# Mashic Compiler: Mashup Sandboxing based on Inter-frame Communication

Zhengqin Luo
*INRIA*
*Zhengqin.Luo@inria.fr*

Tamara Rezk
*INRIA*
*Tamara.Rezk@inria.fr*

***Abstract*—We propose a new compiler, called Mashic, for the automatic generation of secure Javascript-based mashups from existing mashup code. The Mashic compiler can effortlessly be applied to existing mashups based on a wide-range of gadget APIs. It offers security and correctness guarantees. Security is achieved via the Same Origin Policy. Correctness is ensured in the presence of benign gadgets, that satisfy confidentiality and integrity constrains with regard to the integrator code. The compiler has been successfully applied to real world mashups based on Google maps, Bing maps, YouTube, and Zwibbler APIs.**

## I. Introduction

Mixing existing online libraries and data into new online applications in a rapid, inexpensive manner, often referred to as mashups, has captured the way of designing web applications. ProgrammableWeb mashup graphs currently report that over 5000 mashup-based web applications and over 3000 gadget APIs currently exist (`http://www.programmableweb.com/`). Since the release of the first major example, `HousingMaps.com` in early 2005, mashups have offered the potential to finally make widespread software reuse a reality.

In a mashup, the *integrator* code integrates *gadgets* from external code providers. Typically, code is written in JavaScript (JS) and executes on the browser as embedded script nodes in the Document Object Model (DOM) [Hors et al., 2000]. 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, permitting to execute untrusted code with otherwise impossible integrator privileges due to browsers security policies;
- or by using the iframe tag, in which case the Same Origin Policy (SOP) applies. The SOP isolates untrusted JS external code, limiting the interaction of gadget and integrator to message sending [Barth et al., 2009b].

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 security in the name of functionality.

Recently, Smash [Keukelaere et al., 2008], AdJail [Louw et al., 2010], and Postmash [Barth et al., 2009a] 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. [2009a] 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.

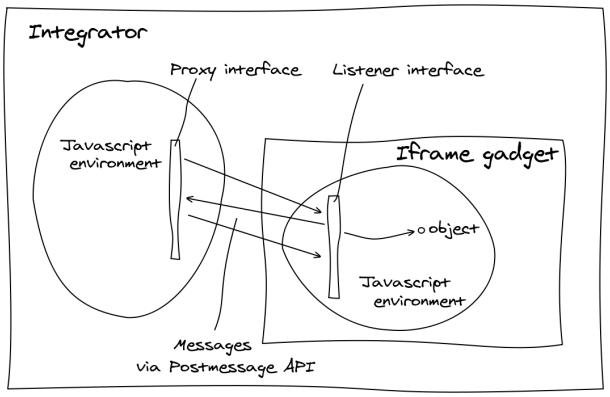



Figure 1. Target Architecture Automatically Generated by Mashic

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 preseving only the good behaviour 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, JS 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 (in the shape of 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 [Sabelfeld and Myers, 2004]. The Mashic compiler[1] 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 [Vinoski, 1997] to interact with the gadget. Due to the asynchronous nature of the PostMessage API, integrator's code is transformed into 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 hypothesis that the gadget is *benign* and a correctness notion of marshaling/unmarshaling for objects that are sent via postMessage.

The correctness notion of marhaling/unmarshaling allow 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 JS semantics extended with HTML constructs by a generalization of colored brackets [Grossman et al., 2000] and resort to equivalences used in information flow security [Sabelfeld and Myers, 2003].

**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.

Because of lack of space, we opt to focus Mashic's presentation on:

- Design and implementation choices and proof techniques used for the compiler (without fully describing the transformations themselves);
- Formalization of correctness (together with restrictions on integrator's code to which it applies) and security properties that are guaranteed;
- The definition of benign gadget, that can independently (from Mashic) be useful for specifying other properties regarding mashups.

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

- Unsupported Constructs: Our integrator transformer currently supports the full JS language [ECMA, 2009] except for a few programming constructs. Specifically, the *for-in* construct and *exception* construct are not supported. Some JS features considered "dangerous" such as *eval* are not supported neither.
- 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. (Note: 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 Barth et al. [2009b]).
- 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. Certain types of mashups do not fall into this category, notably mashups containing advertisement scripts. Louw et al. [2010] 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

[1]Implementation and proofs can be found at `http://www-sop.inria.fr/indes/mashic/`

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.

*Related Work:* The closest works to Mashic are AdJail [Louw et al., 2010], Smash [Keukelaere et al., 2008], and Postmash [Barth et al., 2009a], and are described above. We focus now in other related work. Jang et al. [2010] study on top of 50000 websites privacy violating information flows in JS 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 JS capabilities between different origins [Barth et al., 2009c]. Many mechanisms to prevent JS based attacks have been deployed. For example the Facebook JS subset (FBJS) [Inc., 2011a] was intended to prevent user-written gadgets to attack trusted code but it did not really succeed in its goals [Maffeis and Taly, 2009]. Google Caja [Inc., 2011b] is similar to FBJS, transforming JS programs to insert run-time checks to prevent malicious access. Yahoo ADsafe [Crockford, 2011] statically validates JS programs. Maffeis et al. [2010] resort to language-based techniques to find out a subset of JS that can be used to prove an isolation property for JS code. For that, they identify a capability-safe subset of JS. 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 and real world web applications due to the highly dynamic nature of JS programs and because gadgets in general cannot be restricted to subsets of JS. As a response to the increasing need to get flexible functionality without resigning to security guarantees, the research community has proposed several communication abstractions [Wang et al., 2007, Crockford, 2010, Jackson and Wang, 2007, Keukelaere et al., 2008]. Specifically, OMASH [Crites et al., 2008] proposes a refined SOP to enable mashup communication.These abstractions usually require browser modifications and so far have not been adopted in HTML standards [Hickson, 2011]. There are other works [Bohannon and Pierce, 2010, Akhawe et al., 2010] pursuing the direction of formalizing web applications, but none of them formally model the SOP.

## II. 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. 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:

```
<div id=gadget_canvas></div>
<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 JS 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 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. With PostMessage, only strings can be sent. However, it is possible to marshal objects that do not point to themselves (as e.g. the global object), via a marshaling method, such as the standard JSON stringify. Code in `gadget-msg.js` and `integrator-msg.html` needs to adapt to the asynchronous behaviour. Instead of calling methods or functions, the integrator must send messages to manipulate the untrusted gadget as shown in the following example:

```
1 PostMessage(stringify({action : "newInstance",
2           container : "gadget_div",
3           type : "SIMPLE"}),
4          "http://u-i.com");
5 PostMessage(stringify({action : "setLevel",
6           container : "gadget_div"}),
7           "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.

## III. Decorated Semantics

We propose a decorated semantics to partition a JS 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 [Sabelfeld and Myers, 2003], is 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. Maffeis and Taly [2009]), 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. Hence, we have resorted to ideas from colored brackets [Grossman et al., 2000] and adapt them to a semantics modeling the SOP in the browsers. When decorations are erased, our JS decorated rules are compliant with JS semantics (Maffeis et al. [2008]). For the sake of simplicity in the presentation, we limit this section to the inclusion of only one gadget as a frame, although the JS 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 □ or △ to denote any of them.

*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 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 $o$ does not have property $i$.

*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 $\ell$; 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 $\ell$ in the heap, we use the notation $h(\ell.i_{\{\square\}} = v) = h'$, where $h'$ is the updated heap. We also use $\text{Alloc}(h, o) = h', \ell'$, where $\ell' \notin \text{dom}(h)$, for allocating a fresh location for an object in the heap. After adding the location, the new heap is $h'$. JS 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 lookup process $\text{Scope}(h, \ell, m)$ function 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. 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 $\text{Prototype}(h, \ell, m)$ looks for the $m$ property of the object $h(\ell)$ via the prototype chain.

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

*Integrator and Gadgets Global Objects:* We use subindex $i$ to indicate initial global object for the integrator code.

We define a (simplified) initial 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 ♦,

| | | | |
|---|---|---|---|
| $M$ | ::= | <html> $F$ $J$ </html> | HTML page |
| $F$ | ::= | <iframe src=$u$></iframe> $\mid \epsilon$ | a frame or empty |
| $J$ | ::= | <script$\square$> $s$ </script> $J \mid \epsilon$ | sequence of scripts |
| $P, s$ | ::= | $e \mid \ldots$ | Javascript programs |
| $R$ | ::= | $M \mid F\ J \mid F_{RT}\ J \mid J \mid s\ J$ | run-time programs |
| $F_{RT}$ | ::= | <iframe> $J$ </iframe> | run-time frames |
| | $\mid$ | <iframe> $s$ $J$ </iframe> | |
| $e$ | ::= | $\ldots \mid$ @FunExe$(\ell, s, \square)$ | run-time expressions |
| | $\mid$ | @NewExe$(\ell_o, \ell, s, \square)$ | |
| $v$ | ::= | $\ldots \mid \ell \mid$ *undefined* | run-time values |

Figure 2. JS Syntax with Decorations (excerpt)

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 $\#addlistener_f$. 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 assume that $\mathrm{Alloc}(h, o)$ never allocates those predefined 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 a single 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 omit explanations for other pre-defined native objects.

*Decorated Heap Projections:* 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 $\perp$. 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: $\forall \ell \in \mathsf{dom}(h), h(\ell)\lfloor_\square \neq \perp \Leftrightarrow \ell \in \mathsf{dom}(h')$ & $h'(\ell) = h(\ell)\lfloor_\square$. We define $h = h'$ as equality on heaps. We denote $h' =_\square h$ for $h'\lfloor_\square = h\lfloor_\square$.

*Syntax:* We present in Figure 2 a simplified syntax of the extension of JS 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. Maffeis et al. [2008]). Notice that $f$ ranges over native functions, as the native PostMessage function. We further extend the syntax for run-time expressions denoted as $R$. Run-time expression $e$ is extended with a special context for executing functions. Run-time value $v$ is extended with heap locations $\ell$ and the undefined value.

Before a JS 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 $\ell$ of the current scope object, a statement $s$, and a color $\square$ to bind variables declared by var $x$ with proper decorations to the scope object $\ell$, as in $h(\ell.x_{\{\square\}} = undefined)$.

*Configurations:* Instrumented configurations feature a decoration component that denotes the owner principal of the program being executed. A configuration is a 5-tuple $(\square, h, \ell, R, Q)$ that features: a decoration $\square$ that denotes the principal of the current program in the configuration, a heap $h$, a location $\ell \in \mathcal{L}$ pointing to the current scope object (or $null$ only for the initial configuration), a run-time program $R$ currently being executed, a waiting queue $Q$ in order to give semantics to PostMessage mechanism. A waiting queue is of the form $\langle \ell_i, mq_i \rangle \parallel \langle \ell_f, mq_f \rangle$, where $\ell_i$ and $\ell_f$ are locations for event listeners and $mq_i$ and $mq_f$ are message queues for both, the integrator and the frame, respectively. The syntax for defining a message queue is : $mq ::= m\ mq \mid \varepsilon$ where $m$ is a string. We use $mq_1 + mq_2$ to denote the concatenation of two message queues.

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

The small step semantics is modeled by a relation $\rightarrow$ between configurations. Explanations for the rules in Fig. III follow:

DINIT*:* A mashup execution starts by initializing the heap of the configuration to the initial heap of the integrator $h_{in}$. The scope object is set to the global object $\#global_i$.

DSCRIPT*:* A $\triangle$-decorated script starts by initializing variables defined in $s$. $\mathrm{VD}(h, \ell, s, \triangle)$ adds the variables to the current scope object $\ell$ in $h$. The new configuration takes the color $\triangle$ of the script.

DFRAME*:* A frame fetches the content $\mathsf{Web}(u)$ and merges the initial frame heap $h^f_{in}$ to the current heap. The key point here is that addresses in $h$ do not overlap with

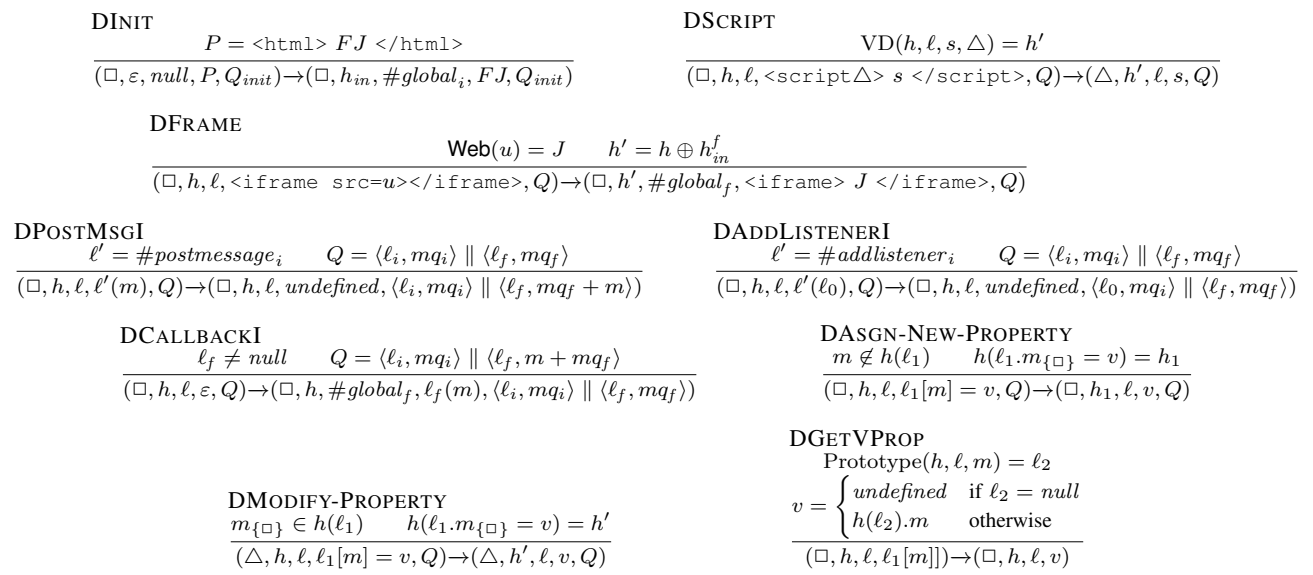



Figure 3. Decorated Semantics Rules (excerpt)

addresses in $h^f_{in}$. Because of this and by the JS semantics, a pointer reference from $h$ cannot be accessed via a program with a scope object in $h^f_{in}$. This DFRAME rule precisely models the SOP since no address in the integrator's heap can be reached from a JS program with a scope object in $h^f_{in}$. Notice that the current scope object is set to the frame's global object.

DPOSTMSGI*:* When the integrator sends a message $m$, the gadget waiting queue is updated.

DADDLISTENERI*:* The integrator sets an event listener $\ell_0$ in the waiting queue by calling the native function from the location $addlistener_i$. Notice that for simplicity, we are assuming only one listener in the formal semantics.

DCALLBACKI*:* This rule together with the symmetric one for frame callback are the only ones introducing non-determinism to the semantics. When no program is executing, pending messages in the waiting queues are handled via the event listeners.

DASGN-NEW-PROPERTY*:* To create a new property $m$, $h(\ell_1)$ is updated with $m$. The new property is decorated with the colour of the current principl. A decoration cannot be changed after creation.

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 a location. When the property $m$ does not exist we return $undefined$.

DCALLFUNC*:* To invoke a function, a new scope object $\ell_s$ is set as current scope object. The $@scope$ property is set to the function's closure scope $h_1(\ell_1).@fscope$. The $@this$ property of $\ell_s$ is set to $\ell_g$ that is 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 $\ell_1$ 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 $\ell$ to return, and the decoration $\Box$ to recover when the function execution finishes. For simplicity, we present functions with one parameter only. Function GetGlobal looks up in the scope chain to get the address of the global object via the window property.

*Example* 1 (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:

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

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.)

## IV. Compilation Overview

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

*Opaque Object Handle:* By the SOP policy, the integrator and the framed gadget cannot exchange JS 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* [Barth et al., 2009a].

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

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

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

*Example* 2. 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:

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

Listing 8. Integrator Bootstrapping

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

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

Listing 9. 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. 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 to operate 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* 3. Recall the mashup from Section II. 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 function `GET_GLOBAL_REF_L` that 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).

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

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

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

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

Figure 4. Integrator Transformation Excerpt

*Integrator Code Transformation:* JS 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 turn the proposal impractical.

*Example* 4. Recall the example in Section II. 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){...});});});
```

The function $\mathcal{C} : s \mapsto s$, see Figure 4, transforms JS code of the integrator into CPS and transforms the integrator code to invoke the proper interfaces from the proxy library.

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.

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

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

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

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

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

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

where

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

## V. Correctness Theorem

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

### A. 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 3** (Correct marshal/unmarshal for $S$)**.** Let $\sim$ be defined as $v \sim v'$ in $h$ iff there exists a bijection $\beta$ 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 3 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 marhaled 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 by ♦) 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 4** (Benign Gadget Heap)**.** A heap $h_g$ is benign if and only if for any heaps $h_0$, $h_1$ such that $h_j\lfloor_{\heartsuit} = h_g$ ($j \in \{0,1\}$), for any function located in $\ell \in \mathsf{dom}(h_g)$, for any $\ell'$ such that $h_0(\ell') = h_1(\ell')$ is an object, and $(\spadesuit, h_j, \ell_i, \ell(\ell'), Q) \rightarrow^* (\spadesuit, h'_j, \ell'_j, v'_j, Q')$, the following conditions holds:

1) $v'_0 = v'_1$;
2) (integrity) $h_j =_{\spadesuit} h'_j$ and $h_j =_{\blacklozenge} h'_j$ ;
3) (confidentiality) $h'_0\lfloor_{\heartsuit} = h'_1\lfloor_{\heartsuit}$;
4) (preservation of benignity) $h'_1\lfloor_{\heartsuit}$ is benign

*Example* 5 (Benign Heap). Recall the integrator's code in Listing 3 in Section II. 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 10. 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 5** (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 $(\heartsuit, h_i, \ell, P_g, Q) \rightarrow^* (\heartsuit, h'_i, \ell, v_i, Q')$, 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* 6 (Benign Gadget Example). Recall the example in Section II, 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 behaviour in gadgets as non-benign behaviour 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 Boudol [2010]).

*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 integrator's usually operate on gadgets via the interfaces provided by it and not directly modifying its properties. Given marshal/unmarshal functions, we also require 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 6** (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 7** (Correct Integrator for $f$,$f^{-1}$,$\mathcal{V}$)**.** Program $P_i$ is a correct integrator, if and only if, for any benign heap $h_g$ such that $(\spadesuit, h_{in} \oplus h_g, \#global, P_i)$ reaches a redex $e$ and a heap $h$ then the following conditions hold:

1) If $e$ is of the form $x = v$ and $\mathrm{Scope}(h, \ell, \text{“}x\text{”}) = \ell_n$, 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 $\ell$ such that $\ell \in \mathsf{dom}(h\lfloor_{\spadesuit})$ and $h(\ell).@prototype = \ell_n$, either $\ell_n = null$ or $h(\ell_n)$ is a ♠-colored object.

4) If $e$ is of the form $\ell_f(\ell')$ and $h(\ell_f)$ is a ♥-colored function, then $h(\ell')$ is an object correct for $f$ and $f^{-1}$ and $\ell_f$ is reachable from $\mathcal{V}$ in $h$.

*Example* 7 (Correct integrator prototype chain). We illustrate why an integrator's object cannot have a gadget's object as its prototype object (bullet 3). Assume that in the heap of the original mashup, $h(\ell_i)$ is a ♠-colored object, and $h(\ell_g)$ is a ♥-colored object such that

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

By reading the property "$a$" of $\ell_i$ in the original mashup we get 3. The heap of the compiled code will contain a pointer to a handle:

$$h(\ell_i) = \{@prototype : \ell_o\}$$

$$h(\ell_o) = \left\{ \begin{array}{ll} \text{"}\_id\text{"} & n \\ \text{"}\_is\_ohandle\text{"} & true \end{array} \right\}$$

Hence, by reading the property "$a$" of $\ell_i$ in the compiled mashup we do not get 3.

### B. 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 use $\ell \in \ell_1\ell_2 \dots \ell_n$ to denote that scope object $\ell$ (either the global object or an object featuring a $@scope$ property) is included in a list of scope objects in a heap $h$ linked by the $@scope$ property as a chain $\ell_1\ell_2 \dots \ell_n$ (i.e. $h(\ell_i).@scope = \ell_{i+1}$).

We define the $\beta$-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 8** (Scope Chain Indistinguishability)**.** Let $\ell_1$ be a scope object in $h$ and $\ell'_1$ be a scope object in $h'$, and $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ be a partial injective function. Let $\ell_1\ell_2 \dots \ell_n$ be the scope chain of $\ell_1$ in $h$, let $\ell'_1\ell'_2 \dots \ell'_m$ be the scope chain of $\ell'_1$ 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) $\beta(\ell_1)\beta(\ell_2) \dots \beta(\ell_n)$ is a sub-sequence of $\ell'_1\ell'_2 \dots \ell'_m$;
2) for $\ell \notin \beta(\ell_1)\beta(\ell_2) \dots \beta(\ell_n)$, and $\ell \in \ell'_1\ell'_2 \dots \ell'_m$, $\forall i \in \mathsf{dom}(h'(\ell))$, $i \in \{@scope, @prototype, @this, \text{"}\_k\text{"}, \text{"}\_l\text{"}, \text{"}\_m\text{"}, \text{"}\_x_i\text{"}\}$

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. 5. 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.

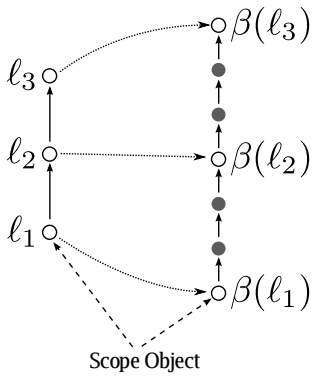



Figure 5. Example: Scope Indistinguishability

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

**Definition 9** (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:

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

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

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

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

*Example* 8 (Object indistinguishability). Let $o_1$, $o_2$ $o_3$ be:

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

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

If $\ell'_1 \neq \beta(\ell_1)$ and $\ell_2 \neq \ell'_2$, then we have $o_1 \simeq_\beta o_2$ and $o_1 \not\simeq_\beta o_3$.

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

**Definition 11** (Heap indistinguishability)**.** We say that $(h_1, \ell_1)$ and $(h_2, \ell_2)$, are indistinguishable with respect to $\beta : \mathcal{L} \rightharpoonup \mathcal{L}$ with $\mathsf{dom}(\beta) = \mathsf{dom}(h_1)$ and $\mathsf{rng}(\beta) \subseteq \mathsf{dom}(h_2)$, denoted $(h_1, \ell_1) \simeq_\beta (h_2, \ell_2)$, if and only if:

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

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

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 $(\spadesuit, \varepsilon, null, \tilde{M}(P_i, P_g), Q_{init}) \rightarrow^* (\square, h_0, \ell_0, \varepsilon, Q_{init})$ then,*

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

*where $Q_1$ has no message waiting, and there exists $\beta$ 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.

## VI. 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 JS execution environment of the integrator as shown in the DFrame rule in Fig. III.

*Example* 9 (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 JS functions, is associated as a property of the global object. As many native functions the code associated to the `setTimeout` function can be changed at execution time, changing in this way the assumed behavior for `setTimeout`.

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

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

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

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

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

**Definition 12** (Decorated Mashic Compilation (for security theorem))**.** Given an integrator script $P_i$ and a gadget script $P_g$, a set of variable $\mathcal{V}$ denoting global names exported

by the gadget script, we define the Mashic compilation $M_c(P_i, P_g, \mathcal{V})$ to be:

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

where

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

*Example* 10 (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. By using decoration of Definition 12 and semantics rules, we get that the projection $h\lfloor_{♠}$ of the integrator heap before execution of the gadget and projection $h'\lfloor_{♠}$ 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* 11 (Confidentiality violation). Recall variable `secret` in the example of Section 2. Let us assume that the gadget's heap is $h\lfloor_{♥}$.

After execution of the non-benign gadget in Listing 4 with an integrator's global object containing "$secret$"$_{\{♠\}}$ : "$yes$" the gadget heap has $h\lfloor_{♥}(\#global_f).$"$steal$" = "$yes$". But starting with integrator's global object containing "$secret$"$_{\{♠\}}$ : "$no$" the gadget heap is $h\lfloor_{♥}(\#global_f).$"$steal$" = "$no$". 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 $M_c(P_i, P_g, \mathcal{V})$ provides integrity and confidentiality guarantees:

**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 $M_c(P_i, P_g, \mathcal{V})$:*

$$(♠, \varepsilon, null, M_c(P_i, P_g, \mathcal{V}), Q_{init}) \rightarrow^* (♥, h, \ell, s, Q)$$

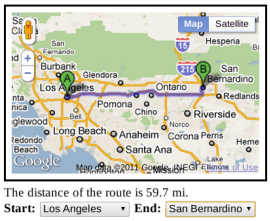
*if*

$$(♥, h, \ell, s, Q) \rightarrow (♥, h', \ell', s', Q')$$

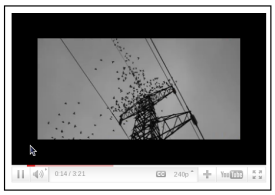
*then we have*

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

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




(a) Map Directions




(b) Youtube Player Controls

Figure 6. Case Studies of Applying Mashic Compiler

## VII. Implementation and Case Studies

The Mashic compiler is written in Bigloo [2] (a dialect of Scheme) and JS. It has 3.3k lines of Bigloo code and 0.8k lines of JS code.

*CPS in Javascript:* Since JS 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 [2009]. We define a global variable `counter` to count the depth of current call stacks. If the counter exceeds a certain limit a tail call will return a trampoline object instead of invoking the function.

This is shown in Listing 11.

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

Listing 11. Trampoline of Tail Call

A guard loop, on the top level, detects if a trampoline object is returned, as shown in Listing 12. If a trampoline object is detected, the loop restarts the execution of the tail call.

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

Listing 12. 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 event handler. Similar to the opaque object handle, we associate opaque function handles with function objects on the integrator side.

*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

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

| 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 7. Selected Case Studies

examples involve non-trivial interactions between the integrator and the gadget.

In Figure 6 we show two concrete examples. The first example is a mashup using 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 though buttons in the integrator.

We report a selected list of mashups in Fig. 7. In the first column of the figure, the mark P means that the integrator's code was obtained from public available code in the web, whereas mark O means that the code is ours.

*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 mainly comes from the unoptimized CPS-transformation and message-passing. We discuss different optimizations.

- CPS Optimization: In order to optimize trampolines, a solution is to use a partial CPS-transformation where code segments not involved in the interaction with gadgets are not transformed. Other techniques proposed by Pettyjohn et al. [2005], Loitsch [2007] to implement call-with-current-continuation in JavaScript are also suitable in our case.
- Batching Optimization: Message-passing is the main cause of performance penalty, especially inside a loop. For example in the *mark drawing* mashup we show in Figure 7, 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, often is 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. We plan to implement batching [Ibrahim et al., 2009] in Mashic to reduce message-passing penalties.

## VIII. 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 plan to investigate the integration of ADJail policy language and mechanisms (as discussed in Section VII) in order to safely apply the compiler to gadgets which are advertisement scripts.

*Acknowledgments:* We thank Bernard Serpette, Sergio Maffeis, Shriram Krishnamurthi, José Santos, and anonymous reviewers for their comments on how to improve this work.

## References

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.

Adam Barth, Collin Jackson, and William Li. Attacks on JavaScript Mashup Communication. In *W2SP2009*, 2009a.

Adam Barth, Collin Jackson, and John C. Mitchell. Securing Frame Communication in Browsers. *Commun. ACM*, 52 (6):83–91, 2009b.

Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin Javascript Capability Leaks: Detection, Exploitation, and

Defense. In *USENIX security symposium*, pages 187–198, 2009c.

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.

Gérard Boudol. Typing termination in a higher-order concurrent imperative language. *Inf. Comput.*, 208:716–736, 2010.

Steven Crites, Francis Hsu, and Hao Chen. OMash: Enabling Secure Web Mashups via Object Abstractions. In *CCS*, pages 99–108, 2008.

Douglas Crockford. The `<module>` Tag , 2010. http://www.json.org.

Douglas Crockford. ADsafe, 2011. http://www.adsafe.org/.

ECMA. ECMAScript Language Specification. Technical report, ECMA, 2009. http://www.ecma-international.org/.

Dan Grossman, J. Gregory Morrisett, and Steve Zdancewic. Syntactic type abstraction. *TOPLAS*, 22:1037–1080, 2000.

Ian Hickson. HTML5. Technical report, W3C, May 2011.

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.

Ali Ibrahim, Yang Jiao, Eli Tilevich, and William R. Cook. Remote batch invocation for compositional object services. In *ECOOP*, 2009.

Facebook Inc. Facebook Javascript Subset, 2011a. https://developers.facebook.com/docs/fbjs/.

Google Inc. Google Caja Project, 2011b. http://code.google.com/p/google-caja/.

Collin Jackson and Helen J. Wang. Subspace: Secure Cross-domain Communication for Web Mashups. In *WWW*, 2007.

Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. In *CCS*, 2010.

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.

Florian Loitsch. Exceptional continuations in JavaScript. In *Workshop on Scheme and Functional Programming*, September 2007.

Florian Loitsch. *Scheme to JavaScript Compilation*. PhD thesis, Université de Nice - Sophia Antipolis, March 2009.

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.

S. Maffeis and A. Taly. Language-based Isolation of Untrusted Javascript. In *CSF*, pages 77–91. IEEE, 2009.

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

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

Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from generalized stack inspection. In *ICFP*, pages 216–227, 2005.

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

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.

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

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.