Mash-IF: Practical Information-Flow Control within Client-side Mashups

Zhou Li, Kehuan Zhang, XiaoFeng Wang
Indiana University at Bloomington
{lizho, kehzhang, xw7}@indiana.edu

Abstract

Mashup is a representative of Web 2.0 technology that needs both convenience of cross-domain access and protection against the security risks it brings in. Solutions proposed by prior research focused on mediating access to the data in different domains, but little has been done to control the use of the data after the access. In this paper, we present Mash-IF, a new technique for information-flow control within mashups. Our approach allows cross-domain communications within a browser, but disallows disclosure of sensitive information to remote parties without the user's permission. It mediates the cross-domain channels in existing mashups and works on the client without collaborations from other parties. Also of particular interest is a novel technique that automatically generates declassification rules for a script by statically analyzing its code. Such rules can be efficiently enforced through monitoring the script's call sequences and DOM operations.

Keywords: Web, Browser, Mashup, Protection, Security Model, Information-Flow Control

1. Introduction

The rapid progress of Web 2.0 technologies has brought in a whole new category of web services, such as Flickr, YouTube, Facebook and Wikipedia. Among them is client-side web application hybrid, commonly known as client-side *mashup*, a service that syndicates data and components from different sources into an single tool that runs within a web client's browser. Examples of mashups include online realtors that label locations on Google Maps with real estate data [12], financial aggregators that compile information from one's multiple accounts [3], and news aggregators that integrate different news websites [17]. Those services are gaining support from major web service providers such as Google and Microsoft, who provide APIs for mashup development.

The concept of mashups, unfortunately, fundamentally contravenes the security model adopted by current web browsers, i.e., the Same Origin Policy (SOP). The policy prevents the documents or scripts loaded from one *origin*, defined as a combination of protocol, port and host, from accessing properties of a document from a different origin [36]. It is meant to protect web contents against cross-domain attacks on the client side [30]. For a mashup, however, cross-domain communication among its components becomes a necessity. Without proper security controls in place, this opens the door to the attacks. To get out of this dilemma, both academia and industry are actively seeking effective solutions that permit but regulate the interactions among mutually-untrusting web services within browsers. Prominent examples include SMash [31], MashupOS [39] and OMash [22]. These approaches suggest new mashup-level abstractions that allow content providers and integrators to specify policies on how their contents are accessed, as well as new cross-domain channels that are mediated according to the policies.

A fundamental problem of the recently proposed approaches is that they only control the access to the contents from different domains, not the use of the contents after the access. This is often insufficient for information protection within mashups. As an example, consider a client-side account aggregator, a mashup version of personal financial management software such as `Mint.com` [3]. Such an aggregator is designed to consolidate information from a user's multiple financial accounts into a single web page. To this end, the integrator script needs to cross its domain so as to access the user's bank accounts and the passwords of these accounts [3]. However, the last thing we want is that the script transfers such sensitive information to the party we do not trust. Similar problems also happen, for example, when a gmail user uses a twitter gadget to access her twitter account but is afraid that her emails and contacts are exfiltrated by the gadget. Such a risk cannot be mitigated without proper management of information flows within mashups. Moreover, existing proposals are designed for guiding the development of new mashups and therefore, might not be suitable for managing the cross-domain channels within a large number of existing mashups. They also need cooperation from content providers and integrators to label information.

In this paper, we present a new technique, called *Mash-IF*, that makes a first step towards practical information-flow control within mashups. Mash-IF is not meant to replace existing browser-level security mechanisms, such as

SOP and those proposed in prior research [31, 39, 22]. Instead, it is designed to add an additional layer of protection of private user data through mediating dissemination. Specifically, our approach includes a new information-flow model that permits cross-domain access within a browser, but in the absence of an explicit consent from the user, disallows sensitive information to flow into any remote party other than its origin. For example, under our model, a financial aggregator can cross domains to gather one's credit or debit data from her accounts; however such access and aggregation can only happen within the user's browser and the information from the accounts is not allowed to be sent to unauthorized recipients, for example, an advertisement website. To enforce this model, our technique mediates DOM (Document Object Model) operations and function calls within scripts at the add-on level. This enables us to effectively control existing channels for cross-domain communication, because such communication typically goes through API functions supplied by web service providers. We also developed a tool to let the user label and identify information important to her without involvement of content providers and integrators.

Classic information-flow models like BLP treat a *subject* (e.g., a script) as a black-box, and as a result, their policies (e.g., "no read up" and "no write down") can become overstrict for web applications. For example, the BLP model will completely forbid a script that inputs sensitive data to communicate with untrusted websites, even if such communication does not involve any sensitive data (e.g., downloading a picture). Mitigation of this problem often relies on *declassifying* some outputs of a subject. In our research, we propose a new technique that automatically generates declassification rules for scripts. Whenever a script reads sensitive data, our approach statically analyzes its code to identify all the execution paths that could propagate the sensitive information to other part of a mashup or remote recipients. These paths are fingerprinted by their corresponding function call sequences. The rule created thereby declassifies the outputs of the script unrelated to the sequences. A prominent property of this approach is that it only needs to analyze a script once and can reuse the same rules afterwards on the script, as long as its code does not change, which can be verified using its hash value. This strategy works particularly well for mashups, as their scripts do not change often. We summarize the contributions of the paper as follows:

•**Management of mashup information flows.** We propose a new model to mediate information flows within mashups. The model is enforced through interposing on DOM operations and function calls within scripts, which enables information flow tracking and control without changing browser code. Also different from prior work is that Mash-IF does not rely on content providers' cooperation, which is critical to the practical deployment of our techniques.

•**Novel declassification approach.** We propose novel techniques that automatically analyze scripts to generate declassification rules. These rules allow our approach to achieve finer-grained control than the black-box models like BLP, which is necessary for preserving legitimate functionalities of mashups. Enforcement of these rules is based upon monitoring scripts' call sequences. Compared with the prior research [38], this approach avoids potentially intensive instruction-level monitoring.

•**Implementation and evaluation.** We implemented Mash-IF as an add-on for the Mozilla Firefox, and evaluated it using 10 real mashups. Our experimental study demonstrates the efficacy of our techniques.

The rest of the paper is organized as follows. Section 2 and 3 describe our model and its enforcement techniques. Section 4 reports our evaluation study. Section 5 compares our approach with prior work. Section 6 discusses the limitations of our techniques and Section 7 concludes the paper.

2. The Model

Basic concepts. Here, we introduce some basic concepts in Mash-IF, including *objects*, *subjects*, *flows*, *labels* and *owners*. An *object* is a repository for information. It can be local, for example, a node in a DOM tree or a cookie, or remote, for example, a web server identified by protocol, port and host. A *subject* is an information-processing unit that works on objects. In mashups, it typically refers to the scripts running in a browser[1]. A subject can operate on an object, which includes read and write. When these operations happen, *flows* that carry information moves from the object to the subject, or from the subject to the object.

Individual objects, subjects and flows are associated with a set of labels. A label can be described as a tuple (T, O), where T is a tag for a *sensitivity level* and O represents an *origin*. Our model includes three sensitivity levels, *low* (non-sensitive or public), *high* (sensitive) or *very high* (highly sensitive). These levels correspond to three trust relationships identified by MashupOS [39]: *open content* (public) that can be accessed by any domain, *access-controlled content* (sensitive) that is only accessible to other domains through mediation and *isolated content* (highly sensitive) that is completely isolated from other domains. Another relationship proposed by the prior work, *unauthorized content*, that describes the web content without the privileges of any domain, could be modeled by an additional sensitivity level *very low*. This level, however, does not seem to be necessary in the absence of the cooperation from content providers or integrators, as such content is supposed to be identified by these parties. An origin is indicative of the domain from which information within objects/subjects/flows comes. The domain a subject belongs to is described by its owner tag.

[1] Note that scripts are web content and therefore also objects.

A *reference monitor* within a browser is in charge of setting, changing and removing the labels attached to a local object, a subject or a flow. A remote object's labels, once set, cannot be modified without the user's consent. Mash-IF includes a tool that helps the user indicate to the reference monitor her sensitive information within a mashup, for example, a password field, a table including her account balance. The labels of subjects and flows are generated automatically, according to a set of rules (described below).

Security policies. The security objective of Mash-IF is to protect sensitive or highly sensitive information from being leaked to unauthorized parties. More specifically, for sensitive information, we do not want it to flow into a remote object in a different domain; for highly sensitive information, it should not be accessed from other domains even locally. The control on the highly sensitive information is more restrictive because cross-domain access to the information like authentication tokens, even happening within a browser, can lead to compromise of the user's privacy. A prominent example is cross-site request forgery (CSRF) in which the attacker disrupts the integrity of the user's session with a web site using the user's cookie [19]. To achieve that objective, our model offers three sets of security policies: propagation rules, declassification rules and control rules.

Propagation rules specify how the labels of objects/subjects/flows can be inherited by other parties:

- Without declassification, a flow from an object or a subject inherits all its labels.
- An object or a subject that receives a flow *combines* all the labels of the flow with the object/subject's own labels.
- Labels of the same origin are combined into a single label with the highest sensitivity level among them.

Declassification rules describe when to remove labels from objects/subjects/flows. An object can be declassified if all of its content has been cleaned or overwritten. Declassification of a script's outbound flows, however, cannot be achieved without understanding how the script processes sensitive information. Mash-IF includes a technique that automatically analyzes a script to generate such rules, which we elaborate in Section 3.4. Control rules disallow the flows incompatible with the security objective:

- Highly sensitive flows of one origin cannot be received by an object with a label of different origin and a subject with a different owner.
- A flow to a remote object is permitted if and only if none of the flow's labels unrelated to the remote object is sensitive or highly sensitive.

Intuitively, Mash-IF prevents sensitive information from flowing into the remote host that does not belong to the origin of the information. On the other hand, it gives green light to cross-domain operations necessary for a mashup to work properly, as long as they do not involve highly sensitive information, and keep the sensitive information that comes from other domains within browser. The threat of CSRF, for example, can be eliminated by labeling the cookie as highly sensitive.

Integrity protection. Though the current design of our model is for confidentiality protection, we still need to consider some basic integrity protection, because otherwise information leaks can happen once a subject from one domain has been compromised by the code from another domain. To this end, Mash-IF includes an integrity rule that prevents a script from writing to another script tagged with a different owner. We also disallow cross-domain modifications on important DOM objects and their properties, such as `document.location` and `document.domain`.

3. Enforcing the Model in Mashups

3.1. Overview

The enforcement mechanism of Mash-IF includes three components, a labeling tool that assists the user to mark sensitive objects, a reference monitor (RM) that tracks, controls and declassifies sensitive information flows, and a declassification rule generator (DRG) that analyzes the scripts within a mashup to build the rules for identifying their non-sensitive outputs. The interactions among these components are illustrated in Figure 1.

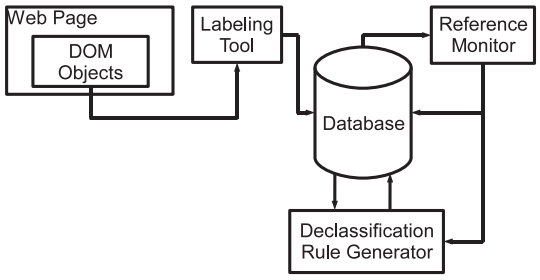

Figure 1. The System Architecture

The enforcement mechanism works as described below. Once the mashup is loaded into a browser, the labeling tool automatically identifies a set of highly sensitive objects such as cookies of different domains, and also provides an interface to let the user mark other objects, such as the text fields for accommodating passwords or displaying account balances. The DOM identifiers of these objects are recorded by our mechanism, together with their origins, for identifying them in the future.

The reference monitor interposes on all the DOM operations and scripts' function calls. Whenever a script reads a sensitive object, the label of the object is propagated to the script. If it writes to a local object, the RM determines how to adjust the labels of the object according to a declassification rule for the script. Such a rule includes sequences of DOM operations and function calls. Once the input of the script, which can be a parameter of a DOM `get` operation, is found to be sensitive or highly sensitive, the RM starts

monitoring the script's call sequences afterwards. If one of the sequences in the rule is observed, the labels of the input are propagated to the output, and the RM takes further actions according to the propagation rules and the control rules. For example, if sensitive data are about to be delivered to a remote host in a different domain, the RM can ask for the permission to proceed.

Declassification rules are automatically generated by the DRG through analyzing scripts' source code. Once the RM detects that a script invokes a DOM operation or a call that involves sensitive or highly sensitive information, and the script has not been evaluated with regards to the operation or the call before, it passes the script to the DRG. Starting from the input, the DRG statically analyzes the script to find out all the call/operation sequences that can propagate the sensitive information to the script's outputs. Such sequences do not need to be accurate, as long as they do not have false negatives and incur false positives with little impact on a mashup's normal operations. They are recorded into an embedded database shared between the RM and the DRG, together with a hash value of the script for identifying it in the future and with the entry (e.g., a DOM `get`) from which the analysis starts.

3.2. Labeling Tool

We built a prototype of the labeling tool based upon *Aardvark* [20], a Firefox extension, as illustrated in Figure 2. The tool first identifies a set of objects with highly sensitive information.Of particular importance is the authentication data such as `document.cookie` and the information related to other HTTP authentication (e.g., Basic, Digest and NTLM) and properties under `history`. The labeling tool further enables the user to mark the objects visible to them. One can use the mouse to frame elements on web content under cursor and specify their sensitivity levels. As a result, the IDs of these elements and their labels (sensitivity levels and origins) are saved into a lightweight database like *SQLite 3* [13] used in our prototype. This approach ensures that the labels cannot be accessed by scripts but can be easily retrieved by the reference monitor. A potential problem here is that a website could alter the IDs of those items, making the RM unable to locate them next time when the same content is loaded. This, however, rarely happens in practice: we monitored web pages from 10 different domains including Gmail, several banks, Facebook, and found that though their contents changed from time to time, the IDs of the objects accommodating sensitive data (e.g., passwords) were always the same. A check can also be performed as soon as web content is loaded: once the IDs of previously-labeled objects are not there, our approach can communicate this to the user and ask her to re-identify them.

3.3. Reference Monitor

The reference monitor is designed to track, declassify and control sensitive or highly sensitive information flows

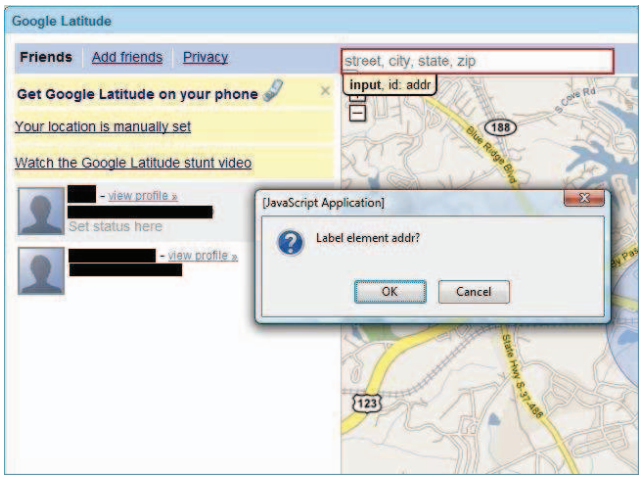

Figure 2. Labeling Tool

within a mashup. To this end, it needs to mediate all DOM accesses, including methods such as `get`, `set` and functions like `getElementById()`. This can be achieved in Firefox by extending SecurityManager, a module of XPCOM [18], by hooking the callback functions it provides. In Windows Internet Explorer (IE), these operations could be intercepted by a plug-in [32] or a DLL wrapper [37]. Also interposed by the RM are JavaScript function calls, which can be done in Firefox through call hooking. This allows Mash-IF to mediate existing cross-domain channels such as fragment identifier [21] and the `postMessage` method in HTML 5 [23].

Data-flow tracking. The RM enforces the propagation rules to track and label sensitive or highly sensitive information. Once a script is observed to read from an object whose sensitivity level is either high or very high, a declassification rule for the script is chosen to identify its sensitive output. If no appropriate rule has been found, the script is handed to the DRG for analysis.

Information-flow tracking within Mash-IF can be facilitated by the knowledge of the mashup APIs provided by content providers, such as Google AJAX APIs [28]. Such knowledge is represented by an API model in our research: for each API function, our model describes its attributes with a 3-tuple (A, O, I), where A is a set of actions, including "read" and "write", O indicates the object on which the action is taken, and I contains the function's description. Using such a model, the RM can identify a read operation on sensitive or highly sensitive objects from API calls, and follow the information flows created thereby until they are about to get into other API calls, where mediation happens according to the control rules and the descriptions of these calls. This simplifies the task of tracing sensitive data flows across a script: in many cases, the only scripts we need to care about are those from integrators.

For example, consider a mashup that extracts the ad-

dresses of one's friends from Facebook and then marks them on Google Maps. When monitoring the mashup, the RM detects that an integrator script reads from Facebook APIs the friends' profiles, which are marked as sensitive. The information derived from the profiles is tracked down across the script according to a declassification rule until a Google Maps API `getLatLng()` is invoked with the addresses. This API is associated with a write to a remote object, the domain of Google. Since the information involved in the operation is sensitive, declassification is required. The RM then uses the description of the function to explain to the user what she could leak out once the call is allowed to proceed. With the user's consent, her response can be saved as either a control rule or a declassification rule for handling the same situation in the future.

This approach could put too much faith in the content provider's specifications of its APIs. If this is deemed too risky, we can always choose to follow sensitive information into individual API functions. Another issue the RM needs to deal with is how to label the content dynamically downloaded from websites, for example, the data that come out of `XMLHttpRequest`. Our solution is a policy that infers the sensitivity levels of an output according to its corresponding input: if sensitive or highly sensitive information (e.g., cookies) is involved in `XMLHttpRequest`, the content it obtains will be labeled as high or very high.

Declassification. Whenever a subject writes an object, or passes information to another subject, an information flow that carries the subject's labels occurs. This flow can be declassified by the RM in accordance with a set of declassification rules. A declassification rule can be specified by the user directly or automatically identified from a script. A user-defined rule is associated with a specific mashup, and can be represented as (d, l), where d can be a script's function or an object receiving a flow with a set of labels l. For example, one can declassify the flow with a label (sensitive, Facebook) when it enters the Google Maps API `getLatLng`. The rule generated for a script includes sequences of DOM accesses and function calls. These sequences are organized into a set of trees, with each tree defined as $T(N, E)$, where $N = \{n | n \text{ is function name } \}$, and $E = \{\langle x, y \rangle | x \in N, y \in N\}$. Examples are illustrated in Figure 3. Each tree is rooted at a specific program location where a function or a DOM access inputs from sensitive or highly sensitive objects, and its leaves output data to other subjects or objects. A path from the root to a leaf describes an execution path within the script that propagates information from the input to the output. Such a rule declassifies the output produced by the execution that does not correspond to any path on these trees. To enforce the rule, the RM monitors all the calls and DOM operations from a script once it reads a sensitive or highly sensitive object, and attempts to identify their sequence from a tree rooted at the input instruction. If this fails, the output is regarded as non-sensitive. Otherwise, control will be taken to prevent information leaks. Since this type of rules is associated with individual scripts, the RM needs to check the hash value of a script once it is downloaded, to ensure that its code has not been modified. All declassification rules and scripts' hash are stored in a SQLite database in our prototype. A concern for the declassification rules is that

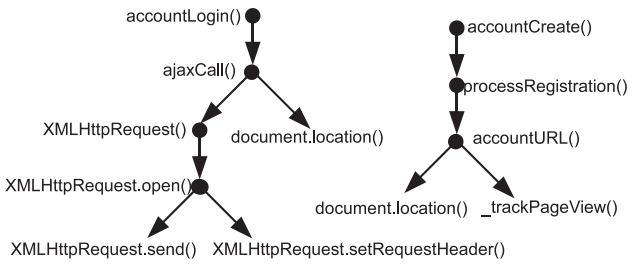

Figure 3. The Declassification Tree

they are modeled in a *public-default* way: as long as no rule matches a call sequence, the output of a script is deemed clean. Behind this treatment is the pragmatic consideration that most data flows within legitimate mashups can be non-sensitive, and tracking only sensitive flows can avoid a significant performance impact. On the other hand, such a benefit does not necessarily come at the cost of the privacy assurance Mash-IF can achieve: as elaborated in Section 3.4, the DRG includes a static analysis mechanism that works effectively on a subset of the JavaScript language demonstrated in prior research [26] to be completely analyzable; for the statements outside that subset, our approach treats them as black-boxes, as did the BLP model, and considers all of their outputs to be sensitive if any of their inputs is sensitive. In this way, Mash-IF ensures that no execution path propagating sensitive data will slip under the radar.

Data-flow control. Information-flow control happens whenever the RM observes cross-domain access to highly sensitive objects or delivery of sensitive information to a remote host in a different domain. Access to objects like `domain.cookie` needs to go through DOM operations, which our approach mediates. Networking activities of a script, such as `XMLHttpRequest`, can be monitored through function-call hooking. Setting the `location` property of `document` or the source of an image in web page can also leak out sensitive information to a different domain, which can be monitored through interception of DOM operations. In addition, the RM controls the invocation of the mashup APIs that could cause information leaks, such as `getLatLng`. Actually, scripts are not the only mashup components in touch with the outside: for example, sensitive data can be directly sent out by form submission. Such activities, however, will trigger DOM events, which are monitored by the RM. Once a violation of the control rule is found, the RM either directly stops the corresponding activities, or suspends them and asks for the user's permission to proceed. To minimize the user's involvement in

this process, our approach can record her decision on one access request, and given her consent, extend it to a policy that can be enforced automatically under the same circumstance, without consulting the user again. On the other hand, all the policy settings and rules are allowed to be customized by the user.

3.4. Declassification-Rule Generator

Once a script's DOM operation or function call imports sensitive or highly sensitive information, the RM identifies the program location where the access happens, and attempts to retrieve a declassification rule from the database according to the location. If the attempt fails, the RM passes the script to the DRG to generate the rule. The DRG performs a static analysis on the script's source code to extract all its execution paths that could propagate sensitive information to an output, and then uses the sequences of the calls and DOM operations on these paths to build the rule.

JavaScript is well known to be hard to analyze statically [26], due to some of its language features, for example, lack of static typing, Lambda functions and `eval` function, that make the information flows within a script difficult to determine without running the code. However, a thorough and accurate analysis of JavaScript code is not necessary here: what we want are just a set of "good enough" declassification rules with small false positives and negligible false negatives. The classic BLP model treats a subject as a black box. Whenever the subject receives a sensitive input, all of its outputs are deemed sensitive. This treatment is just too coarse-grained to be useful for controlling real scripts: many of them need to communicate with the website inappropriate for receiving the data they read, though such communication often involves no sensitive data. The DRG is designed to refine such an information flow model. Our approach seeks all the execution paths that propagate data from a fixed input to outputs. This analysis does not need to be accurate: we can always treat a complicated JavaScript command as a black boxes, and track all of its outputs once it operates on sensitive data.

Prior research shows that accurate static analysis can be done on a subset of JavaScript language [26]. The DRG we implemented works on this subset, called JavaScript$_{SA}$, treating other language structures as black boxes. As a result, we can achieve a very good coverage, identifying all paths that propagate sensitive data. On the other hand, the chance of false positive, i.e., involving the path that actually does not leak information, is reasonably low, as the JavaScript statements not in the subset are actually not frequently used by legitimate scripts [26]. In our research, we built the DRG prototype as a Firefox plug-in. Following we elaborate our design and implementation.

Building CFG. Before analyzing information flow within JavaScript codes, the DRG first generates a control flow graph (CFG). It starts from the Abstract Syntax Tree (AST) produced by calling a set of API functions [25] of *SpiderMonkey*, the JavaScript virtual machine of the Firefox [24]. The CFG consists of a set of nodes and arcs, where each node represents a *basic block*, and each arc represents the transition of an execution from one basic block to another. A basic block is a statement sequence that begins with a jump target, ends at a branch statement and does not include another branching.

Dataflow analysis on JavaScript$_{SA}$. Static analysis of JavaScript is hard, but can still be achieved if we focus on a subset of the language, which has been shown to be practical and effective by the prior work [26]. In our research, we identified such a subset, JavaScript$_{SA}$, that can be thoroughly evaluated through static analysis, and came up with a set of rules to describe how the labels of sensitive data are propagated by the statement within that set.

The analysis starts with a set of variables that accommodate the data a script reads from DOM objects. Each of such variables v is given a set of labels $label_v = \{v\}$. For every statement along the CFG, our analyzer propagates labels from the sets associated with its input variables to those of its output variables, according to the statement's propagation rule. An execution path that moves sensitive data from an input to an output is identified if the variables holding the data to be delivered to the Internet are found to be associated with some labels. The path is then extracted for building a declassification rule. The problem of finding such a path is essentially the well-known *reaching definition problem* [33] and thus can be solved using existed classic algorithm for iterative data flow analysis [33]. Our approach also includes an inter-procedure analysis that tracks sensitive data flows across different functions.

One important issue we need to deal with is *alias*. In JavaScript, assigning an object variable "`a`" to another variable "`b`" will establish an alias relation between them, and as a result, every operation on one variable will also happen to the other. Such a relation is transitive, and can therefore involve a large set of variables. This can impede our data-flow analysis, as the DRG does not know that an access to a variable actually generates a sensitive information flow from another one. We solved this problem in our research by annotating the alias relation among different variables in a data structure that the DRG maintains to track them. Those variables are linked together by an alias chain. Whenever an operation is found to transfer sensitive data to one of them, the DRG propagates the sensitive label to every member in the chain.

Analysis of the statements outside JavaScript$_{SA}$. Dynamic features of JavaScript, such as `eval()` and variable index of array, are not included in JavaScript$_{SA}$. These features are known to be the part of the language hard to analyze statically. To track data flows in their presence, the DRG adopts a strategy that treats the statement involving these features as a black box and applies the BLP model to

pessimistically estimate its sensitive outputs.

As an example, consider the statement `id = name[x]`, where `x` is an index variable and an element in the array `name` is known to be sensitive. The question here is how to determine the label of `id`, which is sensitive if `x` points to the sensitive element, and nonsensitive otherwise. The hurdle here, however, is that the value of `x` can only be observed during the runtime. As a result, a static analysis cannot determine whether the sensitivity label should be propagated to `id`. Our solution here is to "black-box" the statement, labeling `id` as sensitive as long as a single element of `name` is sensitive, as the BLP does. Similar treatments also happen to objects and complex expressions: for an object with a sensitive attribute, we deem that all its attributes are sensitive if the DRG cannot statically determine which attribute a statement will operate on; for a complex expression, we label all its outputs as sensitive, once one of its sub-expressions is found to be sensitive and the static analysis does not know whether an operation on the expression involves that sub-expression. Inevitably, such an approach will cause false positives. This is acceptable, however, as long as the declassification rule generated thereby does not significantly impair the legitimate functionality of a script, which we found was often true according to our experimental study (Section 4).

Problems happen, however, when statements like `eval()` are encountered. The function `eval()` runs a string as code, which can read or write any variables that carry sensitive labels. Since the input of `eval` may not be observed without running a script, the only option we have is rolling back to the BLP model, treating all its outputs as sensitive once a JavaScript reads a sensitive object and also executes such a statement. Fortunately, both our research and prior work [26] found that most legitimate scripts do not include `eval()`. This ensures that in most cases, our approach can still do much better than the BLP.

Rule generation. After discovering the execution paths that could leak sensitive or highly sensitive information, the DRG moves to build a tree that fingerprints individual paths with sequences of function calls and DOM operations. Such sequences are actually recorded while the script is being analyzed. Once a data-leaking path has been found, the DRG dumps its sequence to the database. Each sequence is also annotated by the program location where sensitive data are read, and an output function through which sensitive data flows get out of the script, for example, `XMLHttpRequest .open()`. The program location, together with the hash value of a script, is used to retrieve from the database the sequences associated with the script.

4. Evaluation

The purpose of our experimental study is to understand the efficacy of our techniques in controlling information flows within mashups. To this end, we ran our prototype on 10 real client-side mashups. Half of them came from iGoogle Gadget [9], including Google Gmail Gadget [6], Google Finance Portfolios [5], Facebook Gadget [2], Google Search eBay [8] and Google Latitude [7]. Those mashups are all very popular, with tens of thousands or even millions of users. The other half were from other integrators, including WalkJogRun [16], ImageLoop [10], AuctionReminder [1], Twitterfall [14] and Pingfm [11]. . Following we elaborate the study that evaluated the effectiveness (Section 4.1) and performance (Section 4.2) of our techniques.

4.1. Effectiveness

The experiment. To evaluate our prototype against each mashup, we first used our labeling tool to mark a set of sensitive objects in different web domains, including login boxes, the entries accommodating user input and the text items including account information. Then, we performed the operations on the labeled objects, such as login, as described in Table 1. Scripts involved in these operations, except the APIs whose models were known to our prototypes, were analyzed by the DRG. The declassification rules created in this way were utilized by the RM to enforce the two control rules of our model.

To evaluate the effectiveness of the enforcement, we performed a differential analysis in which each mashup was executed twice, with different contents in a labeled object: if an output was found to be different in these two runs, it was treated as sensitive. This analysis was based upon Firebug [34], a Firefox add-on that intercepts and records all the traffic generated by the browser, including HTML requests (including GET, POST and PUT) and `XMLHttpRequest`. In the experiment, whenever a sensitive output was found, we checked the Firebug log to find out the party to receive data. If the recipient was not in a same domain as the labeled object, a false negative was recorded. Once our prototype blocked an attempt to transfer data, we also checked the log to determine whether the data was sensitive. If not, a false positive was logged.

Findings. For each mashup, we found that our prototype successfully mediated its sensitive information flows. Our differential analysis discovered sensitive outputs in eight of those mashups. Among them, seven actually transferred the data to the same domains. Only one was found to leak information to a different domain. Our prototype identified all of them, but only blocked the last one. As a result, the experiment reported zero false positive and false negative. The detailed outcomes are described in Table 1.

Twitterfall and Pingfm are the two mashups that were not found to export any sensitive data. All their information flows derived from user data were kept local, within the browser. Google Finance Portfolios is among the seven that transferred sensitive information to the same domains. What it did was delivery of the stock code we en-

Table 1. Effectiveness

Mashup	API Providers	Operation	Labeled Objects	API Model	Test Result	Log Result	Output Objects	Input and Output Origins
WalkJogRun	Google Map	Login	Input box	_trackPageView	Block	Leak	User ID	Different
ImageLoop	Facebook	Upload photo	Function	doImport	No Block	No Leak	User Photo	Same
Facebook Gadget	Facebook	Search friends	Input box	_gel	No Block	No Leak	Friends Name	Same
Gmail Google Gadget	Gmail	Open one mail	Div		No Block	No Leak	Mail Subject	Same
Google Finance Portfolios	Google Financial	Add one stock	Input box		No Block	No Leak	Stock ID	Same
AuctionReminder	Google Calendar	Add one event	Select Option		No Block	No Leak	Event Type	Same
Google Search eBay	eBay	Search Goods	Input box		No Block	No Leak	Item Name	Same
Twitterfall	Twitter Google Map	Search twitter	Input box	getLatLng	No Block	No Leak		
Pingfm	Facebook Twitter	Type message	Textarea		No Block	No Leak		
Google Latitude	Google Contact Google Map	Set Location	Input box	getLatLng	No Block	No Leak	User Address	Same

Table 2. Performance

Case	Operation	Native (μs)	Analysis and Monitor (μs)	Overhead	Monitor Only (μs)	Overhead
1	Login	66	108	63.64%	74.9	13.48%
2	Upload photo	575.8	1118	94.16%	687.6	19.42%
3	Search friends	127	221	74.02%	162.7	28.11%
4	Open one mail	99.2	154	55.24%	110.2	11.09%
5	Add one stock	9.5	55	478.95%	10.6	11.58%
6	Add one event	14.7	39	165.31%	17	15.65%
7	Search Goods	14.6	44	201.37%	20.7	41.78%
8	Search twitter	43.6	74	69.72%	60.3	38.30%
9	Type message	4.4	12	172.73%	6.1	38.64%
10	Set Location	4.1	8	95.12%	4.4	7.32%

tered to Google Financial. WalkJogRun [16] is the one that leaked out sensitive user information. It required `userid` and password to log in. Once the login button was clicked, we found that a function from the integrator, `accoutLogin`, was triggered. The function not only authenticated the user input and sent the data to its website, but also passed `userid` to `_trackPageView`, an API provided by Google Analytics [4]. Our model for the API showed that it would deliver the input to the UCFE (Urchin Collector Front-end) [15], a different domain. This cross-domain operation was captured by our prototype and blocked according to the control rule.

4.2. Performance

The performance of our prototype was evaluated under three scenarios. In the first one, which we call "*native*", the mashup was executed in a browser that did not include our implementation. The second scenario, "*analysis and monitor*", involved statically analyzing scripts and utilizing the declassification rules generated thereby to track information flows. The third one, "*monitor only*", emulated the situation when a script analyzed before was executed again. In this case, the rule associated with it (sequences of DOM access and function calls) was applied to propagate sensitive labels across the script. In all three scenarios, a set of operations, as illustrated in Table 2, were performed within each mashup, and delays incurred by them were recorded. The experimental results reported here were averaged over multiple runs and the computation platforms we used included a 2.00 GHz AMD Turion-PC with 3 GB RAM, on which Windows Vista and Firefox 3.0.8 was installed.

From Table 2, we can observe that a significant overhead was incurred when the prototype had to analyze a script: the delays reported here ranges from 50% to 500%. This, however, happens only when a script is first encountered, or when it exhibits the behavior never seen before. Mashups are typically used repeatedly by users and their code rarely changes. As a result, the overhead for "monitor only" is actually more likely to reflect the delay those users experience. In this scenario, we observed the latencies below 20% in six mashups, 30% in one and around 40% in the other three. Such an overhead is often completely overshadowed by other operation delays such as communication time. For example, we measured the login process of Facebook for 5 times, and found the delay the user experienced was only 2.9%, though the overhead for the login operation alone was 13.48%. We also found that API models could greatly reduce the delays, which is understandable, as many complicated operations for analyzing scripts and monitoring its behaviors were avoided in this case. Another finding is that our prototype did not affect the operation of a browser at all when no sensitive objects were labeled: we used the browser equipped with Mash-IF to surf hundreds of websites with no sensitive information, and did not experience any noticeable delay.

5. Related Work

Access control in mashup. Mashups are built upon cross-domain access, which runs contradictory to the Same Ori-

gin Policy, the security policy widely used in today's web browsers. A solution to the problem is mediated cross-domain access, which has been intensively studied recently. Prominent examples include MashupOS [39], OMash [22] and Smash [31]. MashupOS [39] applied the abstractions from operating systems to protect the resources within mashups and proposes four trust relationships between content providers and integrators. Those relationships can also be described in our model. OMash [22] suggests a new abstraction that treats web pages as objects, using a predefined method "`getPublicInterface()`" to expose its public data. SMash [31] adopts an abstraction of *components* and communication *channels*, where principles are mapped to the components connected to an event hub through input/output ports. The event hub here is a publish/subscribe system with many-to-many channels through which messages are published and distributed. A fundamental limitation of those models is their focus on the access to the data, rather than the use of the data after the access happens. This is insufficient for many practical mashups that need to use one's sensitive data but cannot be trusted not to send them to an unauthorized party. Moreover, those approaches require the cooperation among content providers, integrators and the clients, which render them hard to deploy. Mash-IF is designed to address these issues: it controls sensitive data at information-flow level and works on the existing mashups even in the absence of collaborations from other parties.

Script Analysis. Program analysis techniques have been increasingly used to analyze scripts and other Web applications. In [38], both dynamic and static analyses are applied to track the sensitive information flow at the instruction level to mitigate the threat of cross-site scripting. Instruction-level dynamic taint analysis is certainly more accurate than our approach, which is based upon monitoring call sequences. However, such an analysis needs to monitor every binary instruction. It also needs to modify browsers, particularly, the source code of JavaScript virtual machine, while Mash-IF can be built into browser add-ons. The static analysis Mash-IF utilizes can thoroughly analyze a subset of JavaScript language, as did in prior research [26]. Unlike that work, which intends to detect untrusted widget [26], our aim is to roughly track data flows across scripts, and therefore can always employ the black-box model like the BLP whenever hard-to-analyze statements (those outside the subset) are encountered. Another important contribution of our approach is utilization of a fingerprint of the execution (sequences of calls and DOM access) that could leaks to information leaks to efficiently track sensitive data during the runtime. This approach is novel, up to our knowledge, and turns out to be effective, according to our experiment study (Section 4). Script analysis has also been used in BrowserShield [35] and Pixy [29]. BrowserShield relies on a proxy on the firewall to interpose on JavaScript statements in web pages and then analyze it using the interposition functions. It is designed for detecting exploits of the vulnerabilities within a browser, not for tracking information flows. On the other hand, the technique it uses to wrap statements might also be applied to mediate DOM access, a necessary step for extending Mash-IF to the Internet Explorer (IE). Pixy also statically analyzes scripts. However, it is meant to work on PHP scripts on the server side, for discovering such vulnerabilities as SQL injection, while our focus is scripts running in browsers.

DOM operation interception. An important technique used in Mash-IF is DOM operation interception. Hallaraker et al. [27] proposed a model to audit the execution of JavaScript and implemented it in the Firefox, which is similar to the technique we adopted to mediate DOM access. SpyShield [32] mediates DOM events from the IE to contain malicious add-ons. In the absence of source code, further study is needed to explore the potential to achieve a full mediation of DOM access within the IE.

BFlow. Concurrently with our research and independently, Yip et al. proposes BFlow [40], a technique that controls the information flows within browsers. However, Mash-IF differs significantly from BFlow in the following perspectives. First, BFlow tracks data at the granularity of *protection zones*, groups of browser frames, while our approach can achieve much finer-grained control, to individual DOM objects. Second, BFlow requires an explicit declassification either by the user or the developer, while our work can automatically generate declassification rules through analyzing scripts. Third, BFlow needs the support from web servers to label sensitive data. In contrast, Mash-IF is designed to work on the client side, and can therefore record and identify sensitive objects without the help from the server.

6. Discussion

The information-flow model we propose here is designed to work on the client side, in the absence of the cooperation from content providers and integrators. This is in contrast to the prior work [40] that needs effort from web clients, servers and integrators. However, client-side control can be limited, as a web client may not have sufficient information to determine the sensitivity and even the origins of some contents. On the other hand, requirement of collaborations from all parties can impede the practical deployment of a technique. We are considering design of a new model to take advantage of the information from different parties when available, and also incrementally deployable. Another important extension we plan to work on is incorporation of integrity protection into our system.

Identifying and labeling subjects/objects need to be further automated. An important issue here is how to determine the sensitivity of the inputs from a user. A straightforward solution is to let the user mark an object to which she will key in sensitive data like password. This, however, can be inconvenient. A better solution should be more automatic: for example, a labeling tool can mark an input

from the user as sensitive if the same content was found in a "password" field before. Such labeling techniques will be investigated in our follow-up research.

Effective mediation of DOM access and function calls is the very foundation for information-flow control. We demonstrated that this can be conveniently achieved in Firefox through browser extension. A further study will seek effective ways to apply our technique to Internet Explorer, to which mediation without changing browser code is critical.

Our current implementation of the DRG only focuses on data flows and relies on static analysis. Natural extensions of this approach include control-flow analysis, and combination of static and dynamic analysis. These technologies have been demonstrated in prior research [38] to be effective, and there is little doubt that they can be integrated into our system. The special part of our approach is fingerprinting information-leaking paths discovered by the analysis with call sequences, which can be efficiently checked when the same script is used again. We believe that the application of this technique is clearly beyond mashups. It is conceivable that one can use the same approach to effectively track and control information flows within server-side scripts, and other applications.

7 Conclusion

In this paper, we present Mash-IF, a new technique for information-flow control on the client side. Our technique enables cross-domain access to sensitive information within a browser, but forbid propagation of such information to an unauthorized remote host. We designed our technique in a way that it can work without the collaborations from other parties and mediate the existing channels for cross-domain communications. We also developed a new technique that automatically builds declassification rules for a script by statically analyzing its code. Such rules can be efficiently enforced within a browser by monitoring sequences of function calls and DOM operations. We evaluated Mash-IF against real mashups, and discovered a previous-unknown privacy problem in one of them.

Acknowledgement

We thank our shepherd Christopher Stewart for his guidance on preparing the final version of the paper, and anonymous reviewers for their valuable comments. This work was supported in part by the National Science Foundation under Grant No. CNS-0716292.

References

[1] Auctionreminder. http://www.auctionreminder.net/.
[2] Facebook gadget. http://www.google.com/ig/directory?hl=en&type=gadgets&url=www.brianngo.net/ig/facebook.xml.
[3] Free personal finance software, budget software, online money management and budget planner, mint.com. http://www.mint.com/.
[4] Google analytics. http://www.google.com/analytics/.
[5] Google finance portfolios. http://www.google.com/ig/directory?hl=en&type=gadgets&url=www.google.com/ig/modules/finance_portfolios.xml.
[6] Google gmail gadget. http://www.google.com/ig/directory?hl=en&type=gadgets&url=www.google.com/ig/modules/builtin_gmail.xml.
[7] Google latitude. http://www.google.com/ig/directory?hl=en&type=gadgets&url=www.google.com/ig/modules/fv.xml.
[8] Google search ebay. http://www.google.com/ig/directory?hl=en&type=gadgets&url=www.netremote.com/rss/ebay.xml.
[9] igoogle. http://www.google.com/ig.
[10] Imageloop. http://www.imageloop.com.
[11] Ping.fm. http://ping.fm/.
[12] Savvyrent.com. http://www.savvyrent.com.
[13] Sqlite. http://www.sqlite.org/.
[14] Twitterfall. http://twitterfall.com/.
[15] Urchin 5 web analytics software. http://www.google.com/analytics/urchin_software.html.
[16] Walkjogrun. http://www.walkjogrun.net.
[17] What's out? http://whatsout.net/.
[18] Xpcom - mdc. https://developer.mozilla.org/en/XPCOM, 2008.
[19] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88, New York, NY, USA, 2008. ACM.
[20] R. Brown. Aardvark firefox extension. http://karmatics.com/aardvark/, 2005.
[21] J. Burke. Cross domain frame communication with fragment identifiers. http://tagneto.blogspot.com/2006/06/cross-domain-frame-communication-with.html, 2006.
[22] S. Crites, F. Hsu, and H. Chen. Omash: enabling secure web mashups via object abstractions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108. ACM New York, NY, USA, 2008.
[23] I. H. et al. Html 5 working draft. http://www.whatwg.org/specs/web-apps/current-work/.
[24] M. Foundation. Spidermonkey (javascript-c) engine. http://www.mozilla.org/js/spidermonkey/, 2009.
[25] M. Foundation. Spidermonkey jsparse.c cross-reference. http://mxr.mozilla.org/mozilla/source/js/src/jsparse.c, 2009.
[26] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th USENIX Security Symposium*. USENIX society, 2009.
[27] O. Hallaraker and G. Vigna. Detecting malicious javascript code in mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.
[28] G. Inc. Google ajax apis. http://code.google.com/apis/ajax/, 2009.
[29] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *2006 IEEE symposium on security and privacy*, pages 258–263, 2006.
[30] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 58–71. ACM New York, NY, USA, 2007.
[31] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Proceeding of the 17th international conference on World Wide Web*, pages 535–544. ACM New York, NY, USA, 2008.
[32] Z. Li, X. Wang, and J. Y. Choi. Spyshield: Preserving privacy from spy add-ons. In *Recent Advances in Intrusion Detection (RAID)*, pages 296–316, 2007.
[33] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[34] I. Parakey. Firebug - web development evolved. http://getfirebug.com/, 2009.
[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. In *Proc. OSDI*, 2006.
[36] J. Ruderman. The same origin policy. http://www.mozilla.org/projects/security/components/same-origin.html, 2008.
[37] K. Skilling. Function call tracing in jscript. http://www.codeproject.com/KB/scripting/JScriptDebug.aspx, 2007.
[38] 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 *Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, 2007.
[39] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in mashupos. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 1–16, 2007.
[40] A. Yip, N. Narula, M. Krohn, and R. Morris. Privacy-preserving browser-side scripting with bflow. In *EuroSys'09*, 2009.