# A Lattice-based Approach to Mashup Security

Jonas Magazinius
Chalmers

Aslan Askarov
Cornell University

Andrei Sabelfeld
Chalmers

## ABSTRACT

A web mashup is a web application that integrates content from different providers to create a new service, not offered by the content providers. As mashups grow in popularity, the problem of securing information flow between mashup components becomes increasingly important. This paper presents a security lattice-based approach to mashup security, where the origins of the different components of the mashup are used as levels in the security lattice. Declassification allows controlled information release between the components. We formalize a notion of *composite delimited release* policy and provide considerations for practical (static as well as runtime) enforcement of mashup information-flow security policies in a web browser.

## Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Information flow controls

## General Terms

Security, Languages

## Keywords

Web mashups, security policies, lattices, information flow, declassification, noninterference

## 1. INTRODUCTION

A web mashup is a web application that integrates content from different providers to create a new service, not provided by the content providers. As mashups are becoming increasingly popular, the problem of securing information flow between mashup components is becoming increasingly important.

### 1.1 Web mashups

Web mashups consist of a hosting page, usually called the integrator, and a number of third-party components, often called widgets, gadgets, blocks, or pipes. An example of a mashup-based application is a web site that combines the data on available apartments from one source (e.g., Craigslist) with the visualization functionality of another source (e.g., Google Maps) to create an easy-to-use map interface.

The number of web mashups is rapidly increasing. For example, a directory service for mashups `programmableweb.com` registers on average three new mashups every day. This directory contains more than 4000 registered mashups and 1000 registered content provider API's.

### 1.2 Mashup security

Mashup applications, by their nature, involve interaction between various page components. Often these components are loaded from different origins. Here, origins are identified by an Internet domain, protocol, and a port number, a standard convention which we also follow in this paper.

Cross-origin interaction within the browser is currently regulated by the so-called Same-Origin Policy (SOP). SOP classifies documents based on their origins. Documents from the same origin may freely access each other's content, while such access is disallowed for documents of different origins.

Unfortunately, the SOP mechanism turns out to be problematic for mashup security. First, origin tracking in SOP is only partial and allows content from different sources to coexist under the same origin. For example, an HTML tag with an *src* attribute can load content from some other origin and integrate it in the current document. Once integrated, such content is considered to be of the same origin as the integrating document. This means that the content is accessible to scripts in other documents from the same origin.

Of particular concern here is document inclusion via *script* tags. When a *script* tag is used to load JavaScript code from a different origin, the loaded script is integrated into the document, and thereby can freely interact with it. For the same reasons, interaction between different components loaded in this fashion is unrestricted.

The problem of *script*-tag inclusion for mashup applications is that the integrator must trust the third parties to protect its secrets and not to override trusted data with untrusted. Effectively, the security of the integrator no longer relies only upon itself, but also on the security of the third parties whose scripts are included.

So far, these issues have been resolved using the *iframe* tag. The *iframe* tag borrows a part of the integrator's window space to display another document. Since the integrated content is loaded in a separate document, the SOP applies, and the sensitive information of the integrator is protected.


*ASIACCS'10* April 13–16, 2010, Beijing, China.

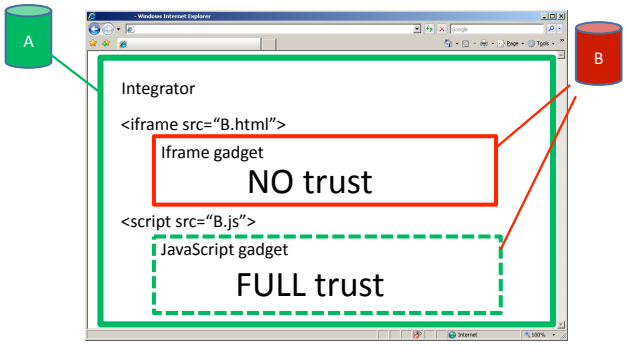


**Figure 1: Polarized trust in mashups**

However, this also severely reduces the possibilities for interaction between the documents. A number of techniques for secure communication between documents have been proposed to bypass the restrictions, but, due to JavaScript's dynamic nature, ensuring confidentiality has proved to be complicated. See Barth et al. [10] for a number of attacks on mashup communication techniques.

The phenomenon is illustrated in Figure 1, where there are two inclusions from site $B$ into site $A$. The first inclusion (B.html) is by an *iframe* tag, while the second inclusion (B.js) is by a *script* tag. This implies two levels of trust: either full or no trust, but also two modes of interaction. Either the content is fully trusted and integrated in the document with full interactivity, or the content is not trusted at all and loaded in a separate document with very limited interactivity.

To sum up, today's mashups trade the users' confidentiality and integrity for functionality. In order to deal with this problem, we aim at requiring the same separation between cross-origin content within documents as we have between documents.

## 1.3 Lattice-based approach

We propose a lattice-based approach to mashup security. The security lattice is built from the origins of the mashup components so that each level in the lattice corresponds to a set of origins. The key element in the approach is that the security lattice is inferred directly from the mashup itself.

The security lattice is used to label the classified objects, where an object's label corresponds to the origin from which it is loaded. The labels are used to track information flow within the browser. One may use a range of techniques, such as static and/or dynamic analysis to ensure that information from one origin does not flow to another one unless the information has been declassified (in case of confidentiality) or endorsed (in case of integrity) for that specific target.

The enforcement mechanism controls possible channels for communicating data from within the page to the outside environment, such as by following links or submitting forms.

In order for the components of one origin to securely release information to another origin, declassification [33] is required. We propose a mechanism that allows origins to specify *escape hatches* [31] for declassifying objects. The novelty of our mechanism is that a piece of data may be released only if all origins that *own* the data agree that it can be released. This approach provides a much-desired flexibility for composite secure data release.

At two extreme instances of our framework, we obtain an isolation of iframes and the flexibility of the *script* tag for including third-party content. The main benefit of our approach is that it allows a fine-grained control over information propagation within the browser.

## 1.4 Attacker model

We assume an honest user that runs a trusted browser on a trusted machine. The *web attacker* [9] is an owner of malicious web sites that the user might be accessing. The web attacker is weaker than, for example, the classical Dolev-Yao attacker [18], because the web attacker may not intercept or modify arbitrary messages. This implies that the web attacker is unable to mount man-in-the-middle attacks. Instead, the network capabilities of the web attacker are restricted to the standard network protocols of communication with servers in the domain of attacker's control.

In contrast to Barth et al. [9], we do not assume a particular separation of web sites on trusted and untrusted. Instead, different component of a web site (or mashup) have different interests and only trust themselves and their security policies.

The *gadget attacker* [9] is a web attacker with the possibility that an integrator embeds a gadget of the attacker's choice. Our attacker is richer than the gadget attacker. First, we take into account that different content providers might have different interests and protect gadgets from each other. Second, the integrator itself might be a malicious web site. Hence, we refer to our attacker as a *decentralized gadget attacker*.

Social engineering attacks such as *phishing* are not in the scope of the paper. Note that since we focus on distinguishing intended vs. unintended inter-domain communication, injection attacks (such as by *cross-site* scripting) are not prevented, but the payload of the injection is restricted from unintended inter-domain communication.

## 1.5 Sources and sinks

Security sources and sinks correspond to the end-points, where security-sensitive data enters and leaves the system. For confidentiality, we consider secret sources, where secret information enters the system, and public sinks, where public output happens. For integrity, untrusted sources and trusted sinks are of the respective importance. Most of the discussion in this paper is focused on confidentiality. Section 5.1 briefly discusses an integrity extension.

User-sensitive data can be stored in browser cookies, form input, browsing history, and other secret sources (cf. the list of sensitive sources used by Netscape Navigator 3 [27]). Client-side scripts have full read access to such data. The need for read access is often well-justified: one common usage is form validation, where (possibly sensitive) data is validated on the client side by a script, before it is passed over to the server. Read access is necessary for such a validation.

We assume that public sinks are observable by the attacker. A script must not leak information from secret sources to public sinks. Examples of public sinks are communication to attacker-observable web sites or interaction with parts of the host site that the script is not allowed to. As we describe further, fine granularity of the lattice-based approach allows us to express such policies.

## 1.6 Scenarios

Below are some motivating scenarios for our approach.

### 1.6.1 Dangerous goods

Consider a scenario of a trucking company that uses a web-based application for managing truck data. In this context, sensitive data that this application needs to operate on includes information such as truck load and scheduled stops. In order to visualize the location of the trucks to the user, the application uses the Google Maps API [2]. This visualization requires that the web application supplies coordinates of each truck when making API calls. With the current technology, Google Maps API can only be used through script inclusion, which means that the code supplied by Google has access to the entire page it is part of. Due to the limitations of the Same-Origin Policy, the company must trust that Google's code is not malicious or that Google's security is not compromised.

### 1.6.2 Advertising

In online advertisement, ad providers seek tight interaction of the ads with pages that provide context for advertisements. Hence, the iframe-based solution often turns out to be too restrictive. On the other hand, ad scripts need to be constrained from giving full trust, since a malicious advertiser can compromise sensitive data.

Unlike previous work that restricts language for advertisement to a strict subset, e.g., AdSafe [14], we allow interactions between trusted code and ads as long as information-flow policies of the trusted origin are respected. Such policies may prevent any flows from the trusted origin to the ad provider, or perhaps, allow some restricted flow, such as releasing certain keywords or releasing some part of user behavior.

## 2. LATTICE-BASED FLOW IN MASHUPS

To deal with the problem of cross-origin content within a document, we propose an approach based on security lattices. An interesting aspect in the mashup setting is that we can infer the levels of the lattice from the mashup itself. The origins of the components of the mashup become the levels of the security lattice. The security lattice is used to label nodes in the *Document Object Model-tree* (DOM-tree), a tree representation of the underlying document. To allow a controlled release of information we also propose a declassification mechanism.

### 2.1 Information lattice

We draw on the *information lattice* [16, 21] in our model of secure information flow. The lattice model is a natural fit for modeling both confidentiality and integrity. In general, a lattice is a partially ordered set with *join* ($\sqcup$) and *meet* ($\sqcap$) operations for computing the *least upper bound* and the *greatest upper bound*, respectively. The security lattice is based on a *security order* on *security levels*, elements of the lattice. The security order expresses the relative restrictiveness of using information at a given security levels. Whenever two elements $\ell_1$ and $\ell_2$ are ordered $\ell_1 \sqsubseteq \ell_2$, then the usage of information at level $\ell_2$ is at least as restrictive as usage of information at level $\ell_1$. More restrictive levels correspond to higher confidentiality and to lower integrity, in the respective cases of modeling confidentiality and integrity. The intention is that higher confidentiality (lower integrity) information does not affect lower confidentiality (higher integrity) in a secure system.

The lattice operators $\sqcup$ and $\sqcap$ are useful for computing the security level of information that is produced by combining information at different security levels. A simple example of an information lattice is a lattice with two elements *low* and *high*, where $low \sqsubseteq high$ and $high \not\sqsubseteq low$. These levels may correspond to public and secret information in a confidentiality setting and to malicious and non-malicious information in an integrity setting.

### 2.2 The domain lattice

The elements of the security lattice are simply sets of origins ordered by the set relation. At the bottom of lattice, denoted by $\bot$, is the empty set. Single origins (i.e., singleton origin sets) form a flat structure. In the notation above, origins $o_1, \ldots, o_n$ correspond to levels $\ell_1, \ldots, \ell_n$, where $\bot \sqsubseteq \ell_1, \ldots, \bot \sqsubseteq \ell_n$ so that for any $\ell$ and $i$ we have $\ell \neq \bot \ \& \ \ell \sqsubseteq \ell_i \Longrightarrow \ell = \ell_i$.

When content from one origin is combined with content from another origin, the level of the result is the join of the origins. Indeed, the levels in the lattice correspond to sets of origins $\ell = \{o_1, \ldots, o_n\}$, where $\ell \sqsubseteq \ell'$ if and only if we have the set inclusion $\ell \subseteq \ell'$. This allows data to be combined and used within the browser and still prevents it from leaking to external targets.

DOM-tree nodes (including the affiliated variables) are labeled with security levels when a new document is loaded from an origin server. The origin of the document is the base level of the lattice. As the document is being parsed and the DOM-tree is built, we use the origins of the contents in the document for labeling new objects. All HTML tags that have an *src* attribute can fetch content (e.g., images, scripts, or style sheets) from any origin, which will be incorporated into the current document.

One interesting aspects of the lattice model is the treatment of subdomains. The Same Origin Policy treats subdomains the in same way as any other domain, with the exception of one case. In current browsers one may change the *document.domain* property of a document loaded from a subdomain to the domain it is a subdomain of. When this is done, the subdomain is considered as a part of the domain. This means that the subdomain can access and can be accessed from any document loaded from the domain, since they are now considered to be of the same origin according to the SOP.

Translating this behavior to the security lattice would mean a merge of the origins or an uncontrolled declassification of all contents belonging to either the subdomain or the domain. This behavior can be supported using a lattice-based approach, but since we aim at a more fine-grained control over information flow in the browser, we prefer that subdomains are treated as any other domains.

### 2.2.1 Examples

The examples below clarify how the lattice model reflects security goals in different contexts.

#### Single domain.

We start with a simple example of a page, loaded from a single domain, that does not reference any third-party content. This represents most regular pages that only contain content from the origin domain. In this case, the interest-


**Figure 2: Single domain lattice**

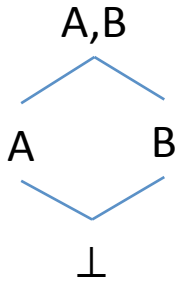


**Figure 3: Two-domain mashup lattice**

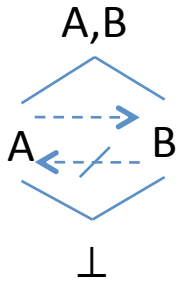


**Figure 4: Declassification**

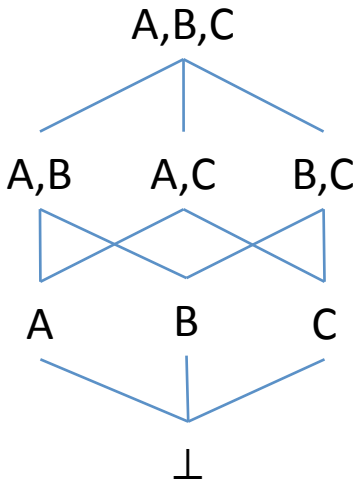


**Figure 5: Three-domain mashup lattice**

ing part of the lattice consists only of that domain and the bottom label, as can be seen in Figure 2.

*Simple mashup with two domains.*

In the scenarios of Dangerous Goods and Advertising from Section 1.6, we have content from two origin domains combined to create a mashup. Figure 3 shows the lattice for the Advertising scenario. Information flow between the content provider and advertisement provider is disallowed. The Dangerous Goods scenario features declassification of certain content from one domain to the other. This is a one-way flow of information, portrayed in Figure 4.

As we elaborate in Section 3, each origin can provide a set of escape hatches that specifies what information can be released and to what origin. In the Dangerous Goods scenario, this corresponds to the coordinates of the truck.

*Complex mashup with multiple domains.*

More complex mashups, such as the iGoogle portal or the social networking site Facebook, include content from multiple domains. In such mashups, when a content is combined from two origins, the level of the result is raised to the join of the levels. Figure 5 shows the lattice for a mashup combining content from three origins.

## 2.3 Embedded third-party content

When communication between the content and its origin is allowed by default, as is the case with the SOP, one needs to identify how third-party content, that is included in a document, is labeled.

In browsers today, any third-party content included in a document is considered to have the document's origin regardless of the actual origin of the included content. This turns problematic in a mashup setting, because the third-party content may be freely send to the document's origin. Instead, we associate the third-party content with its actual origin. This choice has an important security implication: information has to be declassified before it is communicated to other origins. That is, third-party content may not be sent to the document's origin without being declassified by the third-party.

## 2.4 Declassification

While mashups without cross-domain communications exist (cf. the simple version of an advertisement scenario), flexible information flow within mashups is often desired (cf. combination of Craigslist and Dangerous Goods with Google Maps). It is crucial to have a permissive and yet secure cross-domain mechanism that does not allow one component to leak information from another without an explicit permission. How do we ensure that information release intended by one component is not abused by another component or, perhaps, by a malicious integrator? For example, one component of a mashup may agree to release a piece of data but only if is averaged with a piece of data of another component. Or, an email service agrees to release the addresses from the user's contact list but only to a certain type of social network web sites. What we seek is a framework, where individual components may declare what they are willing to release. The information may include data that is controlled by other components, but the actual release is only allowed if all the *owner* components *agree* on releasing the data. This brings us to the next section, where

we formalize this kind of policies.

# 3. FORMAL POLICY

Our formal security policy is an extension of the delimited release [31] policy to multiple origins. Delimited release defines the declassification policy as a set of *escape hatches* which declare what information about secrets can be declassified. In a multiple-origin setting, the policy declaration is spread across multiple origins. We let every origin define its set of escape hatches. This reflects the origin's own view on declassification. An origin can freely declassify expressions that are as restrictive as its own level, but is limited in declassification of expressions that involve other origins. In order for such declassifications to be allowed, a corresponding declaration has to be present in the declassification policies of the other involved origins. The rest of this section specifies how a composite declassification policy is derived based on the individual policies, and defines our security condition which we dub *composite delimited release*.

An escape hatch is represented as a pair $(e, \ell)$, where $e$ is an expression to be declassified and $\ell$ is a target level of the declassification. For a given origin $o$, denote by $E(o)$ a set of escape hatches of that origin.

Consider a simple example of declassifying expression $x + y$, where $x$ has a security level of origin $A$ and $y$ has a security level of origin $B$. We want to allow release of $x + y$ only if both $A$ and $B$ agree on the declassification of $x + y$. We call all origins willing to declassify a particular expression *declassifying origins* or *declassifiers*.

DEFINITION 1 (DECLASSIFIERS). *Given an expression $e$ that is to be declassified to a target security level $\ell$, and a set of origins $o_1 \dots o_n$ with respective declassification policies $E(o_1) \dots E(o_n)$, define* declassifying origins *for $e$ to $\ell$ as follows:*

$$\mathit{declassifiers}(e, \ell, o_1 \dots o_n) = \{o_i \mid (e, \ell') \in E(o_i) \wedge \ell' \sqsubseteq \ell\}$$

The expression $e$ is simply looked up in the set of escape hatches of $E(o_i)$ in the definition above.

Note that $\mathit{declassifiers}(e, \ell, o_1 \dots o_n)$ by itself corresponds to a security level. We next define when a declassification is *allowed*. Informally, when an expression $e$ is declassified from a source level $\ell_{source}$ to a target level $\ell_{target}$, there needs to be enough origins willing to declassify that expression. Formally, this is captured by the following definition.

DEFINITION 2 (ALLOWED DECLASSIFICATIONS). *For an expression $e$ of level $\ell_{source}$, declassification of $e$ to a target level $\ell_{target}$ is* allowed *if*

$$\ell_{source} \sqsubseteq \ell_{target} \sqcup \mathit{declassifiers}(e, \ell_{target}, o_1 \dots o_n)$$

*We use notation $\mathit{allowed}(e, \ell_{source}, \ell_{target}, \mathbf{O})$ for allowed declassifications, where $\mathbf{O}$ abbreviates a set of origins $o_1 \dots o_n$.*

### *Example.*

Assume variables $x$ and $y$ with levels $\Gamma(x) = \{A\}$ and $\Gamma(y) = \{B\}$. Consider origin $A$ with declassification policy $E(A) = \{(x + y, \bot)\}$. $A$ allows declassification of $x + y$ to the public level. Assume also that $B$ has no reference of $y$ in its declassification policy $E(B)$. The composite of these policies allows declassification of $x + y$ to target level $\{B\}$, because $\mathit{declassifiers}(x + y, \{B\}, AB) = \{A\}$ and $\{A, B\} \sqsubseteq \{B\} \sqcup \{A\}$. However, declassifying $x + y$ to $\bot$ is disallowed, because the inequality for allowed declassifications does not hold if the target level is $\bot$: $\{A, B\} \not\sqsubseteq \bot \sqcup \{A\}$.

An example scenario for this kind of policy is a challenge-response pattern, where $B$ poses the challenge $y$, $A$ performs some computation with $y$ and $A$'s private value $x$ and declassifies the result of the computation to $B$.

### *Composite policy.*

We now show how a composite declassification policy can be constructed from individual policies of every origin.

DEFINITION 3. *Given origins $\mathbf{O}$, define by $\mathit{Compose}(\mathbf{O})$ escape hatches $(e, \ell)$ that are allowed according to the declassification policies of $\mathbf{O}$:*

$$\mathit{Compose}(\mathbf{O}) = \{(e, \ell) \mid (e, \ell') \in \tilde{E}(o) \text{ for some } \ell' \text{ and } o \in \mathbf{O} \wedge \mathit{allowed}(e, \Gamma(e), \ell, \mathbf{O}\}$$

Note that $\mathit{Compose}(o_1 \dots o_n)$ is monotonic in origins. Adding a new origin never makes declassification policy more restrictive.

### *Composite delimited release.*

Based on the definition of composite policy, we can now extend the condition of delimited release [31] to a setting with multiple origins.

We associate every object $x$ in the browser model with a security level $\Gamma(x)$, where $\Gamma$ is a mapping from object names to security levels. We model the browser as a transition system $\langle S, \mapsto \rangle$, where $S$ ranges over possible states $s$, and $\mapsto$ defines transitions between states. We denote by $s(x)$ the value of a variable $x$ in a state $s$, and lift this notation to values of expressions in a given state. Denote by $\sim_\ell$ equivalence of two states up to a level $\ell$:

$$s_1 \sim_\ell s_2 \triangleq \forall x \,.\, \Gamma(x) \sqsubseteq \ell \,.\, s_1(x) = s_2(x)$$

We write $s \Downarrow s'$ whenever $s'$ is a terminal state in a sequence of transitions $s \mapsto s_1 \dots \mapsto s'$.

For a set of escape hatches, we define indistinguishability of states up to a security level $\ell$ based on this set of escape hatches:

DEFINITION 4 (INDISTINGUISHABILITY OF STATES). *For a set of escape hatches $E$ say that states $s$ and $s'$ are* indistinguishable *by $E$ up to $\ell$ (written $s\ I(E, \ell)\ s'$) if $\forall (e, \ell') \in E$ such that $\ell' \sqsubseteq \ell$ it holds that $s(e) = s'(e)$.*

This allows us to define our security condition.

DEFINITION 5 (COMPOSITE DELIMITED RELEASE). *For origins $\mathbf{O}$, a system $\langle S, \mapsto \rangle$ satisfies* composite delimited release *if for every level $\ell$ and any states $s_1$ and $s_2$ such that $s_1 \sim_\ell s_2$ and $s\ I(\mathit{Compose}(\mathbf{O}), \ell)\ s'$ then whenever $s_1 \Downarrow s_1'$ and $s_2 \Downarrow s_2'$ it holds $s_1' \sim_\ell s_2'$.*

This definition uses composite policies to filter out disallowed declassifications. For instance, in a system with two origins $A$ and $B$, such that $A$'s declassification policy is empty, $B$'s code cannot declassify any information about $A$'s data. In browser-specific settings this prevents unintended leakage of information.

The composite delimited release precisely regulates *what* information can be declassified, because the escape hatches

```
document.location = "http://evil.com/leak?secret="+encodeURI(form.CardNumber.value);
```
(a) Leak via URL

```
if (form.CardType.value == "VISA") new Image().src="http://evil.com/leak?VISA=yes";
else new Image().src="http://evil.com/leak?VISA=no";
```
(b) Implicit flow

**Figure 6: Explicit and implicit flows**

are related to the initial values in the program. For example, assume that both $A$ and $B$ contain only $(x + y, \bot)$ in their escape hatch sets, where $\Gamma(x) = \{A\}$ and $\Gamma(y) = \{B\}$. Assume there is also $x'$ with $\Gamma(x') = \{A\}$. Composite delimited release allows declassification of the initial value of $x+y$. If, however, $x$ is updated to $x'$, which is different from the initial values of $x$, then the declassification of $x + y$ is rejected.

Composite delimited release can be enforced in two steps. The first step checks that all declassifications are allowed, i.e., all involved origins agree on the declassified escape hatches. Second step has to ensure that the value of an escape hatch expression is not changed since the start of the system. Such an enforcement can be done both statically [31] and dynamically [5].

# 4. ENFORCEMENT CONSIDERATIONS

This section provides practical considerations for implementing an enforcement mechanism for the policies that we have discussed. Enforcement can be realized by a collection of different techniques, which we bring up in this section. Regardless of the technique used, we need to consider all possible communication channels. This includes direct communication channels such as XMLHttpRequest, but also indirect ones such as modification of the DOM tree or communication requests that happen after the user follows a link on a page.

## 4.1 Information-flow tracking

When tracking the actual information flow in JavaScript code, a combination of standard information-flow control [17, 26, 34] can be used with tracking information flow in the presence of language features such as dynamic code evaluation.

### 4.1.1 Explicit and implicit flow

To illustrate simple flows, consider an application that processes a credit card number. Such applications often employ simple validating scripts on the client side before the number is sent to the server. Assume fields `CardNumber` and `CardType` contain the actual number and type of the card. Figure 6(a) corresponds to an *explicit* flow, where secret data is explicitly passed to the public sink via URL. Figure 6(b) illustrates an *implicit* [17] flow via control flow: depending on the secret data, there are different side effects that are visible for the attacker. The program branches on whether or not the credit card number type `form.CardType.value` is VISA, and communicates this sensitive information bit to the attacker through the URL. These flows are relatively well understood [30]. Note that these attacks demonstrate different sinks for communicating data to the attacker: the former uses the redirection mechanism, and the latter creates a new image with the source URL leading to the attacker's web site.

### 4.1.2 Beyond simple flows

While tracking explicit and implicit flows is relatively well-understood [17, 26, 34], JavaScript and DOM open up further channels for leaking information. One particular challenge is the dynamic code evaluation feature of JavaScript, which that evaluates a given string by the function *eval*(). Static analysis is bound to be conservative when analyzing programs that include *eval*(), especially if strings to be evaluated are not known at the time of analysis. However, recent progress on dynamically analyzing programs for secure information flow [32, 6, 5] shows how to enforce versions of security that are insensitive to program nontermination either purely dynamically or by hybrids of static and dynamic techniques.

Vogt et al.[35] show how a runtime monitor can be used for tracking information flow. They modify the source code of the Firefox browser, adding a monitor to the JavaScript engine. Although they adopt the simplistic high-low security lattice (see the discussion in Section 6), their enforcement can be extended with our lattice model in a straightforward fashion. With Vogt's implementation as a starting point, our larger research program pursues modular enforcement by hybrid mechanisms that combines monitoring with on-the-fly static analysis for a languages with dynamic code evaluation [5], timeout [28], tree manipulation [29], and communication primitives [5].

## 4.2 Communication channels

Any action that results in a request being sent is potentially a communication channel. While some of the actions were intended for this purpose, some have unintentionally arisen from the design of the browser. These channels need to be controlled in order to prevent information leaks. The channels can be categorized in *navigation* channels and *content-request* channels.

### 4.2.1 Navigation channels

Navigation channels are the result of navigation in the browser. When the browser navigates to a new page, a request that is sent to the target location may include any information from the current document. Some navigation channels are one-way, since the document initiating the navigation is usually unloaded to make place for the new document. Below we list possible navigation channels.

#### *Window navigation.*

When a browser window navigates to a new page, a new document is requested and loaded inside that window, replacing the current document. Window navigation is initiated by setting the *location* attribute of the window to a new address. Another way to navigate windows is by spawning

new ones, using the *window.open*() method, or navigating previously spawned windows to a new address.

*Frame navigation.*

Frame navigation may happen when a frame parent resets the *src* attribute of the frame node. This applies to frame nodes created by both the *frame* tag and the *iframe* tag. This replaces the document currently loaded in the frame with the document being requested. However, the parent of the frame persists, and access to the content from the parent frame or other frames is restricted by SOP.

*Links and forms.*

An often disregarded form of navigation is user interaction with links and forms. Note that the target of a link or a form of one of the components may be modified by another component. As a result, information about the user interaction may be leaked to an arbitrary origin.

### 4.2.2 Content-request channels

Content-request channels stem from the different possibilities for requesting new content within the browser. These channels are two-way channels, since the requested content is included in the current document.

*The XMLHttpRequest object.*

The XMLHttpRequest object allows JavaScript code to request content from the origin of the document. In mashups, this corresponds to the origin of the integrator. In current browsers, the XMLHttpRequest communication channel is the only communication channel restricted by the SOP. In a mashup, this prevents components from requesting content from arbitrary origins.

In our approach, components can communicate with their respective origin regardless of the origin of the document. The information that can be communicated in this manner is restricted by the information flow policy. This makes our approach more permissive than the current standards, while still maintaining confidentiality.

*DOM-tree modification.*

When DOM nodes are added or modified this can result in new content being requested from arbitrary origins. These requests can carry information in the URL being requested as well as in the content received. This creates an unintentional communication channel through which an attacker may leak information.

# 5. EXTENSIONS

We discuss an extension with integrity policies and applicability of our approach to server-side mashups.

## 5.1 Integrity

While the primary focus of this paper is confidentiality, we briefly discuss integrity extensions to our approach. In an extension to integrity, the security levels need to reflect both confidentiality and integrity of data. Such levels are denoted as pairs $\ell_C; \ell_I$, where $\ell_C$ is a confidentiality component, and $\ell_I$ is an integrity component of the level. Each of the components is, as previously, a set of involved origins, where integrity component enlists origins that trust that level. Therefore, integrity ordering is dual to the one

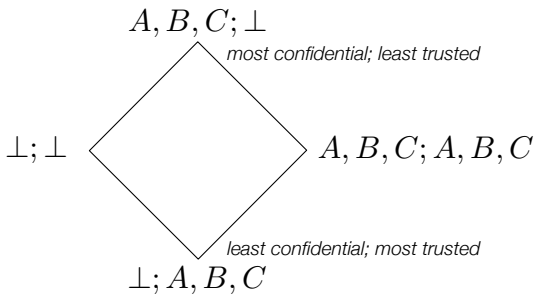


**Figure 7: Combined security lattice**

of confidentiality: the more origins the label includes, the more trusted it is. The bottom $\bot$ corresponds to trust by no origin, the least trusted level.

Figure 7 shows the combined security lattice for both confidentiality and integrity, assuming three origins $A, B$, and $C$. The least restrictive level is $\bot; A, B, C$, corresponding to the least confidential and the most trusted data. The most restrictive level is $A, B, C; \bot$ corresponding to the most confidential and least trusted data.

Note that this extension allows one to reason not only about pure integrity policies, but also about the relationship between integrity and confidentiality [25]. In both cases, policies for endorsing untrusted data [3] are important.

## 5.2 Server-side mashups

This paper has so far considered client-side mashups, where the components are combined in the browser. However, our approach may just as well be applied on the server side. If none of the mashup components contains user-specific information or if the integrator has access to all required user information, then the components may be combined on the server side. This opens up for the possibility of statically analyzing all code before delivering it to the client. A popular example of such a mashup is the social network Facebook [1], which combines static analysis of the third-party code with rewriting of the code to ensure isolation.

# 6. RELATED WORK

We discuss most related work on declassification, monitoring information-flow in browsers, and access control in mashups.

*Declassification.*

Much progress has been recently made on policies along the *dimensions of declassification* [33] that correspond to *what* information is released, *where* in the systems it is released, *when* and by *whom*. Combining the dimensions remains an open challenge [33]. Recently, the *what* and *where* dimensions, and sometimes their combinations, received particular attention [23, 4, 8, 11, 5].

The *who* dimension of declassification has been investigated in the context of *robustness* [25, 3], but in separation from *what* is declassified. Lux and Mantel [22] investigate a bisimulation-based condition that helps expressing who (or, more precisely, what input channels) may affect declassification.

The composite delimited release policy we suggest in this paper combines the *what* and *who* dimensions. The escape

hatches express the *what* and the ownership of the origins of the escape-hatch policies expresses the *who*.

### *Monitoring information flow in browsers.*

Vogt et al.[35] show how a runtime monitor can be used for tracking information flow. They modify the source code of the Firefox browser, adding a monitor to the JavaScript engine. However, their experiments show that it is often desirable for JavaScript code to leak some information outside the domain of origin: they identify 30 domains such as `google-analytics.com` that should be allowed *some* leaks. Their solution is to white-list these domains, and therefore allow *any* leaks to these domains, opening up possibilities for laundering. With our approach, these domains can be integrated into a policy with declassification specifications of exactly what can be leaked to which security level, avoiding information laundering.

Mozilla's ongoing project FlowSafe [19] aims at empowering Firefox with runtime information-flow tracking, with dynamic information-flow reference monitoring [6, 7] at its core.

Yip et al. [37] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server. By applying our method within documents, we obtain a finer-grained information-flow tracking than that of BFlow.

### *Access control in mashups.*

Access control in mashups has been an active area of recent research. Access-control policies have the known limitation compared to information-flow policies that once the data is allowed access, it can be used by an application arbitrarily, and potentially, in an insecure way. We discuss some recent highlights in this area below.

Wang et al. [36] draws analogies between mashups and operating systems and defines protection and communication abstractions for the browser in their proposal MashupOS. MashupOS expands the trust model of the SOP to better match the trust relationships found in a mashup. Two HTML tags are suggested to implement the abstractions, *ServiceInstance* and *Friv*. The tag *ServiceInstance* is used to load a service into an isolated region of memory, which can then be connected to a *Friv* which is responsible for displaying the content. The combination is similar to an iframe, but with more flexibility and protection. The isolation between content is controlled by the SOP.

OMash, by Crites et al., [13] simplifies the abstractions of MashupOS. They propose that every document should declare a public interface through which all communication with other documents is handled. OMash does not rely on the SOP for isolation. Instead, each document is isolated apart from the public interface. This does not handle cross-origin content within the same document.

Jackson et al. proposed Subspace [20], a framework for secure communication between mashup components based on existing browser features. Each component is loaded in an iframe originating from a subdomain of the integrator and communication is achieved by relaxing the domain attribute of the documents so that a communication object can be shared.

Smash [15], proposed by De Keukelaere et al., is another high-level communication framework for mashups. The components are isolated from each other, but can communicate using a high-level interface in the framework. Isolation is achieved by loading each component in an iframe. The fragment identifier channel [12] is used as a communication primitive. The communication primitive can be exchanged for a more suitable solution, as the actual communication is managed at a lower level in the framework. As this framework relies on existing browser features, it can be easily adapted. However, once a piece of information has been communicated to another component, control over its use is lost.

## 7. CONCLUSION

We have proposed a lattice-based approach to the mashup security problem. By representing origin domains as incomparable security levels in a lattice, we have a natural model, where no information between the origins is allowed, unless explicitly prescribed by a declassification policy. We have formalized the security guarantees that combine the aspects of *what* can be released and by *who*. We have discussed practical issues with security policies and integrating their enforcement into browsers.

Compared to much work on access-control policies in web browsers, we are able to track the flow of information in a more fine-grained way. Compared to other work on tracking information flow in the browser, we are able to offer a rich decentralized security-policy model.

Future work includes a formalization of a fully-fledged combination of the *what* dimension of declassification (as expressed by escape hatches) and the *who* dimension (as expressed by the *decentralized label model* [24]). Another line of work is a practical evaluation by implementation. Yet another intriguing direction focuses on integrity aspects, as sketched in Section 5.1.

## 8. REFERENCES


[1] Facebook. http://www.facebook.com.

[2] Google Maps API. http://code.google.com/apis/maps.

[3] A. Askarov and A. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS. Springer-Verlag, 2010. To appear.

[4] A. Askarov and A. Sabelfeld. Localized delimited release: Combining the what and where dimensions of information release. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 53–60, June 2007.

[5] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[6] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, June 2009.

[7] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. Technical Report UCSC-SOE-09-34, University of California, Santa Cruz, 2009.

[8] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, pages 339–353, May 2008.

[9] A. Barth, C. Jackson, and J. C. Mitchell. Securing frame communication in browsers. In *Proc. USENIX Security Symposium*, 2008.

[10] Adam Barth, Collin Jackson, and William Li. Attacks on javascript mashup communication. In *Proc. of Web 2.0 Security and Privacy 2009 (W2SP 2009)*, May 2009.

[11] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. In *Proc. IEEE Computer Security Foundations Symposium*, June 2008.

[12] J. Burke. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html, June 2006.

[13] Steven Crites, Francis Hsu, and Hao Chen. Omash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108, New York, NY, USA, 2008. ACM.

[14] D. Crockford. Making javascript safe for advertising. adsafe.org, 2009.

[15] 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 '08: Proceeding of the 17th international conference on World Wide Web*, pages 535–544, New York, NY, USA, 2008. ACM.

[16] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[18] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 2(29):198–208, August 1983.

[19] B. Eich. Flowsafe: Information flow security for the browser. https://wiki.mozilla.org/FlowSafe, October 2009.

[20] Collin Jackson and Helen J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 611–620, New York, NY, USA, 2007. ACM.

[21] J. Landauer and T. Redmond. A lattice of information. In *Proc. IEEE Computer Security Foundations Workshop*, pages 65–70, June 1993.

[22] A. Lux and H. Mantel. Who can declassify? In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, volume 5491 of *LNCS*, pages 35–49. Springer-Verlag, 2009.

[23] H. Mantel and A. Reinhard. Controlling the what and where of declassification in language-based security. In *Proc. European Symp. on Programming*, volume 4421 of *LNCS*, pages 141–156. Springer-Verlag, March 2007.

[24] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. ACM Symp. on Operating System Principles*, pages 129–142, October 1997.

[25] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.

[26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2009.

[27] Netscape. Using data tainting for security. http://wp.netscape.com/eng/mozilla/3.0/handbook/javascript/advtopic.htm, 2006.

[28] A. Russo and A. Sabelfeld. Securing timeout instructions in web applications. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[29] A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *Proc. European Symp. on Research in Computer Security*, LNCS. Springer-Verlag, September 2009.

[30] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[31] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.

[32] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[33] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, January 2009.

[34] V. Simonet. The Flow Caml system. Software release. Located at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[35] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. Network and Distributed System Security Symposium*, February 2007.

[36] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in mashupos. *SIGOPS Oper. Syst. Rev.*, 41(6):1–16, 2007.

[37] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 233–246, New York, NY, USA, 2009. ACM.