Mashic compiler: Mashup sandboxing based on inter-frame communication

Zhengqin Luo, José Fragoso Santos, Ana Almeida Matos, Tamara Rezk

Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication

Zhengqin Luo[1], José Fragoso Santos[2,3], Ana Almeida Matos[4,5], and Tamara Rezk[2]

[1]Google Inc.
[2]INRIA
[3]Imperial College London
[4]SQIG, Instituto de Telecomunicações
[5]Instituto Superior Técnico, Universidade de Lisboa

1 Introduction

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 constraints 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.

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 [1]. 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

This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008.

[1] `http://www.programmableweb.com/`

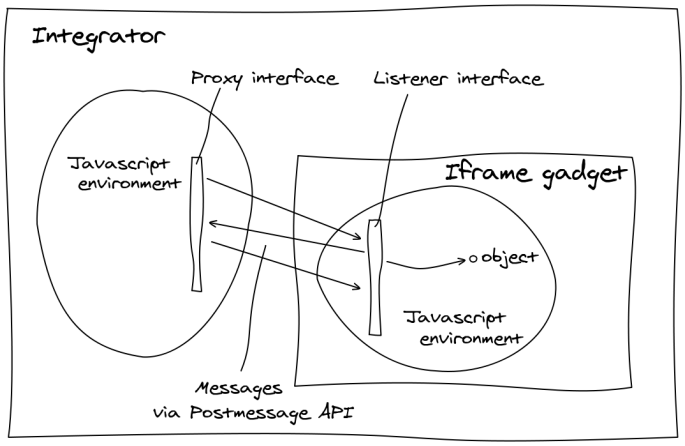


Figure 1: Target Architecture Automatically Generated by Mashic

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].

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] reveals 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 ADsafe [10] statically validates JavaScript programs. In contrast to the Mashic compiler, all of these mechanisms are gadget-code dependent. 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 compiler from ML programs to JavaScript programs. Their result is similar to ours in that compiled JavaScript programs will execute in isolation, but it requires integrator code to be written in ML, which Mashic does not require.

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:

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

```
var secret = document.getElementById("secret_input");
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:

```
var steal;
steal = secret;
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:

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

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

```
<div id=gadget_canvas></div>
<script src='http://untrusted.com/gadget-msg.js'>
</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[2] 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 standard JSON stringify. Code in `gadget-msg.js`

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

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 properties 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. 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.

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

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

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.

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

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

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.

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:

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

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

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

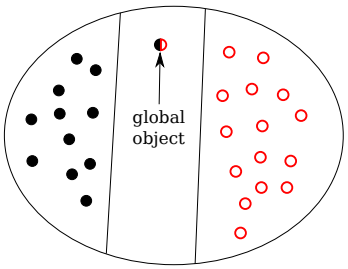


Figure 2: Example: Uniformly Colored Heap

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

M	::=	<html> F J </html>	HTML page
F	::=	<iframe src=u></iframe> $\mid \epsilon$	a frame or empty
J	::=	<script□> 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	::=	+ | - | < | > | === | =	binary operators

Figure 3: JavaScript Syntax with Decorations

is presented in Figure 4.

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

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

3.4 Configurations

Instrumented global configurations feature a decoration component that denotes the owner principal of the program being executed. Decorations are propagated via seman-

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

Figure 4: Variable Declaration Function

tics 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 $\langle \Box, h, \ell, R, Q \rangle_x$ 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 $\Box$ 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 :

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

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 $\langle \Box, \varepsilon, null, M, Q_{init} \rangle_I$ where $Q_{init} = \langle null, \varepsilon \rangle \parallel \langle null, \varepsilon \rangle$.

We also define a configuration for the core JavaScript semantics

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

to be a 4-tuple featuring:

- A decoration $\Box$ 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:

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

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 Figures 6, 7, and 8.

3.6 DOM Semantics Rules

We extend the syntax with run-time expressions:

R	::=	$M \mid F\ J \mid F_{RT}\ J \mid J \mid s\ J$	run-time programs
F_{RT}	::=	<iframe> J </iframe>	run-time frames
	$\mid$	<iframe> $s\ J$ </iframe>	
e	::=	$\ldots \mid$ @FunExe$(\ell, s, \square) \mid$ @NewExe$(\ell_o, \ell, s, \square)$	run-time expressions
v	::=	$pv \mid \ell \mid$ *undefined*	run-time values
i	::=	$@x \mid m$	properties of objects

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 @FunExe$(\ell, s, \square)$ and @NewExe$(\ell_o, \ell, s, \square)$, v denotes run-time values which consist of primitive values pv, heap locations ℓ, and undefined value $undefined$.

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 $\triangle$.

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

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

Figure 5: Decorated Semantics Rules (DOM)

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.

EINIT, 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.

3.7 Core Semantics Rules

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

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

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.

DTHIS

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

DOBJ-LITERAL

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

DCALLFUNC

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

DCALLMETHOD

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

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

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

DCALLRET

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

DNEW

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

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

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

Figure 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

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

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

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

Figure 7: Decorated Semantics Rules (Core JavaScript , continued)

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

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

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

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

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

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

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

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

Figure 8: Decorated Semantics Rules (Core JavaScript, continued)

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.

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 $\Box$, keeping track the owner of the function.

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

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

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 ♥, $x[\text{“}b\text{”}] = 3$ results in the decorated object on the right side:

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

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:

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

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:

```
1 function OHandle(id){
2    if (id == undefined) id = handle_id_gen();
3    this._id = id;
4    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:

```
1 var handle_list = {};
2 function add_handle_obj(ohandle,obj){
3  handle_list[ohandle._id] = obj;}
4 function get_obj_by_handle(ohandle){
5  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.

```
function GET_GLOBAL_REF(global_name,cont){
  var m_id = gen_id();
  var msg = {msg_id : m_id,
             msg_type : 'GET_GLOBAL_REF',
             global_name : global_name};
  PostMessage(stringify(msg));
  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:

```
1  function GET_GLOBAL_REF_L(recv){
2   var obj = window[recv.global_name];
3   return make_resp_msg(recv,obj);
4  }
5  function make_resp_msg(recv,obj){
6   var ohandle, msg;
7   if (obj != null &&
8      (typeof(obj) == "object" ||
9       typeof(obj) == "function"))
10      {ohandle = new OHandle();
11       add_handle_obj(ohandle,obj);
12       msg = {msg_id : recv.msg_id,
```

```
13            msg_type:'EXE_CONT',
14            return_val : ohandle};}
15  else {msg = {msg_id: recv.msg_id,
16          msg_type:'EXE_CONT',
17          return_val : obj};}
18  return msg;
19 }
```

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:

```
1 { msg_id : m_id,
2   msg_type: 'CALL_METHOD',
3   object : opq_obj,
4   method_name: method,
5   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:

```
1 { msg_id: m_id,
2   msg_type:'EXE_CONT',
3   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.

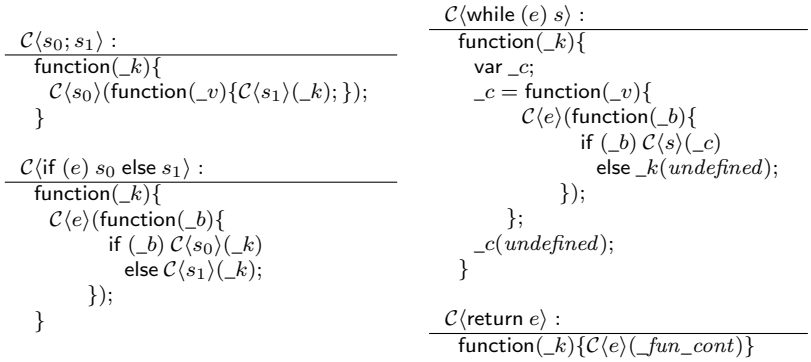


Figure 9: Transformation of Statements

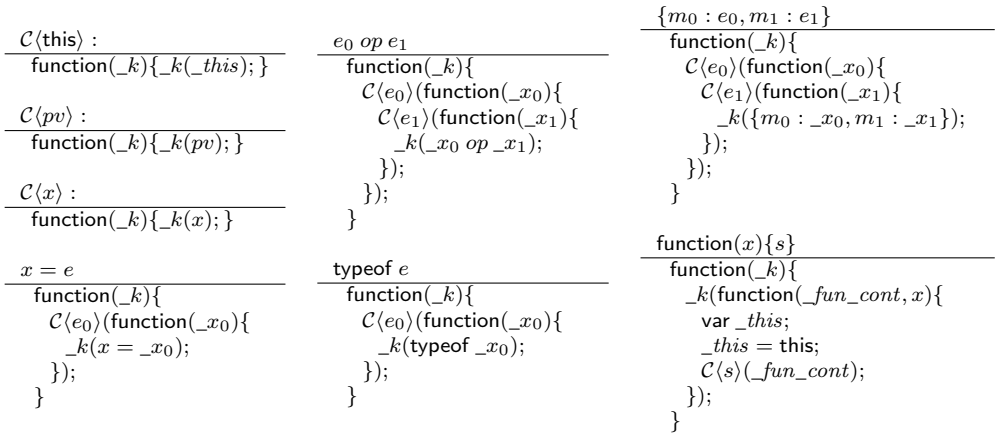


Figure 10: Transformation of Expressions, Non-Message-Passing part

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♥> P_g </script>
    <script♠> P_i </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 changes to the integrator's heaps by the original and the compiled versions are indistinguishable. 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]);
      }
    }); }); }
```

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

```
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);
        }
      }); }); }); }
```

Figure 11: Transformation of Expressions, Message-Passing Part

Definition 2 (Mashic Compilation)**.** Let P_i be an integrator script, P_g be a gadget script, and $\mathcal{V}$ 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 $j \in \{0 \ldots n\}$, as

$$\mathsf{var}\ x_j; x_j = \mathsf{new}\ OHandle(j);$$

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

$$add\_handle\_obj(\mathsf{new}\ OHandle(j), x_j);$$

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 $h_j\downarrow_{\heartsuit} = h_g$ ($j \in \{0, 1\}$), 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 $(\spadesuit, h_j, \ell_j, \ell(\ell')) \rightarrow^* (\spadesuit, h'_j, \ell'_j, v'_j)$, the following conditions hold:

1. $v'_0 = v'_1$;
2. (integrity) $h_j =_{\spadesuit} h'_j$ and $h_j =_{\blacklozenge} h'_j$;
3. (confidentiality) $h'_0 =_{\heartsuit} h'_1$;
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-♦-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 for marshaling/unmarshaling functions f, f^{-1} and variable set $\mathcal{V}$ if and only if, for any benign heap h_g such that $(♠, h_{in} \oplus h_g, \#global, P_i)$ 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 $\ell \in \mathsf{dom}(h\lfloor_{♠})$ 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 for which f and f^{-1} are correct and ℓ_f is reachable from $\mathcal{V}$ in h.

Example 9 (Correct integrator prototype chain). We illustrate why a correct 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:

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

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.

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, and 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.

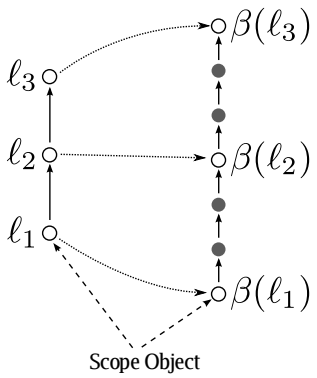


Figure 12: Example: Scope Indistinguishability

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 that o_1 and o_2 are indistinguishable with respect to β, written $o_1 \simeq_\beta o_2$, 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 $o_1.@this \sim_\beta o_2.$"$\_this$";
4. $i = @body$ then $o_1.@body = \mathsf{function}(x)\{s\}$, then $@body \in \mathsf{dom}(o_2)$ and $o_2.@body = \mathsf{function}(\_fun\_cont, x)\{s\}$, where

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

We give an example illustrating object indistinguishability.

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

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

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

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)**.** 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 $(h_1, \ell_1) \simeq_\beta (h_2, \ell_2)$, 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) $o_1 \simeq_\beta o_2$
   (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 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 for marshaling/unmarshaling functions f, f^{-1} and variable set $\mathcal{V}$, then $\mathcal{C}\langle P_i \rangle$ and $\mathcal{C}'\langle P_i \rangle$ are both correct integrators for the same functions and variable set.*

Proof. Straightforward by structural induction on the 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

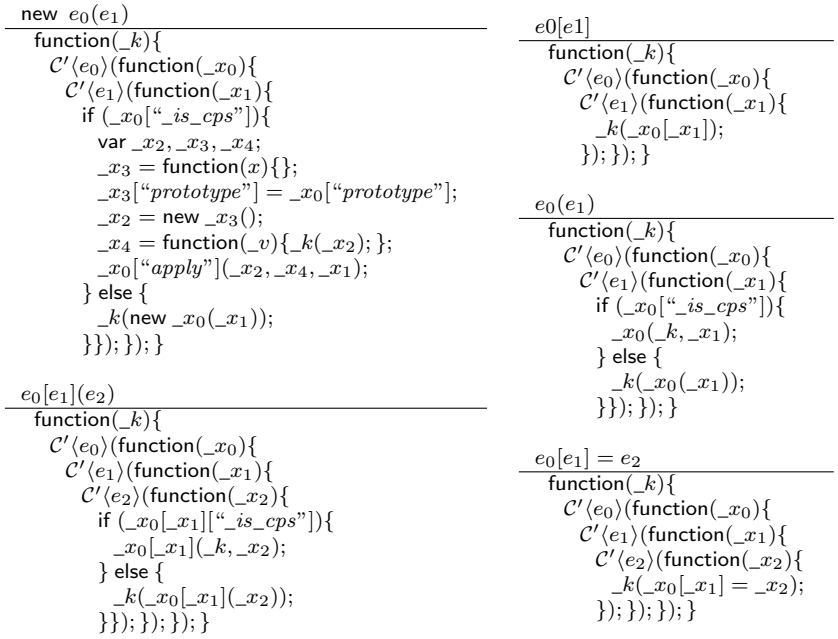


Figure 13: CPS Transformation (Intermediate)

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:

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

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

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

Proof. Straightforward by Definition 11 and Definition of $\mathrm{Scope}(\_, \_, \_)$ 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

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

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. $\mathsf{dom}(h_c \lfloor_\spadesuit) = \hat{\text{A}} < om(h_i \lfloor_\spadesuit)$;
3. $\forall \ell \in \mathsf{dom}(h_c \lfloor_\spadesuit), o_c = h_c(\ell)$ and $o_i = h_i(\ell)$, we have 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 the 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

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

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.

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

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 $\text{“}\mathit{secret}\text{”}_{\{\spadesuit\}} : \text{“}\mathit{yes}\text{”}$

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

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

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

the gadget heap is $h\lfloor_{\heartsuit}(\#\mathit{global}_f).\text{“}\mathit{steal}\text{”} = \text{“}\mathit{no}\text{”}$. 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.)* $h =_{\spadesuit} h'$;
2. *(confidentiality.) For any h_0 such that $h_0 =_{\heartsuit} h$, we have $\langle \heartsuit, h_0, \ell, s, Q \rangle_F \rightarrow \langle \heartsuit, h_0', \ell', s', Q' \rangle_F$, and $h_0' =_{\heartsuit} h'$.*

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

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

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.

```
res_or_tramp=fun(arg);
while (res_or_tramp instanceof Trampoline)
   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.

```
function wrap_fun(fhandle){
   return function(arg){
      var msg = { fun : fhandle,
                  msg_type : 'CALLBACK',
                  argument : arg};
      PostMessage(stringify(msg));
      return;};}
```

Listing 18: Wraping Function Handle

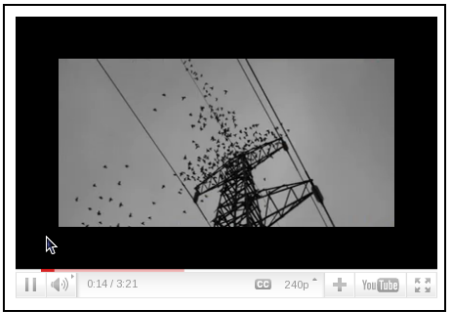
The wrapped function, upon each invocation, sends a message to notify the integrator to invoke the function associated with the function handle.

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.

(a) Map Directions



(b) Youtube Player Controls

Figure 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.

Figure 15: Selected Case Studies

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.

```
1 function GET_REAL_VALUE(v, cont) {
2   if (isMashicObject(v)) {
```

```
3         _cont = function() { cont.call(null, v._value);};
4         FLUSH()
5     } else { cont.call(null, v); }
6 }
```

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.

```
1 function FLUSH() {
2     var i, requests;
3
4     requests = [];
5     for (i=0; i<_batched_futures.length; i++) {
6         if (isComplexBatchedFuture(batched_futures[i])) break;
7         requests[i] = createMsg(batched_futures[i]);
8     }
9
10    postMessage(requests);
11    _current_batch_index = i;
12 }
```

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.

```
1 function _message_handler(m) {
2     var responses;
3
4     responses = parse(m);
5     for (i=0; i<responses.length; i++) {
6         _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

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

```
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); });
      });
}
```

Figure 16: Optimised Mashic-CPS Transformation

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.

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

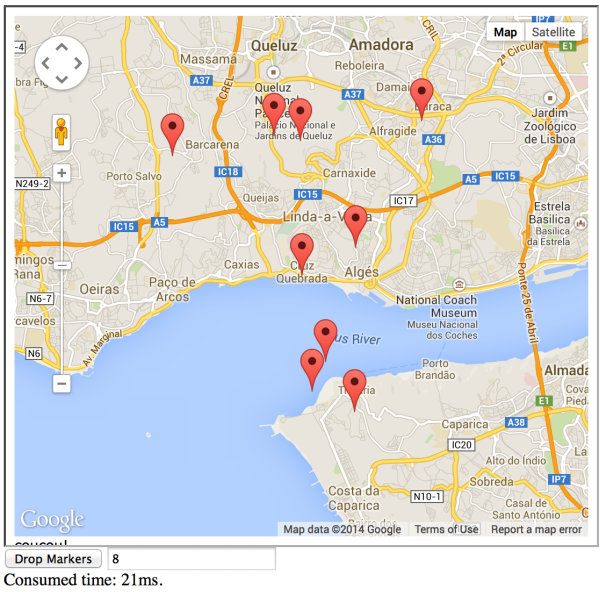


Figure 17: Map Markers

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 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

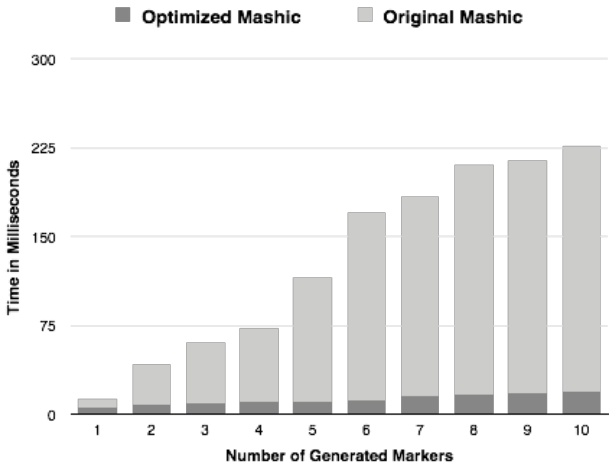


Figure 18: Comparison between mashic optimised version and original version

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.