statement $@\mathsf{FunExe}(L, s)$ for keeping track of the caller's scope chain during the execution of the function's body. Rule If - True checks if the guard of the conditional does not belong to the set of *falsy* values –$\{\mathsf{false}, 0, \mathsf{undefined}, \mathsf{null}\}$– and replaces the whole conditional with its then-branch followed by an internal statement $@\mathsf{EI}$ for notifying the dynamic analysis of the end of that branch. Contextual Propagation is standard.

3 Static Typing Secure Information Flow in Core JS

In this section, we present both a new type language for controlling information flow in JavaScript and the static component of our analysis. Here, the specification of security policies relies on two key elements: a lattice of security levels and a typing environment that maps resources to security types, which can be viewed as safety types annotated with security levels. In the examples, we use $\mathcal{L} = \{H, L\}$ with $L \sqsubset H$, meaning that L-labelled resources (*low* resources) are less confidential than those labelled with H (*high*). We use $\sqcup$, $\bot$, and $\top$ for the least upper bound (*lub*), the *bottom* level, and the *top* level, respectively.

Security Types. A security type $\dot\tau = \tau^\sigma$ is obtained by pairing up a *raw type* τ with a security level σ, called its *external level*. The external level of a security type establishes an upper bound on the levels of the resources on which the values

of that type may depend. For instance, a primitive value of type $\textsf{PRIM}^L$ may only depend on *low* resources. The syntax of raw types is given and explained below:

$$\begin{aligned}\tau ::= \textsf{PRIM} \mid\ & \langle \dot\tau.\dot\tau \xrightarrow{\sigma} \dot\tau\rangle \mid\ \langle \kappa.\dot\tau \xrightarrow{\sigma} \dot\tau\rangle \\ & \mid \mu\kappa.\langle f^\sigma : \dot\tau, \cdots, f^\sigma : \dot\tau, *^\sigma : \dot\tau\rangle \mid \mu\kappa.\langle f^\sigma : \dot\tau, \cdots, f^\sigma : \dot\tau\rangle\end{aligned}$$

- The type PRIM is the type of expressions which evaluate to primitive values.
- The type $\langle \dot\tau_0.\dot\tau_1 \xrightarrow{\sigma} \dot\tau_2\rangle$ is the type of expressions which evaluate to functions that map values of type $\dot\tau_1$ to values of type $\dot\tau_2$ and during the execution of which, the keyword this is bound to an object of type $\dot\tau_0$. Level σ is the *writing effect* [14] of functions of this type, that is, a lower bound on the levels of the resources updated or created during their execution. When specifying a function type inside an object type, one can use the type variable bound by that object type as the type of the keyword this (in the syntax of types, κ ranges over the set of type variables).
- The type $\mu\kappa.\langle f_0^{\sigma_0} : \dot\tau_0, \cdots, f_n^{\sigma_n} : \dot\tau_n, *^{\sigma_*} : \dot\tau_*\rangle$ is the type of expressions which evaluate to objects that *may* define the fields f_0 to f_n mapping each field f_i to a value of security type $\dot\tau_i$. The security type assigned to $*$ is the *default security type*, which is the security type of all fields not in $\{f_0, \cdots, f_n\}$. Every field f_i is further associated with an *existence level* σ_i that establishes an upper bound on the levels of the contexts in which the field can be created or deleted. The level σ_* is the *default existence level*. When no default security type is declared, the objects of the type may only define explicitly declared fields.
  The reason why we do not precisely track the presence of fields in object types is that we do not want the type of an object to change at runtime even though its structure may change. Notice that the absence of a field in a type does not mean it cannot be accessed in objects of that type: this field may still be defined in the prototype chain. We could have flattened security types for objects by requiring every object type to explicitly declare all the fields accessible through the prototype chains of the objects of that type, but this would have two disadvantages. First, object types would be less precise, and second, they would be much larger as the types of prototype fields would be duplicated. The cost of this design choice is a more complex Static Field Projection typing rule that has to take the prototype chain into account.

Given a security type $\dot\tau$, the expression $\mathsf{lev}(\dot\tau)$ denotes its external level and $\lfloor\dot\tau\rfloor$ its raw type (for instance, $\mathsf{lev}(\textsf{PRIM}^L) = L$ and $\lfloor\textsf{PRIM}^L\rfloor = \textsf{PRIM}$). We define $\dot\tau^\sigma$ as $\lfloor\dot\tau\rfloor^{\mathsf{lev}(\dot\tau)\sqcup\sigma}$ (for example, $(\textsf{PRIM}^L)^H = \textsf{PRIM}^H$). Given a function security type $\dot\tau = \langle \dot\tau_0.\dot\tau_1 \xrightarrow{\sigma} \dot\tau_2\rangle^{\sigma'}$, we use $\dot\tau.\mathsf{this}$, $\dot\tau.\mathsf{arg}$, $\dot\tau.\mathsf{ret}$, and $\dot\tau.\mathsf{wef}$ to denote $\dot\tau_0$, $\dot\tau_1$, $\dot\tau_2$, and σ, respectively. Given an object security type $\dot\tau$, we use $\mathsf{dom}(\dot\tau)$ for the set containing all field names explicitly declared in $\dot\tau$ (including $*$, if present). Given a field name f and an object security type $\dot\tau$, $\dot\tau.f$ ($\dot\tau.\overline{f}$, resp.) denotes either the security type (existence level resp.) with which $\dot\tau$ associates f or its default security type (existence level, resp.) when $f \notin \mathsf{dom}(\dot\tau)$ and $* \in \mathsf{dom}(\dot\tau)$. The ordering $\sqsubseteq$ on security levels induces a simple ordering $\preceq$ on security types: $\dot\tau_0 \preceq \dot\tau_1$ *iff* $\mathsf{lev}(\dot\tau_0) \sqsubseteq \mathsf{lev}(\dot\tau_1)$ and $\lfloor\dot\tau_0\rfloor = \lfloor\dot\tau_1\rfloor$. We use $\dot\tau_g$ for the type of the global

$$
\begin{array}{rl}
\Gamma(\texttt{public}) = \textsf{PRIM}^L & \\
\Gamma(\texttt{secret}) = \textsf{PRIM}^H & \\
\Gamma(\texttt{secret\_input}) = \langle \dot{\tau}_g \xrightarrow{H} \textsf{PRIM}^H \rangle^L & \dot{\tau}_o = \mu\kappa \cdot \left\langle \begin{array}{l} \texttt{public1}^L : \textsf{PRIM}^L, \\ \texttt{public2}^L : \textsf{PRIM}^L, \\ \texttt{secret1}^H : \textsf{PRIM}^H \\ \texttt{secret2}^H : \textsf{PRIM}^H \end{array} \right\rangle^L \\
\Gamma(\texttt{public\_input}) = \langle \dot{\tau}_g \xrightarrow{H} \textsf{PRIM}^L \rangle^L & \\
\Gamma(\texttt{g}) = \langle \dot{\tau}_g \xrightarrow{H} \textsf{PRIM}^L \rangle^L & \Gamma(\texttt{o0}) = \mu\kappa.\langle \texttt{\_proto\_}^H : \dot{\tau}_o \rangle^L \\
\multicolumn{2}{c}{\Gamma(\texttt{o}) = \Gamma(\texttt{o1}) = \Gamma(\texttt{o2}) = \dot{\tau}_o}
\end{array}
$$

Table 3. Typing Environment for the Examples 1 to 6

object. Finally, a typing environment Γ is simply a mapping from variables to security types.

Example 2. Table 3 presents the typing environment used to type the programs given in Examples 1 to 6. Since `secret_input`, `public_input`, and `g` are to be used as functions, their respective types use the type of the global object as the type of the keyword this. Since none of these three functions expects an argument or updates the heap, their respective types omit the type of the argument and declare a *high* writing effect. Our design choice of not flattening object types can also be seen in this example: the type of `o0` is much shorter as it does not need to mention at top level the fields declared in $\dot{\tau}_o$.

Static Type System. The key insight of the static type system is that it wraps program regions which cannot be precisely analysed at static time within a boundary statement $@monitor(\Gamma, pc, \dot{\tau}_r, s)$ responsible for turning on the typing analysis at runtime. The parameters Γ, pc, and $\dot{\tau}_r$ are the typing environment, the context level [14], and the type of the function whose body is being typed, respectively. Given a typing environment Γ, a level pc, and an expression e, the typing judgment $\Gamma, pc \vdash_e e \hookrightarrow e' : \dot{\tau}$ means that e is rewritten as a semantically equivalent expression e', which may include boundary statements, has raw type $\lfloor \dot{\tau} \rfloor$, and reads variables or fields of level at most $\mathsf{lev}(\dot{\tau})$. Typing judgements for statements, with the form $\Gamma, pc, \dot{\tau}_r \vdash_s s \hookrightarrow s'$, differ from typing judgements for expressions in that they do not assign a type to the statement. When e (s resp.) coincides with e' (s' resp.), we omit $\hookrightarrow e'$ ($\hookrightarrow s'$ resp.) from the typing rules. The most relevant typing rules are given in Figure 2 and described below. (We omit the explanations of Rules Literal, Variable, and Assignment as they are standard.)

Static field projection As a given field may be defined anywhere in the prototype chain of the inspected object, this rule needs to take into account the whole prototype chain of that object. To this end, we overload function π to model a static prototype chain inspection procedure. Informally, $\pi(\dot{\tau}, f)$ computes the *lub* between the security types of f in the prototype chain of objects of type $\dot{\tau}$ and upgrades the external level of this type with the *lub* between the existence levels of the field *_proto_* in that prototype chain.

Example 3 (Leaks via Prototype Mutations). The program below creates three empty objects bound to: `o0`, `o1`, and `o2`. Then, it creates a field named `public1` in

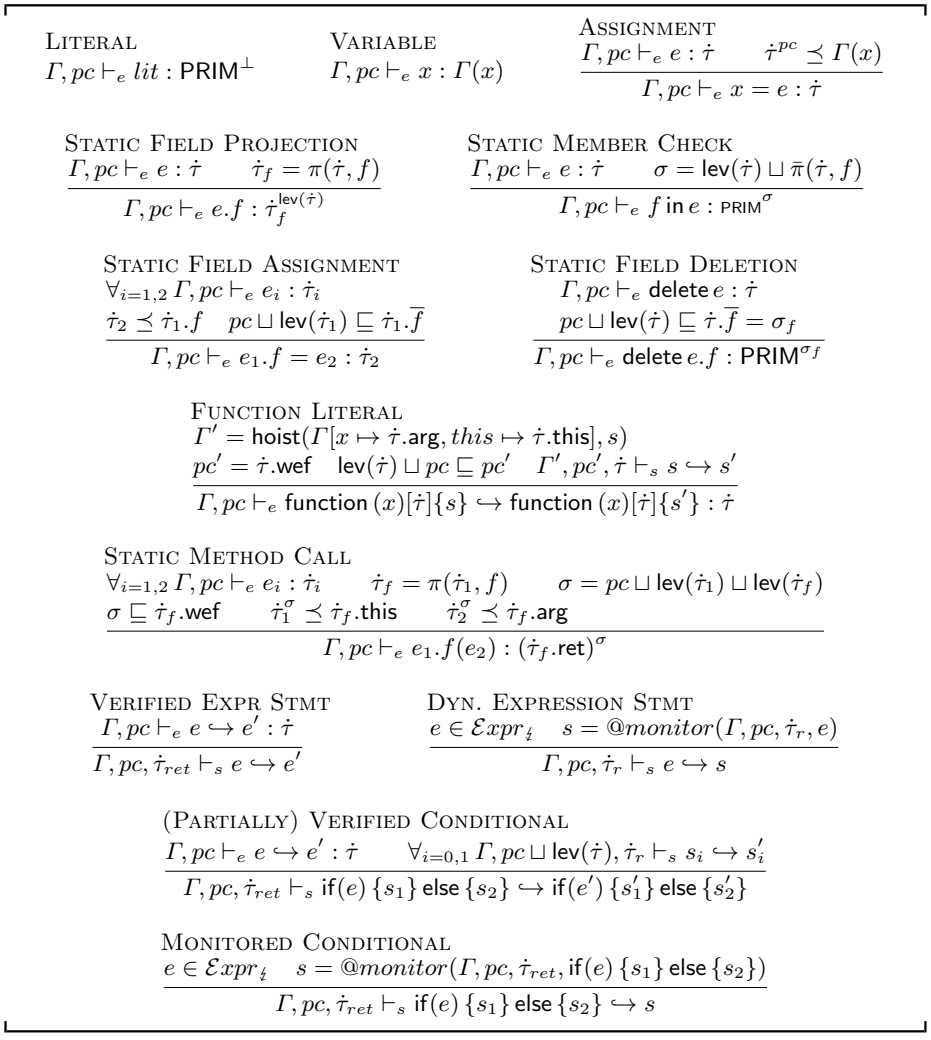


Fig. 2. Static Typing Core JS Expressions

both `o1` and `o2`, which is set to 0 in `o1` and to 1 in `o2`. Depending on the value of a *high* variable `secret`, the prototype of `o0` is either set to `o1` or to `o2`. Finally, the *low* variable `public1` is set to the value of the field `public1` of the prototype of `o0` (because `o0` does not define that field), thereby creating an implicit information flow between `secret` and `public`.

```
o0 = {}; o1 = {}; o2 = {}; o1.public1 = 0; o2.public1 = 1;
if(secret){o0._proto_ = o1} else {o0._proto_ = o2}; public = o0.public1
```

Letting Γ be the typing environment of Table 3, it follows that $\pi(\Gamma(\texttt{o0}), \texttt{public1}) = \mathsf{PRIM}^{H}$ because $\Gamma(\texttt{o0}).\overline{\texttt{\_proto\_}} = H$. Hence, the assignment `public = o0.public1` is not typable as the type of `o0.public1`, PRIM^{H}, is not lower than or equal to PRIM^{L}.

Static Member Check Since the domain of an object can change at execution time and since programs can check if a given field is defined using the keyword in, the mere existence of a field may disclose secret information. The existence security levels declared in object security types serve to control this type of information flows. However, analogously to field projections, this rule needs to take into account the whole prototype chain of the inspected object, because the field whose existence is being checked may be defined anywhere in that prototype chain. To this end, we make use of the static function $\bar{\pi}(\dot{\tau}, f)$ that computes the *lub* between the existence levels of f and *_proto_* in the prototype chain of objects of type $\dot{\tau}$.

Example 4 (Leaks via Membership Checks). The program below creates an object with two fields `secret1` and `secret2`. Then, depending on the value of a *high* variable `secret`, it deletes either `secret1` or `secret2` from the domain of `o`. Finally, the *low* variable `public` is assigned to `true` if `secret1` is defined in the prototype chain of `o` or to `false` if it is not, thereby creating an implicit flow between `secret` and `public`.

```
o = {}; o.secret1 = 0; o.secret2 = 0;
if (secret) { delete o.secret1 } else { delete o.secret2 }; public = secret1 in o
```

Letting Γ be the typing environment of Table 3, it follows that $\bar{\pi}(\Gamma(\texttt{o}), \texttt{secret1}) = \textsc{prim}^H$ because $\Gamma(\texttt{o}).\overline{\texttt{secret1}} = H$. Hence, the last assignment is not typable as the type of the expression `secret1 in o`, $\textsc{prim}^H$, is not lower than or equal to $\textsc{prim}^L$.

Static field assignment The first constraint of the rule checks if the type of the assigned expression is a subtype of the assigned field type, thus preventing the assignment of a secret value to a public field. The second constraint checks if the context level is lower than or equal to the existence level of the assigned field, thereby preventing the creation of a visible field depending on secret information.

Field Deletion The rule checks if the context level is lower than or equal to the field's existence level, thereby preventing visible fields from being deleted in invisible contexts.

Functional literal This rule checks if the context level is lower than or equal to the writing effect of the type of the function literal, thereby preventing the evaluation of function literals that update or create *public* resources inside secret contexts. Then, the type system types the body of the function literal using the typing environment obtained by extending the current one with the type of the the formal argument, the type of the keyword this, and the types of the variables declared in the body of the function literal. To this end, we make use of a syntactic function hoist that extends the typing environment given as its first argument with the mappings from the variables declared in the statement given as its second argument to their respective security types. Note that this rule may re-write the the body of the function literal in order to enable the dynamic analysis.

Method call This rule first verifies if the context level is lower than or equal to the writing effect of the method to call, thereby preventing the calling of a method that creates or updates *public* resources depending on *secret* values.

Then, the rule checks if the type of the object on which the method is called and the type of the function argument match the type of the keyword this and the type of the formal parameter. The method call is finally typed with the return type of the method type upgraded with the context level.

DYN. EXPRESSION STATEMENT This rule wraps every expression that contains a dynamic field operation inside a boundary statement. Recall that $\mathcal{E}xpr_{\lightning}$ denotes the set of Core JS dynamic field operations.

CONDITIONAL If the conditional guard contains a dynamic field operation, the whole conditional is wrapped inside a boundary statement. In the opposite case, the type system types both branches, upgrading the context level with the external level of the security type of the conditional guard.

Example 5 (Hybrid versus Static Typing of the Running Example). Consider the program from Example 1 and the typing environment of Table 3. When typing the assignment `public = o[g()]`, which contains a dynamic field operation, the type system applies the DYN. EXPRESSION STATEMENT rule and wraps the whole assignment inside a boundary statement. All the other statements, which do not contain dynamic field operations, are fully statically verified and, therefore, left unchanged. Hence, the resulting program is given by:

```
o = {}; o.secret1 = secret_input(); o.public1 = public_input();
o.public2 = public_input(); @monitor(@type_env, @pc, @ret, public = o[g()])
```

If, instead, the type system tried to statically type this assignment, it would need to check that the type of `o[g()]` was less than or equal to the type of `public`, PRIM^L. Since we do not know the value to which the call to `g` evaluates, the type system would need to use the *lub* between the types of all the fields declared in the type of o. Consequently, as one of those fields has type PRIM^H, the assignment would not be typable.

4 Dynamic Typing Secure Information Flow in Core JS

The goal of a boundary statement is to enable and disable the information flow analysis at runtime. In this section, we define the semantics of the boundary operator by extending the semantics of Core JS with optional tracking of security types and verification of security constraints.

Monitored Semantics A configuration of the monitored semantics has the form $(\!|\Psi, \Omega|\!)$ where Ψ is a Core JS configuration and Ω is a possibly empty set of monitor configurations. A monitor configuration ω is associated to a specific function call and has the form $\omega = \langle \Gamma, \dot{\tau}_r, l, o, \rho \rangle$ where: **(1)** Γ is a typing environment, **(2)** $\dot{\tau}_r$ is the type of the function that is executing, **(3)** l is the identifier of the environment record associated to the function call that is being monitored, **(4)** o is a *control context*, which is a list containing the levels of the expressions on which the monitored statement branched in order to reach the current context, and **(5)** ρ is an *expression context*, which is a list consisting of the security types of the values of the current evaluation context. The rules of the monitored

$$\textsc{Monitor sync}\quad \frac{\Psi \xrightarrow{\alpha_l} \Psi' \quad \omega \xrightarrow{\alpha_l} \omega'}{(\!|\Psi, \Omega \cup \{\omega\}|\!) \to (\!|\Psi', \Omega \cup \{\omega'\}|\!)} \qquad \textsc{Unmonitored step}\quad \frac{\Psi \xrightarrow{\alpha_l} \Psi' \quad \forall_{\omega \in \Omega}\, \mathsf{er}(\omega) \neq l}{(\!|\Psi, \Omega|\!) \to (\!|\Psi', \Omega|\!)}$$

$$\textsc{Monitor configuration +}\quad \frac{l = \mathsf{head}(L) \qquad \forall_{\omega \in \Omega}\, \mathsf{er}(\omega) \neq l \qquad \omega' = \langle \Gamma, \dot{\tau}_r, l, pc :: [\,], [\,] \rangle}{(\!|\langle H, L, E[@monitor(\Gamma, \dot{\tau}_r, pc, s)] \rangle, \Omega|\!) \to (\!|\langle H, L, E[@monitor(s)] \rangle, \Omega \cup \{\omega'\}|\!)}$$

$$\textsc{Monitor configuration - 1}\quad \frac{\Psi = \langle H, L, E[@monitor(\underline{v})] \rangle \quad \mathsf{head}(L) = \mathsf{er}(\omega) \quad \Psi' = \langle H, L, E[\underline{v}] \rangle}{(\!|\Psi, \Omega \cup \{\omega\}|\!) \to (\!|\Psi', \Omega|\!)}$$

$$\textsc{Monitor configuration - 2}\quad \frac{\Psi = \langle H, L, E[@monitor(\mathsf{return}\ \underline{v})] \rangle \quad \mathsf{head}(L) = \mathsf{er}(\omega) \quad \Psi' = \langle H, L, E[\mathsf{return}\ \underline{v}] \rangle}{(\!|\Psi, \Omega \cup \{\omega\}|\!) \to (\!|\Psi', \Omega|\!)}$$

Fig. 3. Monitored Semantics Rules

semantics are given in Figure 3 and described below. We use $\mathsf{er}(\omega)$ to denote the location of the environment record associated with ω.

Rule Monitor sync corresponds to a monitored step. The transition of the monitor is synchronised with the transition of Core JS semantics through an *internal event* α_l, where l identifies the running function that performed a computation step.

Rule Unmonitored step models the case where there is no matching monitor configuration for the current computation step. In this case, Core JS semantics performs an unconstrained computation step (that takes place outside a boundary statement).

Rule Monitor configuration + generates a new monitor configuration for verifying the statement inside a boundary statement. In order to account for computation steps inside boundary statements, we extend the syntax of evaluation contexts with a special boundary context: $E = @monitor(E')$.

Rules Monitor configuration - 1 and Monitor configuration - 2 remove a monitor configuration from the current set of monitor configurations when its corresponding statement finishes executing.

Monitoring Rules Monitor transitions are defined in Figure 4. We use $\Gamma, \dot{\tau}_r, l \vdash \langle o, \rho \rangle \xrightarrow{\alpha_l} \langle o', \rho' \rangle$ as shorthand for $\langle \Gamma, \dot{\tau}_r, l, o, \rho \rangle \xrightarrow{\alpha_l} \langle \Gamma, \dot{\tau}_r, l, o', \rho' \rangle$. The constraints enforced by the monitor are the same as the constraints enforced by the type system of Section 3. However, in contrast to the type system, the monitor can precisely type dynamic expressions, since it has access to field names computed at runtime.

Example 6 (Monitoring a Dynamic Field Look-up). In the following, we present the sequence of monitor configurations generated when executing the statement: `@monitor(@type_env, @pc, @ret, public = o[g()])` (check the running example).

$$\langle \bot, [\,] \rangle \xrightarrow{\mathsf{var}_l(\mathtt{o})} \langle L, \dot{\tau}_o \rangle \xrightarrow{\mathsf{var}_l(\mathtt{g})} \langle L, \langle \dot{\tau}_g .\_ \xrightarrow{H} \mathsf{PRIM}^L \rangle^L :: \dot{\tau}_o \rangle \xrightarrow{\mathsf{f\text{-}call}_l} \langle L, \mathsf{PRIM}^L :: \dot{\tau}_o \rangle$$

If `g()` returns `public1`:
$$\xrightarrow{\mathsf{f\text{-}proj}_l(\mathtt{public1})} \langle L, \mathsf{PRIM}^L \rangle \xrightarrow{\mathsf{v\text{-}ass}_l(\mathtt{public})} \langle L, \mathsf{PRIM}^L \rangle$$

If `g()` returns `private1`:
$$\xrightarrow{\mathsf{f\text{-}proj}_l(\mathtt{private1})} \langle L, \mathsf{PRIM}^H \rangle \not\xrightarrow{\mathsf{v\text{-}ass}_l(\mathtt{public})}$$

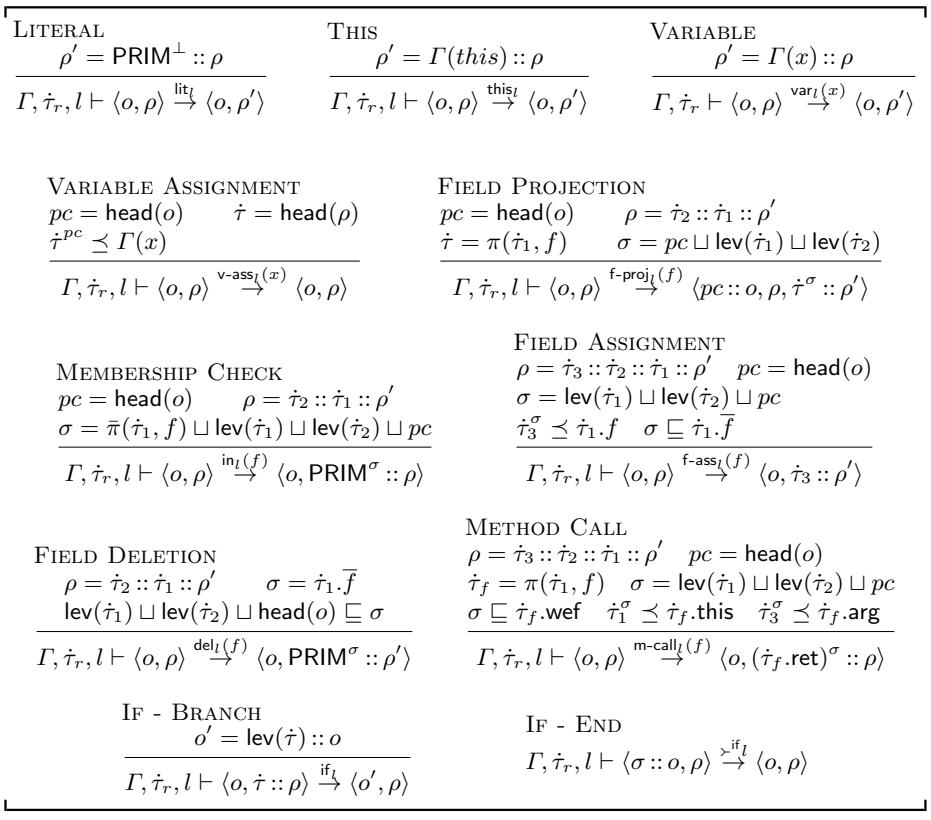


Fig. 4. Dynamic Typing Core JS Expressions and Statements

We consider two different cases: the case in which `g()` evaluates to `public1` and the case in which it evaluates to `secret1`. While in the first case, the execution is allowed to go through, in the second one it gets stuck, because the program tries to assign a secret value to a public variable.

Let us now briefly explain the rules that better illustrate our choices when designing the monitor. Since, by default, all literal values are public, when a *literal* value is evaluated, the monitor simply pushes $\mathsf{PRIM}^{\perp}$ onto the expression stack. In contrast, when a *variable* is evaluated, the monitor has to read its type from the typing environment and push it onto the expression stack. When a *field projection* is evaluated, the first two types on the expression stack are the types of the expressions that evaluate to the field name and to the inspected object, respectively. Furthermore, the name of the inspected field is available in the internal event that labels the transition. Hence, the monitor simply has to replace the first two types of the expression stack with the type of the inspected field upgraded with the external levels of the types of the current subexpressions. When an *if statement* is evaluated, the type of the conditional guard is on top of the expression stack. Hence, the monitor simply pops that type out of the

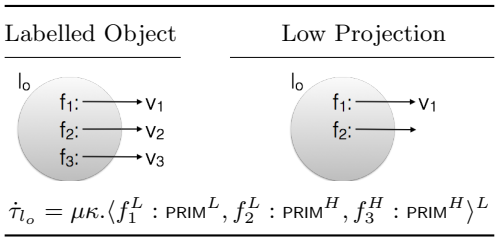


Fig. 5. A Labelled Object and Its Low Projection

expression stack and pushes its external level (upgraded with the current *pc*) onto the control stack. Complementarily, when the execution leaves the branch of a conditional, the monitor just pops out the top of the control stack.

Implementation. Instead of wrapping statements containing dynamic field operations within boundary statements, which are not part of the JavaScript language, the prototype of the hybrid type system [15] in-lines the monitoring logic in the statement itself [16]. This approach has the advantage of being immediately deployable. The prototype implementation was used to secure simple Web application accessible online [15].

5 Security Guarantees

This section describes the security guarantees offered by the proposed analysis. To formally define the absence of information leaks, we rely on an intuitive notion of *low-projection* [14] that establishes the part of a heap that an attacker at a given security level can see. Informally, given a heap H, an attacker at level σ can observe:

1. the existence of a field f in the domain of an object whose type has external level $\leq \sigma$ and associates f with an existence level $\leq \sigma$ and
2. the value of a field f in the domain of an object whose type has external level $\leq \sigma$ and associates f with a security type with external level $\leq \sigma$.

Figure 5 presents a labelled object together with its low-projection at level L. The object in the figure has three fields: f_1, f_2, and f_3. An attacker at level L can observe both the existence and the value of f_1 since it has *low* existence level and is associated with a visible value and the existence but not the value of f_2, since it has *low* existence level but is associated with an invisible value. The attacker can neither observe the value nor the existence of f_3 because it has *high* existence level and is associated with an invisible value. Two heaps H_0 and H_1 are said to be *low-equal* at level σ, written $H_0 \sim_\sigma H_1$, if they coincide in their respective low-projections. Theorem 1 states that the monitored successfully-terminating execution of a program generated by the static type system on two low-equal heaps always yields two low-equal heaps. A sketch of the proof of Theorem 1 is given in the long-version of the paper available online at [15].

Theorem 1 (Noninterference). *For any typing environment Γ, levels σ and pc, security type $\dot{\tau}$, statement, s, and two heaps H_0 and H_1, such that $\Gamma, pc, \dot{\tau} \vdash_s s \hookrightarrow s'$, $H_0 \sim_\sigma H_1$, and $(\!|\langle H_i, [\,], s'\rangle, \{\}|\!) \rightarrow^* (\!|\langle H_i', [\,], v_i\rangle, \{\}|\!)$ for $i = 0, 1$, it holds that $H_0' \sim_\sigma H_1'$.*

6 Related Work

There is a wide variety of mechanisms for enforcing and verifying secure information flow, ranging from purely static type systems [18, 14] to different flavours of dynamic analysis [13, 2]. The main mechanisms for securing information flow in JavaScript [1, 8, 6] are mostly-dynamic due to the dynamicity of the language.

There is a long line of research on safety types for JavaScript which dates back to the seminal work of Thieman [17]. Since then, the TypeScript programming language [11] was proposed as a flexible language that adds optional types to JavaScript with the goal of harnessing the flexibility of real JavaScript, while at the same time providing some of the advantages otherwise reserved for statically typed languages, such as informative compiling errors. Recently, Rastogi et al. [12] designed and implemented a new gradual type system for safely compiling TypeScript to JavaScript. The soundness of the proposed approach is guaranteed by combining strict static checks with residual runtime checks. We believe that our work can serve as a basis for extending TypeScript types with security labels in order to verify secure information flow in TypeScript web applications.

Gradual type systems for secure information flow have been proposed for a pure lambda calculus [3] and for a core ML-like language with references [4]. The goal of these two works is significantly different from ours, as their main intent is to cater for the use of *polymorphic* security labels. For instance, the type language proposed in [4] includes a special annotation "?" representing an unknown security level at static time. Expressions that use variables whose types contain the unknown level annotation, "?", cannot be precisely typed at static time. The programmer can introduce runtime casts in points where values of a pre-determined security type are expected. Then the dynamic analysis checks whether or not a cast can be *securely* performed during execution. However, in order to verify such casts at runtime, these analyses must track security labels during the execution of both dynamically verified and statically verified program regions. In contrast, our analysis only needs to dynamically verify the execution of program regions which were not statically verified.

7 Conclusions

We propose a sound hybrid typing analysis for enforcing secure information flow in a core of JavaScript that includes dynamic field operations. Furthermore, our analysis can be easily extended to handle other dynamic constructs of the language such as eval or unknown code, which only need to be wrapped inside the proposed boundary statement. Finally, we have implemented our analysis and used it to verify a web application described available online [15].

This work follows a well-established trend on combining static and dynamic analysis to devise more permissive and efficient hybrid mechanisms [13]. Our approach can be applied to other scenarios, such as the verification of isolation properties [9], where it could be used to derive mostly-static lightweight enforcement mechanisms from prior purely static specifications.

Acknowledgments We acknowledge funding from the EPSRC grant reference EP/K032089/1 (Fragoso Santos) and the ANR project AJACS ANR-14-CE28-0008 (Jensen, Rezk, and Schmitt). No new data was collected in the course of this research.

References

1. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information flow control in WebKit's JavaScript bytecode. In *POST*, 2014.
2. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *SP*, 2010.
3. T. Disney and C. Flanagan. Gradual information flow typing. In *STOP*, 2011.
4. L. Fennell and P. Thiemann. Gradual security typing with references. In *CSF*, 2013.
5. C. Flanagan. Hybrid type checking. In *POPL*, 2006.
6. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a web browser with flexible and precise information flow control. In *CCS*, 2012.
7. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*, 2014.
8. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF*, 2012.
9. S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *CSF*, 2009.
10. J. Matthews and R. B. Findler. Operational semantics for multi-language programs. *ACM TOPLAS*, 2009.
11. Microsoft. TypeScript language specification. Technical report, Microsoft, 2014.
12. A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *POPL*, 2015.
13. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF*, 2010.
14. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
15. J. Fragoso Santos. Online materials - hybrid type system. http://www.doc.ic.ac.uk/~jfaustin, 2015.
16. J. Fragoso Santos and T. Rezk. An information flow monitor-inlining compiler for securing a core of JavaScript. In *IFIP SEC*, 2014.
17. P. Thiemann. Towards a type system for analysing JavaScript programs. In *ESOP*, 2005.
18. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 1996.
19. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 1994.

Modular Monitor Extensions for Information Flow Security in JavaScript

José Fragoso Santos, Tamara Rezk, Ana Almeida Matos

Modular Monitor Extensions for Information Flow Security in JavaScript

José Fragoso Santos[1], Tamara Rezk[2], and Ana Almeida Matos[3]

1 **Imperial College London**
`jose.fragoso.santos@imperial.ac.uk`
2 **Inria**
`tamara.rezk@inria.fr`
3 **University of Lisbon, SQIG-Instituto de Telecomunicações**
`ana.matos@ist.utl.pt`

Abstract

Client-side JavaScript programs often interact with the web page into which they are included, as well as with the browser itself, through APIs such as the DOM API, the XMLHttpRequest API, and the W3C Geolocation API. Precise reasoning about JavaScript security must therefore take API invocation into account. However, the continuous emergence of new APIs, and the heterogeneity of their forms and features, renders API behavior a moving target that is particularly hard to capture. To tackle this problem, we propose a methodology for modularly extending sound JavaScript information flow monitors with a generic API. Hence, to verify whether an extended monitor complies with the proposed noninterference property requires only to prove that the API satisfies a predefined set of conditions. In order to illustrate the practicality of our methodology, we show how an information flow monitor-inlining compiler can take into account the invocation of arbitrary APIs, without changing the code or the proofs of the original compiler. We provide an implementation of such a compiler with an extension for handling a fragment of the DOM Core Level 1 API. Furthermore, our implementation supports the addition of monitor extensions for new APIs at runtime.

1 Introduction

Isolation properties guarantee protection for trusted JavaScript code from malicious code. The noninterference property [9] provides the mathematical foundations for reasoning precisely about isolation. In particular, noninterference properties guarantee absence of flows from confidential/untrusted resources to public/trusted ones.

Although JavaScript can be used as a general-purpose programming language, many JavaScript programs are designed to be executed in a browser in the context of a web page. Such programs often interact with the web page in which they are included, as well as with the browser itself, through Application Programming Interfaces (APIs). Some APIs are fully implemented in JavaScript, whereas others are built with a mix of different technologies, which can be exploited to conceal sophisticated security violations. Thus, understanding the behavior of client-side web applications, as well as proving their compliance with a given security policy, requires cross-language reasoning. The size, complexity, and number of commonly used APIs poses an important challenge to any attempt at formally reasoning about the security of JavaScript programs [13]. To tackle this problem, we propose a methodology for extending JavaScript monitored semantics. This methodology allows us to verify whether a monitor complies with the proposed noninterference property in a modular way. Thus, we make it possible to prove that a security monitor is still noninterferent when extending it with a new API, without having to revisit the whole model. Generally, an API can be viewed as a particular set of specifications that a program can follow to make use of the

resources provided by another particular application. For client-side JavaScript programs, this definition of API applies both to: (1) interfaces of services that are provided to the program by the environment in which it executes, namely the web browser (for instance, the DOM, the XMLHttpRequest, and the W3C Geolocation APIs); (2) interfaces of JavaScript libraries that are explicitly included by the programmer (for instance, jQuery, Prototype.js, and Google Maps Image API). In the context of this work, the main difference between these two types of APIs is that in the former case their semantics escapes the JavaScript semantics, whereas in the latter it does not. The methodology proposed here was designed as a generic way of extending security monitors to deal with the first type of APIs. Nevertheless, we can also apply it to the second type whenever we want to execute the library's code in the original JavaScript semantics instead of the monitored semantics.

▶ **Example 1** (Running example: A Queue API)**.** Consider the following API for creating and manipulating priority queues. The API is available to the programmer through the global variable *queueAPI*, and variable *queueObj* is bound to a concrete queue:

queueAPI.queue(): creates a new priority queue;
queueObj.push(*el*, *priority*): adds a new element to the queue;
queueObj.pop(): pops the element with the highest priority.

The method calls from this API cannot be verified by the JavaScript monitor, as we are assuming that the code of the methods is not available to the JavaScript engine. Furthermore, the specification of the queue API may not obey the JavaScript semantics and hence prevention of the security leaks may need different constraints.

In order to extend a JavaScript security monitor to control the behavior of this API, one has to define what we call an *API Register* to set the security constraints associated to the corresponding API method calls on *queueAPI* and *queueObj*. API method calls should be implemented as interception points of the monitor semantics and the API Register should then make the invocation of these methods if the security constraints are satisfied.

The following questions then arise: What constraints must we impose on the new API register in order to preserve the noninterference guarantees of the JavaScript monitor? Is it possible to modularly prove noninterference of the extended monitor without revisiting the whole set of constraints, including those of the JavaScript monitor?

There are two main approaches for implementing a monitored JavaScript semantics: either one modifies a JavaScript engine so that it also implements the security monitor (as in [15]), or one inlines the monitor in the original program (as in [16], [8], and [10]). Both these approaches suffer from the problem of requiring ad-hoc security mechanisms for all targeted APIs. We show how to extend an information flow monitor-inlining compiler so that it also takes into account the invocation of APIs. Our extensible compiler requires each API to be associated with a set of JavaScript methods that we call its *IFlow Signature*, which describes how to handle the information flows triggered by its invocation. We provide a prototype of the compiler, which is available online [20]. A user can easily extend it by loading new IFlow signatures. Using the compiler, we give realistic examples of how to prevent common security violations that arise from the interaction between JavaScript and the DOM API. In a nutshell, the benefit of our approach is that it allows us to separate the proof of security for each API from the proof of security for the core language. This separation is, to the best of our knowledge, new and useful as new APIs are continuously emerging.

The contributions of the paper are: (1) a methodology for extending JavaScript monitors with API monitoring (Section 3.2), (2) the design of an extensible information flow monitor-inlining compiler that follows our methodology (Section 4), (3) an implementation [20] of a

JavaScript information flow monitor-inlining compiler (Section 5) that handles an important subset of the DOM API and is extensible with new APIs by means of IFlow Signatures.

2 Related Work

We refer the reader to a recent survey [7] on web scripts security and to [19] for a complete survey on information flow enforcement mechanisms up to 2003, while focusing here on the most closely related work on dynamic mechanisms for enforcing noninterference.

Flow-sensitive monitors for enforcing noninterference can be divided into *purely dynamic monitors* [3–5] and *hybrid monitors* [12, 22]. While hybrid monitors use static analysis to reason about untaken execution paths, purely dynamic monitors do not rely on any kind of static analysis. There are three main strategies in designing sound purely dynamic information flow monitors. The *no-sensitive-upgrade* (NSU) strategy [3] forbids the update of public resources inside private contexts. The *permissive-upgrade* strategy [4] allows sensitive upgrades, but forbids programs to branch depending on values upgraded in private contexts. Finally, the *multiple facet* strategy [5] makes use of values that appear differently to observers at different security levels. Here, we show how to extend information flow monitors that follow the NSU discipline.

Hedin and Sabelfeld [15] are the first to propose a runtime monitor for enforcing noninterference for JavaScript. The technique that we present for extending security monitors can be applied to this monitor, which is purely dynamic and follows the NSU discipline. In [14], the authors implement their monitor as an extended JavaScript interpreter. Their implementation makes use of the informal concepts of shallow and deep information flow models in order to cater for the invocation of built-in libraries and DOM API methods. However, these concepts are not formalised. In fact, our definition of monitored API can be seen as a formalisation of the notion of deep information flow model for libraries.

Both Chudnov and Naumann [8] and Magazinius et al. [16] propose the inlining of information flow monitors for simple imperative languages. In [10], we present a compiler that inlines a purely dynamic information flow monitor for a realistic subset of JavaScript. In the implementation presented in this paper we extend the inlining compiler of [10] with the DOM API, applying the methodology proposed here.

Taly et al. [21] study API confinement. They provide a static analysis designed to verify whether an API may leak its confidential resources. Unlike us, they only target APIs implemented in JavaScript, whose code is available for either runtime or static analysis.

Russo et al. [18] present an information flow monitor for a WHILE language with primitives for manipulating DOM-like trees and prove it sound. They do not model references. In [2], we present an information flow monitor for a simple language that models a core of the DOM API based on the work of Gardner et al. [11]. In contrast to [18], we can handle references and live collections. Here, we apply the techniques of [2] to develop monitor extensions for a fragment of the DOM Core Level 1 API [17]. Recent work [23] presents an information flow monitor for JavaScript extended with the DOM API that also considers event handling loops. To the best of our knowledge, no prior work proposes a generic methodology to extend JavaScript monitors and inlining compilers with arbitrary web APIs.

3 Modular Extensions for JavaScript Monitors

In this section we show how to extend a noninterferent monitor so that it takes into account the invocation of web APIs, while preserving the noninterference property.

3.1 Noninterferent JavaScript Monitors

JavaScript Memory Model. In JavaScript [1], objects can be seen as partial functions mapping strings to values. The strings in the domain of an object are called its properties. Memories are mappings from references to objects. In the following, we assume that memories include a reference to a special object called the *global object* pointed to by a fixed reference $\#glob$, that binds global variables. In this presentation, objects, properties, memories, references and values, are ranged over by o, p, μ, r and v, respectively.

We use the notation $[p_0 \mapsto v_0, \ldots, p_n \mapsto v_n]$ for the partial function that maps p_i to v_i where $i = 0, \ldots n$, and $f\,[p_0 \mapsto v_0, \ldots, p_n \mapsto v_n]$ for the function that coincides with f everywhere except in p_0, ..., p_n, which are otherwise mapped to v_0, ..., v_n respectively. Furthermore, we denote by $\mathsf{dom}(f)$ the domain of a function f, and by $f|_P$ the restriction of f to P (when $P \subseteq \mathsf{dom}(f)$). Finally, we write $f(r)(p)$ instead of $(f(r))(p)$, the application of the image of r by function f to p.

Sequences are denoted by stacking an arrow as in $\overrightarrow{v}$, and ϵ denotes the empty sequence. The length of $\overrightarrow{v}$ is given by $|\overrightarrow{v}|$ and $\cdot$ denotes concatenation of sequences.

Security Setting. Information flow policies such as noninterference are specified over security labelings that assign security levels, taken from a given security lattice, to the observable resources of a program. In the following, we use a fixed lattice $\mathcal{L}$ of security levels ranged over by σ. We denote by $\leq$ its order relation, by $\sigma_0 \sqcup \sigma_1$ the least upper bound between levels σ_0 and σ_1, and by $\sqcup\vec{\sigma}$ the least upper bound of all levels in the sequence $\vec{\sigma}$. In the examples, we consider two security levels $\{H, L\}$ such that $L < H$, meaning that resources labeled with *high* level H are more confidential than those labeled with *low* level L.

In our setting, a security labeling is as a pair $\langle \Gamma, \Sigma \rangle$, where Γ maps references, followed by properties, to security levels, and Σ maps references to security levels. Then, given an object o pointed to by a reference r, if defined, $\Gamma(r)(p)$ corresponds to the security levels associated with o's property p, and $\Sigma(r)$ with o's domain. The latter, also referred to as o's *structure security level*, controls the observability of the existence of properties [15].

We say that memory μ is well-labeled by $\langle \Gamma, \Sigma \rangle$ if $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Sigma) \subseteq \mathsf{dom}(\mu)$ and for every reference $r \in \mathsf{dom}(\Gamma)$, $\mathsf{dom}(\Gamma(r)) \subseteq \mathsf{dom}(\mu(r))$.

Security Monitor. JavaScript programs are statements, that include expressions, ranged over by s and e, respectively. We model an information flow monitor as a small-step semantics relation $\rightarrow_{\mathsf{IF}}$ between configurations of the form $\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle$ composed of (1) a memory μ (2) a statement s, that is to execute, (3) a sequence of security levels $\overrightarrow{\sigma_{\mathsf{pc}}}$, matching the expressions on which the original program branched to reach the current context, (4) a security labeling $\langle \Gamma, \Sigma \rangle$, and (5) a sequence of security levels $\overrightarrow{\sigma}$ matching the reading effects of the subexpressions of the expression being computed.

The *reading effect* [19] of an expression is defined as the least upper bound on the security levels of the resources on which the value to which it evaluates depends. Additionally, we assume that the reading effect of an expression is always higher than or equal to the level of the context in which it is evaluated, $\sqcup\overrightarrow{\sigma_{\mathsf{pc}}}$.

Low-equality. In order to account for a non-deterministic memory allocator, we rely on a partial injective function which relates observable references that point to the same resource in different executions of the same program [6]. The β relation is extended to relate observable values via the *β-equality*, which is denoted $\sim_\beta$: two objects are β-equal if they have the same domain and all their corresponding properties are β-equal; primitive values and parsed functions are β-equal if syntactically equal; and, two references r_0 and r_1 are β-equal if the latter is the image by β of the former.

Two memories μ_0 and μ_1 are said to be *low-equal* with respect to labelings $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, a security level σ, and a partial injective function β, written $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \Sigma_1$, if μ_0 and μ_1 are well-labeled by $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$ respectively, and for all references $r_0, r_1 \in \mathsf{dom}(\beta)$, such that $r_1 = \beta(r_0)$, the following hold:

1. The observable domains (i.e. set of observable properties) of the objects pointed by r_0, r_1, coincide: $P_\sigma = \{p \in \mathsf{dom}(\mu_0(r_0)) \mid \Gamma_0(r_0)(p) \leq \sigma\} = \{p \in \mathsf{dom}(\mu_1(r_1)) \mid \Gamma_1(r_1)(p) \leq \sigma\}$;
2. The objects pointed by r_0, r_1 coincide in their observable domain: $\mu_0(r_0)|_P \sim_\beta \mu_1(r_1)|_P$;
3. If the structure security level of either object pointed by r_0, r_1 is observable ($\Sigma_0(r_0) \leq \sigma$ or $\Sigma_1(r_1) \leq \sigma$), then their domains and structure security levels coincide: $\mathsf{dom}(\mu_0(r_0)) = \mathsf{dom}(\mu_1(r_1))$ and $\Sigma_0(r_0) = \Sigma_1(r_1)$.

We extend informally the definition of low-equality to sequences of labeled values and to program continuations (the interested reader can find the formal definitions in [20]). Two sequences of labeled values are low-equal with respect to a given security level σ, denoted $\vec{v_0}, \vec{\sigma_0} \approx_{\beta,\sigma} \vec{v_1}, \vec{\sigma_1}$ if for each position of both sequences, either the two values in that position are low-equal, or the levels that are associated with both of them are not observable. Low-equality between program continuations $s_0, \overrightarrow{\sigma_{pc0}}, \vec{\sigma_0} \approx_{\beta,\sigma} s_1, \overrightarrow{\sigma_{pc1}}, \vec{\sigma_1}$ relaxes syntactic equality between programs in order to relate the intermediate states of the execution of the same original program in two low-equal memories, as illustrated by the following example.

▶ **Example 2** (Low-equal program continuations)**.** Consider the program $x = y$, an initial labeling $\langle \Gamma, \Sigma \rangle$ such that $\Gamma(\#glob)(x) = \Gamma(\#glob)(y) = H$, and two memories μ_0 and μ_1 such that $\mu_i = [\#glob \mapsto [x \mapsto \mathit{undefined}, y \mapsto i]]$, for $i \in \{0, 1\}$. The execution of one computation step of this program in μ_0 and μ_1 yields the programs $x = 0$ and $x = 1$. Since the reading effects associated with the values 0 and 1 are both H, the expressions $x = 0$ and $x = 1$ are low-equal. Formally: $x = 0, \langle L \rangle, \langle H \rangle \approx_{\mathsf{id},L} x = 1, \langle L \rangle, \langle H \rangle$ (where id is the identity function).

Finally, two monitor configurations $\langle \mu_0, s_0, \overrightarrow{\sigma_{pc0}}, \Gamma_0, \Sigma_0, \vec{\sigma_0} \rangle$ and $\langle \mu_1, s_1, \overrightarrow{\sigma_{pc1}}, \Gamma_1, \Sigma_1, \vec{\sigma_1} \rangle$ are said to be *low-equal* w.r.t a level σ and function β, written $\langle \mu_0, s_0, \overrightarrow{\sigma_{pc0}}, \Gamma_0, \Sigma_0, \vec{\sigma_0} \rangle \approx_{\beta,\sigma} \langle \mu_1, s_1, \overrightarrow{\sigma_{pc1}}, \Gamma_1, \Sigma_1, \vec{\sigma_1} \rangle$, if $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta,\sigma} \mu_1, \Gamma_1, \sigma_1$ and $s_0, \overrightarrow{\sigma_{pc0}}, \vec{\sigma_0} \approx_{\beta,\sigma} s_1, \overrightarrow{\sigma_{pc1}}, \vec{\sigma_1}$.

Noninterferent Monitor.

In the remaining of the paper, we consider only *noninterferent* JavaScript monitors. As usual, a monitor $\rightarrow_{\mathsf{IF}}$ is noninterferent, written $\mathbf{NI}_{\mathbf{mon}}(\rightarrow_{\mathsf{IF}})$, if its application on two low-equal configurations produces two low-equal configurations.

▶ **Definition 3** (Monitor Noninterference)**.** A monitor $\rightarrow_{\mathsf{IF}}$ is said to be *noninterferent*, written $\mathbf{NI}_{\mathbf{mon}}(\rightarrow_{\mathsf{IF}})$, if for every programs s_0, s_1, memories μ_0, μ_1, and labeling $\langle \Gamma, \Sigma \rangle$, such that μ_0, μ_1 are well-labeled by $\langle \Gamma, \Sigma \rangle$ and, for all security levels σ, there exists β such that $\langle \mu_0, s_0, \epsilon, \Gamma, \Sigma, \epsilon \rangle \approx_{\beta,\sigma} \langle \mu_1, s_1, \epsilon, \Gamma, \Sigma, \epsilon \rangle$, if $\langle \mu_0, s_0, \epsilon, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{\mathsf{IF}} \langle \mu_0', v_0', \epsilon, \Gamma', \Sigma', \sigma' \rangle$ and $\langle \mu_1, s_1, \epsilon, \Gamma, \Sigma, \epsilon \rangle \rightarrow^*_{\mathsf{IF}} \langle \mu_1', v_1', \epsilon, \Gamma', \Sigma', \sigma' \rangle$ then there is an extension β' of β such that $\langle \mu_0', v_0', \epsilon, \Gamma', \Sigma', \sigma' \rangle \approx_{\beta,\sigma} \langle \mu_1', v_1', \epsilon, \Gamma', \Sigma', \sigma' \rangle$.

3.2 API Extensions to JavaScript Monitors

API relation. Even if the execution of certain APIs escapes the JavaScript semantics, the interaction between JavaScript programs and these APIs is mediated via special API objects that exist in the JavaScript memory. In the following, we assume that (1) the state of the API can be fully encoded in a JavaScript memory and (2) the behavior of each API method only depends on its state. An API is thus modeled as a semantic relation $\Downarrow^{\mathsf{JS}}_{\mathsf{API}}$ of the form $\langle \mu, \overrightarrow{v} \rangle \Downarrow^{\mathsf{JS}}_{\mathsf{API}} \langle \mu', v' \rangle$ where μ is the JavaScript memory in which the API is executed, μ'

is the resulting memory, the sequence of values $\overrightarrow{v}$ corresponds to the arguments of the API invocation, and v' is the value to which the API invocation evaluates. Accordingly, a *monitored API relation*, $\Downarrow_{\mathsf{API}}$, has the form

$$\langle \mu, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \ \Downarrow_{\mathsf{API}} \ \langle \mu', \Gamma', \Sigma', v, \sigma \rangle$$

which adds to the original API configuration the initial and final labelings $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$ (respectively), the sequence of security levels $\overrightarrow{\sigma}$ that is associated with the arguments of the API invocation, and their corresponding reading effect σ.

API register. The bridge between API invocations and the corresponding monitored API semantics is performed by a *API register*, denoted by $\mathcal{R}_{\mathsf{API}}$. We define an API register as a function that, given a memory and a sequence of values, returns a monitored API relation.

▶ **Example 4** (Queue API Register)**.** In order for an extended monitor to take into account the methods of the Queue API from Example 1, the API Register must be extended to handle invocations of the Queue API methods. In the following, $\Downarrow_{QU}$, $\Downarrow_{PU}$, and $\Downarrow_{PO}$ are the API relations corresponding to each one of the methods of the Queue API:

$$\mathcal{R}_Q(\mu, \langle r, m, \ldots \rangle) = \begin{cases} \Downarrow_{QU} & \text{if } m = \text{“queue”} \wedge \$q \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{PU} & \text{if } m = \text{“push”} \wedge \$q \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{PO} & \text{if } m = \text{“pop”} \wedge \$q \in \mathsf{dom}(\mu(r)) \end{cases}$$

The idea is to “mark” the Queue API object (the one bound to variable *queueAPI*) as well as the concrete queue objects, with a special property (in this case, $\$q$).

Monitor-extending Constructor. We now define a monitor-extending constructor $\mathcal{E}$ that, given a monitored small-step semantics $\rightarrow_{\mathsf{IF}}$, a partial function $\mathsf{Intercept}$ mapping statements to sequences of values, and an API register $\mathcal{R}_{\mathsf{API}}$, produces a new monitored small-step semantics $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$. The new extended semantics handles the invocation of APIs by applying the API relation that is returned by $\mathcal{R}_{\mathsf{API}}$. API invocation is triggered by *interception points*, statements containing expression redexes (expressions that only have values as subexpressions) and that are in the set $\mathsf{Intercept}$. Then, if the sequence of values to which its subexpressions evaluate is in the domain of the API register $\mathcal{R}_{\mathsf{API}}$, their image by $\mathcal{R}_{\mathsf{API}}$ is the semantic relation that models the API to be executed.

The definition of $\mathcal{E}$, given in Figure 1, makes use of a syntactic function, $\mathsf{SubExpressions}$, defined on JavaScript statements, such that $\mathsf{SubExpressions}[[s]]$ corresponds to the sequence comprising all the subexpressions of s in the order by which they are evaluated. Rules [Non-Intercepted Program Construct] and [Intercepted Program Construct - Standard Execution] model the case in which the new small-step semantics behaves according to the original semantics $\rightarrow_{\mathsf{IF}}$. Rule [Intercepted Program Construct - API Execution] models the case in which an API is executed. The semantics rule retrieves the semantics relation that models the API to execute (using the API register) and then executes the API. After executing the API, the sequence of values of its subexpressions is replaced with the value to which the API call evaluates. Analogously, the sequence of levels of its subexpressions is replaced with the reading effect of the API call.

3.3 Sufficient Conditions for Noninterferent API Extensions

We identify sufficient conditions to be satisfied by API relations in order for the new extended monitored semantics $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$ to be noninterferent, assuming that the original monitor $\rightarrow_{\mathsf{IF}}$ is noninterferent.

$$\textsc{Non-Intercepted Program Construct}\quad \frac{s \notin \mathsf{Intercept} \qquad \langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \rightarrow_{\mathsf{IF}} \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}$$

$$\textsc{Intercepted Program Construct - Standard Execution}\quad \frac{s \in \mathsf{Intercept} \qquad (\mu, \mathsf{SubExpressions}[[s]]) \notin \mathsf{dom}(\mathcal{R}_{\mathsf{API}}) \qquad \langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle \rightarrow_{\mathsf{IF}} \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma_{\mathsf{pc}}}' \rangle}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', s', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \rangle}$$

$$\textsc{Intercepted Program Construct - API Execution}\quad \frac{\begin{array}{c} s \in \mathsf{Intercept} \qquad (\mu, \mathsf{SubExpressions}[[s]]) \in \mathsf{dom}(\mathcal{R}_{\mathsf{API}}) \\ |\mathsf{SubExpressions}[[s]]| = n+1 \qquad \overrightarrow{\sigma} = \overrightarrow{\sigma}' \cdot \langle \sigma_0, \ldots, \sigma_n \rangle \qquad \Downarrow_{\mathsf{API}} = \mathcal{R}_{\mathsf{API}}(\mu, \mathsf{SubExpressions}[[s]]) \\ \langle \mu, \Gamma, \Sigma, \mathsf{SubExpressions}[[s]], \langle \sigma_0, \ldots, \sigma_n \rangle \rangle\ \Downarrow_{\mathsf{API}}\ \langle \mu', \Gamma', \Sigma', v', \sigma' \rangle \end{array}}{\langle \mu, s, \overrightarrow{\sigma_{\mathsf{pc}}}, \Gamma, \Sigma, \overrightarrow{\sigma} \rangle\ \mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})\ \langle \mu', v', \overrightarrow{\sigma_{\mathsf{pc}}}', \Gamma', \Sigma', \overrightarrow{\sigma}' \cdot \sigma' \rangle}$$

Figure 1 Definition of the monitor-extending constructor $\mathcal{E}$.

The first condition requires that the API relation is *confined*, as formalized in Definition 5. An API relation is *confined* if it only creates/updates resources whose levels are higher than or equal to the least upper bound on the levels of its arguments. This constraint is needed because the choice of which API to execute may depend on all of its arguments.

▶ **Definition 5** (Confined API Relation/Register)**.** An API relation $\Downarrow_{\mathsf{API}}$ is confined if, for every memory μ well-labeled by a labeling $\langle \Gamma, \Sigma \rangle$, every sequence of argument values $\overrightarrow{v}$ and corresponding sequence of security levels $\overrightarrow{\sigma}$, if $\langle \mu, \langle \Gamma, \Sigma \rangle, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \Downarrow_{\mathsf{API}} \langle \mu', \langle \Gamma', \Sigma' \rangle, v', \sigma' \rangle$ for some memory μ', labeling $\langle \Gamma', \Sigma' \rangle$, value v', and level σ'; then, for all security levels $\hat{\sigma}$:

$$\sqcup \overrightarrow{\sigma} \not\leq \hat{\sigma} \quad \Rightarrow \quad \mu, \Gamma, \Sigma \approx_{\mathsf{id}, \hat{\sigma}} \mu', \Gamma', \Sigma' \ \wedge\ \sigma' \not\leq \hat{\sigma}$$

Furthermore, we say that the API Register function $\mathcal{R}_{\mathsf{API}}$ is confined, written $\mathbf{Conf}(\mathcal{R}_{\mathsf{API}})$, if all the API relations in its range are confined, and if for every memories μ and μ', labelings $\langle \Gamma, \Sigma \rangle$ and $\langle \Gamma', \Sigma' \rangle$, sequence of values $\overrightarrow{v}$, security level σ, and function β, such that $\mu, \Gamma, \Sigma \approx_{\beta, \sigma} \mu', \Gamma', \Sigma'$, then $\mathcal{R}_{\mathsf{API}}(\mu, \overrightarrow{v}) = \mathcal{R}_{\mathsf{API}}(\mu', \beta(\overrightarrow{v}))$.

The second condition requires that the API relation is *noninterferent*, as formalized in Definition 6. In order to relate the outputs of the API Register in two low-equal memories, we extend the notion of low-equality to API registers. Informally, two API registers are said to be low-equal if, whenever they are given as input two low-equal memories and two low-equal sequences of values, they output the same noninterferent API relation. Then, an API relation is *noninterferent* if whenever it is executed on two low-equal memories, it produces two low-equal memories and two low-equal values.

▶ **Definition 6** (Noninterferent API Relation/Register)**.** An API relation $\Downarrow_{\mathsf{API}}$ is said to be noninterferent, written $\mathbf{NI}(\Downarrow_{\mathsf{API}})$, if for every two memories μ_0 and μ_1 respectively well-labeled by $\langle \Gamma_0, \Sigma_0 \rangle$ and $\langle \Gamma_1, \Sigma_1 \rangle$, any two sequences of values $\overrightarrow{v_0}$ and $\overrightarrow{v_1}$, respectively labeled by two sequences of security levels $\overrightarrow{\sigma_0}$ and $\overrightarrow{\sigma_1}$, and any security level σ for which there exists a function β such that $\overrightarrow{v_0}, \overrightarrow{\sigma_0} \approx_{\beta, \sigma} \overrightarrow{v_1}, \overrightarrow{\sigma_1}$ and $\mu_0, \Gamma_0, \Sigma_0 \approx_{\beta, \sigma} \mu_1, \Gamma_1, \Sigma_1$, if:

$$\langle \mu_0, \Gamma_0, \Sigma_0, \overrightarrow{v_0}, \overrightarrow{\sigma_0} \rangle \Downarrow_{\mathsf{API}} \langle \mu_0', \Gamma_0', \Sigma_0', v_0', \sigma_0' \rangle \ \wedge\ \langle \mu_1, \Gamma_1, \Sigma_1, \overrightarrow{v_1}, \overrightarrow{\sigma_1} \rangle \Downarrow_{\mathsf{API}} \langle \mu_1', \Gamma_1', \Sigma_1', v_1', \sigma_1' \rangle$$

then there is an extension β' of β s.t. $\mu_0', \Gamma_0', \Sigma_0' \approx_{\beta', \sigma} \mu_1', \Gamma_1', \Sigma_1'$ and $\langle v_0' \rangle, \langle \sigma_0' \rangle \approx_{\beta', \sigma} \langle v_1' \rangle, \langle \sigma_1' \rangle$.

Furthermore, we say that the API Register function $\mathcal{R}_{\mathsf{API}}$ is noninterferent, written $\mathbf{NI}(\mathcal{R}_{\mathsf{API}})$, if all the API relations in its range are noninterferent.

▶ **Example 7** (Noninterferent JavaScript program using the Queue API)**.** Assume that the APIs given in Example 1 are noninterferent and consider the following program that starts by computing two objects `o0` and `o1`:

```
1  q = queueAPI.createQueue();
2  if (h) { q.push(o1, 1); }
3  q.push(o0, 0); l = q.pop();
```

If this program starts with memories μ_i ($i \in \{0,1\}$) using labeling $\langle \Gamma, \Sigma \rangle$ and assuming that in both executions the invocations of all the external APIs go through (i.e. the execution is never blocked), then it must terminate with memories μ_i' labeled by Γ', Σ:

$$\mu_i = \left[\begin{array}{c}(\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1,\\ (\#glob, h) \mapsto i\end{array}\right] \qquad \Gamma = \left[\begin{array}{c}(\#glob, h) \mapsto H, (\#glob, l) \mapsto L,\\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L\end{array}\right]$$

$$\mu_i' = \left[\begin{array}{c}(\#glob, o0) \mapsto r_0, (\#glob, o1) \mapsto r_1,\\ (\#glob, h) \mapsto i, (\#glob, l) \mapsto r_i, (\#glob, q) \mapsto r_q\end{array}\right] \quad \Gamma' = \left[\begin{array}{c}(\#glob, h) \mapsto H, (\#glob, l) \mapsto H,\\ (\#glob, o0) \mapsto L, (\#glob, o1) \mapsto L\end{array}\right]$$

Since initial memories are low-equal, $\mu_0, \Gamma, \Sigma \approx_{\mathsf{id}, L} \mu_1, \Gamma, \Sigma$, we use the hypothesis that all three API relations are noninterferent to conclude that the memories yielded by the invocation of the API relations in lines `1`, `2`, and `3` are also low-equal. Furthermore, in the execution that maps `h` to 1, the value of `l` clearly depends on the value of `h`, from which we conclude that it is also the case in the execution that maps `h` to 0.

Our main result states that if the API relation is confined and noninterferent, then the extension of the noninterferent JavaScript monitor with the API monitor is noninterferent.

▶ **Theorem 8** (Security)**.** *For every monitored semantics* $\rightarrow_{\mathsf{IF}}$*, API register* $\mathcal{R}_{\mathsf{API}}$ *and set of interception points* Intercept*:*

$$\mathbf{NI_{mon}}(\rightarrow_{\mathsf{IF}}) \wedge \mathbf{NI}(\mathcal{R}_{\mathsf{API}}) \wedge \mathbf{Conf}(\mathcal{R}_{\mathsf{API}}) \Rightarrow \mathbf{NI_{mon}}(\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}}))$$

4 A Meta-Compiler for Securing Web APIs

We now propose a way of extending an information flow monitor inlining compiler to take into account the execution of arbitrary APIs.

Input compilers. We assume available two inlining compilers specified by compilation functions $\mathcal{C}_{\mathsf{e}}$ and $\mathcal{C}_{\mathsf{s}}$ for compiling JavaScript expressions and statements, respectively. Function $\mathcal{C}_{\mathsf{s}}$ makes use of function $\mathcal{C}_{\mathsf{e}}$. The compilers $\mathcal{C}_{\mathsf{e}}/\mathcal{C}_{\mathsf{s}}$ map every expression e/statement s to a pair $\langle s', i \rangle$, where:

1. statement s' simulates the execution of e/s in the monitored semantics;
2. index i is such that, after the execution of s', (1) the compiler variable $\$\hat{v}_i$ stores the value to which e/s evaluates in the original semantics and (2) the compiler variable $\$\hat{l}_i$ stores its corresponding reading effect.

We assume that the inlining compiler works by pairing up each variable/property with a new one, called its *shadow* variable/property [8, 16], that holds its corresponding security level. Since the compiled program has to handle security levels, we include them in the set of program values, which means adding them to the syntax of the language as such, as well as adding two new binary operators corresponding to $\leq$ (the order relation) and $\sqcup$ (the least upper bound). Besides adding to every object o an additional shadow property $\$l_p$ for every property p in its domain, the inlined monitoring code is also assumed to extend o with a special property $\$struct$ that stores its structure security level.

▶ **Example 9** (Instrumented Labeling)**.** Given an object $o = [p \mapsto v_0, q \mapsto v_1]$ pointed to by r_o and a labeling $\langle \Gamma, \Sigma \rangle$, such that $\Gamma(r_o) = [p \mapsto H, q \mapsto L]$ and $\Sigma(r_o) = L$, the instrumented counterpart of o labeled by $\langle \Gamma, \Sigma \rangle$ is $\hat{o} = [p \mapsto v_0, q \mapsto v_1, \$l_p \mapsto H, \$l_q \mapsto L, \$struct \mapsto L]$.

$$\textsc{Intercepted Expression}$$

$$\frac{\begin{array}{c}\mathsf{SubExpressions}[[e]] = \langle e_0, \ldots, e_n\rangle \qquad \mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e_0\rangle = \langle s_0, i_0\rangle \ \cdots \ \mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e_n\rangle = \langle s_n, i_n\rangle \\ \hat{e} = \mathsf{Replace}[[e, \$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}]] \qquad \langle\mathcal{C}_{\mathsf{e}}\rangle\hat{e} = \langle\hat{s}, i\rangle \\ s_{\mathsf{api}} = \left\{\begin{array}{l} s_0 \ \ldots \ s_n \\ \$if_{\mathsf{sig}} = \$apiRegister(\$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}); \\ \mathsf{if}(\$if_{\mathsf{sig}})\{ \\ \quad \$if_{\mathsf{sig}}.check(\$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}, \$\hat{l}_{i_0}, \ldots, \$\hat{l}_{i_n}); \\ \quad \$\hat{v}_i = \hat{e}; \\ \quad \$\hat{l}_i = \$if_{\mathsf{sig}}.label(\$\hat{v}_i, \$\hat{v}_{i_0}, \ldots, \$\hat{v}_{i_n}, \$\hat{l}_{i_0}, \ldots, \$\hat{l}_{i_n}); \\ \} \ \mathsf{else} \ \{\hat{s}\} \end{array}\right. \qquad s' = \left\{\begin{array}{ll} s_{\mathsf{api}} & \text{if } \hat{e} \in \mathsf{Intercept} \\ \hat{s} & \text{otherwise} \end{array}\right. \end{array}}{\mathcal{C}_{\mathsf{API}}\langle\mathcal{C}_{\mathsf{e}}\rangle\langle e\rangle = \langle s', i\rangle}$$

Figure 2 Extended Compiler $\mathcal{C}_{\mathsf{API}}$.

4.1 IFlow Signatures

We propose *IFlow signatures* to simulate monitored executions of API relations. IFlow signatures are composed of three methods – *domain*, *check*, and *label*. Method *domain* checks whether or not to apply the API, *check* checks if the constraints associated with the API are verified, and *label* updates the instrumented labeling and outputs the reading effect associated with a call to the API. Functions *check* and *label* must be specified separately because *check* has to be executed before calling the API (in order to prevent its execution when it can potentially trigger a security violation), whereas *label* must be executed after calling the API (so that it can label the memory resulting from its execution). Formally, we define an *IFlow Signature* as a triple $\langle\#check, \#label, \#domain\rangle$, where: $\#check$ is the reference of the *check* function object, $\#label$ is the reference of the *label* function object, and $\#domain$ is the reference of the *domain* function object.

Runtime API Register. We assume the existence of a runtime function called the *runtime API register*, that simulates the API Register, which we denote by $\$apiRegister$. The function $\$apiRegister$ makes use of the *domain* method of each API in its range to decide whether there is an API relation associated with its inputs, in which case it outputs an object containing the corresponding IFlow Signature, otherwise it returns *null*.

Meta-compiler. Figure 2 presents a new meta-compiler, $\mathcal{C}_{\mathsf{API}}$, that receives as input an inlining compiler for JavaScript expressions, $\mathcal{C}_{\mathsf{e}}$, and outputs a new inlining compiler that can handle the invocation of the APIs whose signatures are in the range of the API register simulated by $\$apiRegister$. Since statement redexes are not intercepted, the compilation function $\mathcal{C}_{\mathsf{s}}$ is left unchanged except that it uses the new compilation function for expressions for compiling the subexpressions of the given statement. The specification of the meta-compiler makes use of a syntactic function Replace that receives as input an expression and a sequence of variables and outputs the result of substituting each one of its subexpressions by the corresponding sequence variable. Intercept is the set of all statements that contain an expression redex whose execution is to be intercepted by the monitored semantics. Each expression that can be an interception point of the semantics is compiled by the compiler generated by the meta-compiler to a statement, which: (1) executes the statements corresponding to the compilation of its subexpressions, (2) tests if the sequence of values corresponding to the subexpressions of the expression to compile is associated with an IFlow signature, (3) if the test is true, it executes the *check* method of the corresponding IFlow signature, an expression equivalent to the original expression, and the *label* method of the corresponding IFlow signature. If the test is false, it executes the compilation of an

expression equivalent to the original one by the original inlining compiler. For simplicity, we do not take into account expressions that manipulate control flow, meaning that the evaluation of a given expression implies the evaluation of all its subexpressions. Therefore, we do not consider the JavaScript conditional expression. This limitation can be surpassed by re-writing all conditional expressions as IF statements before applying the compiler.

4.2 Correctness

We say that an inlining compiler is *correct* with respect to a given monitored semantics $\rightarrow_{\mathsf{IF}}$ if, provided that a program and its compiled counterpart are evaluated in "similar" memories, the evaluation of the original one in the monitored semantics terminates *if and only if* the evaluation of its compilation also terminates in the original semantics, in which case the final memories as well as the computed values are again "similar". Here we use a notion of *similarity* between labeled memories in the monitored semantics and instrumented memories in the original semantics, denoted by $\mathcal{S}_\beta$. This relation requires that for every object in the labeled memory, the corresponding labeling coincides with the instrumented labeling and that the property values of the original object be similar to those of its instrumented counterpart. (The formal definition of $\mathcal{S}_\beta$ can be found in the companion report [20].)

The correctness of the compiler generated by the meta-compiler depends on the correctness of the compiler given as input and the correctness of the IFlow signatures in the runtime API register. Definitions 10 and 11 formally specify the conditions that the instrumented API register must verify in order for the generated compiler to be correct. We use $\rightarrow^*_{\mathsf{JS}}$ as the semantics relation for JavaScript configurations.

▶ **Definition 10** (Correct IFlow Signature)**.** An IFlow Signature $\langle \#c, \#l, \#d \rangle$ is *correct* with respect to an API $\Downarrow_{\mathsf{API}}$ if for all memories μ_0 and μ_1, labeling $\langle \Gamma, \Sigma \rangle$, sequence of values $\overrightarrow{v}$, and sequence of security levels $\overrightarrow{\sigma}$, such that $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu_1$ for some function β, then: $\langle \mu_0, \Gamma, \Sigma, \overrightarrow{v}, \overrightarrow{\sigma} \rangle \ \Downarrow_{\mathsf{API}} \ \langle \mu_0', \Gamma', \Sigma', v_0, \sigma \rangle$ if and only if (1) $\langle \mu_1, \#c(\beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1', true \rangle$, (2) $\langle \mu_1', \beta(\overrightarrow{v}) \rangle \ \Downarrow^{\mathsf{JS}}_{\mathsf{API}} \ \langle \mu_1'', v_1 \rangle$, and (3) $\langle \mu_1'', \#l(v_1, \beta(\overrightarrow{v}), \overrightarrow{\sigma}) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1''', \sigma \rangle$, in which case $\langle \mu_0', \Gamma', \Sigma' \rangle \ \mathcal{S}_{\beta'} \ \mu_1'''$ and $v_0 \ \mathcal{S}_\beta \ v_1$, for some β' extending β.

▶ **Definition 11** (Correct Runtime API Register)**.** A runtime API register corresponding to a function object pointed by $\#\$apiRegister$ is correct with respect to an API register $\mathcal{R}_{\mathsf{API}}$ if for all memories μ_0 and μ_1, labeling $\langle \Gamma, \Sigma \rangle$ and sequence of values $\overrightarrow{v}$, such that $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_\beta \ \mu_1$ for some function β, then: $\mathcal{R}_{\mathsf{API}}(\mu_0, \overrightarrow{v}) = \Downarrow_{\mathsf{API}}$ if and only if (1) $\langle \mu_1, \#apiRegister(\beta(\overrightarrow{v})) \rangle \rightarrow^*_{\mathsf{JS}} \langle \mu_1', r_{\mathsf{sig}} \rangle$, (2) $\langle \mu_0, \Gamma, \Sigma \rangle \ \mathcal{S}_{\beta'} \ \mu_1'$ for some β' extending β, and (3) signature $\langle o_{\mathsf{sig}}(\text{"check"}), o_{\mathsf{sig}}(\text{"label"}), o_{\mathsf{sig}}(\text{"domain"}) \rangle$ is correct with respect to $\Downarrow_{\mathsf{API}}$, where $o_{\mathsf{sig}} = \mu_1'(r_{\mathsf{sig}})$.

Theorem 12 states that provided that the compiler given as input to the meta-compiler is correct and the runtime API register is correct, the generated compiler is also correct.

▶ **Theorem 12** (Correctness)**.** *If compiler $\mathcal{C}$ is correct w.r.t. $\rightarrow_{\mathsf{IF}}$, then $\mathcal{C}_{\mathsf{API}}\langle \mathcal{C} \rangle$ is correct w.r.t. $\mathcal{E}(\rightarrow_{\mathsf{IF}}, \mathsf{Intercept}, \mathcal{R}_{\mathsf{API}})$ provided that the runtime API register is correct w.r.t. $\mathcal{R}_{\mathsf{API}}$.*

The meta-compiler proposed in this section allows the developer of the inlining compiler to extend it in a modular way, developing and proving each API IFlow signature at a time.

5 Implementation of the Meta Compiler and DOM API Extension

An implementation of a meta-compiler based on the JavaScript inlining compiler of [10] can be found in [20] together with an online testing tool and a set of IFlow signatures that

$$\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}(\mu, \langle r, m, \ldots\rangle) = \begin{cases} \Downarrow_{\mathsf{cre}} & \text{if } m = \text{“createElement”} \wedge r = \#doc \\ \Downarrow_{\mathsf{app}} & \text{if } m = \text{“appendChild”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{rem}} & \text{if } m = \text{“removeChild”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{len}} & \text{if } m = \text{“length”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{par}} & \text{if } m = \text{“parentNode”} \wedge @tag \in \mathsf{dom}(\mu(r)) \\ \Downarrow_{\mathsf{ind}} & \text{if } m \in \mathsf{Number} \wedge @\mathsf{tag} \in \mathsf{dom}(\mu(\mathsf{r})) \\ \Downarrow_{\mathsf{sib}} & \text{if } m = \text{“nextSibling”} \wedge @tag \in \mathsf{dom}(\mu(r)) \end{cases}$$

Figure 3 API register $\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}$ for the DOM API.

includes all those studied in the paper. As a case study, we give a high-level description of our the DOM API extension.

Interaction between client-side JavaScript programs and the HTML document is done *via* the DOM API [17]. In order to access the functionalities of this API, JavaScript programs manipulate a special kind of objects, here named *DOM objects*. In contrast to the ECMA Standard [1] that specifies in full detail the internals of objects created at runtime, the DOM API only specifies the behavior that DOM interfaces are supposed to exhibit when a program interacts with them. Hence, browser vendors are free to implement the DOM API as they see fit. In fact, in all major browsers, the DOM is not managed by the JavaScript engine. Instead, there is a separate engine, often called the *render engine*, whose role is to do so. Therefore, interactions between a JavaScript program and the DOM may potentially stop the execution of the JavaScript engine and trigger a call to the render engine. Thus, a monitored JavaScript engine has no access to the implementation of the DOM API.

We model DOM objects as standard JavaScript objects and we assume that every memory contains a *document* object denoted *doc*, which is accessed through the property “doc” and stored in fixed reference $\#doc$. Each DOM object defines a property $@tag$ that specifies its tag (for instance, $\langle div\rangle$, $\langle html\rangle$, $\langle a\rangle$) and, possibly, an arbitrary number of indexes 0, ..., n each pointing to one of its $n + 1$ children. DOM Element objects form a *forest*, such that the displayed HTML document corresponds to the tree hanging from the object pointed to by $\#doc$. Due to lack of space, we only present the labeled API relation for removing a DOM Element object from its parent object in the DOM forest. This API method gives rise to implicit information flows [2, 18, 23] that its labeled version needs to take into account.

▶ **Example 13** (Leak via removeChild - Order Leak)**.** Suppose that in the original memory there are three orphan DIV nodes bound to variables `div1`, `div2`, and `div3`.

```
1 div1.appendChild(div2); div1.appendChild(div3);
2 if(h) { div1.removeChild(div2); }
3 l = div1[0];
```

After the execution of this program, depending on the value of the high variable `h`, the value of the low variable `l` can be either that of `div2` or `div3`, meaning that the final level associated with variable `l` must be H in both executions. This example shows that, when removing a node, the new indexes of its right siblings are affected. To tackle this problem, the labelled DOM API methods enforce that the level of the property through which a DOM object is accessed is always lower than or equal to the levels of the properties corresponding to its right siblings.

Below we give the specification of the labeled API relation $\Downarrow_{\mathsf{rem}}$ for removing a DOM object from its parent in the DOM forest. This rule receives a sequence of arguments $\langle r_0, m_1, r_2\rangle$ as input and removes the object pointed to by r_2 from the children of the object pointed to by r_1. To this end, it first checks that $\mu(r_0)$ is in fact the parent of $\mu(r_2)$. Then,

```
domain = function(o0, m){
    return o0[@tag] && (m == "removeChild");
}

check = function(o0, m1, o2, σ0, σ1, σ2){
    var i = $index(o0, o2);
    return [$check(σ0 ⊔ σ1 ⊔ σ2 ≤ o0[$shadow(i)]);]†
}

label = function(ret, o0, m1, o2, σ0, σ1, σ2){
    var j = $index(o0, o2);
    while(j < o0.length − 1){
        [o0[$shadow(j)] = o0[$shadow(j + 1)];]††
    }
    return [σ0 ⊔ σ1 ⊔ σ2]†††;
}
```

Figure 4 IFlow Signature of $\Downarrow_{\mathsf{rem}}$.

the object $\mu(r_0)$ is updated by shifting by -1 all the indexes equal to or higher than i (the index of the object being removed) and by removing its index n. The levels of the indexes of the right siblings of the node to remove are accordingly shifted by -1. The constraint of the rule prevents a program from removing in a high context a node that was inserted in a low context. Function $\mathcal{R}_{\#\mathsf{Children}}$ receives a memory μ as input and outputs a binary relation such that if $\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu)$, then the DOM node pointed to by r has n children (with indexes $0, \ldots, n-1$).

REMOVECHILD

$$\frac{\begin{array}{c}\mu(r_0)(i) = r_2 \qquad \langle r_0, n+1\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \mathsf{dom}(o_0) = \mathsf{dom}(\gamma_0) = \mathsf{dom}(\mu(r_0))\backslash\{n\} \\ \forall_{0\le j<i}\ .\ o_0(j) = \mu(r_0)(j) \qquad \forall_{i\le j<n}\ .\ o_0(j) = \mu(r_0)(j+1) \qquad o_0(@tag) = \mu(r_0)(@tag) \\ \forall_{0\le j<i}\ .\ \gamma_0(j) = \Gamma(r_0)(j) \qquad \forall_{i\le j<n}\ .\ \boxed{\gamma_0(j) = \Gamma(r_0)(j+1)}^{\dagger\dagger} \qquad \gamma_0(@tag) = \Gamma(r_0)(@tag) \\ \mu' = \mu\,[r_0 \mapsto o_0] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \gamma_0] \qquad \boxed{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Gamma(r_0)(i)}^{\dagger}\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, m_1, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \ \Downarrow_{\mathsf{rem}}\ \langle \mu', \Gamma', \Sigma, r_2, \boxed{\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2}^{\dagger\dagger\dagger}\rangle}$$

In order for DOM API relations to be added to the semantics, one has to add them to the API register. Hence, we assume that the $\mathcal{R}_{\mathsf{API}}$ extends the API register given in Figure 3. The following lemma validates the hypotheses of the security theorem (Theorem 8) for $\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}$, allowing us to conclude that the extension of a noninterferent JavaScript monitor with the DOM API relations here defined is noninterferent.

▶ **Lemma 14** (Confinement and Noninterference for the DOM API). $\mathbf{Conf}(\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}}) \wedge \mathbf{NI}(\mathcal{R}^{\mathsf{DOM}}_{\mathsf{API}})$

Figure 4 presents a possible IFlow signature for the API relation $\Downarrow_{\mathsf{rem}}$, which makes use of the following runtime functions: (1) $\$check$ diverges if its argument is different from *true* and returns *true* otherwise; (2) $\$shadow$ receives as input a property name and outputs the name of the corresponding shadow property; and (3) $\$index$ outputs the index of its second argument in the list of children of its first argument. The labeled boxes in the API relation rule and in the code of the IFlow signature are intended to emphasize the correspondence between the two.

6 Conclusion

In summary, we have proposed a methodology for extending arbitrary monitored JavaScript semantics with secure APIs, which allows to prove the security of the extended monitor in a modular way. As a case study, we extend the inlining compiler of [10] with a fragment of the DOM Core Level 1 API. Further related technical developments, as well as an implementation that includes the IFlow signatures of the APIs studied in the paper, can be found in [20].

This work has been partially supported by the EPSRC Grant Reference EP/H008373/1.

References

1. The 5.1th edition of ECMA 262 June 2011. ECMAScript Language Specification. Technical report, ECMA, 2011.
2. A. Almeida-Matos, J. Fragoso Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *TGC*, 2014.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS*, 2009.
4. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS*, 2010.
5. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL*, 2012.
6. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW*, 2002.
7. N. Bielova. Survey on javascript security policies and their enforcement mechanisms in a web browser. *Special Issue on Automated Specification and Verification of Web Systems of JLAP*, 2013.
8. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *CSF*, 2010.
9. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.
10. J. Fragoso Santos and T. Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of Javascript. In *SEC*, 2014.
11. P. Gardner, G. Smith, M. J. Wheelhouse, and U. Zarfaty. Dom: Towards a formal specification. In *PLAN-X*, 2008.
12. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University, 2007.
13. A. Guha, B. Lerner, J. Gibbs Politz, and S. Krishnamurthi. Web api verification: Results and challenges. 2012.
14. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *SAC*.
15. D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*, 2012.
16. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 2012.
17. W3C Recommendation. DOM: Document Object Model (DOM). Technical report, W3C, 2005.
18. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
19. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
20. José Fragoso Santos and Tamara Rezk. Information flow monitor-inlining compiler. http://www-sop.inria.fr/indes/ifJS/.
21. A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated analysis of security-critical javascript apis. In *SP*, 2011.
22. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In *ICICS*, 2006.
23. Deepak Garg Vineet Rajani, Abhishek Bichhawat and Christian Hammer. Information Flow control for Event Handling and the DOM in Web Browsers. In *CSF*, 2015. to appear.

A DOM API Relations

This appendix describes the labeled API relations with which we extend the JavaScript semantics for interaction with DOM objects.

A.1 Auxiliary Semantic Functions

Our specification of the DOM API relations makes use of the following semantic functions:

- $\mathcal{R}_{\#\mathsf{Children}}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathbb{N}$, such that if $\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu)$, then the DOM node pointed to by r has n children (meaning that it defines the indexes $0, \cdots, n-1$).
- $\mathcal{R}_{\mathsf{Ancestor}}$ receives a memory μ as input and outputs a binary relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{\mathsf{Ancestor}}(\mu)$, then the DOM node pointed to by r_0 is an ancestor of the DOM node pointed to by r_1 in the DOM forest stored in μ.
- $\mathcal{R}_{\mathsf{Parent}}$ receives a memory μ as input and outputs a relation in $\mathcal{R}ef \times \mathcal{R}ef$, such that if $\langle r_0, r_1\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu)$, then the DOM node pointed to by r_0 is the *parent* of the DOM node pointed to by r_1 (meaning that there is an index i such that $\mu(r_0)(i) = r_1$).
- $\mathcal{O}rphan$ receives a memory μ as input and ouputs a set of references, such that if $r \in \mathcal{O}rphan(\mu)$, then the DOM node pointed to by r is an *orphan* node, that is, it does not have a parent in the DOM forest stored in μ (meaning that it is the root of a dangling tree).

A.2 DOM API Relations - Invariants

Indexes Invariant. When appending a new node to a given node, its index depends on the indexes of the nodes that were already appended. Analogously, when removing a node, the new indexes of its right siblings depend on the index of the node that is to be removed. To tackle this problem, we specify the semantic relations corresponding to the DOM methods *removeChild* and *appendChild* in such a way that, for every DOM node, the level of the property through which it is accessed is always lower than or equal to the levels of the properties corresponding to its right siblings. We refer to this invariant as the *indexes invariant.*

Parent Node Invariant. In the formal model, a DOM object does not define a property pointing to its parent. However, the API relations are specified in such a way that the structure security level of a DOM node works as the level of a "ghost" property pointing to its parent node. Hence, if the structure of a DOM object is observable, it also means that its parent is also observable.

A.3 DOM API Relations - Specification

In the following, we explain the monitored API rules given in Figure 5. In the specification of each API, when an element of the initial configuration is not used in the premises of the corresponding rule, we denote it by _.

- [CREATEELEMENT] The API relation $\Downarrow_{\mathsf{cre}}$ creates a new DOM Element node with tag m and binds a free reference r to it. The structure security level of the newly created node as well as the level of its property @*tag* are both set to $\sigma_0 \sqcup \sigma_1 \sqcup \sigma_2$ in order to verify the confinement property (Definition 5).

createElement

$$\frac{r \notin \mathsf{dom}(\mu) \qquad \mu' = \mu\,[r \mapsto [@tag \mapsto m]] \qquad \Gamma' = \Gamma\,[r \mapsto [@tag \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]] \qquad \Sigma' = \Sigma\,[r \mapsto \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2]}{\langle \mu, \Gamma, \Sigma, \langle \#doc, \_, m\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{cre}}\; \langle \mu', \Gamma', \Sigma', r, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

removeChild

$$\frac{\begin{array}{c}\mu(r_0)(i) = r_2 \qquad \langle r_0, n+1\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \mathsf{dom}(o_0) = \mathsf{dom}(\gamma_0) = \mathsf{dom}(\mu(r_0))\backslash\{n\} \\ \forall_{0\le j<i}\cdot o_0(j) = \mu(r_0)(j) \qquad \forall_{i\le j<n}\cdot o_0(j) = \mu(r_0)(j+1) \qquad o_0(@tag) = \mu(r_0)(@tag) \\ \forall_{0\le j<i}\cdot \gamma_0(j) = \Gamma(r_0)(j) \qquad \forall_{i\le j<n}\cdot \gamma_0(j) = \Gamma(r_0)(j+1) \qquad \gamma_0(@tag) = \Gamma(r_0)(@tag) \\ \mu' = \mu\,[r_0 \mapsto o_0] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \gamma_0] \qquad \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Gamma(r_0)(i)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{rem}}\; \langle \mu', \Gamma', \Sigma, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

appendChild - orphan node

$$\frac{\begin{array}{c}\langle r_2, r\rangle \notin \mathcal{R}_{\mathsf{Ancestor}}(\mu) \qquad r_2 \in \mathcal{O}rphan(\mu) \qquad \langle r_0, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \\ \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Sigma(r_0) \sqcap \Sigma(r_2) \\ \mu' = \mu\,[r_0 \mapsto \mu(r_0)\,[n \mapsto r_2]] \qquad \Gamma' = \Gamma\,[r_0 \mapsto \Gamma(r_0)\,[n \mapsto \Sigma(r_0) \sqcup \Sigma(r_2)]]\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu', \Gamma', \Sigma, r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

appendChild - non-orphan node

$$\frac{\begin{array}{c}\langle r_p, r_2\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \qquad \langle \mu, \Gamma, \Sigma, \langle r_p, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{rem}}\; \langle \mu', \Gamma', \Sigma', \_, \_\rangle \\ \langle \mu', \Gamma', \Sigma', \langle r_0, \_, r_2\rangle, \langle \sigma_0 \sqcup \Sigma(r_2), \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu'', \Gamma'', \Sigma'', \_, \_\rangle \qquad \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2 \le \Sigma(r_2)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r_0, \_, r_2\rangle, \langle \sigma_0, \sigma_1, \sigma_2\rangle\rangle \;\Downarrow_{\mathsf{app}}\; \langle \mu'', \Gamma'', \Sigma'', r_2, \sigma_0 \sqcup \sigma_1 \sqcup \sigma_2\rangle}$$

length

$$\frac{\langle r, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \qquad \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r)}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{len}}\; \langle \mu, \Gamma, \Sigma, n, \sigma\rangle}$$

parentNode

$$\frac{v = \begin{cases} r_p & \text{if } \langle r_p, r\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \\ \mathit{undefined} & \text{otherwise}\end{cases} \qquad \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \Sigma(r_2)}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{par}}\; \langle \mu, \Gamma, \Sigma, v, \sigma\rangle}$$

index

$$\frac{\langle v, \sigma\rangle = \begin{cases} \langle \mu(r)(i), \Gamma(r)(i)\rangle & \text{if } i \in \mathsf{dom}(\mu(r)) \\ \langle \mathit{undefined}, \Sigma(r)\rangle & \text{otherwise}\end{cases}}{\langle \mu, \Gamma, \Sigma, \langle r, i\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{ind}}\; \langle \mu, \Gamma, \Sigma, v, \sigma_0 \sqcup \sigma_1 \sqcup \sigma)\rangle}$$

nextSibling

$$\frac{\begin{array}{c}\langle r_p, r\rangle \in \mathcal{R}_{\mathsf{Parent}}(\mu) \qquad \langle r_p, n\rangle \in \mathcal{R}_{\#\mathsf{Children}}(\mu) \\ \langle v_i, \sigma_i\rangle = \begin{cases} \langle \mu(r_p)(i+1), \Gamma(r_p)(i+1)\rangle & \text{if } i+1 < n \\ \langle \mathit{undefined}, \Gamma(r_p)\rangle & \text{otherwise}\end{cases} \\ \sigma = \sigma_0 \sqcup \sigma_1 \sqcup \sigma_i \sqcup \Sigma(r)\end{array}}{\langle \mu, \Gamma, \Sigma, \langle r, \_\rangle, \langle \sigma_0, \sigma_1\rangle\rangle \;\Downarrow_{\mathsf{sib}}\; \langle \mu, \Gamma, \Sigma, v_i, \sigma\rangle}$$

Figure 5 DOM API Relations

- [removeChild] The API relation $\Downarrow_{\mathsf{rem}}$ removes the node pointed to by r_2 from the list of children of the node pointed to by r_0, after checking that $\mu(r_0)$ is in fact the parent of $\mu(r_2)$. The object $\mu(r_0)$ is updated by shifting by -1 all the indexes equal to or higher than i (the index of the object being removed) and by removing index n. The levels of

the indexes of the right siblings of the node to remove are accordingly shifted by -1. The constraint of the rule prevents a program from removing in a high context a node that was inserted in a low context (see Example 13).

- [APPENDCHILD] The API relation $\Downarrow_{\mathsf{app}}$ has two different behaviors depending on the fact that the node pointed to by r_2 is or is not an orphan node. If the node pointed to by r_2 is an orphan node, the behavior of $\Downarrow_{\mathsf{app}}$ is the following: (1) it first checks that the node to append ($\mu(r_2)$) is not an ancestor of the node to which it is to be appended ($\mu(r_0)$); (2) it creates a new property n in $\mu(r_0)$ and sets it to point to $\mu(r_2)$ (where n is the previous number of children of $\mu(r_0)$); (3) the level of the new index property n is set to the least upper bound on the levels of the arguments and the level of its new left sibling provided that it exists (in order to enforce the *Indexes Invariant*); (4) the least upper bound on the level of the arguments must be equal to or lower than the structure security level of $\mu(r_0)$ because adding an index to a node changes its domain; (5) the least upper bound on the level of the arguments must be equal to or lower than the structure security level of $\mu(r_2)$ (in order to enforce the *Parent Node Invariant*). If the node pointed to by r_2 is not an orphan node, the behavior of $\Downarrow_{\mathsf{app}}$ is the following: (1) it removes $\mu(r_2)$ from the list of children of its current parent (using the $\Downarrow_{\mathsf{rem}}$ API relation); (2) the API relation $\Downarrow_{\mathsf{app}}$ calls itself recursively.
- [LENGTH] The API relation $\Downarrow_{\mathsf{len}}$ evaluates to the number of children of $\mu(r)$. The reading effect of a call to this API must be higher than or equal to the structure security level of $\mu(r)$ because it leaks information about the domain of $\mu(r)$. Concretely, by calling this API relation, one finds out which are the index properties that the node defines.
- [PARENTNODE] The API relation $\Downarrow_{\mathsf{par}}$ evaluates either to the reference that points to the parent of $\mu(r)$, or to *undefined* if $\mu(r)$ is an orphan node. The reading effect of a call to this API is higher than or equal to the structure security level of $\mu(r)$ because it acts as the level of a "ghost" property pointing to the corresponding parent node.
- [INDEX] The API relation $\Downarrow_{\mathsf{ind}}$ evaluates to the ith child of $\mu(r)$. If $\mu(r)$ has less than $i + 1$ children the call to this API returns *undefined*. Besides the security levels of the arguments, the reading effect of a call to this API must take into account either the security level associated with index i (provided that it is defined), or the structure security level of $\mu(r)$ (if it does not exist).
- [NEXTSIBLING] The API relation $\Downarrow_{\mathsf{sib}}$ evaluates either to the reference that points to the right sibling of $\mu(r)$, or to *undefined* if $\mu(r)$ does not have a right sibling. In the former case, the reading effect of a call to this API is higher than or equal to the security level associated with the index pointing to the right sibling, whereas in the latter case it must be higher than or equal to the structure security level of the parent node of $\mu(r)$.

A Taxonomy of Information Flow Monitors

Nataliia Bielova and Tamara Rezk

Inria, France
name.surname@inria.fr

Abstract. We propose a rigorous comparison of information flow monitors with respect to two dimensions: soundness and transparency.
For soundness, we notice that the standard information flow security definition called Termination-Insensitive Noninterference (TINI) allows the presence of termination channels, however it does not describe whether the termination channel was present in the original program, or it was added by a monitor. We propose a stronger notion of noninterference, that we call Termination-Aware Noninterference (TANI), that captures this fact, and thus allows us to better evaluate the security guarantees of different monitors. We further investigate TANI, and state its formal relations to other soundness guarantees of information flow monitors.
For transparency, we identify different notions from the literature that aim at comparing the behaviour of monitors. We notice that one common notion used in the literature is not adequate since it identifies as better a monitor that accepts insecure executions, and hence may augment the knowledge of the attacker. To discriminate between monitors' behaviours on secure and insecure executions, we factorized two notions that we call true and false transparency. These notions allow us to compare monitors that were deemed to be incomparable in the past.
We analyse five widely explored information flow monitors: no-sensitive-upgrade (NSU), permissive-upgrade (PU), hybrid monitor (HM), secure multi-execution (SME), and multiple facets (MF).

1 Introduction

Motivated by the dynamic nature and an extensive list of vulnerabilities found in web applications in recent years, several dynamic enforcement mechanisms in the form of information flow monitors [5–7, 10, 13, 15, 18, 25, 29, 35], have been proposed. In the runtime monitor literature [8, 14], two properties of monitors are considered specially important: soundness and transparency. In this work, we rigorously compare information flow monitors with respect to these two dimensions. We analyse five widely explored information flow monitor techniques: no-sensitive-upgrade (NSU) [35], permissive-upgrade (PU) [6], hybrid monitor (HM) [15], secure multi-execution (SME) [13], and multiple facets (MF) [7].

Soundness An information flow monitor is sound when it ensures that observable outputs comply with a given information flow policy. In the case of noninterference, the monitor must ensure equal observable outputs if executions start in

equal observable inputs. We notice that some monitoring techniques introduce new termination channels, whereas others don't. The standard information flow security definition called *Termination-Insensitive Noninterference (TINI)* does not account for termination: only initial memories in which the program terminates should lead to equal observable outputs. Thus, TINI allows the presence of termination channels, however it does not describe whether the termination channel was present in the original program, or it was added by a monitor. *Termination-Sensitive Noninterference* (TSNI), on the other hand, is a stronger policy that disallows the presence of any termination channel. However, most information flow monitors do not satisfy TSNI. Hence, existing definitions do not allow us to discriminate between different monitors with respect to the security guarantees that they provide. We propose a notion of noninterference, stronger than TINI but weaker than TSNI, that we call *Termination-Aware Noninterference (TANI)*, that captures the fact that the monitor does not introduce a new termination channel, and thus allows to better evaluate the security guarantees of different monitors. We discovered that HM, SME, and MF do satisfy TANI, while NSU and PU do not satisfy TANI.

Example 1 (NSU introduces a termination channel). Consider the following program, where each variable can take only two possible values: 0 and 1.

Program 1

```
1 if h = 0 then l = 1;
2 output l
```

This program is leaking confidential information – upon observing output `l=0` (`l=1`), it's possible to derive that `h=1` (`h=0`). In spite of this fact, NSU allows the execution of this program starting in a memory [`h=1, l=0`] and blocks the execution otherwise, thus introducing a new termination channel.

Transparency An information flow monitor is transparent when it preserves program semantics if the execution complies with the policy. In the case of noninterference, the monitor must produce the same output as an original program execution with a value that only depends on observable inputs. We identify different common notions from the literature that aim at comparing the behaviour of monitors: precision, permissiveness, and transparency. We notice that permissiveness is not adequate since it identifies as better a monitor that accepts insecure executions, and hence may augment the knowledge of the attacker, given that the attacker has knowledge based on the original executions. To discriminate between monitors' behaviours on secure and insecure executions, we factorized two notions that we call true and false transparency. *True transparency* corresponds to the standard notion of transparency in the field of runtime monitoring: the ability of a monitor to preserve semantics of secure executions. An information flow monitor is *false transparent* when it preserves semantics of the original program execution that does not comply with the security policy. False transparency might seem contradictory to soundness at first sight but this is not the case since information flow is not a property of one execution [2, 26] but a property of several executions, also called a hyperproperty [12, 31]. These two

notions of transparency allow us to compare monitors that were deemed to be incomparable in the past. In particular, we prove that HM is *more TSNI precise* (more true transparent for the set of TSNI secure programs) than NSU and NSU is more false transparent than HM. Proofs can be found in the companion technical report [1].

Our contributions are the following:

1. We propose a new information flow policy called termination-aware noninterference (TANI) that allows us to evaluate monitors according to their soundness guarantees. We prove that TANI is stronger than TINI but weaker than TSNI that disallows any termination channels.
2. We identify two different notions of transparency that are used in the literature as the same notion and we call them true and false transparency.
3. We generalize previous results from Hedin et al. [17]: we show that dynamic and hybrid monitors become comparable when the two flavors of transparency are separated into true and false transparency.
4. We analyse and compare five major monitors previously proved sound for TINI: NSU, PU, HM, SME and MF. Table 2 in Section 9 summarizes our results for TANI, true and false transparency.

2 Knowledge

We assume a two-element security lattice with $L \sqsubseteq H$ and we use $\sqcup$ as the least upper bound. A security environment Γ maps program variables to security levels. By $\mu_{\mathtt{L}}$ we denote a projection of low variables of the memory μ, according to an implicitly parameterized security environment Γ. The program semantics is defined as a big-step evaluation relation $(P, \mu) \Downarrow (v, \mu')$, where P is a program that produces only one output v at the end of execution. We assume that v is visible to the attacker at level L and that the program semantics is deterministic. The attacker can gain knowledge while observing output v. Following Askarov and Sabelfeld [3, 4], we define knowledge as a set of low-equal memories, that lead to the program observation v.

Definition 1 (Knowledge). *Given a program P, the low part $\mu_{\mathtt{L}}$ of an initial memory μ, and an observation v, the* knowledge *for semantics relation $\Downarrow$ is a set of memories that agree with μ on low variables and can lead to an observation v: $k_{\Downarrow}(P, \mu_{\mathtt{L}}, v) = \{\mu' \mid \mu_{\mathtt{L}} = \mu'_{\mathtt{L}} \wedge \exists \mu''.(P, \mu') \Downarrow (v, \mu'')\}$.*

Notice that knowledge corresponds to *uncertainty* about the environments in the knowledge set: any environment is a possible program input. The attacker believes that the environments outside of the knowledge set are *impossible* inputs. Upon observing a program output, the uncertainty might decrease because the new output may render some inputs impossible. This means that the knowledge set may become smaller, thus increasing the knowledge of the attacker.

To specify a security condition, we define what it means for an attacker not to gain any knowledge. Given a program P, and a low part $\mu_{\mathtt{L}}$ of an initial

memory μ, the attacker's knowledge before the execution of the program is a set of memories that agree with μ on low variables. This set is an equivalence class of low-equal memories: $[\mu]_{\mathtt{L}} = \{\mu' \mid \mu_{\mathtt{L}} = \mu'_{\mathtt{L}}\}$.

Definition 2 (Possible outputs). *Given a program P and the low part $\mu_{\mathtt{L}}$ of an initial memory μ, a set of observable outputs for semantics relation $\Downarrow$ is:* $\mathcal{O}_{\Downarrow}(P, \mu_{\mathtt{L}}) = \{v \mid \exists \mu', \mu''.\ \mu_{\mathtt{L}} = \mu'_{\mathtt{L}} \wedge (P, \mu') \Downarrow (v, \mu'')\}$.

In the following, we don't write the semantics relation $\Downarrow$ when we mean the program semantics; the definitions in the rest of this section can be also used with the subscript parameter $\Downarrow$ when semantics has to be explicit.

We now specify the security condition as follows: by observing a program output, the attacker is not allowed to gain any knowledge.

Definition 3 (Termination-Sensitive Noninterference). *Program P is* termination-sensitively noninterferent *for an initial low memory $\mu_{\mathtt{L}}$, written $TSNI(P, \mu_{\mathtt{L}})$, if for all possible observations $v \in \mathcal{O}(P, \mu_{\mathtt{L}})$, we have*

$$[\mu]_{\mathtt{L}} = k(P, \mu_{\mathtt{L}}, v)$$

A program P is termination-sensitively noninterferent*, written $TSNI(P)$, if for all possible initial memories μ, $TSNI(P, \mu_{\mathtt{L}})$.*

The above definition is *termination-sensitive* because it does not allow an attacker to learn the secret information from program divergence. Intuitively, if the program terminates on all low-equal memories, and it produces the same output v then it satisfies TSNI. If the program doesn't terminate on some of the low-equal memories, then for all possible observations v, the knowledge $k(P, \mu_{\mathtt{L}}, v)$ becomes a subset of $[\mu]_{\mathtt{L}}$ and doesn't satisfy the definition.

Example 2. Consider Program 2. If the attacker observes that `l=1`, then he learns that `h` was 0, and if the attacker doesn't see any program output (divergence), the attacker learns that `h` was 1. TSNI captures this kind of information leakage, hence TSNI doesn't hold.

Program 2

```
1 l = 1;
2 (while (h=1) do skip);
3 output l
```

Proposition 1. *$TSNI(P)$ holds if and only if for all pairs of memories μ^1 and μ^2, we have: $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} \wedge \exists \mu'.(P, \mu^1) \Downarrow (v_1, \mu') \Rightarrow \exists \mu''.(P, \mu^2) \Downarrow (v_2, \mu'') \wedge v_1 = v_2$.*

Termination-sensitive noninterference sometimes is too restrictive as it requires a more sophisticated program analysis or monitoring that may reject many secure executions of a program. A weaker security condition, called *termination-insensitive noninterference* ($TINI$), allows information flows through program divergence, while still offering information flow security.

To capture this security condition, we follow the approach of Askarov and Sabelfeld [4], by limiting the allowed attacker's knowledge to the set of low-equal memories where the program terminates. Since termination means that some output is observable, a set that we call a *termination knowledge*, is a union of all knowledge sets that correspond to some program output: $\bigcup_{v'} k(P, \mu_{\mathtt{L}}, v')$.

Definition 4 (Termination-Insensitive Noninterference). *Program P is* termination-insensitively noninterferent for an initial low memory $\mu_{\mathtt{L}}$*, written* $TINI(P, \mu_{\mathtt{L}})$*, if for all possible observations $v \in \mathcal{O}(P, \mu_{\mathtt{L}})$, we have*

$$\bigcup_{v' \in \mathcal{O}(P,\mu_{\mathtt{L}})} k(P, \mu_{\mathtt{L}}, v') = k(P, \mu_{\mathtt{L}}, v).$$

A program P is termination-insensitively noninterferent*, written $TINI(P)$, if for all possible initial memories μ, $TINI(P, \mu_{\mathtt{L}})$.*

Example 3. TINI recognises the Program 2 as secure, since the attacker's *termination knowledge* is only a set of low-equal memories where the program terminates. For example, for $\mu_{\mathtt{L}}$= [`l=0`], only one observation `l=1` is possible when `h=0`, therefore $TINI$ holds: $\bigcup_{v' \in \{1\}} k(P, \mathtt{l=0}, v') = [\mathtt{h=0,\ l=0}] = k(P, \mathtt{l=0}, 1)$.

Proposition 2. *$TINI(P)$ holds if and only if for all pairs of memories μ^1 and μ^2, we have: $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} \wedge \exists \mu'.(P, \mu^1) \Downarrow (v_1, \mu') \wedge \exists \mu''.(P, \mu^2) \Downarrow (v_2, \mu'') \Rightarrow v_1 = v_2$.*

3 Monitor soundness

In this section, we consider dynamic mechanisms for enforcing information flow security. For brevity, we call them "monitors". The monitors we consider are purely dynamic monitors, such as NSU and PU, hybrid monitors in the style of Le Guernic et al. [22,23] that we denote by HM, secure multi-execution (SME), and multiple facets monitor (MF). All the mechanisms we consider have deterministic semantics denoted by $\Downarrow_M$, where M represents a particular monitor. All the monitors enforce at least *termination-insensitive noninterference (TINI)*.[1] Since TINI accepts termination channels, it also allows the monitor to introduce new termination channels even if an original program did not have any. In the next section, we will propose a new definition for soundness of information flow monitors, capturing that a monitor should not introduce a new termination channel. But, first, we set up the similar definitions of termination-sensitive and -insensitive noninterference for a monitored semantics. Instead of using a subscript $\Downarrow_M$ for a semantics of a monitor M, we will use a subscript M.

Definition 5 (Soundness of TSNI enforcement). *Monitor M soundly enforces* termination-sensitive noninterference*, written $TSNI(M)$, if for all possible programs P, $TSNI_M(P)$.*

[1] This is indeed a lower bound since some monitors, like SME, also enforce termination- and time-sensitive noninterference.

Proposition 1 proves that this definition of TSNI soundness is equivalent to the standard two-run definition if we substitute the original program semantics with the monitor semantics. Similarly, to define a sound TINI monitor, we restate Definition 4 of TINI with the monitored semantics. The definition below is equivalent to the standard two-run definition (see Proposition 2).

Definition 6 (Soundness of TINI enforcement). *Monitor M soundly enforces* termination-insensitive noninterference*, written $TINI(M)$, if for all possible programs P, $TINI_M(P)$.*

This definition compares the initial knowledge and the final knowledge of the attacker under the monitor semantics. But in practice, an attacker has also the initial knowledge of the original program semantics (see Example 1).

4 Termination-Aware Noninterference

We propose a new notion of soundness for the monitored semantics, called *Termination-Aware Noninterference (TANI)* that does not allow a monitor to introduce a new termination channel.

Intuitively, all the low-equal memories, on which the original program terminates, should be treated by the monitor in the same way, meaning the monitor should either produce the same result for all these memories, or diverge on all of them. In terms of knowledge, it means that the knowledge provided by the monitor, should be smaller or equal than the knowledge known by the attacker before running the program. Additionally, in the case the original program always diverges, TANI holds if the monitor also always diverges or if the monitor always terminates in the same value.

Definition 7 (Termination-Aware Noninterference). *A monitor $\Downarrow_M$ is* Termination-Aware Noninterferent (TANI)*, written $TANI(M)$, if for all programs P, initial memories μ, and possible observations $v \in \mathcal{O}_M(P, \mu_{\mathtt{L}})$, we have:*

- $\mathcal{O}(P, \mu_{\mathtt{L}}) \neq \emptyset \implies \bigcup_{v' \in \mathcal{O}(P, \mu_{\mathtt{L}})} k(P, \mu_{\mathtt{L}}, v') \subseteq k_M(P, \mu_{\mathtt{L}}, v)$
- $\mathcal{O}(P, \mu_{\mathtt{L}}) = \emptyset \implies (\mathcal{O}_M(P, \mu_{\mathtt{L}}) = \emptyset \vee [\mu]_{\mathtt{L}} = k_M(P, \mu_{\mathtt{L}}, v))$

Notice that, for the case that the original program sometimes terminate ($\mathcal{O}(P, \mu_{\mathtt{L}}) \neq \emptyset$)), we do not require equality of the two sets of knowledge since the knowledge set of the monitored program can indeed be bigger than the knowledge set of the attacker before running the program[2]. The knowledge set may increase when a monitor terminates on the memories where the original program did not terminate (e.g., SME from Section 6 provides such enforcement).

Example 4 (TANI enforcement). Coming back to Program 1, TANI requires that on two low-equal memories [`h=0, l=0`] and [`h=1, l=0`] where the original program terminates, the monitor behaves in the same way: either it terminates on both memories producing the same output, or it diverges on both memories.

[2] Remember that the bigger knowledge set corresponds to the smaller knowledge or to the increased uncertainty.

It is well-known that TSNI is a strong form of noninterference that implies TINI. We now formally state the relations between TINI, TANI and TSNI.

Theorem 1. *$TSNI(M) \Rightarrow TANI(M)$ and $TANI(M) \Rightarrow TINI(M)$.*

5 Which monitors are TANI?

We now present five widely explored information flow monitors and prove whether these monitors comply with TANI. In order to compare the monitors, we first model all of them in the same language. Thus, our technical results are based on a simple imperative language with one output (see Figure 1). The language's expressions include constants or values (v), variables (x) and operators ($\oplus$) to combine them. We present the standard big-step program semantics in Figure 2.

$P ::= S$; output x
$S ::=$ skip $\mid x := e \mid S_1; S_2 \mid$ if x then S_1 else $S_2 \mid$ while x do S
$e ::= v \mid x \mid e_1 \oplus e_2$

Fig. 1: Language syntax

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, \mu) \Downarrow \mu} \qquad \textsc{assign}\ \frac{}{(x := e, \mu) \Downarrow \mu[x \mapsto [\![e]\!]_\mu]} \qquad \textsc{seq}\ \frac{(S_1, \mu) \Downarrow \mu' \qquad (S_2, \mu') \Downarrow \mu''}{(S_1; S_2, \mu) \Downarrow \mu''}$$

$$\textsc{if}\ \frac{[\![x]\!]_\mu = \alpha \qquad (S_\alpha, \mu) \Downarrow \mu'}{(\mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow \mu'} \qquad \textsc{while}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ S; \mathsf{while}\ x\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow \mu'}{(\mathsf{while}\ x\ \mathsf{do}\ S, \mu) \Downarrow \mu'}$$

$$\textsc{output}\ \frac{[\![x]\!]_\mu = v}{(\mathsf{output}\ x, \mu) \Downarrow (v, \mu)}$$

where $[\![x]\!]_\mu = \mu(x)$, $[\![v]\!]_\mu = v$ and $[\![e_1 \oplus e_2]\!]_\mu = [\![e_1]\!]_\mu \oplus [\![e_2]\!]_\mu$

Fig. 2: Language semantics

The semantics relation of a command S is denoted by $pc \vdash (\Gamma, S, \mu) \Downarrow_M (\Gamma', \mu')$ where pc is a program counter, M is the name of the monitor and Γ is a security environment mapping variables to security levels. All the considered monitors are flow-sensitive, and Γ may be updated during the monitored execution. We assume that the only output produced by the program is visible to the attacker at level L. Since our simple language supports only one output at the end of the program, the OUTPUT rule of the monitors is defined only for $pc = L$, and thus only checks the security level of an output variable x.

No-sensitive upgrade (NSU) The *no-sensitive upgrade approach* (NSU) first proposed by Zdancewic [35] and later applied by Austin and Flanagan [5] is

based on a purely dynamic monitor that controls only one execution of the program. To avoid implicit information flows, the NSU disallows any upgrades of a low security variables in a high security context. Consider Program 1: since the purely dynamic monitor accepts its execution when `h=1`, it should block the execution when `h=0` to enforce TINI. NSU does so by blocking the second execution since the low variable `l` is updated in a high context.

$$\textsc{skip}\ \frac{}{pc \vdash (\Gamma, \mathsf{skip}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma, \mu)}$$

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad pc \sqsubseteq \Gamma(x) \qquad \Gamma' = \Gamma[x \mapsto \Gamma(e) \sqcup pc]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{seq}\ \frac{pc \vdash (\Gamma, S_1, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu') \qquad pc \vdash (\Gamma', S_2, \mu') \Downarrow_{\mathbf{NSU}} (\Gamma'', \mu'')}{pc \vdash (\Gamma, S_1; S_2, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma'', \mu'')}$$

$$\textsc{if}\ \frac{[\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}$$

$$\textsc{while}\ \frac{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S; \mathsf{while}\ x\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{while}\ x\ \mathsf{do}\ S, \mu) \Downarrow_{\mathbf{NSU}} (\Gamma', \mu')}$$

$$\textsc{output}\ \frac{[\![x]\!]_\mu = v \qquad \Gamma(x) = L}{L \vdash (\Gamma, \mathsf{output}\ x, \mu) \Downarrow_{\mathbf{NSU}} (v, \Gamma, \mu)}$$

Fig. 3: NSU semantics

Our NSU formalisation for a simple imperative language is similar to that of Bichhawat *et al.* [11]. The main idea of NSU appears in the ASSIGN rule: the monitor blocks "sensitive upgrades" when a program counter level pc is not lower than the level of the assigned variable x. Figure 3 represents the semantics of NSU monitor. We use $\Gamma(e)$ as the least upper bound of all variables occurring in expression e. If e contains no variables, then $\Gamma(e) = L$. NSU was proven to enforce termination-insensitive noninterference (TINI) (see [5, Thm. 1]).

Example 5 (NSU is not TANI). Consider Program 1 and an initial memory [`h=1, l=0`]. NSU does not satisfy TANI, since the monitor terminates only on one memory, i.e., $k_M(P, \mu_{\mathtt{L}}, v) =$ [`h=1, l=0`], while the original program terminates on both memories, low-equal to [`l=0`].

Permissive Upgrade (PU) The NSU approach suffices to enforce TINI, however it often blocks a program execution pre-emptively. Consider Program 3. This program is TINI, however NSU blocks its execution starting in memory [`h=0, l=0`] because of a sensitive upgrade under a high security context.

```
1 if h = 0 then l = 1;
2 l := 0;
3 output l
```

Program 3

Austin and Flanagan proposed a less-restrictive strategy called *permissive upgrade* (PU) [6]. Differently from NSU, it allows the assignments of low variables under a high security context, but labels the updated variable as *partially-leaked* or 'P'. Intuitively, P means that the content of the variable is H but it may be L in other executions. If later in the execution, there is a branch on a variable marked with P, or such variable is to be output, the monitor stops the execution.

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad \Gamma' = \Gamma[x \mapsto \Gamma(e) \sqcup \mathsf{lift}(pc, \Gamma(x))]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{if}\ \frac{\Gamma(x) \neq P \qquad [\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{PU}} (\Gamma', \mu')}$$

where

$$\mathsf{lift}(pc, l) = \begin{cases} L & \text{if } pc = L \\ H & \text{if } pc = H \wedge l = H \\ P & \text{if } pc = H \wedge l \neq H \end{cases}$$

Fig. 4: PU semantics

We present a permissive upgrade monitor (PU) for a two-point lattice extended with label P with $H \sqsubset P$. The semantics of PU is identical to the one of NSU (see Fig. 3) except for the ASSIGN and IF rules, that we present in Fig. 5. Rule ASSIGN behaves like the ASSIGN rule of NSU, if $pc \sqsubseteq \Gamma(x)$ and $\Gamma(x) \neq P$. Otherwise, the assigned variable is marked with P. Rule IF is similar to the rule IF in NSU, but the semantics gets stuck if the variable in the test condition is partially leaked. PU was proven to enforce TINI (see [6, Thm. 2]). However, PU is not TANI since it has the same mechanism as NSU for adding new termination channels.

Example 6 (PU is not TANI). Consider Program 1 and an initial memory [`h=1, l=0`]. PU does not satisfy TANI, since the monitor terminates only on one memory, i.e., $k_M(P, \mu_{\mathtt{L}}, v) =$ [`h=1, l=0`], while the original program terminates on both memories, low-equal to [`l=0`].

Hybrid Monitor (HM) Le Guernic *et al.* were the first to propose a *hybrid monitor* (HM) [15] for information flow control that combines static and dynamic analysis. This mechanism statically analyses the non-executed branch of each test in the program, collecting all the possibly updated variables in that branch. The security level of such variables are then raised to the level of the test, thus preventing information leakage.

Example 7. Consider Program 1 and its execution starting in [h=1, l=0]. This execution is modified by HM because the static analysis discovers that variable l could have been updated in a high security context in an alternative branch.

$$\textsc{assign}\ \frac{[\![e]\!]_\mu = v \qquad \Gamma' = \Gamma[x \mapsto pc \sqcup \Gamma(e)]}{pc \vdash (\Gamma, x := e, \mu) \Downarrow_{\mathbf{HM}} (\Gamma', \mu[x \mapsto v])}$$

$$\textsc{if}\ \frac{\begin{array}{c}\Gamma'' = \mathsf{Analysis}(S_{\neg\alpha}, pc \sqcup \Gamma(x), \Gamma) \\ [\![x]\!]_\mu = \alpha \qquad pc \sqcup \Gamma(x) \vdash (\Gamma, S_\alpha, \mu) \Downarrow_{\mathbf{HM}} (\Gamma', \mu')\end{array}}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \Downarrow_{\mathbf{HM}} (\Gamma' \sqcup \Gamma'', \mu')}$$

$$\textsc{output}\ \frac{\Gamma(x) = L \Rightarrow v = [\![x]\!]_\mu \qquad \Gamma(x) \neq L \Rightarrow v = \mathsf{def}}{L \vdash (\Gamma, \mathsf{output}\ x, \mu) \Downarrow_{\mathbf{HM}} (v, \Gamma, \mu)}$$

Fig. 5: HM semantics

The semantics of HM is identical to NSU except for the ASSIGN, IF and OUTPUT rules that we show in Figure 6. The ASSIGN rule does not have any specific constraints. The static analysis $\mathsf{Analysis}(S, pc, \Gamma)$ in the IF rule explores variables assigned in S and upgrades their security level according to pc. We generalize the standard notation $\Gamma[x \mapsto l]$ to sets of variables and use $\mathsf{Vars}(S)$ for the sets of variables assigned in command S.

$$\mathsf{Analysis}(S, pc, \Gamma) = \Gamma[\{y \mapsto pc \sqcup \Gamma(y) \mid y \in \mathsf{Vars}(S)\}$$

HM was previously proven to enforce TINI [15, Thm. 1] and we prove in the companion technical report [1] that HM satisfies TANI.

Theorem 2. *HM is TANI.*

Secure Multi-Execution (SME) Devriese and Piessens were the first to propose secure multi-execution (SME) [13]. The main idea of SME is to execute the program multiple times: one for each security level. Each execution receives only inputs visible to its security level and a fixed def value for each input that should not be visible to the execution. Different executions are executed with a low priority scheduler to avoid leaks due to divergence of high executions because SME enforces TSNI.

Example 8 (SME "fixes" termination channels). Consider Program 4:

Program 4

```
1 if l = 0 then
2   while h=0 do skip;
3 else
4   while h=1 do skip;
5 output l
```

Assume $\mu_{\mathtt{L}} = [\mathtt{l{=}0}]$ and the default high value used by SME is `h=1`. Then, there exists a memory $\mu' = [\mathtt{h{=}0,\ l{=}0}]$, low-equal to $\mu_{\mathtt{L}}$, on which the original program doesn't terminate: $\mu' \notin \bigcup_{v'} k(P, \mu_{\mathtt{L}}, v')$, but SME terminates: $\mu' \in k_M(P, \mu_{\mathtt{L}}, \mathtt{l{=}0})$. Notice that SME makes the attacker's knowledge smaller.

$$\text{SME}\ \frac{(P, \mu|_\Gamma) \Downarrow (v, \mu') \qquad \mu''' = \begin{cases} \mu' \odot_\Gamma \mu'' & \text{if } \exists \mu''.(P,\mu) \Downarrow (v', \mu'') \\ \mu' \odot_\Gamma \bot & \text{otherwise} \end{cases}}{pc \vdash (\Gamma, P, \mu) \Downarrow_{\mathbf{SME}} (v, \Gamma, \mu''')}$$

$$\text{where} \quad \mu|_\Gamma(x) = \begin{cases} \mu(x) & \Gamma(x) = L \\ \mathsf{def} & \Gamma(x) = H \end{cases} \qquad \mu' \odot_\Gamma \mu''(x) = \begin{cases} \mu'(x) & \Gamma(x) = L \\ \mu''(x) & \Gamma(x) = H \end{cases}$$

Fig. 6: SME semantics

The SME adaptation for our while language is given in Figure 7, with executions for levels L and H. The special value $\bot$ represents the idea that no value can be observed and we overload the symbol to also denote a memory that maps every variable to $\bot$. Using memory $\bot$ we simulate the low priority scheduler of SME in our setting: if the execution corresponding to the H security level does not terminate, the SME semantics still terminates. In this case all the variables with level H, which values should correspond to values obtained in the normal execution of the program, are given value $\bot$.

SME was previously proven TSNI [13, Thm. 1] and we prove that it also enforces TANI: this can be directly inferred from our Theorem 1.

Theorem 3. *SME is TANI.*

Multiple Facets Austin and Flanagan proposed multiple facets (MF) in [7]. In MF, each variable is mapped to several values or facets, one for each security level: each value corresponds to the view of the variable from the point of view of observers at different security levels. The main idea in MF is that if there is a sensitive upgrade, MF semantics does not update the observable facet. Otherwise, if there is no sensitive upgrade, MF semantics updates it according to the original semantics.

Example 9. Consider the TINI Program 5. In MF, the output observable at level L (or the L facet of variable `l`) is always the initial value of variable `l` since MF will not update a low variable in a high context. Therefore, all the executions of Program 5 starting with `l=1` are modified by MF, producing the output `l=1`.

Program 5

```
1 if h = 0 then l = 0 else l=0;
2 output l
```

Our adaptation of MF semantics is given in Figure 8 where we use the following notation: a faceted value, denoted $\langle v_1 : v_2 \rangle$, is a pair of values v_1 and v_2.

$$\text{MF Rule}\ \boxed{\frac{pc \vdash (\Gamma, P, \mu \uparrow_\Gamma) \downarrow_{MF} (\langle v_1 : v_2\rangle, \Gamma', \hat{\mu})}{pc \vdash (\Gamma, P, \mu) \Downarrow_{\mathbf{MF}} (v_2, \Gamma', \hat{\mu} \downarrow_{\Gamma'})}}\ \text{SKIP}\ \frac{}{pc \vdash (\Gamma, \mathsf{skip}, \hat{\mu}) \downarrow_{MF} (\Gamma, \hat{\mu})}$$

$$\text{ASSIGN}\ \frac{[e]\hat{\mu} = \langle v_1 : v_2\rangle \qquad \hat{v} = \begin{cases}\langle v_1 : \hat{\mu}(x)_2\rangle & \text{if } pc = H \wedge \Gamma(x) = L\\ \langle v_1 : v_2\rangle & \text{if } pc = L \vee \Gamma(x) \neq L\end{cases} \qquad \Gamma'(y) = \begin{cases}\Gamma(e) & \text{if } pc = L \wedge y = x\\ \Gamma(y) & \text{otherwise}\end{cases}}{pc \vdash (\Gamma, x := e, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}[x \mapsto \hat{v}])}$$

$$\text{SEQ}\ \frac{pc \vdash (\Gamma, S_1, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}') \qquad pc \vdash (\Gamma', S_2, \hat{\mu}') \downarrow_{MF} (\Gamma'', \hat{\mu}'')}{pc \vdash (\Gamma, S_1; S_2, \hat{\mu}) \downarrow_{MF} (\Gamma'', \hat{\mu}'')}$$

$$\text{IF-HIGH}\ \frac{[x]\hat{\mu} = \langle \alpha_1 : \alpha_2\rangle \qquad pc = H \vee \Gamma(x) = H \qquad H \vdash (\Gamma, S_{\alpha_1}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}$$

$$\text{IF-LOW}\ \frac{[x]\hat{\mu} = \langle \alpha_1 : \alpha_2\rangle \qquad pc = L \wedge \Gamma(x) = L \qquad L \vdash (\Gamma, S_{\alpha_1}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}_1) \qquad L \vdash (\Gamma, S_{\alpha_2}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}_2)}{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S_{true}\ \mathsf{else}\ S_{false}, \mu) \downarrow_{MF} (\Gamma', \hat{\mu}_1 \otimes_\Gamma \hat{\mu}_2)}$$

$$\text{WHILE}\ \frac{pc \vdash (\Gamma, \mathsf{if}\ x\ \mathsf{then}\ S; \mathsf{while}\ x\ \mathsf{do}\ S\ \mathsf{else}\ \mathsf{skip}, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}{pc \vdash (\Gamma, \mathsf{while}\ x\ \mathsf{do}\ S, \hat{\mu}) \downarrow_{MF} (\Gamma', \hat{\mu}')}$$

$$\text{OUTPUT}\ \frac{[x]\hat{\mu} = \hat{v}}{L \vdash (\Gamma, \mathsf{output}\ x, \hat{\mu}) \downarrow_{MF} (\hat{v}, \Gamma, \hat{\mu})}$$

where

$$[\hat{v}]\hat{\mu} = \hat{v}, \quad [x]\hat{\mu} = \hat{\mu}(x)$$

$$[e_1 \oplus e_2]\hat{\mu} = \langle v_1 \oplus v_2 : v_1' \oplus v_2'\rangle, \text{ where } [e_1]\hat{\mu} = \langle v_1 : v_1'\rangle, [e_2]\hat{\mu} = \langle v_2 : v_2'\rangle$$

$$\hat{\mu}_1 \otimes_\Gamma \hat{\mu}_2(x) = \begin{cases}\hat{\mu}_1(x) & \text{if } \Gamma(x) = H\\ \langle \hat{\mu}_1(x)_1 : \hat{\mu}_2(x)_2\rangle & \text{if } \Gamma(x) = L\end{cases}$$

$$\mu \uparrow_\Gamma (x) = \begin{cases}\langle \mu(x) : \mu(x)\rangle & \text{if } \Gamma(x) = L\\ \langle \mu(x) : \bot\rangle & \text{if } \Gamma(x) = H\end{cases} \qquad \hat{\mu} \downarrow_\Gamma (x) = \begin{cases}\hat{\mu}(x)_1 & \text{if } \Gamma(x) = H\\ \hat{\mu}(x)_2 & \text{if } \Gamma(x) = L\end{cases}$$

Fig. 7: Multiple Facets semantics

The first value presents the view of an observer at level H and the second value the view of an observer at level L. In the syntax, we interpret a constant v as the faceted value $\langle v : v\rangle$. Faceted memories, ranged over $\hat{\mu}$, are mappings from variables to faceted values. We use the notation $\hat{\mu}(x)_i$ ($i \in \{1, 2\}$) for the first or second projection of a faceted value stored in x. As in SME, the special value $\bot$ represents the idea that no value can be observed. MF was previously proven TINI [7, Thm. 2] and we prove that it satisfies TANI.

Theorem 4. *MF is TANI.*

6 Precision, permissiveness and transparency

A number of works on dynamic information flow monitors try to analyse precision of monitors. Intuitively, precision describes how often a monitor blocks (or modifies) secure programs. Different approaches have been taken to compare precision of monitors, using definitions such as "precision", "permissiveness" and "transparency". We propose a rigorous comparison of these definitions.

In the field of runtime monitoring, a monitor should provide two guarantees while enforcing a security property: soundness and transparency. *Transparency* [8] means that whenever an execution satisfies a property in question, the monitor should output it without modifications[3].

Precision (versus well typed programs) Le Guernic et al. [23] were among the first to start the discussion on transparency for information flow monitors. The authors have proved that their hybrid monitor accepts all the executions of a program that is well typed under a flow-insensitive type system similar to the one of Volpano et al. [33]. Le Guernic [21] names this result as *partial transparency*. Russo and Sabelfeld [27] prove a similar result: they show that a hybrid monitor accepts all the executions of a program that is well typed under the flow-sensitive type system of Hunt and Sands [19].

Precision (versus secure programs) Devriese and Piessens [13] propose a stronger notion, called *precision*, that requires a monitor to accept all the executions of all secure programs. Notice that this definition is stronger because not only the monitor should recognise the executions of well typed programs, but also of secure programs that are not well typed. Devriese and Piessens have proven that such precision guarantee holds for SME versus TSNI programs.

Transparency (versus secure executions) As a follow-up, Zanarini et al. [34] have proven that another monitor based on SME satisfies *transparency for TSNI*. This monitor accepts all the TSNI executions of a program, even if the program itself is insecure.

Permissiveness (versus executions accepted by other monitors) In his PhD thesis, Le Guernic [21] compares his hybrid monitor with another hybrid monitor that performs a more precise static analysis, and proves an *improved precision* theorem stating that whenever the first hybrid monitor accepts an execution, the second monitor accepts it as well. Following this result, Besson et al. [10] investigate other hybrid monitors and prove relative precision in the style of Le Guernic, and Austin and Flanagan [6, 7] use the same definition to compare their dynamic monitors. Hedin et al. [17] name the same notion by *permissiveness* and compare the sets of accepted executions: one monitor is more permissive than another one if its set of accepted executions contains a set of accepted executions of the other monitor.

To compare precision of different information flow monitors, we propose to distinguish two notions of transparency. *True transparency* defines the secure

[3] Bauer et al. [8] actually provide a more subtle definition, saying a monitor should output a semantically equivalent trace.

executions accepted by a monitor, and *false transparency* defines the insecure executions accepted by a monitor.

True Transparency We define a notion of *true transparency* for TINI. Intuitively, a monitor is true transparent if it accepts all the TINI executions of a program.

Definition 8 (True Transparency). *Monitor M is* true transparent *if for any program P, and any memories μ, μ' and output v, the following holds:*

$$TINI(P, \mu_{\mathtt{L}}) \wedge (P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu')$$

There is a well-known result that a truncation automata cannot recognise more than computable safety properties [16, 30]. Since noninterference can be reduced to a safety property that is not computable [31], and NSU and PU can be modeled by truncation automata, it follows that they are not true transparent. We show that the monitors of this paper, that cannot be modeled by truncation automata, are not true transparent for TINI neither.

Example 10 (HM is not true transparent). Consider Program 5: it always terminates with `l=0` and hence it is secure. Any execution of this program will be modified by HM because `l` will be marked as high.

Example 11 (MF is not true transparent). Consider again TINI Program 5. The MF semantics will not behave as the original program semantics upon an execution starting in [`h=1, l=1`]. The sensitive upgrade of the test will assign faceted value [`l=`$\langle 0 : 1\rangle$] to variable `l` and the output will produce the low facet of `l` which is 1, while the original program would produce an output 0. Hence, this is a counter example for true transparency of MF.

Example 12 (SME is not true transparent for TINI). Since SME enforces TSNI, it eliminates all the termination channels, therefore even if the original program has TINI executions, SME might modify them to achieve TSNI.

Consider TINI Program 4 and an execution starting in [`h=0,l=1`]. SME (with default value `h=1`) will diverge because it's "low" execution will diverge upon `h=1`. Therefore, SME is not true transparent for TINI.

Even though none of the considered monitors are true transparent for TINI, this notion allows us to define a relative true transparency to better compare the behaviours of information flow monitors when they deal with secure executions.

Given a program P and a monitor M, we define a set of initial memories that lead to secure terminating executions of program P, and a monitor M does not modify these executions:

$$\mathcal{T}(M, P) = \{\mu \mid TINI(P, \mu_{\mathtt{L}}) \wedge \exists \mu', v.\ (P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu')\}$$

Definition 9 (Relative True Transparency). *Monitor A is* more true transparent *than monitor B, written $A \sqsupseteq_{\mathcal{T}} B$, if for any program P, the following holds: $\mathcal{T}(A, P) \supseteq \mathcal{T}(B, P)$.*

Austin and Flanagan [5, 6] have proven that MF is more true transparent than PU and PU is more true transparent than NSU. We restate this result in our notations and provide a set of counterexamples showing that for no other couple of analysed monitors relative true transparency holds.

Theorem 5. *$MF \supseteq_{\mathcal{T}} PU \supseteq_{\mathcal{T}} NSU$.*

Example 13 ($NSU \not\supseteq_{\mathcal{T}} PU, NSU \not\supseteq_{\mathcal{T}} HM$). Consider TINI Program 3: an execution in initial memory with [h=0] is accepted by PU and HM because the security level of l becomes low just before the output, and it is blocked by NSU due to sensitive upgrade.

Example 14 ($NSU \not\supseteq_{\mathcal{T}} SME, NSU \not\supseteq_{\mathcal{T}} MF, PU \not\supseteq_{\mathcal{T}} HM$, $PU \not\supseteq_{\mathcal{T}} SME$ and $PU \not\supseteq_{\mathcal{T}} MF$). Program 7 is TINI since l' does not depend on h. With initial memory [h=0, l=1], HM, SME (with default value chosen as 0) and MF terminate with the same output as normal execution. However, NSU will diverge due to sensitive upgrade and PU will diverge because of the branching over a partially-leaked variable l.

Program 6

```
1 if h = 0 then l = 1;
2 if l = 1 then l = 0;
3 output l';
```

Example 15 ($HM \not\supseteq_{\mathcal{T}} NSU, HM \not\supseteq_{\mathcal{T}} PU, HM \not\supseteq_{\mathcal{T}} SME, HM \not\supseteq_{\mathcal{T}} MF$). Consider Program 1 and its secure execution starting in [h=1, l=1]. NSU, PU, SME (the default value of SME does not matter in this case) and MF terminate with the same output as original program execution, producing l=1. However, HM modifies it because the security level of l is raised by the static analysis of the non-executed branch.

Example 16 ($SME \not\supseteq_{\mathcal{T}} NSU, SME \not\supseteq_{\mathcal{T}} PU, SME \not\supseteq_{\mathcal{T}} HM, SME \not\supseteq_{\mathcal{T}} MF$). All the terminating executions of TINI Program 4 are accepted by NSU, PU, HM and MF, while an execution starting in [h=0, l=1] with default value for SME set to h=1 doesn't terminate in SME semantics.

Example 17 ($MF \not\supseteq_{\mathcal{T}} HM$). Program 8 is TINI for any execution. HM with [h=1,l=0,l'=0] terminates with the original output because the output variable [l'] is low. However, MF with [h=1,l=0,l'=0] doesn't terminate.

Program 7

```
1 if h=0 then l=0 else l=1;
2 if l=0 then
3   while true do skip;
4 else
5   l=0
6 output l'
```

Example 18 ($MF \not\supseteq_{\mathcal{T}} SME$). Program 5 is TINI for any execution. With [h=0, l=1] it terminates in the program semantics and SME semantics (with any default value) producing l=0. However, the MF semantics produces l=1.

Precision We have discovered that certain monitors (e.g., HM and NSU) are incomparable with respect to true transparency. To compare them, we propose a more coarse-grained definition that describes the monitors' behaviour on secure programs.

Definition 10 (Precision). *Monitor M is* precise *if for any program P, the following holds:*

$$TINI(P) \wedge \forall \mu.(\exists \mu', v.(P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu'))$$

This definition requires that all the executions of secure programs are accepted by the monitor. NSU, PU, HM and MF are not precise since they are not true transparent. SME is precise for TSNI, and this result was proven by Devriese and Piessens [13], however SME it not precise for TINI (see Example 15).

To compare monitors' behaviour on secure programs, we define a set of a TINI programs P, where a monitor accepts all the executions of P:

$$\mathcal{P}(M) = \{P \mid TINI(P) \wedge \forall \mu.(\exists \mu', v.(P, \mu) \Downarrow (v, \mu') \Rightarrow (P, \mu) \Downarrow_M (v, \mu'))\}$$

Definition 11 (Relative Precision). *Monitor A is more precise than monitor B, written $A \supseteq_{\mathcal{P}} B$, if $\mathcal{P}(A) \supseteq \mathcal{P}(B)$.*

We have found out that no couple of the five monitors are in relative precision relation. Below we present the counterexamples that demonstrate our findings.

Example 19 ($HM \not\supseteq_{\mathcal{P}} SME$). Consider TINI Program 5. All the executions of this program are accepted by SME. However, HM modifies the program output to default because the security level of `l` is upgraded to H by the static analysis of the non-executed branch.

Example 20 ($HM \not\supseteq_{\mathcal{P}} NSU$, $HM \not\supseteq_{\mathcal{P}} PU$). Consider the following program:

Program 8

```
1 l = 0;
2 if h = 0 then skip
3 else
4   while true do l = 1;
5 output l
```

This TINI program terminates only when [`h=0`]. This execution is accepted by NSU and PU, but the program output is modified by HM since HM analyses the non-executed branch and upgrades the level of `l` to H.

Example 21 ($HM \not\supseteq_{\mathcal{P}} MF$). Consider TINI Program 11. MF accepts all of its executions, while HM modifies the program output to default because the security level of `l` is raised to high.

Program 9

```
1 l = 0;
2 if h = 0 then l = 0 else skip;
3 output l
```

The rest of relative precision counterexamples demonstrated in Table 2 of Section 9 are derived from the corresponding counterexamples for relative true transparency.

Since relative precision does not hold for any couple of monitors, we propose a stronger definition of relative precision for TSNI programs. We first define a set of a TSNI programs P, where a monitor accepts all the executions of P:

$$\mathcal{P}^*(M) = \{P \mid TSNI(P) \wedge \forall \mu.(\exists \mu', v.(P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu'))\}$$

Definition 12 (Relative TSNI precision). *A monitor A is more TSNI precise than a monitor B, written $A \supseteq^*_{\mathcal{P}} B$, if $\mathcal{P}^*(A) \supseteq \mathcal{P}^*(B)$.*

Theorem 6. *For all programs without dead code, $HM \supseteq^*_{\mathcal{P}} NSU, HM \supseteq^*_{\mathcal{P}} PU$.*

Notice that SME was proven to be precise for TSNI programs (see [13, Thm. 2]), therefore SME is more TSNI precise than any other monitor. We demonstrate this in Table 2 of Section 9.

False Transparency To compare monitors with respect to the amount of insecure executions they accept, we propose the notion of *false transparency*. Notice that false transparency violates soundness.

Definition 13 (False Transparency). *Monitor M is* false transparent *if for any program P, for all executions starting in a memory μ and finishing in memory μ' with value v, the following holds:*

$$\neg TINI(P,\mu) \wedge (P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu').$$

Given a program P and a monitor M, we define a set of initial memories, where a program P terminates, and a monitor M is false transparent for P:

$$\mathcal{F}(M,P) = \{\mu \mid \neg TINI(P,\mu_{\mathtt{L}}) \wedge \exists \mu', v.(P,\mu) \Downarrow (v,\mu') \Rightarrow (P,\mu) \Downarrow_M (v,\mu')\}$$

Definition 14 (Relative False Transparency). *Monitor A is* more false transparent *than monitor B, denoted $A \supseteq_{\mathcal{F}} B$, if for any program P, the following holds: $\mathcal{F}(A,P) \supseteq \mathcal{F}(B,P)$.*

Theorem 7. *The following statements hold: $NSU \supseteq_{\mathcal{F}} HM$, $PU \supseteq_{\mathcal{F}} NSU$, $PU \supseteq_{\mathcal{F}} HM$, $SME \supseteq_{\mathcal{F}} HM$, $MF \supseteq_{\mathcal{F}} NSU$, $MF \supseteq_{\mathcal{F}} PU$ and $MF \supseteq_{\mathcal{F}} HM$.*

Example 22 ($NSU \not\supseteq_{\mathcal{F}} PU$). Execution of Program 12 in the initial memory μ= [`h=0, l=0, l'=0`] is interfering since it produces an output `l=0`, while an execution in the low-equal initial memory where [`h=1`] produces `l=1`. An execution started in μ is accepted by PU but blocked by NSU.

```
1 if h = 0 then l' = 1 else l = 1;
2 output l
```
Program 10

Example 23 (NSU $\not\supseteq_{\mathcal{F}}$ SME, PU $\not\supseteq_{\mathcal{F}}$ SME). Execution of Program 13 starting in memory [h=0, l=0] is not TINI and it is accepted by SME (with default value h=0). However, it is rejected by NSU because of sensitive upgrade and by PU because on the branching over a partially-leaked variable l.

```
1 if h = 0 then l = 0 else l = 1;
2 if l = 0 then l' = 0 else l' = 1;
3 output l'
```
Program 11

Example 24 (NSU $\not\supseteq_{\mathcal{F}}$ MF). The following program always terminates in the normal semantics coping the value of h into l. Hence all of its executions are insecure. Every execution leads to a sensitive upgrade and NSU will diverge with any initial memory. However, in the MF semantics the program will terminate with l=0 if started with memory [h=0,l=0] since the sensitive upgrade of the true branch will assign faceted value [l=$\langle 0 : 0\rangle$] to variable l. Hence, this is a counter example for NSU being more false transparent than MF.

```
1 if h=0 then l=0 else l=1;
2 output l
```
Program 12

Example 25 (PU $\not\supseteq_{\mathcal{F}}$ MF). Program 13 is not TINI for all executions. However MF with [h=1,l=1,l'=1] terminates in the same memory as normal execution, while PU will diverge because l is marked as a partial leak.

Example 26 (HM $\not\supseteq_{\mathcal{F}}$ NSU, HM $\not\supseteq_{\mathcal{F}}$ PU, HM $\not\supseteq_{\mathcal{F}}$ SME, HM $\not\supseteq_{\mathcal{F}}$ MF). Consider Program 1 and an execution starting in memory [h=1, l=0]. This execution is not secure and it is rejected by HM, however NSU, PU and MF accept it. SME also accepts this execution in case the default value for h is 1.

Example 27 (SME $\not\supseteq_{\mathcal{F}}$ NSU, SME $\not\supseteq_{\mathcal{F}}$ PU). Execution of Program 15 starting in memory [h=0, l=0] is interfering and it is accepted by both NSU and PU, producing an output l=0. However, SME (with default value chosen as 1) modifies this execution and produces l=1.

```
1 if l = 0 then
2   if h = 1 then l = 1 else skip
3 else
4   if h = 0 then l = 0 else skip
5 output l
```
Program 13

Example 28 (SME $\not\supseteq_{\mathcal{F}}$ MF and MF $\not\supseteq_{\mathcal{F}}$ SME). Program 16 is not TINI if possible values of h are 0, 1, and 2. MF with [h=1,l=1] terminates in the same memory than normal execution but SME (with default value 0) always diverges.

```
1 if h = 0 then                                          Program 14
2    while true do skip;
3 else
4   if h=1 then l=1 else l=2;
5 output l;
```

On the other hand, with initial memory [`h=1, l=0`], SME (using default value 1) terminates in the same memory as the normal execution, producing `l=1` but MF produces a different output `l=0`.

7 Related Work

In this section, we discuss the state of the art for taxonomies of information flow monitors with respect to soundness or transparency.

For soundness, no work explicitly tries to classify information flow monitors. However, it is folklore that TSNI, first proposed in [32], is a strong form of non-interference that implies TINI. Since most well-known information flow monitors are proven sound only for TINI [5–7, 15, 35], it is easy, from the soundness perspective, to distinguish SME from other monitors because SME is proven sound for TSNI [13]. However, to the best of our knowledge, no work tries to refine soundness in order to obtain a more fine grain classification of monitors as we achieve with the introduction of TANI.

For transparency, Devriese and Piessens [13] prove that SME is precise for TSNI and Zanarini et al. [34] notice that the result could be made more general by proving that SME is true transparent for TSNI, which makes of SME an effective enforcement [24] for TSNI. In this work, we first compare transparency for TINI: none of the monitors that we have studied is true transparent for TINI. Hedin et al. [17] compare hybrid (HM) and purely dynamic monitors (NSU and PU), and conclude that for these monitors permissiveness is incomparable. By factorizing the notion of permissiveness, we can compare HM and NSU: HM is more precise for TSNI than NSU and PU, and NSU and PU are more false transparent than HM. Using the same definition of permissiveness, Austin and Flanagan [6,7] prove that PU is more permissive than NSU and that MF is more permissive than PU. Looking at this result and the definition of MF, our intuition was that MF could accept exactly the same false transparent executions as NSU and PU. However, we discovered that not only MF is more true transparent than NSU and PU (this is an implication of Austin and Flanagan results) but also MF is strictly more false transparent than NSU and PU. Bichhawat et al. [11] propose two non-trivial generalizations of PU, called puP and puA, to arbitrary lattices and show that puP and puA are incomparable w.r.t. permissiveness. It remains an open question if puP and puA can be made comparable by discriminating true or false transparency, as defined in our work.

	NSU	PU	HM	SME	MF
NSU		$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
PU	$\supseteq_{\mathcal{T}}$ $\supseteq_{\mathcal{F}}$		$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
HM	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$		$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
SME	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	$\supseteq^*_{\mathcal{P}} \supseteq_{\mathcal{F}}$		$\supseteq^*_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$
MF	$\supseteq_{\mathcal{T}} \supseteq_{\mathcal{F}}$	$\supseteq_{\mathcal{T}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \supseteq_{\mathcal{F}}$	$\not\supseteq_{\mathcal{P}} \not\supseteq_{\mathcal{F}}$	

$\supseteq_{\mathcal{T}}$ more true TINI transparent than
$\supseteq_{\mathcal{P}}$ more TINI precise than ($\not\supseteq_{\mathcal{P}} \implies \not\supseteq_{\mathcal{T}}$)
$\supseteq^*_{\mathcal{P}}$ more TSNI precise than
$\supseteq_{\mathcal{F}}$ more false TINI transparent than
Monitor is TANI
Monitor is TSNI, hence TANI

Table 1: **Taxonomy of five major information flow monitors**

8 Conclusion

In this work we proposed a new soundness definition for information flow monitors, that we call *Termination-Aware Noninterference* (TANI). It determines whether a monitor adds a new termination channel to the program. We have proven that HM, SME and MF, do satisfy TANI, whereas NSU and PU introduce new termination channels, and therefore do not satisfy TANI.

We compare monitors with respect to their capability to recognise secure executions, i.e., true transparency [8]. Since it does not hold for none of the considered monitors, we weaken this notion and define *relative true transparency*, that determines "which monitor is closer to being transparent". We then propose even a more weaker notion, called *precision*, that compares monitor behaviours on secure programs, and allows us to conclude that HM is more TSNI precise than NSU and PU that previously were deemed incomparable [17]. We show that the common notion of permissiveness is composed of *relative true and false transparency* and compare all the monitors with respect to these notions in Table 2.

For simplicity, we consider a security lattice of only two elements, however we expect our results to generalise to multiple security levels. In future work, we plan to compare information flow monitors with respect to other information flow properties, such as declassification [28].

Acknowledgment

We would like to thank Ana Almeida Matos for her valuable feedback and interesting discussions that has lead us to develop the main ideas of this paper, Aslan Askarov for his input to the definition of TANI, and anonymous reviewers for feedback that helped to improve this paper. This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008.

References

1. A Taxonomy of Information Flow Monitors Technical Report. `https://team.inria.fr/indes/taxonomy`.

2. M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programmming Languages and Systems.*, 1993.
3. A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*, pages 207–221, 2007.
4. A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proceedings of the 2009 22Nd IEEE Computer Security Foundations Symposium*, CSF '09, pages 43–59. IEEE Computer Society, 2009.
5. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.
6. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.
7. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th Symposium of Principles of Programming Languages*. ACM, 2012.
8. L. Bauer, J. Ligatti, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.
9. L. Bello, D. Hedin, and A. Sabelfeld. Value sensitivity and observable abstract values for information flow control. In *Proceedings of the International Conferences on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, 2015.
10. F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *CSF'13*, pages 240–254. IEEE, 2013.
11. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Generalizing permissive-upgrade in dynamic information flow analysis. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, PLAS'14, pages 15:15–15:24. ACM, 2014.
12. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.
13. D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. of the 2010 Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.
14. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.
15. G. L. Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. Automata-based confidentiality monitoring. In *Proc. of the 11th Asian Computing Science Conference (ASIAN'06)*, volume 4435, pages 75–89. Springer-Verlag Heidelberg, 2006.
16. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.
17. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *IEEE 28th Computer Security Foundations Symposium, CSF*, 2015.
18. D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *Proc. of the 25th Computer Security Foundations Symposium*, pages 3–18. IEEE, 2012.
19. S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06*, pages 79–90, New York, NY, USA, Jan. 2006. ACM.
20. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428, 2011.
21. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University and University of Rennes 1, 2007.

22. G. Le Guernic. Precise Dynamic Verification of Confidentiality. In *Proc. of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proc.*, pages 82–96, 2008.
23. G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *Proc. of the Annual Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.
24. J. Ligatti, L. Bauer, and D. Walker. Enforcing Non-Safety Security Policies with Program Monitors. In *ESORICS 05*, 2005.
25. A. G. A. Matos, J. F. Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *Trustworthy Global Computing - 9th International Symposium, TGC*, 2014.
26. J. McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Transactions on Software Engineering*, 1996.
27. A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proc. of the 23rd Computer Security Foundations Symposium*, pages 186–199. IEEE, 2010.
28. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
29. J. F. Santos and T. Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of Javascript. In *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014*, 2014.
30. F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
31. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium*, pages 352–367, 2005.
32. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *In Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168. Society Press, 1997.
33. D. Volpano, G. Smith, and C. Irvine. A Sound Type System For Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.
34. D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *IEEE 26th Computer Security Foundations Symposium*, pages 18–32, 2013.
35. S. A. Zdancewic. *Programming languages for information security.* PhD thesis, Cornell University, 2002.

On access control, capabilities, their equivalence, and confused deputy attacks

Vineet Rajani
MPI-SWS

Deepak Garg
MPI-SWS

Tamara Rezk
INRIA

***Abstract*—Motivated by the problem of understanding the difference between practical access control and capability systems formally, we distill the essence of both in a language-based setting. We first prove that access control systems and (object) capabilities are fundamentally different. We further study capabilities as an enforcement mechanism for confused deputy attacks (CDAs), since CDAs may have been the primary motivation for the invention of capabilities. To do this, we develop the first formal characterization of CDA-freedom in a language-based setting and describe its relation to standard information flow integrity. We show that, perhaps suprisingly, capabilities cannot prevent all CDAs. Next, we stipulate restrictions on programs under which capabilities ensure CDA-freedom and prove that the restrictions are sufficient. To relax those restrictions, we examine provenance semantics as sound CDA-freedom enforcement mechanisms.**

***Keywords*-Access control; Capability; Confused deputy problem; Provenance tracking; Information flow integrity**

I. Introduction

Access control and capabilities are the most popular mechanisms for implementing authorization decisions in systems and languages. Roughly, whereas in access control a token (authentication credential) that represents the current principal is associated to a list of authorization rights, in capabilities the authorization right is the token (capability). Although, both mechanisms have been widely studied and deployed at various levels of abstraction (see e.g. [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]), there seems to be no clear consensus on the fundamental difference in their modus operandi. Our broad goal is to formalize fundamental properties that distinguish access control and capability systems from each other. Our motivation is partly pedagogic, and partly to discover the limits of what can and cannot be enforced using access control systems and capabilities. Against the backdrop of this broad goal, we make three contributions in this paper.

First, we reflect upon the question of whether or not access control and capability systems are equivalent in a formal language-based setting. Specifically, we are interested in this question in the context of capability systems that possess what Miller *et al.* [11] call Property A or "no designation without authority": If a principal acquires a capability, it also acquires the authority to use it.[1] Property A is very interesting because it is fundamental to many types of capability systems including all object capability systems, which are used to obtain isolation and security in large code bases [5], [3], [2]. In an object capability system, a capability is a reference to an ordinary language object or a memory location and there are no checks on using references, so the possessor of a capability can always read or write it.

To formalize access control and capabilities, we design a small core calculus with regions (principals) and memory references (objects/capabilities), and equip it with two different semantics—an access control semantics and a capability semantics with property A. We then show that access control and capabilities with property A are fundamentally different: The access control semantics is strictly more permissive than the capability semantics. (This formally justifies an earlier *informal* argument to the same effect by Miller *et al.* [11].)

Our access control semantics is the expected one. It intervenes on every use (read/write) of a reference and checks that the use is compliant with a given access policy. The capability semantics is less obvious, so we briefly describe its design here. By contraposition of the definition of property A, it follows that to limit authority in a capability system with property A, we must limit the designation of capabilities. In general, principals may acquire capabilities either by generating them (e.g., by guessing them or computing them from existing values) or by receiving them from other principals. Hence, to get security, i.e., to control authority, a capability system must ensure that:

1) A principal cannot generate a capability he is not authorized to use, and
2) A principal cannot receive (from another principal) a capability he is not authorized to use.

In practical systems with property A, (1) is ensured by using abstract, unforgeable tokens for capabilities. (1) is closely related to, and usually implied by, a property called "capability safety" [12]. Capability safety requires that a principal may acquire a capability only if the capability, as an object, is reachable in the initial heap starting from the principal's initial set of capabilities. So capability safety immediately implies (1). However, (1) in itself is not enough to get security; we also need (2).

How do we enforce (2)? One option is to *define* authority as the set of all capabilities that are obtained during program execution. Then, (2) holds trivially. However, this implicit definition of authority allows bugs in the code to leak

[1]The word principal should be interpreted broadly here. It may refer to a section of code, a function, or a user.

capabilities that the programmer never intended, without breaking allowed authority in a formal sense.

An alternative to this implicit specification of authority is to specify, via an explicit access policy, what references (capabilities) each principal is authorized to access and to ensure that each principal can only obtain references that it can legitimately use. This approach is taken in some practical implementations of object capabilities, e.g., Firefox's security membrane [5], which intercepts all transmissions from one domain to another and restricts objects (capabilities) in accordance with relevant policies (Firefox's policies are drawn from web standards like the same-origin policy). Our capability semantics models a simplified version of this general pattern. It intervenes on every transmitted and computed value and checks that if the value is a reference (capability), then the executing principal is authorized to use the capability according to the access policy. This intervention is computationally expensive, since every value must be checked. Nonetheless, our capability semantics is an ideal model of how security is enforced in the presence of property A and an explicit access policy. Interestingly, our semantics enforces not just (2) but also (1), so there is no need to make capabilities unforgeable. Hence, our approach is compatible with a language that includes pointer arithmetic.[2]

As a second contribution, we formally examine confused deputy attacks (CDAs) [16], which may have been the primary reason for the invention of capabilities. A CDA is a privilege escalation attack where a deputy (a trusted system component) can act on both its own authority and on an adversary's authority. In a CDA, the deputy is confused because it thinks that it is acting on its own authority when in reality it is acting on an attacker's authority. Cross-site request forgery [17], FTP bounce attacks [18] and clickjacking [19] are all prevalent examples of CDAs. It is widely known that access control alone is insufficient to prevent CDAs and it is known that the use of capabilities prevents (at least some) CDAs.

We make two fundamental contributions in the context of CDAs. First, we provide what we believe to be the first formal definition of when a program is free from CDAs. Our definition is extensional and is inspired by information flow integrity [20], [21], [22], but we show that CDA-freedom is strictly weaker than information flow integrity. Second, we use this definition and our capability semantics to formally establish that, perhaps surprisingly, capability semantics is not enough to ensure CDA-freedom. While capabilities prevent many CDAs that are based on explicit designation of authority from the adversary to the deputy, there are other CDAs based on implicit designation that capability semantics cannot prevent. We also stipulate restrictions on programs under which capability semantics prevent all CDAs. However, these restrictions are very strong and render the language useless for almost all practical purposes.

As our final contribution, we investigate alternate approaches for CDA prevention with fewer restrictions. Our approaches rely on provenance tracking (taint tracking). First, we formally show that merely tracking *explicit* provenance (i.e., without taking into account influence due to control flow) suffices to guarantee CDA-freedom with fewer restrictions than capabilities require. In order to remove even these restrictions, we further develop a full-fledged provenance analysis and prove CDA-freedom. We compare the three methods of preventing CDAs (capabilities, explicit provenance tracking, full provenance tracking) in terms of permissiveness through examples.

To summarize, the key contributions of this work are:

- We formally examine the fundamental difference between access control and capabilities in a language-based setting.
- We give the first extensional characterization of CDA-freedom and its relation to information-flow integrity.
- We show that capability semantics are not enough for CDA-freedom in the general case. We then examine conditions under which this implication holds.
- We present provenance tracking as an alternate approach for preventing CDAs with fewer assumptions and prove its soundness.

Proofs and many other technical details are available in a technical report available from the authors' homepages. The technical report also considers an extension of our calculus with computable references (pointer arithmetic).

II. Access control vs Capabilities

Our technical development is based on a region calculus, a simple, formal imperative language with notions of principals (which own a subset of references) and regions (which specify a write integrity policy that we wish to enforce). This simple calculus suffices to convey our key ideas, without syntactic clutter. The syntax of our calculus is shown in Figure 1. We assume two countable sets, Loc of mutable references and $Prin$ of principals. Elements of Loc are written r and elements of $Prin$ are written $\mathbb{P}$. Our calculus has five syntactic categories — values (v), expressions (e), commands (c), regions (ρ), and top-level programs or, simply, programs (P).

Values consist of integers (n), booleans (tt, ff) and pointers or mutable references ${}^{\mathbb{R}}r$ and ${}^{\mathbb{W}}r$. References ${}^{\mathbb{R}}r$ and ${}^{\mathbb{W}}r$ represent the read and write capabilities for the reference r. Capability ${}^{\mathbb{R}}r$ can only be used to read r, whereas capability ${}^{\mathbb{W}}r$ can only be used to write r. Separating these capabilities allows us to make a fine distinction between security checks

[2]Going beyond policy enforcement is the question of whether the policy attains higher-level security goals such as ensuring specific invariants on protected state or limiting observable effects to a desirable set. Such higher-level goals can be attained using static verification, as in [13], [14], [15], but these goals are beyond the scope of this paper.

$v ::=$		Value
	n	Integer
	tt	True
	ff	False
	$^{\mathbb{R}}r$	Read view of a location
	$^{\mathbb{W}}r$	Write view of a location
$e ::=$		Expression
	v	Value
	$!e$	Dereference
$c ::=$		Command
	if e then c else c	Conditional
	while e do c	Loop
	$e := e$	Assignment
	$c; c$	Sequential composition
	skip	Skip
$\rho ::=$		Region
	$\mathbb{P}$	Principal
	$\overline{\mathbb{P}}$	Endorsed principal
$P ::=$		Program
	$\rho\{c\}$	Region command
	$P \circ P$	Region composition

Figure 1. Region calculus

on reads and writes. Expressions e are computations that cannot update references. They include values and reference reading ($!e$). Commands c are standard conditionals, while loops, assignments, sequencing ($c_1; c_2$) and skip.

A region ρ is either a principal $\mathbb{P}$ or an *endorsed principal*, $\overline{\mathbb{P}}$. In both cases, $\mathbb{P}$ represents a ceiling (maximum) authority for executing code. However, in the case of an endorsed region, the principal expresses the explicit willingness to act on another principal's behalf. In our definition of CDA-freedom (Definition 3), we take this intention into account to explicitly exclude endorsed regions as sources of CDAs. For now, readers may ignore endorsed principals $\overline{\mathbb{P}}$, treating them exactly like normal principals $\mathbb{P}$.

A program P is a sequence of commands, executed in possibly different regions. A program has the form $\rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ and means that first command c_1 runs in the region ρ_1, then command c_2 runs in region ρ_2 and so on. When a command runs in a region, the command is subject to the ceiling authority of the region.

Regions and write integrity: The primary property we wish to enforce is *write integrity*. To specify this property, we assume that each reference is *owned* by a principal. This is formalized by an ownership map $\mathbb{O}$, that maps a reference to the principal that owns the reference. Formally, $\mathbb{O} : \mathit{Loc} \to \mathit{Prin}$. Principals are assumed to be organized in a lattice $\mathbb{L}$ whose order is written $\geq_{\mathbb{L}}$. This lattice is a technical representation of a write integrity policy: Code executing in region $\mathbb{P}$ or $\overline{\mathbb{P}}$ can write to reference r, i.e., it can wield the capability $^{\mathbb{W}}r$ only if $\mathbb{P} \geq_{\mathbb{L}} \mathbb{O}(r)$.[3] For convenience, we extend the order $\geq_{\mathbb{L}}$ to regions: $\rho \geq_{\mathbb{L}} \rho'$ when $\rho \in \{\mathbb{P}, \overline{\mathbb{P}}\}$, $\rho' \in \{\mathbb{P}', \overline{\mathbb{P}'}\}$ and $\mathbb{P} \geq_{\mathbb{L}} \mathbb{P}'$.

It should be clear that the lattice $\mathbb{L}$ and the ownership map $\mathbb{O}$ together define an access/authorization policy for write references. We enforce this policy using either access control or capability-based checks, as explained below. Authorization for read references is also important in practice, but is not the focus of this paper. In fact, we allow any command to dereference any read capability the command possesses.

Access control and capability semantics: Since our first goal is to investigate the differences between access control systems and capability systems, we equip our calculus with two different runtime semantics — an access control semantics (ACs) and a capability semantics (Cs). Both enforce write integrity, but in different ways. Whereas ACs checks that the ceiling authority is sufficient when a reference is written (through the policy described above), Cs prevents a region from getting a write capability which it cannot wield in the first place. Technically, Cs must intercept every constructed value and check that, if the value is a write capability, then the executing region is higher (in $\mathbb{L}$) than the region that owns the reference accessible through the capability. While this is cumbersome, in our opinion, this is the formal essence of Miller *et al.*'s Property A of capability systems [11]: possession of a reference (capability) implies the authority to use it. By inference, if a region must not write a reference according to the policy, it must not ever possess the reference. We now formalize the two semantics ACs and Cs.

As usual, a heap H is a map from *Loc* to values and determines the value stored in each reference. Here, values are integers and booleans. Both ACs and Cs are defined by three evaluation judgments: $\langle H, e\rangle \overset{\rho}{\Downarrow_X} v$ for expres-

[3]The lattice specifies only an upper bound or ceiling on the set of references the code in a region can write. However, the code must also explicitly present a write capability to a reference in order to update the reference. Miller *et al.* call this requirement to explicitly present capabilities "property D" or "no ambient authority", and argue that it is a pre-requisite for ruling out confused-deputy attacks [11].

sions, $\langle H, c\rangle \xrightarrow{\rho}_X \langle H', c'\rangle$ for commands and $\langle H, P\rangle \rightarrow_X \langle H', P'\rangle$ for programs. Here X may be A (for the semantics ACs) or C (for the semantics Cs). In the rules for expressions and commands, ρ denotes the region or the ceiling authority in which evaluation happens. Figure 2 shows all the semantic rules. When a rule applies to both ACs and Cs, we use the generic index X in both the name of the rule and on the reduction arrow.

The judgment for expression evaluation $\langle H, e\rangle \overset{\rho}{\Downarrow}_X v$ means that when the heap is H, expression e evaluates to value v. The ACs rules (top of Figure 2, left panel) are straightforward. For dereferencing, we need the read capability ${}^{\mathbb{R}}r$ (rule A-Deref). The Cs rules (right panel) are exactly like the access control rules, but they all make an additional check: If the value being returned is a write capability, then the executing region ρ must be above the owner of the capability's reference. This ensures that the executing region never gets a write capability whose owner's authority is not below the executing region's authority.

The judgment for command evaluation $\langle H, c\rangle \xrightarrow{\rho}_X \langle H', c'\rangle$ means that c reduces (one-step) to c' transforming the heap from H to H'. The rules for this judgment are mostly standard. The only interesting point is that in the ACs rule for reference update (rule A-Assign), we check that the owner $\mathbb{O}(r)$ of the updated reference r is below the executing region ρ. A corresponding check is not needed in Cs (rule C-Assign) because, there, the assigned reference r cannot even be computed unless $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$. Technically, the rules for $\overset{\rho}{\Downarrow}_C$ ensure that the check $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$ is made in the derivation of the first premise of C-Assign.

Access control more permissive than capabilities: We now prove that ACs is strictly more permissive than Cs, thus accomplishing our first goal. The extra permissiveness of ACs over Cs should be expected because Cs prevents code from obtaining write capabilities that it cannot use whereas ACs allows the region to obtain such capabilities, but prevents it from writing them (later). The following theorem formalizes this intuition. It says that if a reduction is allowed in Cs, then the reduction must also be allowed in ACs.

Theorem 1 (ACs more permissive than Cs)**.** $\langle H, P\rangle \rightarrow_C \langle H', P'\rangle$ implies $\langle H, P\rangle \rightarrow_A \langle H', P'\rangle$.

Proof: By induction on the given derivation of $\langle H, P\rangle \rightarrow_C \langle H', P'\rangle$. ■

The converse of this theorem is false. For example, consider the program $\rho\{{}^{\mathbb{W}}r_1 := {}^{\mathbb{W}}r_2\}$ that runs with ceiling authority ρ and stores the reference r_2 in the reference r_1. Assume that $\rho \geq_{\mathbb{L}} \mathbb{O}(r_1)$ but $\rho \not\geq_{\mathbb{L}} \mathbb{O}(r_2)$, i.e., ρ can write to r_1 but not to r_2. Then, ACs allows the program to execute to completion. On the other hand, Cs blocks this program because ρ will not be allowed to compute the capability ${}^{\mathbb{W}}r_2$, which it cannot wield. Technically, the second premise of rule C-Assign will not hold for this example. Hence, the access control semantics, ACs, is strictly more permissive than the capability semantics, Cs.

Write integrity: Despite their differences, both ACs and Cs provide write integrity in the sense that neither allows a region to write a reference that it is not authorized to write. We formalize and prove this result below.

Theorem 2 (ACs and Cs provide write integrity)**.** If $\langle H, \rho\{c\}\rangle \rightarrow_X{}^* \langle H', \_\rangle$ and $H(r) \neq H'(r)$, then $\rho \geq_{\mathbb{L}} \mathbb{O}(r)$.

Proof: For $X = \text{A}$, the result is proved by induction on the reduction sequence $\rightarrow_A^*$ and, at each step, by induction on the derivation of the given reduction. For $X = \text{C}$, the result follows from the result for $X = \text{A}$ and Theorem 1. ■

Capability Safety: Capability safety is a widely discussed, but seldom formalized foundational property of capability-based languages. Roughly, it says that capabilities to access resources can only be obtained through legal delegation mechanisms. We have proved capability safety for Cs by instantiating a general definition of the property due to Maffeis *et al.* [12]. Since capability safety is largely orthogonal to our goals, we relegate its details to our technical report.

Theorem 3 (Capability Safety)**.** The semantics Cs is capability safe.

III. Confused Deputy Attacks and Capabilities

A confused deputy attack [16], CDA for short, is a privilege escalation attack where the adversary who doesn't have direct access to some sensitive resource, indirectly writes the resource by confusing a deputy, a principal who can access the resource. The confused compiler service is a folklore example of a CDA. In this example, a privileged compiler service is tricked by its unprivileged caller into overwriting a sensitive billing file which the caller cannot update, but the compiler can. The compiler service takes as inputs the names of the source file to be compiled and an output file. It compiles the source file, writes the compiled binary to the output file and, importantly, on the side, writes a billing file that records how much the caller must pay for using the compiler. The caller tricks the compiler by passing to it the name of the billing file in place of the output file, which causes the compiler to overwrite the billing file with a binary, thus destroying the billing file's integrity. (Of course, pay-per-use compilers are rare today, but the example is very illustrative and CDAs remain as relevant as ever.)

CDAs are interesting from our perspective because they distinguish access control semantics, which offer no defense against CDAs from capability semantics, which can prevent at least some CDAs. For instance, Cs would prevent the CDA in the compiler service example above, but ACs would not (see later examples for a proof). It has been claimed in the past that in the presence of Miller *et al.*'s property A, Cs

Expressions:

Access control

$$\text{A-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_A v}$$

$$\text{A-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_A {}^{\mathbb{R}}r \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_A v}$$

Capability

$$\text{C-Val}\ \frac{v = {}^{\mathbb{W}}r \implies \rho \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, v\rangle \overset{\rho}{\Downarrow}_C v}$$

$$\text{C-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_C {}^{\mathbb{R}}r \qquad v = H(r) \qquad v = {}^{\mathbb{W}}r' \implies \rho \geq_{\mathbb{L}} \mathbb{O}(r')}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_C v}$$

Commands:

$$\text{X-if}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = tt}{\langle H, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_1\rangle} \qquad \text{X-else}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = ff}{\langle H, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_2\rangle}$$

$$\text{X-while 1}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = tt}{\langle H, \text{while } e \text{ do } c\rangle \overset{\rho}{\rightarrow}_X \langle H, c; \text{while } e \text{ do } c\rangle} \qquad \text{X-while 2}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_X v \qquad v = ff}{\langle H, \text{while } e \text{ do } c\rangle \overset{\rho}{\rightarrow}_X \langle H, \text{skip}\rangle}$$

Access control

$$\text{A-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_A {}^{\mathbb{W}}r \qquad \rho_r = \mathbb{O}(r) \qquad \rho \geq_{\mathbb{L}} \rho_r \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_A v}{\langle H, e_1 := e_2\rangle \overset{\rho}{\rightarrow}_A \langle H[r \mapsto v], \text{skip}\rangle}$$

Capability

$$\text{C-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_C {}^{\mathbb{W}}r \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_C v}{\langle H, e_1 := e_2\rangle \overset{\rho}{\rightarrow}_C \langle H[r \mapsto v], \text{skip}\rangle}$$

$$\text{X-Seq 1}\ \frac{\langle H, c_1\rangle \overset{\rho}{\rightarrow}_X \langle H', c_1'\rangle}{\langle H, c_1; c_2\rangle \overset{\rho}{\rightarrow}_X \langle H', c_1'; c_2\rangle} \qquad \text{X-Seq 2}\ \frac{}{\langle H, \text{skip}; c_2\rangle \overset{\rho}{\rightarrow}_X \langle H, c_2\rangle}$$

Program:

$$\text{X-Prg 1}\ \frac{\langle H, c\rangle \overset{\rho}{\rightarrow}_X \langle H', c'\rangle}{\langle H, \rho\{c\}\rangle \rightarrow_X \langle H', \rho\{c'\}\rangle} \qquad \text{X-Comp 1}\ \frac{\langle H, P_1\rangle \rightarrow_X \langle H', P_1'\rangle}{\langle H, P_1 \circ P_2\rangle \rightarrow_X \langle H', P_1' \circ P_2\rangle}$$

$$\text{X-Comp 2}\ \frac{}{\langle H, \rho\{\text{skip}\} \circ P\rangle \rightarrow_X \langle H, P\rangle}$$

Figure 2. Access control (A) and Capability (C) semantics

prevent CDAs [16], [11], [23]. However, to the best of our knowledge, there is, thus far, no formal characterization of what it means for a system to be free from a CDA, nor a formal understanding of whether all CDAs can be prevented by Cs. In this section, we address both these issues. First, we provide a formal definition of what it means for a program to be free from CDAs (subsection III-A). Then we show that Cs cannot prevent all CDAs even in a minimalist language such as our region calculus, but they can actually prevent all CDAs under very strong restrictions (subsection III-B). This provides the first formal characterization of a language (fragment) in which capabilities can provably prevent CDAs.

A. Defining CDA-freedom

The goal of this subsection is to define what it means for a program to be CDA-free. To test whether a program is free from CDAs or not, the program must allow for interaction with an adversary. To this end, we define an *authority context* or, simply, context, written $\mathbb{E}_{\rho_A}$, which is a program with one hole, where an adversary's commands can be inserted. We write ρ_A for the adversary region that has a hole.

Definition 1 (Authority Context)**.** An authority context, $\mathbb{E}_{\rho_A}$, is a program with one hole of the form $\rho_A\{\bullet\}$. Formally, $\mathbb{E}_{\rho_A} ::= \rho_1\{c_1\} \circ \ldots \circ \rho_A\{\bullet\} \circ \ldots \circ \rho_n\{c_n\}$. We write $\mathbb{E}_{\rho_A}[c_A]$ for the program that replaces the hole $\bullet$ with the

adversary's commands c_A, i.e., the program $\rho_1\{c_1\} \circ \ldots \circ \rho_A\{c_A\} \circ \ldots \circ \rho_n\{c_n\}$.

Any program P (without a hole) can be trivially treated as an authority context $\mathbb{E}_{\rho_A} = P \circ \rho_A\{\bullet\}$. In some examples, we treat programs as authority contexts in this sense.

In a CDA, the goal of an adversary is to overwrite one or more references. We call these references the "attacker's interest set", denoted AIS. In the sequel, we assume a fixed AIS. Intuitively, a context $\mathbb{E}_{\rho_A}$ is free from a CDA if for every reference $r \in AIS$ either the attacker cannot control what value is written to r, or the attacker can write to r directly. If the first disjunct holds, then there is no attack on r whereas if the second disjunct holds, then there is no need for a confused deputy (the context $\mathbb{E}_{\rho_A}$) to modify r. In either case, there is no confused deputy attack on r. The first disjunct can be formalized by saying that no matter what adversary code we substitute into $\mathbb{E}_{\rho_A}$'s hole, the final value in r is the same. The second disjunct can be formalized by saying that there must be *some* adversary code that, when running with the ceiling authority ρ_A, can write the final value of r to r directly. Based on this, we arrive at the following preliminary definition of CDA-freedom (we revise this definition later). Here, `final` denotes a fully reduced program, of the form $\rho\{\text{skip}\}$.

Definition 2 (CDA-freedom)**.** Context $\mathbb{E}_{\rho_A}$ starting from the initial heap H and running under reduction semantics $\rightarrow_{red}$ is said to be free from CDAs, written $\text{CDAF}(\mathbb{E}_{\rho_A}, H, \rightarrow_{red})$, if for every c_A and H' such that $\langle H, \mathbb{E}_{\rho_A}[c_A]\rangle \rightarrow^*_{red} \langle H', \texttt{final}\rangle$ and for every $r \in AIS$ at least one of the following holds:

1) (No adversary control) For any c'_A, it is the case that if $\langle H, \mathbb{E}_{\rho_A}[c'_A]\rangle \rightarrow^*_{red} \langle H'', \texttt{final}\rangle$ then $H'(r) = H''(r)$, or
2) (Direct adversary write) There exists a c''_A such that $\langle H, \rho_A\{c''_A\}\rangle \rightarrow^*_{red} \langle H''', \texttt{final}\rangle$ and $H'(r) = H'''(r)$.

Note that this definition does not require that the same clause (1 or 2) hold for every $r \in AIS$. Instead, some r may satisfy clause 1 and others may satisfy clause 2. This definition is inspired by and strictly weaker than information flow integrity (we show this formally in Section V).

Example 1 (Compiler service, simplified)**.** We formalize a simplified version of the confused compiler service described at the beginning of this section. The simplification is that this compiler does not contain the code that writes the billing file (we add the billing file in Example 5). Suppose that the compiler runs with authority $\top$ (the highest authority), the compiler service's caller/adversary runs with authority $\bot$ (the lowest authority, $\bot \not\geq_{\mathbb{L}} \top$) and that the compiler reads the source program from the reference r_S and the *name* of the output file from the reference r_O, both of which the caller must write beforehand. Then, we can abstractly model the relevant parts of the compiler service as the context $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{(!^{\mathbb{R}} r_O) := \texttt{compile}(!^{\mathbb{R}} r_S)\}$, where `compile` compiles a program. Note that this program has a CDA, i.e., it is not CDA-free according to Definition 2. For instance, consider adversaries of the form $c_A(S) = ({}^{\mathbb{W}} r_O := {}^{\mathbb{W}} r; {}^{\mathbb{W}} r_S := S)$, where $r \in AIS$ is a reference with $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and S ranges over source programs. Consider the execution of $\mathbb{E}_\bot[c_A(s_1)]$ for some source program s_1. Then, clause (1) does not hold for r because for another source program s_2 with $\texttt{compile}(s_1) \neq \texttt{compile}(s_2)$, the final heaps from the executions of $\mathbb{E}_\bot[c_A(s_1)]$ and $\mathbb{E}_\bot[c_A(s_2)]$ disagree on r. Clause (2) clearly does not hold when the initial heap does not contain $\texttt{compile}(s_1)$ in r. Intuitively, the CDA here is the expected one: The adversary passes whatever reference it wishes to overwrite in place of the output file.

In this case, it is easy to see that ACs do not provide CDA-freedom, because ACs will allow $\mathbb{E}_\bot[c_A(s_1)]$ to run to completion. Technically, $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_A)$ does not hold for all heaps H.

On the other hand, it can be shown that Cs does prevent this CDA, i.e., $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_C)$ holds for all H. The intuition is that if the attacker is able to write any capability ${}^{\mathbb{W}} r$ into r_O, then it must be able to compute ${}^{\mathbb{W}} r$, which implies from the expression evaluation rules of Cs that $\bot \geq_{\mathbb{L}} \mathbb{O}(r)$. Hence, $\bot$ can write to r and, so, clause 2 must hold for r.

Cs do not prevent all CDAs: Based on the above example, one may speculate that Cs prevent all CDAs. However, as the following three examples show, this speculation is false.

Example 2 (Value attack)**.** In this example, we do not allow the adversary to control the location that is written, but instead allow it to control the value that is written. This could, for instance, model a SQL injection attack on a high integrity database, via a confused deputy such as a web server. Assume that $r \in AIS$, $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and $\bot \geq_{\mathbb{L}} \mathbb{O}(r')$, so $\bot$ cannot write to r directly, but it can write to r'. Consider $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{{}^{\mathbb{W}} r := !({}^{\mathbb{R}} r')\}$. This context simply copies the contents of r' into r. This context also does not satisfy $\text{CDAF}(\mathbb{E}_\bot, H, \rightarrow_C)$ for all H. To see this, consider the adversary $c_\bot = ({}^{\mathbb{W}} r' := 42)$ with $H(r) \neq 42$. Then, $\mathbb{E}_\bot[c_\bot]$ ends with 42 in r. Clause 1 does not hold because for $c'_\bot = ({}^{\mathbb{W}} r' := 41)$, $\mathbb{E}_\bot[c'_\bot]$ ends with 41 in r. Clause 2 does not hold because no code running in $\bot\{\bullet\}$ can write 42 (or any value) to r.

Example 3 (Implicit influence)**.** Consider the following context: $\mathbb{E}_\bot = \bot\{\bullet\} \circ \top\{\text{if } (!^{\mathbb{R}} r_A) \text{ then } {}^{\mathbb{W}} r_H := 41 \text{ else } {}^{\mathbb{W}} r_H := 42\}$, where $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$ and $\bot \geq_{\mathbb{L}} \mathbb{O}(r_A)$. This context writes either 41 or 42 to a reference r_H that the adversary cannot write, depending on a boolean read from a reference r_A that the adversary can write. This context has a CDA

and it does not satisfy CDAF($\mathbb{E}_\perp, H, \rightarrow_C$) for all H. To see this consider any H with $H(r_H) \neq 41$ and the adversary command $c_\perp = ({}^{\mathbb{W}}r_A := tt)$. Then, for the reference r_H, neither clause (1) nor (2) holds.

Example 4 (Initial heap attack)**.** Consider the following context: $\mathbb{E}_\perp = \perp\{\bullet\} \circ \top\{!^{\mathbb{R}}r_A := !^{\mathbb{R}}r'_A\}$ where $\perp \geq_{\mathbb{L}} \mathbb{O}(r_A) = \mathbb{O}(r'_A)$. Assume that the initial heap is such that $H(r_A) = {}^{\mathbb{W}}r_H$ with $\perp \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$, i.e., r_A contains a reference ${}^{\mathbb{W}}r_H$ that the adversary cannot write. This context has a CDA — CDAF($\mathbb{E}_\perp, H, \rightarrow_C$) does not hold for all heaps H. To see this, consider the adversary $c_\perp = ({}^{\mathbb{W}}r'_A := 42)$ and an initial heap H such that $H(r_H) \neq 42$. Then, $\mathbb{E}_\perp[c_\perp]$ ends with 42 in r_H. Clause 1 does not hold because for $c'_\perp = ({}^{\mathbb{W}}r'_A := 41)$, $\mathbb{E}_\perp[c'_\perp]$ ends with 41 in r_H. Clause 2 does not hold because no code running in $\perp\{\bullet\}$ can write 42 to r_H.

Note that there is a fundamental difference in the nature of the CDA in Examples 1, 2, 3 and 4. In Example 1, the deputy (region $\top$) obtains the write capability to the reference under attack from the adversary. We refer to this kind of capability designation as *explicit*. In Examples 2, 3 and 4 the deputy already has the capability (either directly in its code or indirectly through the initial heap) but the adversary influences what gets written to it. We refer to this kind of designation as *implicit*. As should be clear from the examples, Cs prevent CDAs caused by explicit designation (Example 1) but do not prevent CDAs caused by implicit designation.

In Section III-B, we show that the language can be restricted to rule out cases with implicit designation. Trivially, for this restricted language, Cs prevent all CDAs. However, this restricted language also rules out many harmless programs. But before going into that, we point out a shortcoming of our current definition of CDA-freedom and propose a fix.

Relaxing CDA-freedom: Our current definition of CDA-freedom *completely* rules out the possibility that the adversary influence any reference of interest that it cannot write directly. In practice, it is possible that the deputy allows the adversary to have *controlled influence* on a privileged reference. The billing file from the compiler example at the beginning of Section III is a good example. There, the adversary (compiler invoker) can legitimately influence the billing file, e.g., by changing the size of the source file, but the deputy (compiler service) wants to limit this control by allowing only legitimate billing values to be written to the billing file.

To permit such controlled interaction between the adversary and the deputy, we introduce a notion of endorsement (on the lines of information flow endorsement [20]). We allow a region to be declared endorsed (denoted by $\overline{\mathbb{P}}$ as opposed to $\mathbb{P}$), and subsequently be taken out of the purview of the CDA-freedom definition. The intuition is to distinguish, via an endorsed region, a confused deputy from a deputy which is acting on an attacker's authority on purpose. We propose the following definition of CDA-freedom with endorsement (denoted by CDAF-E). CDAF-E essentially states that for an authority context ($\mathbb{E}_{\rho_A}$), heap (H) and a reduction relation ($\rightarrow_{red}$), CDAF should hold for all subsequences of region commands which do not include any endorsed principal.

Definition 3 (CDA-freedom with endorsement)**.** Context $\mathbb{E}_{\rho_A} = \rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ is called CDA-free with endorsement under heap H and semantics $\rightarrow_{red}$, written CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{red}$), if for every contiguous subsequence $\mathbb{E}'_{\rho_A} = \rho_i\{c_k\} \circ \ldots \circ \rho_j\{c_{k+m}\}$ of $\mathbb{E}_{\rho_A}$ such that for all $i \in \{k, \ldots, k+m\}$, ρ_i is not of the form $\overline{\mathbb{P}}$ for any $\mathbb{P}$, we have CDAF($\mathbb{E}'_{\rho_A}, H, \rightarrow_{red}$).

The parameter H in CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{red}$) represents any heap starting from which we wish to test non-endorsed subsequences of region commands in $\mathbb{E}_{\rho_A}$. It may sound odd that we use the same heap to test all such subsequences, but the intent is to universally quantify over H outside the definition, so specifying a separate starting heap for each subsequence is not useful.

Example 5 (Compiler service)**.** We extend the compiler service (Example 1) with the billing file. Assume that the billing amount for a source file s is computed by the function $\texttt{billing}(s)$ and that the billing file is represented by the reference r_B with $\perp \not\geq_{\mathbb{L}} \mathbb{O}(r_B)$. Then, we can write the complete compiler as the context: $\mathbb{E}_\perp = \perp\{\bullet\} \circ \top\{(!^{\mathbb{R}}r_O) := \texttt{compile}(!^{\mathbb{R}}r_S)\} \circ \overline{\top}\{{}^{\mathbb{W}}r_B := \texttt{billing}(!^{\mathbb{R}}r_S)\}$. Note that this context has the same CDA as the simplified one from Example 1 (the adversary can confuse the compiler by passing a privileged reference in r_O). Correctly, this context does not satisfy Definition 3. However, importantly, it does not fail this definition because of the third region command $\overline{\top}\{{}^{\mathbb{W}}r_B := \texttt{billing}(!^{\mathbb{R}}r_S)\}$, which writes a controlled value derived from an adversary controlled reference r_S to a privileged reference r_B. That region command is endorsed by $\overline{\top}$ and, hence, excluded from the purview of the definition. Instead, the context fails the definition due its first two region commands $\perp\{\bullet\} \circ \top\{(!^{\mathbb{W}}r_O) := \texttt{compile}(!^{\mathbb{R}}r_S)\}$, which indeed have an undesirable CDA.

B. Capability semantics prevent some CDAs

Examples 2, 3 and 4 show that the capability semantics, Cs, cannot prevent all CDAs even in our simple region calculus. In this subsection, we explore this point further and show that under very strong restrictions on programs (contexts) and heaps, Cs do to prevent all CDAs. We introduce some terminology for discourse. Given an attacker region ρ_A, we call a region ρ low integrity or low if $\rho_A \geq_{\mathbb{L}} \rho$. Dually, a region ρ is high integrity or high if $\rho_A \not\geq_{\mathbb{L}} \rho$. A reference r is called low (high) if $\mathbb{O}(r)$ is low (high), i.e., if ρ_A can

(cannot) directly write the reference.

From our examples, it should be clear that if a high region ends up possessing a high reference r in AIS, then Cs alone may not prevent all CDAs because Cs' checks are limited to references only and, hence, an adversary could confuse the high region by influencing values that the high region writes to a high reference in AIS. Consequently, if we wish to use Cs to prevent all CDAs, we must place enough restrictions on contexts to ensure that high regions never end up possessing high references from AIS (low references are not a concern for preventing CDAs because these references always satisfy clause 2 of Definition 2). The converse is also trivially true: If no high region ever possesses a high reference from AIS, then no high reference from AIS can ever be written under Cs semantics, so clause 1 of Definition 2 must hold for all high references in AIS.

There are three ways in which a high region may end up possessing a high reference from AIS. First, the command that starts running in the high region may have a hard-coded high reference from AIS, as in Examples 2 and 3. Second, the command in the high region may read the high reference from another reference, as in Example 4. Third, the adversary may pass the high reference through another reference, as in Examples 1 and 5. Checks made by Cs prevent the adversary from ever evaluating (let alone passing) a high reference, so the third possibility is immediately ruled out in Cs semantics. It follows, then, that if we can restrict our language and heaps substantially to prevent the first two possibilities, then Cs semantics will imply CDA-freedom.

To prevent the first two possibilities, we restrict initial commands in high regions and the initial heap. Accordingly, we create the following two definitions, which say, respectively, that the commands in non-endorsed high regions and the (initial) heap do not have high references from AIS.

Definition 4 (No interesting high references in high regions)**.** A context $\mathbb{E}_{\rho_A}$ has no interesting high references in non-endorsed high regions, written $nihrP(\mathbb{E}_{\rho_A})$, if $\mathbb{E}_{\rho_A} = \rho_1\{c_1\}\circ\ldots\circ\rho_n\{c_n\}$ and for all $i \in \{1,\ldots,n\}$, if $\rho_A \not\geq_{\mathbb{L}} \rho_i$, $\rho_i \neq \overline{\mathbb{P}}$ and ${}^{\mathbb{W}}r \in c_i$, then either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r)$ or $r \notin AIS$.

Definition 5 (No interesting high references in heap)**.** A heap H has no interesting high references, written $nihrH(H,\rho_A)$ if for all r, $H(r) = {}^{\mathbb{W}}r'$ implies either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r')$ or $r' \notin AIS$.

We now state the main result of this section: If the initial heap has no interesting high references and the context has no interesting high references in non-endorsed high regions, then the context has no CDAs under Cs.

Theorem 4 (Cs prevents some CDAs)**.** If $nihrH(H,\rho_A)$ and $nihrP(\mathbb{E}_{\rho_A})$, then CDAF-E$(\mathbb{E}_{\rho_A}, H, \rightarrow_C)$.

Proof: We first show that the absence of high references of AIS from the heap and non-endorsed high regions is invariant under $\rightarrow_C$. This implies that no high reference from AIS is ever written in the execution of $\mathbb{E}_{\rho_A}[c_A]$. Hence, clause 1 of Definition 2 holds for all high references in AIS and clause 2 holds for all low references in AIS. ■

We note that the restrictions in the condition of this theorem are extremely strong. In the next section, we present alternative mechanisms for obtaining CDA-freedom that relax these restrictions, at the expense of more runtime overhead. Also note that, if preventing CDAs were the only objective, then Cs are very imprecise: They block many programs that have no CDAs.

Example 6. Consider the context $\bot\{\bullet\}\circ\top\{{}^{\mathbb{W}}r := 1\}$ that simply writes 1 to the reference r. Assume $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r)$. Clearly, this context does not have a CDA as the adversary can control neither the reference that is written (always r) nor the value that is written (always 1), so clause 1 of the CDA-freedom definition holds for all references. However, when instantiated with any adversarial command c_A that computes a high reference, the resulting program will be stopped by Cs.

IV. CDA PREVENTION USING PROVENANCE TRACKING

In this section, we describe two mechanisms other than Cs for preventing CDAs. Both mechanisms relax the assumptions needed for CDA prevention (the pre-conditions of Theorem 4) and, at the same time, execute several CDA-free programs like Example 6, which Cs block. We present the mechanisms as two alternative semantics for our calculus. Both semantics start from the same baseline — the access control semantics (ACs) — and add checks based on *provenance tracking* to prevent CDAs. Provenance tracking, which is based on the extensively studied taint tracking (e.g., [24]), augments ACs to *label* each computed value with a principal, which is a lower bound on the principals whose references have been read to compute the value. Since code in region ρ can only write to references below the principal corresponding to the region ρ, the principal labeling a value is also a lower bound on the principals whose *code* has influenced the value. With such a labeling mechanism in place, CDAs can be prevented easily by checking during reference assignment (rule A-Assign) that an attacker-influenced value is not written to a high reference.

Our two new semantics differ in how they compute labels. Our first semantics, called the explicit-only provenance semantics or EPs, tracks regions that have influenced a value but ignores the effect of implicit influences due to control flow. As a result, this semantics prevents CDAs only under some assumptions, but these assumptions are still weaker than those needed for preventing CDAs via Cs (i.e., the assumptions of Theorem 4). Our second semantics, called the full provenance semantics or FPs, tracks all influences on a value, including implicit ones. This semantics prevents

all CDAs without additional assumptions. Since EPs does not track implicit influences, it can be implemented far more efficiently than FPs (this is well-known from work on information flow control), which justifies our interest in both semantics, not just FPs.

A. Explicit-only provenance semantics

The explicit-only provenance semantics (EPs) tracks, for every computed value, the principals whose references have affected the value. Only explicit influences, such as those due to reference copying are tracked. EPs does not track influence due to control constructs (branch conditions in conditionals and loops). We start from the access control semantics, ACs, and modify the expression evaluation judgment $\langle H, e\rangle \overset{\rho}{\Downarrow_A} v$ to include a label (a principal) on the output value v. This label is a lower bound on all principals whose references have been read during the computation of e. To avoid confusion, we denote labels with the letter ℓ, but readers should note that like $\mathbb{P}$'s, labels are drawn from $Prin$.

The revised judgment for expression evaluation is written $\langle H, e\rangle \overset{\rho}{\Downarrow_{EP}} v^{\ell}$. Its rules are shown in Figure 3. In rule EP-Val, the expression e is already a value. Computation of the result does not read any reference, so the label on the output is $\top$ (the highest point of the lattice $\mathbb{L}$). In rule EP-Deref, the expression being evaluated has the form $!e$. In this case, the semantics first evaluates e to the read capability of a reference ${}^{\mathbb{R}}r$ and then dereferences r. The result could be influenced by every region that was dereferenced in computing r from e as well as $\mathbb{O}(r)$. Hence, the output label is the meet or greatest lower bound ($\sqcap$) of the label of r (denoted ℓ_r in the rule) and $\mathbb{O}(r)$.

The command and program evaluation relations of EPs are written $\overset{\rho}{\rightarrow_{EP}}$ and $\rightarrow_{EP}$, respectively. They use the rules of ACs (Figure 2), except the rule for assignment, which now makes an additional check to ensure that low-influenced values are not written to high references. This revised rule, EP-Assign, is also shown in Figure 3. In comparison to the ACs rule, A-Assign, there is one additional last premise. This premise checks that the label on the updated reference (called ℓ_r) and the label on the value written to the reference (called ℓ_v) are both above (higher integrity than) the principal that owns the reference. This ensures that if the updated reference is high (unwritable by the adversary directly), then the value written has no low (adversarial) influence. Importantly, the check is made only if the executing region is not endorsed.

As in Theorem 4, to show that EPs ensures CDA-freedom, we must assume that the initial commands in high regions do not contain high references from AIS (condition $nihrP(\mathbb{E}_{\rho_A})$). However, the condition on the initial heap in Theorem 4—that the heap contain no high references from AIS—can now be weakened slightly: We only require that *high references* in the initial heap not contain any high references from AIS. Intuitively, we do not care about the contents of the low references in the initial heap because anything read from low references will carry a low label (by rule EP-Deref) and, hence, cannot influence anything written to a high reference (rule EP-Assign).

Definition 6 (No interesting high references in high heap)**.** A heap H has no interesting high references in high parts, written $nihrHH(H, \rho_A)$ if for all r, $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r)$ and $H(r) = {}^{\mathbb{W}}r'$ imply either $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r')$ or $r' \notin AIS$.

Note that $nihrH(H, \rho_A)$ immediately implies $nihrHH(H, \rho_A)$ so the latter is a weaker property and hence constitutes a weaker assumption.

Theorem 5 (EPs prevents some more CDAs)**.** If $nihrHH(H, \rho_A)$ and $nihrP(\mathbb{E}_{\rho_A})$, then CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{EP}$).

Proof: We first show that the absence of high references of AIS from the high part of the heap and non-endorsed high regions is invariant under $\rightarrow_{EP}$. This implies that no high reference from AIS is ever written in the execution of $\mathbb{E}_{\rho_A}[c_A]$. Hence, clause 1 of Definition 2 holds for all high references in AIS and clause 2 holds for all low references in AIS. ■

We now discuss the relative permissiveness of Cs and EPs for CDA-prevention.

Example 7. This examples highlights the difference between the assumptions $nihrHH(H, \rho_A)$ of Theorem 5 and $nihrH(H, \rho_A)$ of Theorem 4. Consider the context $\top\{{}^{\mathbb{W}}r := !({}^{\mathbb{R}}r_L)\}$, which doesn't even contain a hole for the adversary (and, hence, trivially, has no CDA). Consider the adversary level $\rho_A = \bot$, assume that $\bot \geq_{\mathbb{L}} \mathbb{O}(r_L) = \mathbb{O}(r)$ and that we start from a heap H with $H(r_L) = {}^{\mathbb{W}}r_H$ with $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$. Then, it is easy to see that $nihrHH(H, \rho_A)$, so the assumption of Theorem 5 does not rule this program out, but it is not the case that $nihrH(H, \rho_A)$, so the assumption of Theorem 4 does rule this program out.

We saw earlier that Example 6 has no CDA, but is halted by Cs. It can be easily checked that EPs allows the example to execute to completion. Based on this, one may ask whether EPs is strictly more permissive than Cs when the program passes the conditions of Theorem 4 (and, hence, also of Theorem 5). However, this is false as the following example shows.

Example 8. Consider the context $\rho\{{}^{\mathbb{W}}r_H := !^{\mathbb{R}}r_L\}$, which has no hole for the adversary and, hence, no CDA. Assume that $\rho_A = \rho$ and $\rho \geq_{\mathbb{L}} \mathbb{O}(r_H) > \mathbb{O}(r_L)$. Then, since the context copies a value from a reference r_L to r_H and the owner of the former is strictly below the owner of the latter, EPs will stop this context from executing. On the other hand, Cs will allow this context to execute to completion.

$$\text{EP-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_{EP} v^{\top}} \qquad \text{EP-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{A}\ {}^{\mathbb{R}}r^{\ell_r} \qquad \mathbb{P}_r = \mathbb{O}(r) \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_{EP} v^{\ell_r \sqcap \mathbb{P}_r}}$$

$$\text{EP-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_{EP}\ {}^{\mathbb{W}}r^{\ell_r} \qquad \rho \geq_{\mathbb{L}} \mathbb{O}(r) \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_{EP} v^{\ell_v} \qquad \rho \neq \overline{\mathbb{P}} \implies \ell_r \sqcap \ell_v \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, e_1 := e_2\rangle \rightarrow^{\rho}_{EP} \langle H[r \mapsto v], \text{skip}\rangle}$$

Figure 3. Explicit provenance semantics (all other rules are same as those of access control semantics, Figure 2)

Hence, EPs prevents all CDAs on a slightly larger language fragment than Cs (Theorem 4 vs Theorem 5). However, the permissiveness of EPs and Cs on CDA-free programs in the *common* fragment is incomparable (Examples 6 and 8).

B. Full provenance semantics

We now show that by tracking complete provenance of values, including implicit influences due to control flow, we can enforce CDA-freedom for our entire calculus (without any restrictions on contexts or heaps). To do this, we build a full provenance semantics or FPs for our calculus. FPs is based upon similar semantics for information flow control [24], with minor adjustments to account for regions. As in EPs, every computed value is labeled with a principal, which is a lower bound on principals whose references (and code) could have influenced the value. To track influence due to control flow, we introduce an auxiliary label to the semantic state. This label, called the program counter or pc in information flow control literature, is a lower bound on all regions that have influenced the reachability of the current command. When we enter the body of a control construct like if-then-else or while, we lower the pc to the meet of the current pc and the label of the branch or loop condition. When we exit the body of the control construct we restore the pc back to its original value (not restoring the pc would make the semantics less permissive). To enable this restoration, we maintain a stack of pc's and push the new pc to the stack when we enter an if-then-else or while construct. We pop the stack when we exit the construct. This stack is denoted PC. Its topmost label is the current pc. At the top-level, we start with $PC = [\top]$.

The rules for FPs are shown in Figure 4. The expression evaluation judgment is identical to that in EPs; it has the form $\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell}$. The judgment for reducing commands is now modified to include the stack PC. It takes the form $\langle H, PC, c\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c'\rangle$. When entering the body of an if-then-else or while construct (rules FP-if, FP-else and FP-while 1), we push $pc \sqcap \ell$ onto PC, where pc is the current topmost label on PC and ℓ is the label of the branch or loop condition. We also add a marker (endif or endwhile) to the code body to indicate when the body ends. When this marker is encountered, we pop the stack PC (rules FP-endif and FP-endwhile). By doing this, we ensure that the top label on PC is a lower bound on the labels of all branch/loop conditions that influence the control flow at the current instruction. The most interesting rule of the semantics is that for assignment (rule FP-Assign), which, in addition to all checks made by the corresponding rule in EPs, also checks that the current pc is above the owning region of the reference being written (last premise). This additional check ensures that an adversary cannot influence the contents of high references even through control constructs, as in Example 3.

The judgment for evaluating programs, $\langle H, PC, P\rangle \rightarrow_{FP} \langle H', PC', P'\rangle$, also carries PC but its rules do not modify PC in any interesting way. It is an invariant that $PC = [\top]$ at the beginning of the program's execution and every time a region's command ends with skip. This explains why $PC = [\top]$ in rule FP-Comp 2.

Next, we show that FPs enforces CDA-freedom without any assumptions. Technically, the definitions of CDA-freedom, Definition 2 and Definition 3, do not directly apply to FPs because those definitions assume that the reduction semantics $\rightarrow_{red}$ rewrite pairs $\langle H, P\rangle$, whereas the FPs reduction $\rightarrow_{FP}$ rewrites triples $\langle H, PC, P\rangle$. However, we can reinterpret $\langle H, P\rangle$ in Definition 2 to mean $\langle H, [\top], P\rangle$. With that implicit change, we can prove the following theorem.

Theorem 6 (FPs prevent all CDAs)**.** CDAF-E($\mathbb{E}_{\rho_A}, H, \rightarrow_{FP}$) holds unconditionally.

Proof: We prove that FPs enforces a strong form of information flow integrity (Definition 8), which in turn implies CDA-freedom with endorsement (Theorem 7). To prove the former, we build a simulation relation between states of the form $\langle H, PC, c\rangle$, as is standard in information flow control [24]. ■

We now discuss the relative permissiveness of FPs and EPs for CDA-prevention.

Example 9. This example demonstrates a CDA-free context that violates the assumptions of Theorem 5 and, hence, is not in the fragment on which EPs enforces CDA-freedom. Consider Example 7, but stipulate that $\bot \not\geq_{\mathbb{L}} \mathbb{O}(r_L)$. The heap now violates $nihrHH(H, \rho_A)$, hence, the resulting example lies outside the fragment allowed by the assumptions of Theorem 5. However, the example will execute to

Expressions:

$$\text{FP-Val}\ \frac{}{\langle H, v\rangle \overset{\rho}{\Downarrow}_{FP} v^{\top}} \qquad \text{FP-Deref}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} {}^{\mathbb{R}}r^{\ell_r} \qquad \mathbb{P}_r = \mathbb{O}(r) \qquad v = H(r)}{\langle H, !e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell_r \sqcap \mathbb{P}_r}}$$

Commands:

$$\text{FP-if}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = tt}{\langle H, pc :: PC, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c_1; \text{endif}\rangle}$$

$$\text{FP-else}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = ff}{\langle H, pc :: PC, \text{if } e \text{ then } c_1 \text{ else } c_2\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c_2; \text{endif}\rangle}$$

$$\text{FP-while 1}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell} \qquad v = tt}{\langle H, pc :: PC, \text{while } e \text{ do } c\rangle \rightarrow^{\rho}_{FP} \langle H, (pc \sqcap \ell) :: pc :: PC, c; \text{endwhile; while } e \text{ do } c\rangle}$$

$$\text{FP-while 2}\ \frac{\langle H, e\rangle \overset{\rho}{\Downarrow}_{FP} v \qquad v = ff}{\langle H, PC, \text{while } e \text{ do } c\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle} \qquad \text{FP-endwhile}\ \frac{}{\langle H, pc :: PC, \text{endwhile}\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle}$$

$$\text{FP-Assign}\ \frac{\langle H, e_1\rangle \overset{\rho}{\Downarrow}_{FP} {}^{\mathbb{W}}r^{\ell_r} \qquad \langle H, e_2\rangle \overset{\rho}{\Downarrow}_{FP} v^{\ell_v} \qquad \rho \geq_{\mathbb{L}} \mathbb{O}(r) \qquad \rho \neq \overline{\mathbb{P}} \implies \ell_r \sqcap \ell_v \sqcap pc \geq_{\mathbb{L}} \mathbb{O}(r)}{\langle H, pc :: PC, e_1 := e_2\rangle \rightarrow^{\rho}_{FP} \langle H[r \mapsto v], pc :: PC, \text{skip}\rangle} \qquad \text{FP-endif}\ \frac{}{\langle H, pc :: PC, \text{endif}\rangle \rightarrow^{\rho}_{FP} \langle H, PC, \text{skip}\rangle}$$

$$\text{FP-Seq 1}\ \frac{\langle H, PC, c_1\rangle \rightarrow^{\rho}_{FP} \langle H', PC' c_1'\rangle}{\langle H, PC, c_1; c_2\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c_1'; c_2\rangle} \qquad \text{FP-Seq 2}\ \frac{}{\langle H, PC, \text{skip}; c_2\rangle \rightarrow^{\rho}_{FP} \langle H, PC, c_2\rangle}$$

Program:

$$\text{FP-Prg 1}\ \frac{\langle H, PC, c\rangle \rightarrow^{\rho}_{FP} \langle H', PC', c'\rangle}{\langle H, PC, \rho\{c\}\rangle \rightarrow_{FP} \langle H', PC', \rho\{c'\}\rangle} \qquad \text{FP-Comp 1}\ \frac{\langle H, PC, P_1\rangle \rightarrow_{FP} \langle H', PC', P_1'\rangle}{\langle H, PC, P_1 \circ P_2\rangle \rightarrow_{FP} \langle H', PC', P_1' \circ P_2\rangle}$$

$$\text{FP-Comp 2}\ \frac{}{\langle H, [\top], \rho\{\text{skip}\} \circ P\rangle \rightarrow_{FP} \langle H, [\top], P\rangle}$$

Figure 4. Full provenance semantics

completion in FPs.

On contexts that lie in the fragment allowed by EPs, FPs is no more permissive than EPs. This is easy to see: FPs makes additional checks for implicit influences, which EPs does not. In fact, due to these checks, FPs is strictly less permissive on the EPs fragment, as the following example shows.

Example 10. Consider the context $\rho\{\text{if } (!^{\mathbb{R}}r_A) \text{ then } {}^{\mathbb{W}}r_H := 41 \text{ else } {}^{\mathbb{W}}r_H := 42\}$ with $\rho_A = \rho \geq_{\mathbb{L}} \mathbb{O}(r_H) > \mathbb{O}(r_A)$. This code has no CDA since it has no hole for the adversary. FPs will stop this program because a location of lower integrity (r_A) influences a location of higher integrity (r_H) via control flow. However, EPs will allow this program to execute to completion because it does not track such influences.

Summary of examples and CDA-prevention: To summarize, the three semantics Cs, EPs and FPs soundly enforce CDA-freedom with endorsement for progressively larger sublanguages, with FPs covering our entire language. However, for programs and heaps that satisfy the assumptions of Cs, EPs and Cs are incomparable in permissiveness. On the subset of programs and heaps that satisfy the assumptions of EPs, EPs is strictly more permissive than FPs. The access control semantics, ACs, is ineffective against confused deputy attacks, even those with only explicit designation. The following table summarizes how all our examples fare on all four semantics. A and R mean that the semantics would accept and reject (halt) the program respectively. For programs with CDA, we write A or R to mean accept or reject when the adversary tries to launch a CDA. NP

means that the example is outside the fragment on which the semantics enforces CDA-freedom.

Example	Has CDA?	ACs	Cs	EPs	FPs
1	CDA	A	R	R	R
2	CDA	A	NP	R	R
3	CDA	A	NP	NP	R
4	CDA	A	NP	R	R
5	CDA	A	R	R	R
6	no CDA	A	NP	NP	A
7	no CDA	A	NP	A	A
8	no CDA	A	A	R	R
9	no CDA	A	NP	NP	A
10	no CDA	A	A	A	R

V. Relation to information flow integrity

Our formalization of CDA-freedom, Definitions 2 and 3, is inspired by the notion of information flow integrity. Here, we compare the two. A standard baseline definition for information flow integrity is Goguen and Meseguer (GM) style non-interference [25], which states that a program satisfies integrity if the high parts of the final memory obtained by executing the program cannot be influenced by the low parts of the initial memory. Since Definition 2 talks about a program with holes, we modify GM-style non-interference to take holes into account. This yields Definitions 7 and 8 (these definitions are inspired by [22]). For an attacker ρ_A, say that H_1 and H_2 are high-equivalent, written $H_1 \sim_{\rho_A} H_2$, if for all r such that $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r)$, it is the case that $H_1(r) = H_2(r)$.

Definition 7 (Non-interference without endorsement)**.** A context $\mathbb{E}_{\rho_A}$ has non-interference against active adversaries under reduction semantics $\rightarrow_{red}$, written *NI-A*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$, if for all heaps H_1, H_2, H_1', H_2' and commands c_A, c_A', if $H_1 \sim_{\rho_A} H_2$, $\langle H_1, \mathbb{E}_{\rho_A}[c_A]\rangle \rightarrow^*_{red} \langle H_1', \texttt{final}\rangle$ and $\langle H_2, \mathbb{E}_{\rho_A}[c_A']\rangle \rightarrow^*_{red} \langle H_2', \texttt{final}\rangle$, then $H_1' \sim_{\rho_A} H_2'$.

Definition 8 (Non-interference with endorsement)**.** Context $\mathbb{E}_{\rho_A} = \rho_1\{c_1\} \circ \ldots \circ \rho_n\{c_n\}$ has non-interference against active adversaries with endorsement under reduction semantics $\rightarrow_{red}$, written *NI-A-E*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$, if for every contiguous subsequence $\mathbb{E}'_{\rho_A} = \rho_i\{c_k\} \circ \ldots \circ \rho_j\{c_{k+m}\}$ of $\mathbb{E}_{\rho_A}$ such that for all $i \in \{k, \ldots, k+m\}$, ρ_i is not of the form $\overline{\mathbb{P}}$ for any $\mathbb{P}$, we have *NI-A*$(\mathbb{E}'_{\rho_A}, \rightarrow_{red})$.

It turns out that Definition 7 is strictly stronger than Definition 2 (and consequently Definition 8 is strictly stronger than Definition 3). Intuitively, Definition 7 ensures that all high references in AIS satisfy clause 1 of Definition 2 (all low references trivially satisfy clause 2 in all four of our semantics).

Theorem 7 (Non-interference implies CDA-freedom)**.** For $\rightarrow_{red} \in \{\rightarrow_A, \rightarrow_C, \rightarrow_{EP}, \rightarrow_{FP}\}$, *NI-A-E*$(\mathbb{E}_{\rho_A}, \rightarrow_{red})$ implies CDAF-E$(\mathbb{E}_{\rho_A}, H, \rightarrow_{red})$ for every H.

The converse of this theorem does not hold, as the following counterexample shows. The reason is that the definition of non-interference quantifies over *two* initial heaps H_1 and H_2, while the definition of CDA-freedom does not.

Example 11. Consider the context $\mathbb{E}_{\rho_A} = \rho_H\{^{\mathbb{W}}r_H := \ !^{\mathbb{R}}r_A\}$, where $\rho_A \not\geq_{\mathbb{L}} \mathbb{O}(r_H)$ and $\rho_A \geq_{\mathbb{L}} \mathbb{O}(r_A)$. The context copies the contents of r_A to r_H, so this context does not satisfy Definition 8. Specifically, we can choose any H_1 and H_2 that agree on all references except r_A. Then, $H_1 \sim_{\rho_A} H_2$. However, the final heaps disagree on r_H, so the final heaps are not equivalent. On the other hand, this context trivially satisfies Definition 3 since it has no hole for the adversary.

VI. Related Work

Understanding precisely which security properties a capability system can enforce and how capabilities and access control differ are long-standing research topics. We focus our discussion only on work dealing with these two issues.

Using informal arguments, Miller *et al.* [11] compare four system models for taking authorization decisions: the first two models are access control as columns and capabilities as rows of the Lampson matrix [6]. The third model is capabilities as keys (as in the Amoeba operating system [4]) and the fourth model is capabilities as objects (as in the Joe-E language [26]). Our capability semantics (Cs) can be seen as an abstraction of either model 2 or a degenerate case of model 4, while our access control semantics (ACs) are an abstraction of model 1. Miller *et al.*'s comparison is based on 7 properties A...G, where only property A is claimed to be impossible to hold in access control as columns. (Although properties B...G are, in practice, not implemented in access control, there is no reason in theory not to have an access control system complying with B...G). Starting from this comparison, we decided to focus our formalization of capabilities only taking property A into account. Miller *et al.* discuss three "myths": the equivalence myth, the confinement or delegation myth, and the irrevocability myth. In this paper, we only consider the equivalence myth.

The confinement myth deals with "capabilities cannot limit the propagation of authority". Miller *et al.* also discuss CDAs in relation to capabilities. CDAs, first described in the literature in 1988 [16], are related to property A — *No Designation without Authority* — since authorization given to one party T (the deputy) is used to access a resource designated by a different party U (the adversary).

The topmost implication of Fig. 14 in [11] and other works such as [16], [23] seem to suggest that CDAs are impossible in capability systems with Property A. Our contribution is to clarify the arguments of [11] and include examples of CDAs that can happen with Property A and those that cannot. Indeed, in some cases of CDAs, designation of a resource can be done *implicitly* by U, in contrast

to explicit designation which would clearly be prevented by Property A.

Chander *et al.* [27] use state-transition systems to model capabilities and access control. They model two versions of capabilities, based on [1]. The first model is capabilities as rows in the Lampson matrix and the second model is capabilities as unforgeable tokens. None of these models have property A. Hence, their capabilities are more similar to our access control model than to our capabilities model. In particular, in their second capabilities model where a capability to access a resource r is an unforgeable token T(r), there is nothing that prevents a subject s from designating r even though s cannot generate a capability T(r). Moreover, once s possesses T(r), s can pass this capability to other subjects. In contrast, our capability semantics would prevent this. With the goal of comparing both systems according to their power of delegation, Chander *et al.* prove various simulation relations between capabilities and access control. One of their main results is the equivalence between access control and capabilities viewed as rows of the Lampson access matrix (without modeling property A). In contrast, we show that the equivalence does not hold with property A. Chander *et al.* also prove that there is no equivalence between access control and capabilities when properties of capabilities seen as unforgeable tokens are taken into account. In their model, this is due to the impossibility of revoking capabilities. We have not modeled revocation in our calculus since it is not needed for exploring the equivalence myth.

Maffeis *et al.* [12] formally connect capabilities that are objects [2] to operational semantics of programming languages. Capability safety refers to the property of a language that guarantees that a component must have a capability to access a resource. They explicitly formalize property A (see §V, Def. 8, cond. 1(b) of [12]) as a basic condition of an object capability system. They do not directly explain how property A can be modeled in an operational semantics, which is central to our results. They also prove that Cajita, a component of Caja [3]—an object capability language based on JavaScript—is capability safe.

Murray *et al.* [28] define an object capability model in the CSP process algebra. In their model, they do not formalize property A and, in particular, they allow for delegation of capabilities as in [27]. These points differ from our model. Using a model checker, Murray *et al.* can detect covert channels of illegal information flows [21]. Their work focuses on the detection of information flow leaks. Our CDA-freedom is similar to, but slightly weaker than, information flow integrity [22] and our work is focused on prevention, not detection. Our full provenance analysis ensures information flow integrity (and thus CDA-freedom).

Reasoning about the correct use of capabilities, separating security policies from implementations, has been studied in [29] and [30]. Drossopoulou and Noble [29] analyze Miller's Mint and Purse example of [2] (in capabilities based on JAVA as in, e.g., Joe-E [26]) using a formal specification language. Building on object capabilities, Drossopoulou *et al.* [31] propose special specification predicates in a specification language based on JavaScript and simpler than the one of [29]. With these specification predicates they can model risk and trust in systems having components with different levels of trust. In their specification language, they cannot directly express the idea of encapsulation of objects. We conjecture that explicitly allowing encapsulation of objects to appear in specifications should be tantamount to assuming property A in object capability systems. Along the same lines, Saghafi *et al.* [23] informally discuss a relation between Property D of [11] and encapsulation in order to compare capabilities with feature-oriented programming. Dimoulas *et al.* [30] propose extensions to capability languages that restrict the propagation of capabilities according to declarative policies. By means of integrity policies, they restrict components that may influence the use of a capability. They do not model property A.

Birgisson *et al.* [32] propose secure information flow enforcement by means of capabilities. They do so by proposing a transformation from arbitrary source programs to a language with capabilities. (In their experiments, the target language is Caja [3].) They present formal guarantees of information flow security [21] and permissiveness.

VII. Conclusion

We examine the relation between access control and capability semantics in a simple language setting. We model Miller *et al.*'s property A "no designation without authority", but without appealing to object encapsulation as in object capability languages [2]. We also present the first extensional characterization of freedom from confused deputy attacks (CDAs) and relate it to information flow integrity. We clarify which classes of CDAs capabilities (as a mechanism) can and cannot prevent and stipulate the exact conditions under which they can prevent all classes of CDAs. Furthermore, we present alternate ways of preventing CDAs using provenance tracking with fewer conditions.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) grant "Information Flow Control for Browser Clients" under the priority program "Reliably Secure Software Systems" (RS^3) and the ANR project AJACS ANR-14-CE28-0008.

References

[1] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[2] M. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, 2006.

[3] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript." [Online]. Available: http://code.google.com/p/google-caja

[4] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, and S. J. Mullender, "Experiences with the amoeba distributed operating system," *Comm. ACM*, 1990.

[5] Mozilla Developer Network, "Script security." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Gecko/Script_security

[6] B. Lampson, "Protection," *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[7] R. Sandhu, "Role-based access control," *Advances in Computers*, vol. 46, pp. 237–286, 1998.

[8] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *IEEE Symposium on Security and Privacy (Oakland)*, 2000.

[9] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode, "Enforcing authorization policies using transactional memory introspection," in *ACM Conference on Computer and Communications Security (CCS)*, 2008.

[10] D. Garg and F. Pfenning, "Noninference in constructive authorization logic," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2006.

[11] M. Miller, K.-P. Yee, J. Shapiro, and C. Inc, "Capability myths demolished," 2003. [Online]. Available: http://www.erights.org/elib/capability/duals/myths.html

[12] S. Maffeis, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[13] S. Drossopoulou and J. Noble, "The need for capability policies," in *Workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2013.

[14] L. Jia, S. Sen, D. Garg, and A. Datta, "A logic of programs with interface-confined code," in *IEEE Computer Security Foundations Symposium (CSF)*, 2015.

[15] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[16] N. Hardy, "The confused deputy (or why capabilities might have been invented)," *Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[17] OWASP, "Cross site request forgery." [Online]. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29

[18] CERT, "Ftp bounce attacks." [Online]. Available: http://www.cert.org/historical/advisories/CA-97.27.FTP_bounce.cfm

[19] L. Huang, A. Moshchuk, H. J. Wang, S. Schecter, and C. Jackson, "Clickjacking: Attacks and defenses," in *USENIX Security Symposium*, 2012.

[20] P. Li, Y. Mao, and S. Zdancewic, "Information integrity policies," in *Workshop on Formal Aspects in Security and Trust (FAST)*, 2003.

[21] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[22] C. Fournet and T. Rezk, "Cryptographically sound implementations for typed information-flow security," in *ACM Symposium on Principles of Programming Languages (POPL)*, 2008.

[23] S. Saghafi, K. Fisler, and S. Krishnamurthi, "Features and object capabilities: Reconciling two visions of modularity," in *International Conference on Aspect-oriented Software Development (AOSD)*, 2012.

[24] G. Boudol, "Secure information flow as a safety property," in *Workshop on Formal Aspects in Security and Trust (FAST)*, 2008.

[25] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy (Oakland)*, 1982.

[26] A. Mettler, D. Wagner, and T. Close, "Joe-E: A security-oriented subset of Java," in *Network and Distributed System Security Symposium (NDSS)*, 2010.

[27] A. Chander, J. C. Mitchell, and D. Dean, "A state-transition model of trust management and access control," in *IEEE Computer Security Foundations Workshop (CSFW)*, 2001.

[28] T. C. Murray and G. Lowe, "Analysing the information flow properties of object-capability patterns," in *Formal Aspects in Security and Trust (FAST)*, 2009.

[29] S. Drossopoulou and J. Noble, "How to break the bank: Semantics of capability policies," in *International Conference on Integrated Formal Methods (iFM)*, 2014.

[30] C. Dimoulas, S. Moore, A. Askarov, and S. Chong, "Declarative policies for capability control," in *IEEE Computer Security Foundations Symposium (CSF)*, 2014.

[31] S. Drossopoulou, J. Noble, and M. S. Miller, "Swapsies on the Internet: First steps towards reasoning about risk and trust in an open world," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2015.

[32] A. Birgisson, A. Russo, and A. Sabelfeld, "Capabilities for information flow," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2011.

Hybrid Monitoring of Attacker Knowledge

Frédéric Besson, Nataliia Bielova and Thomas Jensen
Inria, France

***Abstract*—Enforcement of noninterference requires proving that an attacker's knowledge about the initial state remains the same after observing a program's public output. We propose a hybrid monitoring mechanism which dynamically evaluates the knowledge that is contained in program variables. To get a precise estimate of the knowledge, the monitor statically analyses non-executed branches. We show that our knowledge-based monitor can be combined with existing dynamic monitors for non-interference. A distinguishing feature of such a combination is that the combined monitor is provably more permissive than each mechanism taken separately. We demonstrate this by proposing a knowledge-enhanced version of a no-sensitive-upgrade (NSU) monitor. The monitor and its static analysis have been formalized and proved correct within the Coq proof assistant.**

I. Introduction

Information-flow control provides a promise of a strong information security property [24]. Today most research has focused on *monitors for noninterference* [4], [5], [18], [25], that block executions where secret inputs flow into public outputs. Such flow can happen due to explicit or implicit information flow. An explicit flow occurs when *secret* information is stored in a *public* variable visible to an attacker. An implicit flow happens when assignments to public variables are made under *secret control* (*i.e.*, following a test on a secret variable), like in Program 1.[1]

```
1 l = 0; if (h) then l = 1; output l
```
Program 1

There is an implicit flow from h to l because by observing the value of l the attacker can deduce the secret value h.

Dynamic monitors control one execution of the program and propagate a security label to each program variable. If the monitor suspects a possible flow (explicit or implicit) from secret inputs to a variable labeled as public, it blocks the execution. Such *purely dynamic information flow control* was first proposed by Fenton [11] and has recently regained interest [5], [25] for (at least) two reasons. First, some languages, such as JavaScript, are so dynamic that a precise static analysis is practically impossible. Therefore, an attractive alternative is to resort to dynamic monitoring, either by extending an interpreter for the language or by inlining a monitor in the program. Second, even if a program may have some insecure executions, there may be other executions that are perfectly secure. While a static analysis would reject such programs, dynamic monitors can identify and allow some secure executions of insecure programs.

However, dynamic monitors also have several limitations, due to the fact that they analyse only one execution of a program. As a result, they make the worst-case assumption about what happens later on in the current execution, and what could happen in other executions. Dynamic information flow control first proposed by Zdancewic [25] and later used by Austin and Flanagan [4] is based on the *no-sensitive-upgrade* (NSU) principle: it halts an execution when a public variable gets assigned under secret control. This principle severely limits the *permissiveness* of dynamic monitors in certain cases.

- They may block executions too early: if later a variable is updated, then there is no information leakage.

```
1 if (h) then l = 1;
2 l = 0;
3 output l
```
Program 2

Program 2 is secure but its execution is blocked by NSU when $h = true$.

- If the variable is assigned the same value on both branches, there is no leakage. Program 3 is secure but NSU blocks all its executions.

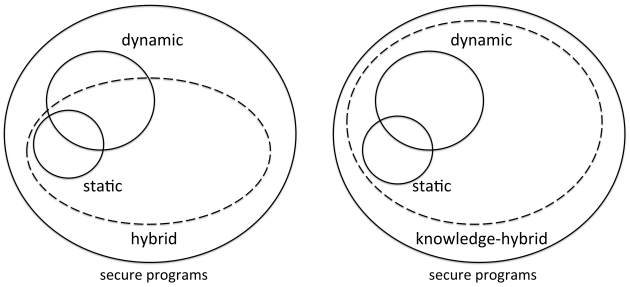
```
1 x = 1;
2 if (h) then l = 1 else l = x;
3 output l
```
Program 3

The first problem of blocking execution too early was addressed by the *permissive-upgrade* (PU) principle [5], by introducing a special *"partially leaked"* label. The second problem requires knowledge about other executions and has motivated a strand of research in hybrid information flow monitors that combine static and dynamic analysis.

Hybrid monitors (e.g. [7], [17], [18], [23]) analyse the source code of each non-executed branch under secret control to detect possible implicit flows. A dynamic monitor can be enhanced with a variety of static analyses. Le Guernic *et al.* [17]–[19] proposed the first combination of dynamic information flow monitors with static dependency analyses. Besson *et al.* [7] extended this work to a more sophisticated constant propagation and dependency analysis. Recently, Hedin *et al.* have shown how to improve their dynamic information flow analysis for JavaScript [14] with a points-to analysis [12].

This research has been partially supported by the French ANR projects AJACS ANR-14-CE28-0008 and ANR-10-LABX-07-01 Laboratoire d'excellence Comin Labs.

[1] In all examples, variables with names starting with "h" are secret, and all the other variables are public.



(a) from Hedin *et al.* [12]. (b) our result

Fig. 1: Relative permissiveness revisited.

Permissiveness. All the monitors discussed above provably enforce noninterference, however some of them may block more program executions than others. Intuitively, *permissiveness* defines how many program executions are accepted by the monitor even if the program may be insecure. Hedin *et al.* [12] have proven that purely dynamic and hybrid monitors are incomparable in their permissiveness. For example, a dynamic NSU monitor allows execution of Program 1 when $h = \mathit{false}$, while the hybrid monitor stops it. On the other hand, the dynamic NSU monitor will stop execution of Program 2 when $h = \mathit{true}$ while the hybrid monitor will allow it. Figure 1a graphically shows the secure programs, and programs, for which the static, dynamic and hybrid analysis identify as secure all its executions (from Hedin *et al.* [12]).

In this paper, we propose a knowledge-based hybrid monitor that is able to reach a level of permissiveness that was deemed impossible for standard hybrid monitors. Figure 1b graphically shows that all the executions of all the secure programs that are accepted by dynamic monitors, are also accepted by an enhanced version of our knowledge-based hybrid monitor.

Modelling attacker knowledge. Basic information flow control detects whether or not a program execution may leak information, but will not provide a more precise description of *what information is being leaked*, or equivalently, *what knowledge an attacker gains from an observation*. However, switching to such knowledge-based analysis can provide a finer control over information flow. Previously [7], the authors have proposed a hybrid information flow control that combines dynamic monitoring and static analysis. This technique computes the leakage of a *concrete program execution* by labeling every program variable with a logical formula over the secret inputs. This formula is a logical description of the knowledge that an attacker can deduce about the initial state of the program when observing the value of a variable.

Program 4

```
1 x = 0; y = 0;
2 if (h1) then y = 1;
3 if (h2) then x = 1 else x = y;
4 output x
```

Consider Program 4 and its execution when $h1 = \mathit{false}$ and $h2 = \mathit{true}$. When Program 4 outputs 1, the attacker learns that either $h1$ or $h2$ was true:

$$\text{Attacker knowledge: } h1 \vee h2$$

Given the same initial memory where $h1 = \mathit{false}$ and $h2 = \mathit{true}$, the hybrid monitor of Besson *et al.* [7] associates the knowledge to the variable after analysing each test: the first test on $h1$ fails, thus the value of y remains unchanged and the knowledge of y is $\neg h1$; upon the second test, the monitor concludes that the value of x is 1 in the true branch and 0 in the false branch. As the values are not the same, the knowledge of the output $x = 1$ might depend on $h2$ and on the knowledge of y. Therefore, the knowledge computed by the monitor is:

$$\text{Approximated knowledge: } \neg h1 \wedge h2$$

As is readily seen, the approximated knowledge is much less precise than the real attacker knowledge. The reason for this gap between estimated and actual knowledge is in the choice of the model of knowledge domain. Intuitively, the knowledge in [7] is limited to a set of environments that can contribute to the *current value of* x.

A more expressive knowledge domain. In this paper, we propose a more general representation of attacker knowledge: the knowledge associated with a variable x is the set of environments that lead to a particular value of x *for several possible values of* x. In other words, the knowledge in x groups the initial environments into equivalence classes, such that two environments are equivalent if they lead to the same value of x. This models much more of the input-output relation of the program.

This new knowledge domain leads to several advantages over the existing previous work [7]. The first advantage is that it empowers the monitor to reason about several executions, and hence *to prove noninterference in more cases* than in previous work [7]. For a concrete example, see Example 1 in Section IV-B. With the previous knowledge representation in [7], we would infer that z depends on x and y; while with the new representation we prove that x and y do not interfere with z.

The second advantage of this more expressive knowledge domain is that it *enables a composition with other monitors*, and we demonstrate such composition with the no-sensitive upgrade (NSU) monitor. The new knowledge domain allows the hybrid monitor to reason about the other executions that would be blocked by the other dynamic monitor for non-executed paths. This leads to a composition of monitors that is strictly more permissive that each monitor separately. We summarise this result in Figure 1b, showing that the knowledge-based hybrid monitor accepts more secure executions than any purely dynamic monitor it is built upon[2].

[2]Notice that the permissiveness result with respect to static analysis is achieved by transitivity of permissiveness: a knowledge-based hybrid monitor is provably more permissive than a standard hybrid monitor, and a standard hybrid monitor is more permissive than a static analysis of Hunt and Sands [15] (see Thm. 3 of [23]).

The third advantage is that the proposed knowledge domain allows us to design a *more precise static analysis*. Our knowledge domain is used for both the dynamic analysis of the executed branch and the static analysis of non-executed branched. This gives a pleasant uniformity to the theory and provides a more general framework, compared to the *ad-hoc* static analyses for the non-executed branches in [7].

Contributions.

- We propose a hybrid monitor that computes the knowledge of the attacker. Our monitor combines a dynamic analysis with a static analysis of the non-executed branches. The knowledge domain allows the monitor to compute an attacker's knowledge more precisely than in previous works [7].
- The knowledge-based hybrid monitor is proved to be correct (it safely over-approximates the attacker's knowledge) and sound (it can be used to enforce non-interference). The proof has been formalized in the Coq proof assistant [1].
- The proposed monitor can be combined with existing dynamic or hybrid monitors for non-interference. A distinguishing feature of such a combination is that the combined monitor is provably more permissive than the monitor it builds upon.
- We have proposed an effective and symbolic representation of knowledge and implemented the computation of knowledge as part of our Coq formalization. The results reported in this paper are all computed with this implementation.

II. Attacker knowledge and non-interference

A. Attacker model

We consider a classical attacker model, following the definition of *gadget attacker* [6]. An attacker provides the program source code and this program runs in an environment that contains secret information, producing some outputs, observable to the attacker.

B. Attacker knowledge

Given an observation at the end of an execution, an *attacker knowledge* is the set of all possible input environments that can lead to that observation. This naturally induces an equivalence relation on input environments. Landauer and Redmond [16] propose a lattice of equivalence classes of environments for representing the knowledge of an attacker. Askarov and Sabelfeld [3] and Askarov and Chong [2] give a characterisation of non-interference in terms of attacker knowledge. The remainder of this section summarizes notions and results from these papers.

We assume a security policy in the form of a lattice of two security levels $(\{\mathsf{L},\mathsf{H}\},\sqsubseteq)$, where $\mathsf{L}\sqsubseteq\mathsf{H}$ and we use $\sqcup$ as the least upper bound. A labelling function Γ assigns security levels to all program variables. We write ρ_L for the L-projection of the environment ρ onto those variables x whose level is lower than L, *i.e.*, for which $\Gamma(x)\sqsubseteq\mathsf{L}$, and in the future notations we drop an implicit labelling function Γ. We write $[\rho]_\mathsf{L}$ for the set of environments that agree with ρ on low variables: $[\rho]_\mathsf{L}=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\}$.

The program semantics is given by a relation $(P,\rho)\downarrow v$, where P is a program that produces output v at the end of the execution. This output is visible to the attacker. The knowledge is defined as the set of low-equivalent environments that can produce the same output v.

Definition 1 (Attacker knowledge)**.** Given a program P, an initial environment ρ, and a final observation v, the *attacker knowledge* is the set of environments that agree with ρ on low variables and leads to the observation of v:

$$\mathcal{K}^{\downarrow}(P,v,\rho)=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\wedge(P,\rho')\downarrow v\}.$$

Notice that a smaller knowledge set represents fewer possible inputs that produce the same program output, thus a smaller set corresponds to a bigger amount of information. Therefore, a smaller knowledge set is a safe approximation of the actual attacker knowledge.

C. Termination-Insensitive Noninterference

In the following, we describe the relationship between knowledge and the standard notion of noninterference. We shall focus on *termination-insensitive noninterference (TINI)*, and hence restrict attention to a termination-insensitive version of knowledge that only considers environments in which the program terminates and where the program output is visible to the attacker. The knowledge obtained just from observing termination, given an initial observation ρ_L is called *initial attacker knowledge*.

Definition 2 (Initial attacker knowledge)**.** Given program P and an environment ρ, the *initial attacker knowledge* is:

$$\mathcal{I}^{\downarrow}(P,\rho)=\{\rho' \mid \rho_\mathsf{L}=\rho'_\mathsf{L}\wedge\exists v.(P,\rho')\downarrow v]\}.$$

Later on, we omit the superscript $\downarrow$ when it can be inferred from the context.

The security condition states that the attacker's knowledge should not grow with the new observation produced by the program execution.

Definition 3 (Knowledge-based security for input environment ρ)**.** Program P is *secure* for an initial input environment ρ if whenever $(P,\rho)\downarrow v$ then

$$\mathcal{K}(P,v,\rho)=\mathcal{I}(P,\rho).$$

Notice that if the program P does not terminate in an environment ρ, then P is considered secure for ρ. However, the program still might be insecure for any other low-equal environment, in which the program terminates.

The standard notion of termination-insensitive noninterference (TINI) is stated by comparing pairs of low-equivalent initial environments.

Definition 4 (TINI)**.** A program P is *termination-insensitively noninterferent (TINI)* if whenever $\rho^1_\mathsf{L}=\rho^2_\mathsf{L}$, and $(P,\rho^1)\downarrow v_1$ and $(P,\rho^2)\downarrow v_2$, then $v_1=v_2$.

Askarov and Sabelfeld [3, Prop. 2] have shown that there is an equivalence between the knowledge-based security and TINI for a lattice with two elements.

Lemma 1. *A program P satisfies TINI if and only if P is secure for all initial environments ρ.*

III. Preliminary definitions

A. Language

We use a simple untyped imperative language extended with a specific output command $\mathsf{output}\ x$ which evaluates the variable x and outputs its value. This output is visible to the attacker at security level L. All the commands of the language are standard, except perhaps for the $\mathsf{assume}(e)$ operator, which evaluates e and continues or halts an execution depending on whether its value is true or false. The syntax of this language is as follows:

$$\mathbb{P} \ni P ::= c; \mathsf{output}\ x \qquad \mathbb{E} \ni e ::= n \mid x \mid e_1 \oplus e_2 \mid \neg e$$
$$\mathbb{C} \ni c ::= \mathsf{skip} \mid x := e \mid c_1; c_2 \mid \mathsf{assume}(\mathrm{e}) \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c$$

The set of expressions contains the usual numeric and Boolean expressions. Every expression can be interpreted as a boolean value, and hence conditional commands take an arbitrary expression as condition. The exact interpretation of expressions as booleans is, however, not essential to the results in this paper and will be left unspecified. We use $\oplus$ to denote an arbitrary binary operator.

An environment $\rho \in Env = Var \to \mathbb{V}$ maps variables to values. The big-step program semantics is presented in Figure 2. The semantics of commands is denoted by a binary relation $(c, \rho) \downarrow \rho'$ meaning that command c when executed in environment ρ will evaluate to ρ' and the semantics of programs is denoted by $(P, \rho) \downarrow v$ meaning that program P when executed in environment ρ will produce an output v.

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, \rho) \downarrow \rho} \qquad \textsc{assign}\ \frac{}{(x := e, \rho) \downarrow \rho[x \mapsto [\![e]\!]_\rho]}$$

$$\textsc{seq}\ \frac{(c_1, \rho) \downarrow \rho' \quad (c_2, \rho') \downarrow \rho''}{(c_1; c_2, \rho) \downarrow \rho''} \qquad \textsc{assume}\ \frac{C[\![e]\!]_\rho = tt}{(\mathsf{assume}(e), \rho) \downarrow \rho}$$

$$\textsc{if}\ \frac{C[\![e]\!]_\rho = \alpha \qquad (c_\alpha, \rho) \downarrow \rho'}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, \rho) \downarrow \rho'}$$

$$\textsc{while}\ \frac{(\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, \rho) \downarrow \rho'}{(\mathsf{while}\ e\ \mathsf{do}\ c, \rho) \downarrow \rho'}$$

$$\textsc{output}\ \frac{(c, \rho) \downarrow \rho' \qquad [\![x]\!]_{\rho'} = v}{(c; \mathsf{output}\ x, \rho) \downarrow v}$$

where $[\![x]\!]_\rho = \rho(x) \quad [\![n]\!]_\rho = n \quad [\![e_1 \oplus e_2]\!]_\rho = [\![e_1]\!]_\rho \oplus [\![e_2]\!]_\rho$

Fig. 2: Language semantics

Let $\mathbb{B}$ denote the set of boolean values. To accommodate the fact that any expression can be used as a condition, we assume a function $C : \mathbb{V} \to \mathbb{B}$ that specifies how each value is interpreted as a boolean. C satisfies the following two constraints which ensure that $\neg e$ represents the negation of the expression e and $e = e'$ models the fact that e and e' evaluates to the same value:

$$C([\![\neg e]\!]_\rho) = \neg C([\![e]\!]_\rho) \quad C([\![e = e']\!]_\rho) = ([\![e]\!]_\rho = [\![e']\!]_\rho)$$

B. Notations

Given sets A and V, we write $V^\sharp$ for $V \cup \{\bot, \top\}$. In domains of the form $V^\sharp$ we write $\lfloor v \rfloor$ to assert that v is an element that is neither $\bot$ nor $\top$. For a function $f \in A \to V^\sharp$, we write $f(a) = \lfloor v \rfloor$ for $f(a) = v \wedge v \notin \{\bot, \top\}$ and $f^{\text{-}1}(v) = \{a \mid f(a) = \lfloor v \rfloor\}$ for the pre-image of v.

Given a domain $V^\sharp$, we get a flat lattice $(V^\sharp, \preccurlyeq, \bot, \top)$ such that $\bot \preccurlyeq x$, $x \preccurlyeq \top$ and $\lfloor x \rfloor \preccurlyeq \lfloor x \rfloor$. Given x, y, we write $x \curlyvee y$ (resp. $x \curlywedge y$) the the least upper bound (resp. greatest lower bound) of x and y. The ordering, least upper bound and greatest lower bound are lifted to functions in the standard pointwise fashion: $f \preccurlyeq g$ iff $\forall x, f(x) \preccurlyeq g(x)$; $(f \curlyvee g)(x) = f(x) \curlyvee g(x)$; $(f \curlywedge g)(x) = f(x) \curlywedge g(x)$.

Given a unary operator $o : V \to W$, we define an operator $o^\sharp : V^\sharp \to W^\sharp$ as follows

$$o^\sharp(x) = \text{if } x \in \{\bot, \top\} \text{ then } x \text{ else } o(x).$$

Similarly, for a binary operator $\oplus : V \to V \to V$ we define $\oplus^\sharp : V^\sharp \to V^\sharp \to V^\sharp$ as

$$x \oplus^\sharp y = \begin{cases} v_1 \oplus v_2 & \text{if } x = \lfloor v_1 \rfloor \wedge y = \lfloor v_2 \rfloor \\ \bot & \text{if } x = \bot \vee y = \bot \\ \top & \text{otherwise.} \end{cases}$$

We define a binary operator $assume : \mathbb{B}^\sharp \to V^\sharp \to V^\sharp$ which returns its second argument if the first argument is either $true$ or $\top$, and undefined otherwise.

$$assume(b, v) = \text{if } (b = true \vee b = \top) \text{ then } v \text{ else } \bot.$$

Finally, we define a conditional operator $if : \mathbb{B}^\sharp \to V^\sharp \to V^\sharp \to V^\sharp$ built from $assume$, where $\neg^\sharp$ is a standard negation operator extended to the domain $\mathbb{B}^\sharp$:

$$if(b, v, v') = assume(b, v) \curlyvee assume(\neg^\sharp b, v').$$

Given $c : A \to \mathbb{V}^\sharp$ and $f, g \in A \to \mathbb{V}^\sharp$, we write $Assume(c, f)$ and $IF(c, f, g)$ for the assume and conditional operators lifted to functions:

$$\begin{aligned} Assume(c, f)(a) &= assume(C^\sharp(c(a)), f(a)) \\ IF(c, f, g)(a) &= if(C^\sharp(c(a)), f(a), g(a)). \end{aligned}$$

where $C^\sharp$ is the function C lifted to the domain $\mathbb{V}^\sharp \to \mathbb{B}^\sharp$.

IV. A hybrid knowledge-based monitor

Our hybrid information flow analysis computes the knowledge of a program's input-output behaviour that the attacker obtains by observing the result of a given execution of the program. More precisely, we shall define the domain $\mathbf{K}$ of knowledge to be the set of functions K that maps environments ρ to values v, with the intention that $K(\rho) = v$ if the program when started in

initial environment ρ will either produce an output v or not terminate. If the hybrid monitor from initial state ρ calculates final value v and knowledge K, then $K^{-1}(v)$ is a safe approximation of the set of all the environments that produce v. This set will allow us to safely approximate the attacker's knowledge (see Theorem 1).

Definition 5 (Knowledge)**.** The domain $\mathbf{K}$ of knowledge is defined by:

$$\mathbf{K} = \mathit{Env} \to \mathbb{V}^\sharp.$$

Notice that the monitor may compute knowledge $K \in \mathbf{K}$ that will map an environment to $\bot$ or $\top$. This is due to the presence of the static analysis. If $K(\rho) = \bot$ then the static analysis has established that computation started in ρ will not terminate. On the other hand, $K(\rho) = \top$ means that approximations in the static analysis made it impossible for the monitor to determine what value will result from an execution starting in ρ. This means that ρ cannot be added to the knowledge set $K^{-1}(v)$ for any possible output v. Recall that a knowledge analysis may always safely under-approximate the knowledge set $K^{-1}(v)$ of output v, so having $K(\rho) = \top$ for some ρ makes the knowledge analysis more conservative than an analysis that is capable of determining the exact output for ρ.

A. Monitor semantics

We now define a hybrid knowledge monitor that combines a dynamic monitor with a static analysis. The hybrid monitor executes the program and, at the same time, computes an over-approximation of the knowledge of the initial state that can be deduced from the current state at a given point in the execution. The semantic state of the hybrid monitor is thus a pair (ρ, κ), where the first component is either an environment Env containing the current values of the variables, or an empty environment, $\cdot$, that will be used by the static analysis. The second component $\kappa \in \mathit{Var} \to \mathbf{K}$ is an environment containing the knowledge present in each variable. At branching points, the monitor will execute one branch and will statically analyse the other, non-executed branch. This static analysis will help refining the computation of the actual knowledge stored in variables.

Given a concrete initial environment ρ, the initial state of the hybrid monitor $\mathit{init}(\rho)$ is such that each variable x has the knowledge of the current value of x in the initial environment: $\mathit{init}(\rho) = (\rho, \kappa_0)$ with $\kappa_0 = \lambda x.\lambda\rho'.\rho'(x)$. To see this, suppose that the program immediately outputs the value v of variable x *i.e.* $v = \rho(x)$. The set of environments $\{\rho' \mid \rho'(x) = v\}$ that produce v is modeled exactly by $\kappa_0(x)^{-1}(v)$.

From the initial state, the monitor executes according to the rules of Figure 3. The concrete execution and the static analysis are combined into one reduction relation $\Downarrow$. The rules SKIP and SEQ are standard. For the other language constructs, there are two rules: a dynamic rule describing the monitored execution of the construct, and a static rule describing the static analysis of it. The dynamic rules will operate on environments with the actual values of the variables. The static analysis, on the other hand, is intended to provide information about all other possible executions so it will not have information about concrete values. In the formalization, this means that the static rules apply only when the environment is undefined (denoted by $\cdot$).

The two rules ASSIGNDYN and ASSIGNSTAT for assignment use the function $(\!|\_|\!)_\kappa$ to evaluate the knowledge about the initial environment contained in the value of the expression e. The function takes the current knowledge environment κ as parameter. In addition, the dynamic rule updates the value of x in the environment ρ.

The rule IFDYN describes the monitored execution of conditional statements of form $\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}$. The outcome of the test α is the value of the expression e computed in the environment ρ and the appropriate branch is executed with that environment, producing a new environment ρ' and a new knowledge environment κ_α. The non-executed branch $c_{\bar{\alpha}}$ is statically analysed, using an undefined environment of values and the current knowledge environment. The knowledge environment $\kappa_{\bar{\alpha}}$ obtained from this static analysis must be combined with the knowledge environment from the execution κ_α. To this end, we construct the function $\mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff})$ that uses a conditional operator $\mathit{IF}(c, f, g)$ from Section III-B. We later show that in programs without loops the IF operator allows us to precisely model the attacker's knowledge.

For the while loop, the dynamic rules WHILEDYNTRUE and WHILEDYNFALSE are standard unfolding semantic rules that apply when the environment is defined. The static rule WHILESTAT states that any s' whose knowledge safely approximates the knowledge before entering the loop (condition $s \preccurlyeq s'$) as well as the knowledge after executing the body of the loop (condition $s_1 \preccurlyeq s'$) is a valid result of the static analysis of the loop. The rule leaves room for an actual implementation to compute more or less precise approximations of the attacker knowledge after a loop. Our implementation (Section VII) employs an iterative fixpoint computation to this end.

We do not define a rule for analysing the output command $\mathsf{output}\ x$, because it does not change the knowledge of any variable. The $\mathsf{output}\ x$ command is important because it is at this point that we must decide what to output and, hence, what security property to enforce. In Section V, we shall propose rules for the output command that will enforce enforce non-interference.

B. Examples

To illustrate the expressive power of our hybrid monitor, we provide a number of examples. They show when the monitor computes precise knowledge but also limitations due to the static analysis of loops. They also illustrate the role played by the static detection of termination.

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, s) \Downarrow s} \qquad \textsc{seq}\ \frac{(c_1, s) \Downarrow s' \qquad (c_2, s') \Downarrow s''}{(c_1; c_2, s) \Downarrow s''}$$

$$\textsc{assignDyn}\ \frac{[\![e]\!]_\rho = v \quad (\!|e|\!)_\kappa = e^\sharp}{(x := e, (\rho, \kappa)) \Downarrow (\rho[x \mapsto v], \kappa[x \mapsto e^\sharp])} \qquad \textsc{assignStat}\ \frac{(\!|e|\!)_\kappa = e^\sharp}{(x := e, (\cdot, \kappa)) \Downarrow (\cdot, \kappa[x \mapsto e^\sharp])}$$

$$\textsc{assumeDyn}\ \frac{C[\![e]\!]_\rho = tt}{(\mathsf{assume}(e), (\rho, \kappa)) \Downarrow (\rho, \mathbb{A}((\!|e|\!)_\kappa, \kappa))} \qquad \textsc{assumeStat}\ \frac{}{(\mathsf{assume}(e), (\cdot, \kappa)) \Downarrow (\cdot, \mathbb{A}((\!|e|\!)_\kappa, \kappa))}$$

$$\textsc{ifDyn}\ \frac{C[\![e]\!]_\rho = \alpha \qquad (c_\alpha, (\rho, \kappa)) \Downarrow (\rho', \kappa_\alpha) \qquad (c_{\bar{\alpha}}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{\bar{\alpha}})}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, (\rho, \kappa)) \Downarrow (\rho', \mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff}))}$$

$$\textsc{ifStat}\ \frac{(c_{tt}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{tt}) \qquad (c_{ff}, (\cdot, \kappa)) \Downarrow (\cdot, \kappa_{ff})}{(\mathsf{if}\ e\ \mathsf{then}\ c_{tt}\ \mathsf{else}\ c_{ff}, (\cdot, \kappa)) \Downarrow (\cdot, \mathbb{IF}((\!|e|\!)_\kappa, \kappa_{tt}, \kappa_{ff}))}$$

$$\textsc{whileDynTrue}\ \frac{C[\![e]\!]_\rho = tt \qquad (\mathsf{if}\ e\ \mathsf{then}\ c; \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, s) \Downarrow s' \qquad s = (\rho, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s'}$$

$$\textsc{whileDynFalse}\ \frac{C[\![e]\!]_\rho = ff \qquad (\mathsf{if}\ e\ \mathsf{then}\ \mathsf{while}\ e\ \mathsf{do}\ c\ \mathsf{else}\ \mathsf{skip}, s) \Downarrow s' \qquad s = (\rho, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s'}$$

$$\textsc{whileStat}\ \frac{(\mathsf{assume}(e); c, s') \Downarrow s_1 \qquad s_1 \preccurlyeq s' \qquad s \preccurlyeq s' \qquad (\mathsf{assume}(\neg e), s') \Downarrow s'' \qquad s = (\cdot, \kappa)}{(\mathsf{while}\ e\ \mathsf{do}\ c, s) \Downarrow s''}$$

where $(\!|x|\!)_\kappa = \kappa(x) \quad (\!|n|\!)_\kappa = \lambda\rho.\lfloor n \rfloor \quad (\!|e_1 \oplus e_2|\!)_\kappa = \lambda\rho.(\!|e_1|\!)_\kappa(\rho) \oplus^\sharp (\!|e_2|\!)_\kappa(\rho)$

$$\mathbb{IF}(c, \kappa_1, \kappa_2)(x) = IF(c, \kappa_1(x), \kappa_2(x)) \qquad \mathbb{A}(c, \kappa)(x) = Assume(c, \kappa(x))$$

$$(\rho, \kappa) \preccurlyeq (\rho', \kappa') \text{ iff } \kappa \preccurlyeq \kappa' \wedge \rho = \rho'$$

Fig. 3: Hybrid knowledge analysis semantics.

```
1 if h then z := x + y                    Program 5
2 else z := y - x;
3 output z
```

Example 1 (Precise knowledge computation)**.** For Program 5, the hybrid monitor computes $\kappa(z) = \lambda\rho.if(C[\![h]\!]_\rho, [\![x+y]\!]_\rho, [\![y-x]\!]_\rho)$. The program is loop-free and therefore $\kappa(z)$ is a function which encodes exactly the function computing the final value of z from the initial environment. Suppose that the final value of z is 1, the knowledge of the output 1 is obtained by $\kappa(z)^{-1}(1) = \{\rho \mid if(C[\![h]\!]_\rho, [\![x+y]\!]_\rho, [\![y-x]\!]_\rho) = 1\}$.

Suppose that initially $h = true$, $x = 0$ and $y = 1$. As a result, L-equivalent environments are $\{\rho \mid \rho(x) = 0 \wedge \rho(y) = 1\}$[3]. This program is indeed secure for the given initial environment (see Definition 3) since all L-equivalent environments output the value 1. Notice that all of them are included in $\kappa(z)^{-1}(1)$: if $x = 0$ and $y = 1$ then the condition $if(h, x+y, y-x) = 1$ always holds: $if(h, 0+1, 1-0) = 1 \Leftrightarrow if(h, 1, 1) = 1 \Leftrightarrow 1 = 1$.

[3]As z is set in both branches, its initial value is irrelevant.

Notice that both dynamic monitors and the standard hybrid monitors block all executions of this program either because there is a low assignment under a high security context in both branches, or because an output variable z explicitly depends on h. We will show in Section VI-C that this power of proving non-interference allows our monitor to be more permissive than other monitors.

Example 2 (Detection of loop non-termination)**.** Interestingly, our monitor may be more precise than other monitors even in the presence of loops in a high security context. Consider Program 6.

```
1 l := 0;                                 Program 6
2 if h then skip
3 else while true do l := 1;
4 output l
```

When h is $true$, the purely dynamic monitors would accept this execution, while the previous hybrid monitors [7], [18] would block it since the hybrid monitor would detect that a value of l might change in the non-executed branch.

However, our monitor is able to detect the nontermination of the while loop. On line 2, we apply the IfDyn

rule, and compute $\kappa(l) = \lambda\rho.if(C[\![h]\!]_\rho, \kappa_0(l), \kappa''(l))$, where $\kappa_0(l) = \lambda\rho.0$ and κ'' is computed by the WhileStat rule. The first three premises of this rule ensure that in state $s' = (\bot, \kappa')$, we have $\kappa'(l) = \lambda\rho.\top$ since l is updated in the loop body. However, the forth premise $((\mathsf{assume}(\neg e), s') \Downarrow s'')$ ensures that $\kappa''(l) = \lambda\rho.\bot$ thus being able to conclude that whenever h is $false$, the program does not terminate.

Example 2 shows that our static analysis allows us to detect non-termination of the loops in some cases. Notice that this capability does not give us soundness for termination-sensitive noninterference, but gives more precision for termination-insensitive noninterference. In contrast, the knowledge monitor in our previous work [7] is not able to prove termination-insensitive noninterference for Example 2 since its static analysis only determines that the output `l` may depend on the secret `h`.

C. Limitations

Our hybrid monitor is not always capable of computing the exact knowledge. A fundamental reason is that the knowledge is computed for each variable independently. Therefore, it cannot express a relation, e.g., the equality of variables.

Example 3 (Imprecise knowledge computation)**.** For Program 7 and its execution when $h = true$, our static analysis does not infer that, at the end of the loop y is equal to x. Hence, it fails at deducing that in the other branch y is 0.

Program 7

```
1 y = 1; x = N;
2 if h then skip;
3 else while x > 0 do x = x-1; y = x;
4 output y
```

When h is true, we statically analyse the while-loop. The loop invariant is $s' = (\cdot, \kappa')$, where κ' defines a knowledge for each variable. To model the fact that x is decremented at each iteration, the static analysis computes

$$\kappa'(x) = \bigvee_{0 \le n \le N} \lambda\rho.n = \lambda\rho.\top.$$

This information is propagated towards y by the assignment $y = x$ and we get $\kappa'(y) = \lambda\rho.\top$. At the end of the loop, the test $\neg x > 0$ *i.e.,* $x = 0$ – providing x is a natural integer – allows to recover the fact that x is necessarily 0. However, as the equality between x and y is not propagated, the value of y cannot be recovered. As a result, the final knowledge in y is $\kappa(y) = \lambda\rho.if(C[\![h]\!]_\rho, 1, \top)$, while the real attacker knowledge is $\lambda\rho.if(C[\![h]\!]_\rho, 1, 0)$.

As a result of the imprecise knowledge computation, the static analysis is not always capable to detect the loop termination. If we have $\kappa(x)(\rho) = \bot$, we know for certain that the program does not terminate for initial environment ρ. However, if $\kappa(x)(\rho) = v$ for some v, there is no certainty. The program either terminates and the value is indeed v or the program does not terminate. Said otherwise, any non-terminating execution from initial environment ρ can soundly approximated by $\kappa(x)(\rho) = v$.

Example 4 (Non-detection of loop non-termination)**.** Consider the program obtained as the sequential composition of Program 7 followed by Program 8. This new composed program is noninterferent (TINI) since it either outputs 1 or does not terminate.

Program 8

```
1 x = 1;
2 while y = 0 do x = 1;
3 output x
```

After an execution of the Program 7 in the initial environment where $h = true$, the computed knowledge in variable y is $\kappa(y) = \lambda\rho.if(C[\![h]\!]_\rho, 1, \top)$.

The static analysis of the while loop detects that the value of x in the loop body is always 1 and the knowledge in variable x is $\kappa(x) = \lambda\rho.1$. However, since the knowledge in y is not precise, the static analysis is unable to determine the non-termination of the loop.

V. Correctness and Soundness

Given a monitor's knowledge κ, a variable x and its value v, we can express the set of possible environments that can produce v as the inverse of κ: $\kappa(x)^{-1}(v)$. The Monitor Correctness Theorem 1 states that this set of environments intersected with the low-equivalence class $[\rho_i]_\mathsf{L}$ is a correct approximation of the attacker knowledge for environment ρ_i, as defined in Definition 1.

Theorem 1 (Monitor Correctness)**.** *Let $c \in \mathbb{C}, \rho, \rho' \in Env, \kappa \in Var \to \mathbf{K}$ and assume that $(c, init(\rho)) \Downarrow (\rho', \kappa)$. Then for all $v \in \mathbb{V}$, $x \in Var$ and $\rho_i \in Env$,*

$$\kappa(x)^{-1}(v) \cap [\rho_i]_\mathsf{L} \subseteq \mathcal{K}(c; \mathsf{output}\ x, v, \rho_i).$$

Proof sketch. The Correctness Theorem is a consequence of a more general, inductive invariant. It states that if we are given sound knowledge κ about a program c_0 executed in environment ρ then monitoring another program c with κ as initial knowledge will produce final knowledge κ' which is sound for the sequential composition $c_0; c$ when executed in environment ρ. To state this invariant, we define the predicate *soundK* which states that the knowledge $\kappa \in Var \to \mathbf{K}$ is sound for an execution of program c in initial environment ρ_i. Formally, we write $soundK(c, \rho_i, \kappa)$ if for all $v \in \mathbb{V}$, $x \in Var$ and $\rho_f \in Env$,

$$\left.\begin{array}{c} \rho_i \in \kappa(x)^{-1}(v) \\ \wedge \\ (c, \rho_i) \downarrow \rho_f \end{array}\right\} \Rightarrow \rho_f(x) = v.$$

The invariant can then be stated as follows:

$$\left.\begin{array}{c} (c, (\rho_0, \kappa)) \Downarrow (\rho_1, \kappa') \\ \wedge \\ soundK(c_0, \rho, \kappa) \end{array}\right\} \Rightarrow soundK(c_0; c, \rho, \kappa').$$

The proof of this invariant is by structural induction over the relation $\Downarrow$ and by case analysis over the hybrid

monitoring rules of Figure 3. Instantiating this invariant with $c_0 = \mathsf{skip}$ and $\kappa = \kappa_0$, we get that

$$(c, \text{init}(\rho)) \Downarrow (\rho', \kappa) \Rightarrow \forall \rho_i. soundK(c, \rho_i, \kappa).$$

Theorem 1 then follows from the observation that all knowledge that is sound according to the predicate *soundK* is a *subset* of the attacker knowledge, as defined in Definition 1. Formally, if $soundK(c, \rho, \kappa))$ then

$$\kappa(x)^{-1}(v) \cap [\rho]_{\mathsf{L}} \subseteq \mathcal{K}(c; \mathsf{output}\ x, v, \rho). \qquad \square$$

We complete the semantics of a hybrid monitor with an additional rule to deal with outputs, presented below. The output rule uses the $NI(\rho, K, v)$ predicate that uses the knowledge K to check whether all low-equal initial environments would either produce the same value v or would not terminate (indicated by a value $\perp$). In Section VII-D we describe how to efficiently implement the computation of predicate *NI*. This predicate is defined by

$$NI(\rho, K, v) \quad \triangleq \quad [\rho]_{\mathsf{L}} \subseteq K^{-1}(v) \cup K^{-1}(\perp).$$

The rule for output is then given by:

$$\textsc{outni}\ \frac{(c, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = v \quad NI(\rho, \kappa(x), v)}{(c; \mathsf{output}\ x, \rho) \Downarrow v}$$

With this output rule, we have the property of a knowledge-based hybrid monitor that the monitor either accepts the output of a program, or blocks.

Lemma 2. *If a knowledge-based hybrid monitor produces a value v for a program P from an initial environment ρ, then the original program P computes the same value:*

$$(P, init(\rho)) \Downarrow v \Rightarrow (P, \rho) \downarrow v.$$

We can then prove (using Theorem 1) that the knowledge-based hybrid monitor completed with an output rule OUTNI enforces knowledge-based security.

Theorem 2 (Monitor Soundness)**.** *A program P, monitored by a knowledge-based hybrid monitor with output rule* OUTNI*, is TINI under the monitor semantics $\Downarrow$: whenever $\rho^1_{\mathsf{L}} = \rho^2_{\mathsf{L}}$ and $(P, init(\rho^1)) \Downarrow v^1$ and $(P, init(\rho^2)) \Downarrow v^2$, then $v^1 = v^2$.*

Example 5 (Permissiveness)**.** Program 6 of Example 2 demonstrates when our analysis is able to detect non-termination of the program. Our monitor computes the knowledge in variable l as $\kappa(l) = \lambda\rho. if(C[\![h]\!]_\rho, 0, \perp)$. Then, when a variable l is to be output, the OUTNI rule ensures that on all possible low-equal environments, either the program outputs 0 or does not terminate – the predicate *NI* holds.

Given that our monitor is sometimes able to model when the program does not terminate, it might be tempting to enforce termination-sensitive noninterference (TSNI). To achieve it, one could substitute the *NI* predicate with a *TSNI* predicate requiring that in all the low-equal memories either the program terminates producing v, or it does not terminate:

$$TSNI(\rho, K, v) \triangleq [\rho]_{\mathsf{L}} \subseteq K^{-1}(v) \vee [\rho]_{\mathsf{L}} \subseteq K^{-1}(\perp).$$

The problem with this approach is that whenever $\kappa(x)(\rho) = v$, there is no certainty that the program terminates, since v approximates $\perp$.

Example 6 (TSNI counterexample)**.** The composed program from Example 4 is TINI but not TSNI since it terminates on $h = true$ and does not terminate when $h = false$. The hybrid monitor computes the knowledge in x as $\kappa(x) = \lambda\rho.1$ that would satisfy *TSNI* predicate, however this would not be a sound enforcement of TSNI.

VI. Combination with other monitors

We now show how an existing dynamic monitor based on security levels can be combined with our knowledge-based hybrid monitor. The combined monitor will admit more executions than each of the monitors taken in separation, and will still be secure. To compare the precision of monitors, Hedin et al. [12] propose the notion of "permissiveness" that compares a set of program executions accepted by two monitors and defines a monitor to be more permissive if it accepts a strictly bigger set of executions.

Hedin et al. [12] observe that purely dynamic monitors (e.g., NSU [4]) and simple hybrid monitors (e.g., Le Guernic et al. [18]) are not necessarily comparable with respect to their permissiveness. For example, the execution of Program 1 in environment $h = false$ is accepted by a dynamic monitor NSU because the test is false, but it is rejected by a hybrid monitor since the static analysis concludes that there might be a leak on the non-executed branch. On the other hand, NSU rejects an execution of Program 2 when $h = true$ (because of a sensitive upgrade on line 1), while a hybrid monitor accepts it (because the security level of l is downgraded to L on line 2).

A. Hybrid monitor reusing an inlined monitor

We assume that a monitor based on security levels (for example, a purely dynamic monitor), is inlined in the program following the inlining technique simultaneously proposed by Chudnov and Naumann [10] and Russo and Sabelfeld [20]. Here, a program c is transformed into a program $\widetilde{c}$, where each variable x has a shadow variable $\widetilde{x}$ representing the security label of x. The monitoring is not intrusive in the sense that the values of x are the same for c and $\widetilde{c}$. In other words, the computation of security levels has no impact on the computed values. We will present the instantiation to NSU in Section VI-B below.

Given a program $P = c; \mathsf{output}\ x$, the monitor usually decides to output x if the label of x is lower or equal than the level L[4]. A hybrid monitor can choose to mimic the

[4]Usually, this condition is $pc \sqcap \Gamma(x) \sqsubseteq \mathsf{L}$, however the program counter *pc* is at the lowest level L because our programs only produce output outside conditionals and while-loops.

$$\textsc{t-skip}\ G, S \vdash \mathsf{skip} \rhd \mathsf{skip} \qquad \textsc{t-seq}\ \frac{\forall i = 1..2. G, S \vdash c_i \rhd \widetilde{c}_i}{G, S \vdash c_1; c_2 \rhd \widetilde{c}_1; \widetilde{c}_2}$$

$$\textsc{t-if}\ \frac{S(x) = \widetilde{x} \qquad y \notin dom(S) \cup rng(S) \wedge y \notin G \qquad \forall i = 1..2. (y :: (pc :: G)), S \vdash c_i \rhd \widetilde{c}_i}{(pc :: G), S \vdash \mathsf{if}\ x\ \mathsf{then}\ c_1 \mathsf{else}\ c_2 \rhd y := \widetilde{x} \sqcup pc; \mathsf{if}\ x\ \mathsf{then}\ \widetilde{c}_1 \mathsf{else}\ \widetilde{c}_2}$$

$$\textsc{t-while}\ \frac{S(x) = \widetilde{x} \qquad y \notin dom(S) \cup rng(S) \wedge y \notin G \qquad (y :: (pc :: G)), S \vdash c \rhd \widetilde{c}}{(pc :: G), S \vdash \mathsf{while}\ x\ \mathsf{do}\ c \rhd y := \widetilde{x} \sqcup pc; \mathsf{while}\ x\ \mathsf{do}\ (\widetilde{c}; y := \widetilde{x} \sqcup y)}$$

$$\textsc{t-assign}\ \frac{S \vdash e \rhd \widetilde{e} \qquad S(x) = \widetilde{x} \qquad \widetilde{c} = \mathsf{if}\ pc \sqsubseteq \widetilde{x}\ \mathsf{then}\ \widetilde{x} = \widetilde{e} \sqcup pc\ \mathsf{else}\ \forall y \in Var. S(y) = B}{pc :: G, S \vdash x := e \rhd \widetilde{c}; x := e}$$

Fig. 4: NSU inlining transformation

original behaviour of the inlined monitor by introducing the following output rule OUTL:

$$\textsc{outL}\ \frac{(\widetilde{c}, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = v \quad \rho'(\widetilde{x}) \sqsubseteq \mathsf{L}}{(\widetilde{c}; \mathsf{output}\ x), \rho \Downarrow v}$$

This rule ensures that the inlined monitor only outputs a value v when the security level of the variable x is below L. We make one assumption about the inlined monitor *viz.*, that it correctly computes the security labels of the variables that can be used to enforce noninterference.

Assumption 1 (Correct labels of inlined monitor)**.** Consider a program $P = \widetilde{c}; \mathsf{output}\ x$. If it outputs a value v according to the following output rule:

$$\textsc{outL}\ \frac{(\widetilde{c}, \rho) \downarrow \rho' \quad \rho'(x) = v \quad \rho'(\widetilde{x}) \sqsubseteq \mathsf{L}}{(\widetilde{c}; \mathsf{output}\ x), \rho \downarrow v}$$

Then the label of variable x is computed correctly in the final environment ρ', meaning

$$\mathcal{K}(P, v, \rho) = \mathcal{I}(P, \rho).$$

As a consequence, a hybrid monitor only using the OUTL output rule soundly enforces noninterference.

To gain more precision, the hybrid monitor can first use its own knowledge about the output variable x (applying the rule OUTNI), and only if it is unable to prove noninterferfence, it can then try to apply the default output rule OUTL of the inlined monitor.

We can gain even more precision by reasoning about the knowledge contained in the shadow variables. This will allow the monitor to output certain values, even if neither the rule OUTNI nor the rule OUTL holds. The idea is to provide the knowledge-based monitor with the extra information that certain variables would never be output because the dynamic monitor would block them. Environments that lead to an output that is certain to be blocked by the dynamic monitor can be disregarded when computing the set of possible outcome of the program. Concretely, we add a new security level B ("will be blocked by the monitor"), such that $H \sqsubset B$. An additional output rule OUTH exploits this new label. It requires that:

- the value of x can be output (because $\rho(\widetilde{x}) \sqsubset B$), and
- all the environments that will not be blocked by a monitor, written $NotB(\widetilde{K})$, and low-equal to ρ would produce the same value v.

Formally, we get the following output rule OUTH that combines information computed by the hybrid monitor and the inlined dynamic monitor.

$$\textsc{outH}\ \frac{\begin{array}{c}(\widetilde{c}, init(\rho)) \Downarrow (\rho', \kappa) \quad \rho'(x) = \lfloor v \rfloor \\ \rho'(\widetilde{x}) \not\sqsubseteq \mathsf{L} \quad \rho'(\widetilde{x}) \sqsubset B \quad single(\rho, \kappa(x), \kappa(\widetilde{x}), v)\end{array}}{(\widetilde{c}; \mathsf{output}\ x), \rho \Downarrow v}$$

The predicate *single* ensures that we only produce a single unique value. It exploits the new *blocked* level B and is defined as follows.

$$\begin{array}{lcl} single(\rho, K, \widetilde{K}, v) & \triangleq & ([\rho]_{\mathsf{L}} \cap NotB(\widetilde{K})) \subseteq K^{-1}(v) \\ NotB(\widetilde{K}) & \triangleq & (Env \backslash \widetilde{K}^{-1}(B)). \end{array}$$

With this rule, the combination of the hybrid monitor and a dynamic monitor can be strictly more permissive than either of them.

B. Application to NSU

In the No-Sensitive Upgrade monitor (NSU) [4], [25] an assignment to a variable is allowed only if the program counter level *pc* is lower than the level of the assigned variable l. Otherwise, the execution is stopped.

Figure 4 presents an inlining transformation for NSU. The inlining technique uses the transformation judgement $G, S \vdash c \rhd \widetilde{c}$, where $S : vars(c) \to Var$ maps each variable of c into its *shadow variable* $S(x)$ which contains the current security level of x; and G is a finite list of variables that store the current *pc*. $S \vdash e \rhd \widetilde{e}$ means that $\widetilde{e}$ is an expression for the level join of shadow variables of the variables in e (see [10, Fig. 9]).

NSU enforces termination-insensitive noninterference [4], so the proposed transformation indeed satisfies Assumption 1. The only difference in our presentation

of inlining is that we do not write an explicit divergence operation (such as `while (true) skip`), but rather assign the highest security level B to all the variables (see T-ASSIGN rule); and still allow the computation $x := e$.

The following short example demonstrates in a concise manner how the knowledge-based hybrid monitor that reuses NSU is more permissive than NSU.

Program 9

```
1 l = 1; if h then l = 0;
2 output h
```

Example 7. All the executions of Program 9 are blocked by NSU since the program tries to output a high variable h on a low channel. However our knowledge-based hybrid monitor combined with NSU accepts its execution when $h = \mathit{false}$. Following the idea of NSU, the monitor transforms an information channel into a termination channel. When $h = \mathit{false}$, the hybrid monitor is able to compute that the other initial environment ρ', where $h = \mathit{true}$ will get a B label ($\kappa(\widetilde{h})(\rho') = B$) and therefore will not be output (unless the rule OUTNI holds). Therefore, the output is accepted by the OUTH rule because the *single* predicate ensures that all the executions not blocked by the monitor (where $h = \mathit{false}$) will compute the same value.

C. Soundness and Permissiveness

We now prove the main soundness result for the hybrid monitor that executes a program with an inlined security monitor. In the following, we write $HM(\textsc{Output})$ for our hybrid monitor using a *set* of output rules $\textsc{Output} \subseteq \{\textsc{outNI}, \textsc{outL}, \textsc{outH}\}$.

Theorem 3. *All the executions of a program $P = (\widetilde{c}; \mathsf{output}\ x)$, monitored by a knowledge-based hybrid monitor $HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})$ are secure for all environments ρ.*

Remark that a hybrid monitor using a smaller set of output rules is also sound. This fact will be useful to prove the relative precision of hybrid monitors.

A monitor A is more permissive than a monitor B if it stops less monitored executions (suppresses less outputs). Following Hedin *et al.* [12], a *productive* environment for a monitor M and a program P, written $E_M(P)$ is a set of environments for which the monitor does not stop, i.e., $E_M(P) = \{\rho \mid \exists v.(P, \rho) \Downarrow_M v\}$. Thus, E_M is a family of productive sets indexed by programs for the monitor M.

Definition 6. A monitor A is *at least as permissive* as a monitor B if $E_B \subseteq E_A$.

Theorem 4 (Hierarchy of hybrid monitors)**.** *Consider a program $P = c; \mathsf{output}\ x$ and a program $\widetilde{P} = \widetilde{c}; \mathsf{output}\ x$ that is instrumented by a sound dynamic monitor for noninterference. The following precision results hold:*

$$
\begin{array}{lcl}
E_{HM(\{\textsc{outNI}\})}(P) & = & E_{HM(\{\textsc{outNI}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outNI}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outL}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) \\
E_{HM(\{\textsc{outNI},\textsc{outL}\})}(\widetilde{P}) & \subseteq & E_{HM(\{\textsc{outNI},\textsc{outL},\textsc{outH}\})}(\widetilde{P})
\end{array}
$$

Theorem 4 shows that the combination of our hybrid monitor with a dynamic monitor is more precise than each of them taken separately. Moreover, as the output rule OUTH requires a cooperation between both, our best hybrid monitor $HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})$ is more precise than a direct parallel composition of both, which can be obtained as $HM(\{\textsc{outNI}, \textsc{outL}\})$.

Example 8. Consider again a Program 9. When $h = \mathit{false}$, the knowledge-based hybrid monitor ($E_{HM(\{\textsc{outNI}\})}$) is not able to prove noninterference and the level of h is H, and hence the output is blocked. However, the more precise monitor $E_{HM(\{\textsc{outNI}, \textsc{outL}, \textsc{outH}\})}$ ensures that h is output since the *single* predicate ensures that the monitor always produces the same output.

VII. Implementation

The knowledge-based hybrid monitor has been implemented and proved correct [1] using the Coq proof assistant. Here, we describe the main data structures and algorithms used for computing knowledge.

Knowledge about initial environments is formalized using knowledge functions $K \in \mathbf{K}$. One algorithmic challenge is to find an efficient algorithm for extracting the knowledge from such a function K *i.e.* computing $K^{-1}(v)$ for some v. We present a concrete representation for a class of knowledge functions that is closed under all the operations needed by the hybrid monitor. With this representation, a logical formula representing the knowledge of an output v is computed in linear time.

A. Concrete domain of knowledge functions

The domain of the monitor $\mathbf{K} = \mathit{Env} \to \mathbb{V}^\sharp$ is encoded with the symbolic domain $\mathbf{K}^\flat \subset \mathcal{P}(\mathbb{F} \times \mathbb{E}) \times \mathbb{F}$ where $\mathbb{E}$ is the set of program expressions and $\mathbb{F}$ is the following set of boolean formulae: $\mathbb{F} \ni \phi ::= \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid e \mid \mathit{tt} \mid \mathit{ff}$ (here $e \in \mathbb{E}$ is a program expression seen as a boolean, see Section III-A).

A pair $(f, e) \in \mathbb{F} \times \mathbb{E}$ denotes a knowledge which returns the value of the expression e when the formula f holds in the initial environment and $\top$ otherwise. The last element $\phi \in \mathbb{F}$ of $\mathbf{K}^\flat$ specifies when the knowledge is $\bot$. Given an element $K = (S, \phi) \in \mathbf{K}^\flat$, we write K^S for the set of pairs S and K^ϕ for the formula ϕ. The denotation of $K \in \mathbf{K}^\flat$ in the knowledge domain $\mathbf{K}$ is then obtained by:

$$
\begin{array}{rcl}
\{\!|K|\!\} & = & \left(\bigwedge_{(f,e)\in K^S} f \mapsto e\right) \curlywedge K^\phi \mapsto \bot \\
\text{where } \psi \mapsto e & = & \lambda\rho.\text{if } [\![\psi]\!]_\rho \text{ then } [\![e]\!]_\rho \text{ else } \top
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{true}[K] & = & \bigvee_{(f,e)\in K^S}(f \wedge e)\\
\mathit{false}[K] & = & \bigvee_{(f,e)\in K^S}(f \wedge \bar{e})\\
\mathit{top}[K] & = & \neg(\mathit{true}[K] \vee \mathit{false}[K] \vee K^\phi)\\
K@\psi & = & (\{(\psi \wedge f, e) \mid (f,e) \in K^S\}, K^\phi \vee \neg\psi)\\
K_1 \curlyvee K_2 & = & (K^= \cup K_{12} \cup K_{21}, K_1^\phi \wedge K_2^\phi), \text{ where}\\
K^= & = & \left\{(f_1 \wedge f_2 \wedge e_1 = e_2, e_1) \middle| \begin{array}{l}(f_1,e_1)\in K_1^S\\(f_2,e_2)\in K_2^S\end{array}\right\}\\
K_{ij} & = & \{(f_i \wedge K_j^\phi, e_i) \mid (f_i,e_i) \in K_i^S\}
\end{array}
$$

Fig. 5: Implementation of the conditional operator.

Our domain $\mathbf{K}^\flat$ also requires well-formedness conditions. Given $(S,\phi) \in \mathbf{K}^\flat$, we add the constraint that any two pairs (f,e) and (f',e') from S must satisfy that if both f and f' hold then the expressions e and e' evaluate to the same value. Moreover, for any $(f,e) \in S$, the conjunction of f and ϕ does not hold. All the operators needed by the hybrid monitor preserve this property.

Example 9. We illustrate below the symbolic encoding of basic knowledge functions.

$$
\{\!|(\emptyset, \mathit{ff})|\!\} = \lambda\rho.\top \qquad \{\!|(\emptyset, \mathit{tt})|\!\} = \lambda\rho.\bot
$$
$$
\{\!|\{(\mathit{true}, n)\}, \mathit{ff})|\!\} = \lambda\rho.n
$$

For Program 3, we get the following knowledge in l in the end of the program:

$$
(\{(h,1);(\neg h, x);(x=1,x)\}, \mathit{ff})
$$

The knowledge is well-formed since if any two formulae among $h, \neg h$ or $x = 1$ hold at the same time, the corresponding expressions evaluate to the same value.

B. Implementation of operators

The assignment rules ASSINDYN and ASSIGNSTAT compute the knowledge of an expression using a function $(\!|\_|\!)_\kappa$. We show how to implement this function for a new domain of knowledge $\mathbf{K}^\flat$ below:

$$
\begin{array}{rcl}
(\!|x|\!)_\kappa = \kappa(x) \qquad (\!|n|\!)_\kappa = (\{(\mathit{tt}, n)\}, \mathit{ff})\\
(\!|k_1 \oplus k_2|\!)_\kappa & = & (S, (\!|k_1|\!)_\kappa^\phi \vee (\!|k_2|\!)_\kappa^\phi) \text{ where}\\
S & = & \left\{(f_1 \wedge f_2, e_1 \oplus e_2) \middle| \begin{array}{l}(f_1,e_1)\in (\!|k_1|\!)_\kappa^S\\(f_2,e_2)\in (\!|k_2|\!)_\kappa^S\end{array}\right\}
\end{array}
$$

One key operator is the conditional operator *IF* which combines the knowledge from different execution paths. The *IF* and *assume* operators can be rewritten using more basic operators, defined in Figure 5:

$$
\begin{array}{lcl}
\mathit{Assume}(c,l) & = & l@\mathit{true}[c] \curlyvee l@\mathit{top}[c]\\
\mathit{IF}(c,l,r) & = & l@\mathit{true}[c] \curlyvee r@\mathit{false}[c] \curlyvee (l \curlyvee r)@\mathit{top}[c]
\end{array}
$$

Theorem 5. *The operators over $\mathbf{K}^\flat$ exactly model the operators over $\mathbf{K}$. In particular, we have*

$$
\begin{array}{lcl}
\{\!|(\!|e|\!)_\kappa|\!\} & = & (\!|e|\!)_{\lambda x.\{\!|\kappa(x)|\!\}}\\
\{\!|K_1 \curlyvee K_2|\!\} & = & \{\!|K_1|\!\} \curlyvee \{\!|K_2|\!\}\\
\{\!|\mathit{IF}(c,e_1,e_2)|\!\} & = & \mathit{IF}(\{\!|c|\!\}, \{\!|e_1|\!\}, \{\!|e_2|\!\})
\end{array}
$$

C. Static analysis of loops

The only place where the specification of the hybrid monitor is not directly executable is the rule WHILESTAT. The implementation of this rule requires an iterative fixpoint computation in order to infer an invariant of the loop body. The domain $\mathbf{K}^\flat$ does not satisfy the finite ascending chain condition. Therefore, a widening operator is needed to ensure convergence and speed up computations. Our widening limits the number of distinct expressions which can occur in formulae. To remove an expression e from a formula f, we compute the formula $f^+ \wedge f^-$ where f^+ is obtained by substituting e for tt and f^- is obtained by substituting e for ff. The obtained formula is by construction stronger which ensures the soundness of the transformation. Remember that $(f \mapsto e)$ returns $\top$ when f does not hold. By bounding the number of expressions to some fixed constant k, and because formulae have a normal, our fixpoint iteration operates over a finite domain of boolean formulae. This ensures convergence.

D. Effective proof of non-interference

To get an effective enforcement and implement the rules OUTNI and OUTH, we need a decision procedure for the predicates *NI* and *single*. These predicates can be encoded as logic formulae $f \in \mathbb{F}$. To get a decidable enforcement, the logic needs to be decidable. As propositional logic is decidable, the decidability depends only on the language of expressions $\mathbb{E}$. It will hold, for instance, for decidable fragment of arithmetic such as linear integer arithmetics or bit-vector arithmetics.

The logic translation is syntax-directed and defined by the function $\langle\cdot\rangle$:

$$
\begin{array}{lcll}
\langle K^{\text{-}1}(v)\rangle & = & \bigvee_{(f,e)\in K^S} f \wedge e = v & \langle K^{\text{-}1}(\bot)\rangle = K^\phi\\
\langle [\rho]_\mathsf{L}\rangle & = & \bigwedge_{\{x \mid \Gamma(x) \sqsubseteq \mathsf{L}\}} x = \rho(x) &
\end{array}
$$

The inverse function $K^{\text{-}1}(v)$ represents the knowledge of the output v. Given the output value v and a symbolic knowledge $K \in \mathbf{K}^\flat$, $\langle K^{\text{-}1}(v)\rangle$ builds a disjunction where all the expressions are constrained to be equal to v. The equivalence class $[\rho]_\mathsf{L}$ of a initial environment ρ is obtained by a conjunction of equality constraints stating that a low variable x, should have the value of $\rho(x)$. Set operations $\cup$, $\cap$ and $\setminus$ have a standard encoding and set inclusion can be done by checking entailment. Theorem 6 states that the security predicates *NI* and *single* can be checked by checking that their logic encoding is a tautology.

Theorem 6. *The logic translation of security predicates is sound and complete.*

$$
\begin{array}{lll}
\mathit{NI}(\rho, K, v) & \mathit{iff} & \langle [\rho]_\mathsf{L}\rangle \Rightarrow \langle K^{\text{-}1}(v)\rangle \vee \langle K^{\text{-}1}(\bot)\rangle\\
\mathit{single}(\rho, K, \widetilde{K}, v) & \mathit{iff} & \langle [\rho]_\mathsf{L}\rangle \wedge \langle \mathit{NotB}(\widetilde{K})\rangle \Rightarrow \langle K^{\text{-}1}(v)\rangle
\end{array}
$$

E. Experiments

From our Coq development, we have extracted an Ocaml proof-of-concept implementation [1]. We have extracted

five programs from this paper and used our implementation to compute the approximated knowledge. These programs were selected to demonstrate the difference in monitors with respect to the attacker knowledge approximation and permissiveness. For these five programs, we provide the actual knowledge gained by an attacker by observing an output and the approximation computed by different monitors, including the best hybrid monitor of Besson *et al.* [7], called HM(Val+Comb). For enforcement of noninterference, we compare the permissiveness of a knowledge-based hybrid monitor HM (Section IV), the standard NSU and their combination (Section VI).

For each program, we analyse an execution for a given initial environment presented in column 2 of Figure 6a. The actual knowledge of the attacker is a formula over the high variables, that we present in column $\mathcal{K}(P_i, v, \rho)$. For columns HM(Val+Comb) and HM, we highlight in light grey the knowledge that was not computed precisely.

For the majority of programs the hybrid monitor HM is able to compute the exact knowledge of the attacker. Only for Program P_7, our monitor approximates the knowledge of the attacker due to the its static analysis limitation. For Programs P_4 and P_5, our hybrid monitor is strictly more precise than the hybrid monitor of Besson *et al.*

Figure 6b gives an insight into permissiveness of the monitors, where HM stands for $HM(\{\text{OUTNI}\})$, NSU stands for $HM(\{\text{OUTL}\})$ and HM+NSU stands for $HM(\{\text{OUTNI}, \text{OUTL}, \text{OUTH}\})$. Given a program execution described in the second column of Figure 6a, we write a ✓ if the value is output and a ✗ if the monitor blocks the execution. All the presented program executions are rejected by NSU except for P_1 and P_7. The execution of P_5 is proved noninterferent by the hybrid monitor HM, while rejected by NSU. Program P_9 illustrates the case where neither HM nor NSU alone is able to ensure termination-insensitive noninterference whereas their combination HM+NSU does. As explained in Section VI-B, the key insight is to exploit the knowledge that other interfering executions would be blocked.

VIII. Discussion

1) Scalability: The formal development and implementation of our hybrid monitor are given for a minimalistic imperative language. A relevant question is whether our core monitor could be efficiently implemented for a full-fledged language. Core dynamic monitors have already been adapted to very dynamic languages. For instance, the JSFlow project [13] implements a dynamic information flow monitor for JavaScript. Hedin *et al.* [12] also proposed a hybrid monitor that covers a large subset of JavaScript. For hybrid monitors, the main challenge is to mitigate the overhead incurred by the static analysis. Note that it is always possible to trade precision for efficiency – for instance by making an aggressive use of widening operators (see Section VII-C). At the extreme, static analysis can even be momentarily switched off if it is deemed too costly or unfeasible. In that case, the computed knowledge for the non-executed branch would be $\lambda\rho.\top$ i.e. the absence of knowledge which is sound but imprecise. Regarding functions, a reasonable trade-off could be to limit the static analysis to the current function boundaries i.e intra-procedural analysis. Yet, getting the desired trade-off between precision and efficiency requires more investigation.

2) Extension to programs with I/O and strategies: The proposed approach can be extended to programs with I/O and strategies [8]. We could define a special global variable that contains all the knowledge of the previous outputs and each new input would immediately contain that knowledge. Like this, we could track the knowledge that would be an upper bound for any possible strategy. The current representation would not change in this case.

IX. Related work

Zdancewic [25] proposed the *no-sensitive-upgrade* principle for dynamic information flow control that halts execution if a program assigns to low variables under secret control. Austin and Flanagan [4], [5] extended this to *permissive upgrade* which takes the future use of the assigned variable into account before halting the execution. *Hybrid monitors* for information flow control combine static and dynamic program analysis [17], [18], [21], [23]. One of the first techniques was proposed by Le Guernic *et al.* [18] where the static analysis only performs syntactic checks on non-executed branches. Russo and Sabelfeld [23] introduced a generic framework of hybrid monitors, where non-executed branches are analysed syntactically and formally proved that the permissiveness of such monitors is incomparable with the purely dynamic monitors. In a follow-up work, Le Guernic [17] presented a more permissive static analysis, that ignores possible branches that depend only on public variables. Besson *et al.* [7] enhance a dynamic monitor with static constant propagation and dependency analysis, and show how this leads to a hierarchy of increasingly more permissive hybrid monitors. Their knowledge is represented by the domain $\mathbb{F}\times\mathbb{V}$. As explained in Section I, the present work improves permissiveness of the hybrid monitor from [7]: 1) we have a strictly more expressive domain: an element $(f, v) \in \mathbb{F} \times \mathbb{V}$ is exactly modelled in our domain by the knowledge $(\{(f, v)\}, \mathit{ff})$; 2) we have the advantage of capturing certain forms of non-termination. With respect to dynamic monitors, permissiveness of the proposed monitor is incomparable (see Fig. 6b), however it has the power to achieve a strictly higher level of permissiveness by combination with the dynamic monitors.

Chudnov *et al.* [9] propose a hybrid monitor for relational logic. An interesting feature of the work is that the monitor is obtained from a constructive soundness proof. In this work, we consider a specific property (namely termination insensitive non-interference). Yet, our knowledge analysis is not geared to noninterference and could help discharging more general assertions of relational logic.

	$(P_i, \rho) \downarrow v$	$\mathcal{K}(P_i, v, \rho)$	HM(Val+Comb)	HM
P_1	$(l = 0, h = ff) \downarrow 1$	$\neg h$	$\neg h$	$\neg h$
P_4	$(h1 = ff, h2 = tt) \downarrow 1$	$h1 \vee h2$	$\neg h1 \wedge h2$	$h1 \vee h2$
P_5	$(h = tt, x = 0, y = 1) \downarrow 1$	tt	h	tt
P_7	$(h = tt) \downarrow 1$	tt	h	h
P_9	$(h = ff) \downarrow ff$	h	h	h

(a) Computation of Knowledge

	HM	NSU	HM + NSU
P_1	✗	✓	✓
P_4	✗	✗	✗
P_5	✓	✗	✓
P_7	✗	✓	✓
P_9	✗	✗	✓

(b) Permissiveness of Enforcement

Fig. 6: Experimental results

In a recent paper, Hedin *et al.* [12] extend a dynamic information flow monitor for core JavaScript with a static points-to analysis that can approximate the potential write targets in regions with a high security context. The dynamic monitor is based on NSU and prevents implicit flows by forbidding all side effects with targets that are below the security context. The static analysis is used to raise the security labels of the potential write targets to the level of the context before entering this context. This prevents the monitor from stopping when writing to a low target. Interestingly, the static analysis need not be sound or complete, as the dynamic monitor ensures that the hybrid monitor is sound. Precision only affects the permissiveness of the monitor. Their hybrid monitor is more precise than a static information flow analysis such as that of Hunt and Sands [15]. However, they also make the observation that "with the above definition of relative permissiveness, a hybrid monitor cannot subsume a purely dynamic monitor" [12, Thm. 3]. This is not contradictory with our findings because they only consider a particular static analysis. We conjecture that their "NSU + points-to" monitor can benefit from our extension with knowledge computation in order to obtain a monitor that subsumes their dynamic monitor.

The notion of attacker knowledge was first proposed by Askarov and Sabelfeld [3] and then used by Askarov and Chong [2] to study enforcement of noninterference when the security policy changes over time, and for different kind of attackers. The notion of knowledge here is used to state the security conditions but the enforcement mechanism does not compute the knowledge explicitly.

In the area of purely static information flow analysis, Hunt ans Sands [15] proposed a flow-sensitive type system for non-interference that was later proven to be less permissive than a standard hybrid monitor [23, Thm. 3]. Müller *et al.* [22] generalize the type system of Hunt and Sands using the technique of self-composition. They define an abstract weakest precondition calculus for self-composed program that computes logical formulae describing dependencies and equalities between variables.

X. Conclusions

We propose a hybrid monitor to compute the knowledge that an attacker obtains by observing a program output. The monitor is hybrid since it statically analyses non-executed branches. Our symbolic representation of attacker knowledge is powerful and subsumes existing hybrid monitoring approaches. We show that a knowledge-based monitor can be combined with any dynamic monitor for noninterference resulting in an enforcement mechanism that is more permissive than each mechanism taken separately. Therefore, our monitor is able to reach a level of permissiveness that was deemed impossible for the previous hybrid monitors [23].

In this paper we have laid the foundations for designing knowledge-based hybrid monitors. There are several ways in which this work can be further expanded.

The language studied here is voluntarily kept minimalistic and there are interesting semantic questions linked to how to monitor knowledge for more advanced programming languages with features such as objects, arrays and higher-order functions.

Our monitor statically analyses non-executed branches and the theory explains how this integration is designed. However, the current development can go much further and integrate traditional static analyses. In particular, more precise numeric analyses, ranging from constant propagation to polyhedral analysis, would allow the monitor to prove more equalities between variables and, hence, improve permissiveness. Other analyses such as points-to analysis would be required for the extension to the language features mentioned above.

References

[1] Formalisation of the Hybrid Monitor in Coq. Supplementary material.

[2] A. Askarov and S. Chong. Learning is change in knowledge: Knowledge-based security for dynamic policies. In *CSF'12*, pages 308–322. IEEE, 2012.

[3] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *S&P'07*, pages 207–221. IEEE, 2007.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. *CACM*, 52:83–91, 2009.

[7] F. Besson, N. Bielova, and T. Jensen. Hybrid information flow monitoring against web tracking. In *CSF'13*, pages 240–254. IEEE, 2013.

[8] S. Chong. Required information release. *Journal of Computer Security*, 20(6):637–676, 2012.

[9] A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *CSF'14*, pages 48–62. IEEE, 2014.

[10] A. Chudnov and D. A. Naumann. Information Flow Monitor Inlining. In *CSF'10*, pages 200–214. IEEE, 2010.

[11] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.

[12] D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive Hybrid Information Flow Control for a JavaScript-like Language. In *CSF'15*. IEEE, 2015.

[13] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in javascript and its apis. In *SAC'14*, pages 1663–1671. ACM, 2014.

[14] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *CSF'12*, pages 3–18. IEEE, 2012.

[15] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06*, pages 79–90. ACM, Jan. 2006.

[16] J. Landauer and T. Redmond. A lattice of information. In IEEE, editor, *CSFW'93*, pages 65–70, 1993.

[17] G. Le Guernic. Precise Dynamic Verification of Confidentiality. In *Proc. of the 5th International Verification Workshop*, volume 372 of *CEUR Workshop Proc.*, pages 82–96, 2008.

[18] G. Le Guernic, A. Banerjee, T. Jensen, and D. Schmidt. Automata-based Confidentiality Monitoring. In *ASIAN'06*, volume 4435 of *LNCS*, pages 75–89. Springer, 2006.

[19] G. Le Guernic and T. Jensen. Monitoring Information Flow. In A. Sabelfeld, editor, *Workshop on Foundations of Computer Security - FCS'05*, Proceedings of the 2005 Workshop on Foundations of Computer Security (FCS'05), pages 19–30. DePaul University, 2005.

[20] J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. In *SEC'10*, pages 173–186, 2010.

[21] S. Moore and S. Chong. Static analysis for efficient hybrid information-flow control. In *CSF'11*, pages 146–160, 2011.

[22] C. Müller, M. Kovács, and H. Seidl. An analysis of Universal Information Flow based on Self-composition. In *CSF'15*. IEEE, 2015.

[23] A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF'10*, pages 186–199. IEEE, 2010.

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

[25] S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.