2011

ToMaTo: A Trustworthy Code Mashup Development Tool

Jian Chang
University of Pennsylvania, jianchan@cis.upenn.edu

Krishna Venkatasubramanian
University of Pennsylvania, vkris@cis.upenn.edu

Andrew G. West
University of Pennsylvania, westand@cis.upenn.edu

Sampath Kannan
University of Pennsylvania, kannan@cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

See next page for additional authors

ToMaTo: A Trustworthy Code Mashup Development Tool

Abstract
Recent years have seen the emergence of a new programming paradigm for Web applications that emphasizes the reuse of external content, the mashup. Although the mashup paradigm enables the creation of innovative Web applications with emergent features, its openness introduces trust problems. These trust issues are particularly prominent in JavaScript code mashup - a type of mashup that integrated external Javascript libraries to achieve function and software reuse. With JavaScript code mashup, external libraries are usually given full privileges to manipulate data of the mashup application and executing arbitrary code. This imposes considerable risk on the mashup developers and the end users.

One major causes for these trust problems is that the mashup developers tend to focus on the functional aspects of the application and implicitly trust the external code libraries to satisfy security, privacy and other non-functional requirements. In this paper, we present ToMaTo, a development tool that combines a novel trust policy language and a static code analysis engine to examine whether the external libraries satisfy the non-functional requirements. ToMaTo gives the mashup developers three essential capabilities for building trustworthy JavaScript code mashup: (1) to specify trust policy, (2) to assess policy adherence, and (3) to handle policy violation. The contributions of the paper are: (1) a description of JavaScript code mashup and its trust issues, and (2) a development tool (ToMaTo) for building trustworthy JavaScript code mashup.

Keywords
Mashup, Trust, JavaScript, Code Analysis

Comments
5th International Workshop on Web APIs and Service Mashups (Mashups '11), September 14, 2011, Lugano, Switzerland.

Author(s)
Jian Chang, Krishna Venkatasubramanian, Andrew G. West, Sampath Kannan, Oleg Sokolsky, Myuhng Joo Kim, and Insup Lee

ToMaTo: A Trustworthy Code Mashup Development Tool *

Jian Chang[1], Krishna Venkatasubramanian[1], Andrew G. West[1],
Sampath Kannan[1], Oleg Sokolsky[1], Myuhng Joo Kim[2], and Insup Lee[1]

[1]Department of Computer and Information Science - University of Pennsylvania, USA
{jianchan, vkris, westand, kannan, sokolsky, lee}@cis.upenn.edu
[2]Department of Information Security - Seoul Women's University, Republic of Korea
mjkim@swu.ac.kr

ABSTRACT

Recent years have seen the emergence of a new programming paradigm for Web applications that emphasizes the reuse of external content, the *mashup*. Although the mashup paradigm enables the creation of innovative Web applications with emergent features, its openness introduces trust problems. These trust issues are particularly prominent in *JavaScript code mashup* – a type of mashup that integrated external Javascript libraries to achieve function and software reuse. With JavaScript code mashup, external libraries are usually given full privileges to manipulate data of the mashup application and executing arbitrary code. This imposes considerable risk on the mashup developers and the end users.

One major causes for these trust problems is that the mashup developers tend to focus on the functional aspects of the application and implicitly trust the external code libraries to satisfy security, privacy and other non-functional requirements. In this paper, we present ToMaTo, a development tool that combines a novel trust policy language and a static code analysis engine to examine whether the external libraries satisfy the non-functional requirements. ToMaTo gives the mashup developers three essential capabilities for building trustworthy JavaScript code mashup: (1) to specify trust policy, (2) to assess policy adherence, and (3) to handle policy violation. The contributions of the paper are: (1) a description of JavaScript code mashup and its trust issues, and (2) a development tool (ToMaTo) for building trustworthy JavaScript code mashup.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based Services; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (*e.g., *firewalls)*

General Terms

Design, Languages, Verification

Keywords

Mashup, Trust, JavaScript, Code Analysis

1. INTRODUCTION

A popular Web programming paradigm, the *mashup*, has been widely adopted to quickly develop new Web applications by reusing existing third-party content. The word "mashup" also refers to a hybrid Web application that combines content from two or more sources to create a new service(s). Examples of mashup include third-party advertisements in web pages, Web widgets in portal sites such as iGoogle, and external code libraries. For instance, `mlb.com`, a mashup that presents information about Major League Baseball, uses jQuery JavaScript libraries [13] for visual effects, services from `omnitrue.com` for Web traffic analysis, and advertisements from `doubleclick.net`.

External content is usually accessible through certain form of application programming interface (API). The APIs used in mashup can be classified into two categories: data-centric and code-centric. *Data-centric* APIs often serve as an information source for mashup applications. For example, news and video feeds are often considered to be data-centric. In contrast, *code-centric* APIs focus on providing specific functionality and software reuse for mashup applications. Examples include visualization and presentation libraries, such as YUI [25] and jQuery UI [13]. There also exist external APIs that are both data and code centric. They can be viewed as the hybrid of the two categories, and examples include the Google Map API [9] and the AdSense API [8].

Although the mashup paradigm simplifies the creation of new Web applications, this openness to external content also creates new vulnerabilities. In this paper, we focus on examining the trust issues associated with JavaScript code mashups – a specific type of code-centric mashup that utilizes JavaScript. A JavaScript code mashup is a Web application that integrates functionalities from external JavaScript libraries (often developed by third-party) to build a new service(s). To create a JavaScript code mashup, developers identify JavaScript libraries to use as basic building blocks for their mashup, utilize the libraries to create an integrated application, and finally test and publish the mashup. It is important to note that the current development practice is often function-driven. That is, the developers focus

*This research was supported in part by ONR MURI N00014-07-1-0907. POC: Insup Lee, lee@cis.upenn.edu

on choosing code libraries that meet the functional requirements of their mashup(s). Consequently, the developers implicitly trust the code libraries to satisfy security, privacy and other non-functional requirements.

This blind trust in the external libraries used by mashup applications is problematic. As demonstrated in [24], 66.4% of 6805 popular mashup sites include external JavaScript libraries into the top-level document of their Web pages. These external libraries have full privileges to manipulate data and can execute arbitrary JavaScript code. As a result, the information security of these mashups is a serious cause for concern [2]. For example, external libraries can manipulate or leak cookies belonging to the mashup site, which may lead to serious confidentiality (privacy) violations [12]. Further, a malicious external script can potentially navigate the end user's browser to an elaborately designed phishing site, tampering with the integrity of the mashup application.

Motivated by these observations, we propose **ToMaTo**, a **T**rustworthy c**o**de **Ma**shup development **To**ol. ToMaTo allows mashup developers to specify trust policies that capture the information security requirements over external JavaScript libraries used in their mashup applications. The semantics of the trust policy are expressive, which control access to critical function invocations and protect memory access. Taking the trust policy and the mashup application code as its input, ToMaTo evaluates each external library for its policy adherence. The evaluation process uses static code analysis to extract useful program information (*e.g.,* call graph, alias information) and combines with an information flow solver to identify potential policy violations. The policy evaluation process is sound, and ToMaTo will trigger warnings to pinpoint all the potential policy violations. The mashup developers can then use the warning information to design corresponding countermeasures to avoid or minimize the potential trust issues. The principal *contributions* of the paper are: (1) a description of JavaScript code mashup and its trust issues, (2) a development tool for building trustworthy JavaScript code mashup by combining a novel policy description language and a static code analysis-based policy adherence assessment engine.

The remainder of the paper is organized as follows. The formal definitions of JavaScript code mashup and other important notions are introduced in Section 2. Section 3 describes the problem statement and an overview of our approach. The detailed system design of ToMaTo is presented in Section 4, including the trust policy language, the policy adherence assessment process, and the handling of policy violation. Related work is briefly surveyed in Section 5, and Section 6 concludes the paper.

2. BACKGROUND

In this section, we first describe the basics of JavaScript and the Document Object Model (DOM), and define the basic terminologies that we use in the rest of the paper. Finally, the life cycle of JavaScript code mashup is discussed.

2.1 JavaScript and DOM Basics

JavaScript is an implementation of the ECMAScript language standard [7], and it is widely supported by Web browsers to enhance Web pages dynamics. JavaScript is designed to be a prototype-based, object-oriented language with first-class function support and many other dynamic features.

The Document Object Model (DOM) is a language-neutral and platform-independent interface that allows programs and scripts to dynamically access and update the content, the structure, and the style of documents [5]. DOM is also well-supported by most Web browsers and represents every HTML document as a tree structure. An example DOM tree is illustrated in Figure 1. Every HTML element, attribute, and embedded text has a corresponding node in the DOM tree. The DOM tree can be modified through DOM APIs that the scripting languages can utilize.

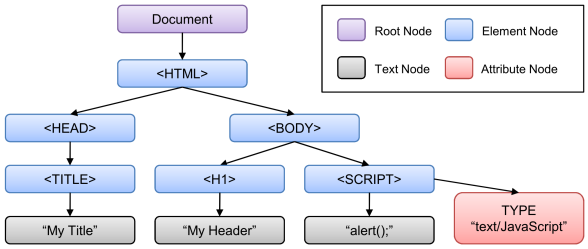

Figure 1: Example of DOM Node Tree

2.2 Definitions

A node in the DOM tree representation of an HTML page is called *JavaScript node* if (1) it has the value `script` in its tag name, and (2) it has a child node of attribute `type` with value "text/JavaScript". The code contained within a JavaScript node can be either *in-line* or *linked*. The code is considered *in-line*, if the script is explicitly embedded in child node. On the other hand, the code is considered *linked* if it is hosted on a remote server whose location `URL` is specified as the value of a child node with attribute `src`. In either case, one can establish a bijection between a piece of script and a DOM node that contains the concrete code or its location information. Figure 2 shows an example of JavaScript node.

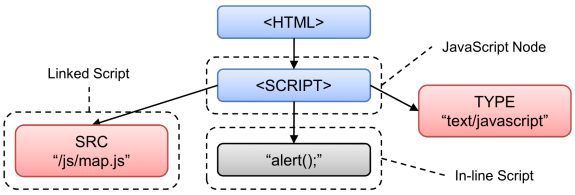

Figure 2: Example of JavaScript Node and Script Code Embedded

Further, the creation of a JavaScript node and its child node can be either *static* or *dynamic*. If all the information contained in a JavaScript node and its children is explicitly encoded within HTML, then we say it is *statically* created. Otherwise, the creation is considered to be *dynamic*, which is often the result of executing other JavaScript code. For example, by invoking the DOM API, such as `document.write`, `appendChild` or `setAttribute`, one can create new JavaScript nodes or modify information contained in existing nodes.

JavaScript code can also be characterized by its *origin*, which is either: *internal* or *external*. JavaScript code is defined to have an *internal* origin, if it meets one of the follow-

ing conditions: (1) it is in-line and statically created; (2) it is linked with the corresponding `src` attribute pointing to a host, which is under the control of the mashup developer (*i.e.,* not third-party); or (3) it is in-line, but was dynamically created through the execution of another script with internal origin. Otherwise, we consider the JavaScript code to have an *external* origin. *An HTML page is defined to be a JavaScript code mashup, if and only if, any JavaScript code contained in the DOM tree has an external origin.* Please note that in the rest of the paper, we will use the terms JavaScript code mashup and code mashup interchangeably.

As we can see, using linked script with the corresponding `src` attribute pointing to externally hosted JavaScript code is the *only* way to *start* the inclusion of external JavaScript code in a mashup. Such external JavaScript code is often encapsulated in text files to facilitate further reuse under current Web programming practices. Thus, we take advantage of this convention to define the notion of *JavaScript library* as a collection of JavaScript code, uniquely identified by their source `URL`, that independently provides functionalities or services through well-defined APIs. In this paper, we do not consider the dependencies between different JavaScript code libraries explicitly. Instead, we assume that it is the mashup developer who will have the domain knowledge to handle this issue by including all the libraries within a dependency closure, in proper order.

2.3 Mashup Life Cycle

The life cycle of a JavaScript code mashup can be divided into two parts – *design* phase and *execution* phase – as illustrated in Figure 3. In the design phase, the mashup developer identifies a set of external JavaScript libraries that serve as building blocks for the mashup application. By including these libraries as linked scripts, the developer sets the stage for constructing the mashup. Finally, the mashup developer writes HTML or JavaScript code to integrate all the building blocks in a meaningful way to provide the desired service(s).

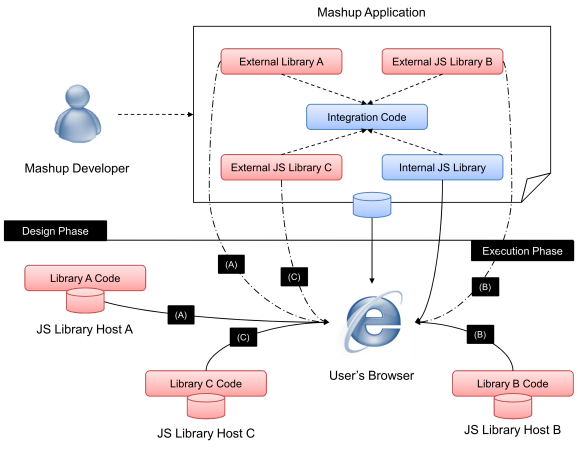

Figure 3: JavaScript Code Mashup Life Cycle

It is important to note that the design phase only defines the logical skeleton of a mashup, the concrete code fusion *only* occurs during the execution phase. To execute a mashup, the end user's browser fetches all the linked scripts from their respective external sources, puts them into the same code space within the browser's JavaScript interpreter, and executes the code as specified. This client-side mashup execution has two main advantages: (1) it saves CPU, bandwidth, and other resources for the mashup host's server by pushing the computation to end users; and (2) it enables the mashup to always use the most recent version of the external libraries with only minimal source modifications.

3. PROBLEM STATEMENT & APPROACH OVERVIEW

In this section, we discuss the problem space of building *trustworthy* JavaScript code mashup and present an overview of our approach to address the problem.

3.1 Problem Statement

The status quo development strategy of JavaScript code mashups is often function-driven. This implies a strong trust by mashup developers on the external JavaScript libraries, especially as it relates to satisfying non-functional properties. However, given the potential for external code libraries to be buggy or even malicious, one needs to provide mashup developers a way to validate their trust assumptions. Currently, there is no convenient way for the mashup developers to (1) explicitly specify various trust requirements (*e.g.,* information security requirement) over different external libraries, and (2) soundly evaluate whether an external JavaScript library meets these trust requirements.

Therefore, the principle problem being addressed in this paper is *to evaluate the trustworthiness of external JavaScript libraries, with respect to preserving the information security requirements of the mashup developers.* Note that, we focus solely on the design phase of the mashup life cycle in this work. Execution phase related trust issues are future work and beyond the scope of this paper. Further, other non-functional properties such as quality-of-service, performance, and reliability, though important, are not addressed herein.

3.2 Approach Overview

In this paper, we propose **ToMaTo**, a **T**rustworthy c**o**de **Ma**shup development **To**ol. ToMaTo assists the mashup developers with three important capabilities: (1) *Trust Policy Specification:* using an expressive but intuitive trust policy language to capture mashup developers' information security requirements over external JavaScript libraries (see Section 4.1); (2) *Policy Adherence Assessment:* soundly evaluating external JavaScript libraries for their adherence to a trust policy, and triggering warnings that pinpoint all the potential trust violations (see Section 4.2) ; (3) *Policy Violation Handling:* enabling mashup developers to plan ahead and to handle the potential trust problems before publishing their mashup application (see Section 4.3).

4. TOMATO SYSTEM DESIGN

In this section, we provide a detailed discussion of the ToMaTo tool. Figure 4 illustrates the principal workflow. After a prototype of the mashup application has been built, the mashup developer can make use of ToMaTo by providing three inputs: (1) the mashup application source code; (2) the `URL`s of the JavaScript libraries that are considered to be external; and (3) the mashup developer's trust policy regarding the privileges granted to different external libraries.

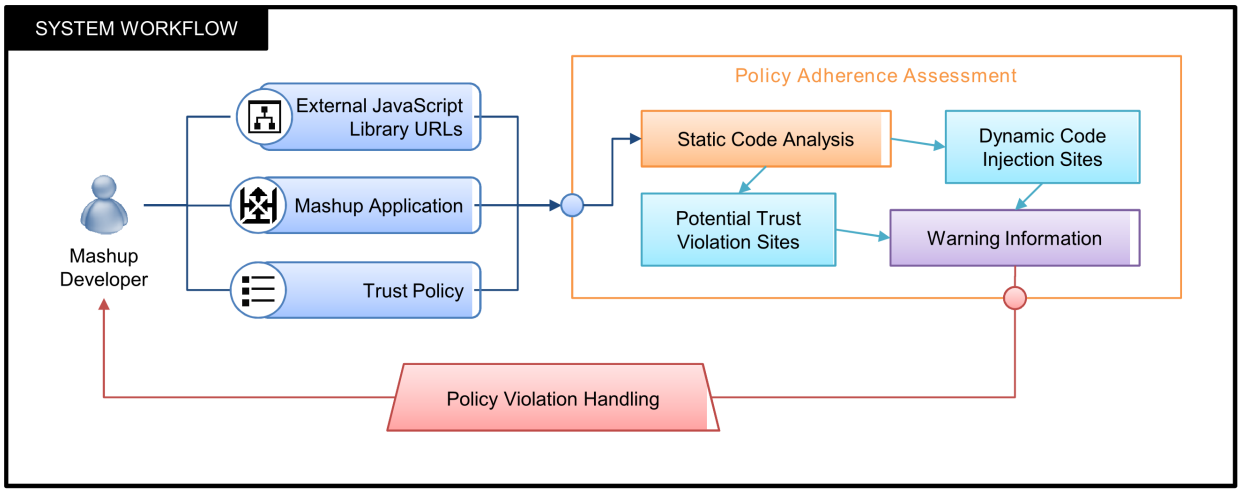

Figure 4: ToMaTo Workflow for Developing Trustworthy JavaScript Code Mashup

The trust policy is expressed as a set of behavior-limiting rules over security-critical JavaScript data structures or operations that need to be protected. ToMaTo then fetches all the JavaScript libraries and performs static code analysis over the entire mashup code space according to the trust policy. The code analysis is sound (*i.e.,* it only suffers from false positives) and triggers warnings for two types of problems: (1) code sections that potentially violate the trust policy, and (2) code sections that enable dynamic code injection during the execution phase (*e.g.,* the invoking of reflective function `eval`). To handle the warnings triggered, the mashup developer has three options: (1) avoid the potential problems by replacing the problematic libraries with alternatives that adhere to the trust policy; (2) avoid the potential problems by carefully skipping the execution of the problematic code sections; and (3) minimize the potential trust issues by handling them with customized countermeasures. We now describe the policy language, assessment and violation handling in more detail.

4.1 Trust Policy Language

```
<principal id="URL">
  DefaultPrivilege
  <function id="FunctionID">
    FuncPrvilege
  </function>
  <object id="ObjectID">
    InfoSecPrivilege
  </object>
  <domnode>
    InfoSecPrivilege
    <domattribute>
      InfoSecPrivilege
    </domattribute>
    <domproperty id="PropertyName">
      InfoSecPrivilege
    </domproperty>
  </domnode>
</principal>
```

Listing 1: Trust Policy Markup Language Syntax

We propose a *Trust Policy Markup Language* (TPML) for mashup developers to encode their trust requirements over the external JavaScript libraries. The TPML adopts the Extensible Markup Language (XML) syntax as shown in Listing 1. The mashup developer will use TPML to specify trust policy in a *top-down* fashion. One starts with the most coarse-grained rules and continues refining them with more fine-grained ones. We believe that this top-down scheme can guide the mashup developer to specify an accurate and complete policy set. In total, the TPML have six basic policy elements:

4.1.1 *The* `principal` *Element*

The `principal` policy element encodes the default rule for external JavaScript libraries. The other five policy elements are children of the `principal` element, and they further refine the default rule. The `principal` element has an `id` attribute to uniquely identify an external JavaScript library. We use the `url` of the external library for this purpose. For example, the `id` of the Google Analytics API is `www.google-analytics.com/ga.js`. In contrast, if the `id` attribute is assigned with value `*`, the policy will then be enforced for all the external libraries used in the mashup. The text of the `principal` element, `DefaultPrivilege`, is defined as:

```
DefaultPrivilege:: Allow | Deny ;
```

Value `Allow` means that the external library in question has no behavior limitation by default, except those specified by the children policy elements. In contrast, `Deny` means the external library cannot exhibit behavior out of the scope its children policy elements explicitly allow. This default rule eases the policy specification process by reducing the number of rules that need to be explicitly written.

4.1.2 *The* `function` *Element*

The `function` policy element encodes behavior-limiting rules for invoking security-critical JavaScript functions. Its `id` attribute identifies the function, with `FunctionName` being the name of the JavaScript function. The text of the `function` element, `FuncPrvilege`, is defined as:

```
FuncPrivilege:: AllowInvoke | DenyInvoke ;
```

Value `AllowInvoke` means that the library is allowed to invoke the specified function. In contrast, `DenyInvoke` means the external library is forbidden from invoking that function.

Example Policy – Restricting Pop-ups: Pop-ups can be a major annoyance when using Web sites. The following example policy can restrict pop-ups by forbidding the invocation of JavaScript functions `alert` and `window.open`:

```
<principal id="*">
  Allow
  <function id="alert">
    DenyInvoke
  </function>
  <function id="window.open">
    DenyInvoke
  </function>
</principal>
```

Listing 2: Restricting Pop-ups

4.1.3 The `object` Element

The `object` policy element encodes information security rules for variables in JavaScript programs. Its `id` attribute uniquely identifies the variable in question, which is denoted in the form `FunctionName/VN`. `VN` is the name of the variable. `FunctionName` identifies the JavaScript function where the variable is defined or used. For global variables, `FunctionName` should be an empty string. The text field `InfoSecPrivilege` is a two-dimension vector defined as:

```
InfoSecPrivilege:: (CFDOption,ITGOption);
CFDOption:: AllowRead | DenyRead ;
ITGOption:: AllowWrite | DenyWrite;
```

The first dimension, `CFDOption`, encodes the *confidentiality* property of the variable. Value `AllowRead` means that the information of the variable can flow to the external library specified by the parent `principal` element. In contrast, `DenyRead` means such flows are forbidden. The second dimension, `ITGOption`, encodes the *integrity* property. Value `AllowWrite` means that the information of the external library is allowed to flow into the variable in question. In contrast, `DenyWrite` means such information flows are forbidden.

Example Policy – Restricting Browser Redirection: The location of Web pages can be changed by JavaScript code, which might lead to the redirection of the browser to malicious sites. The following policy can restrict this behavior by protecting the critical data-structures:

```
<principal id="*">
  Allow
  <object id="window.location">
    (AllowRead, DenyWrite)
  </object>
</principal>
```

Listing 3: Restricting Browser Redirection

4.1.4 The `dom{node,property,attribute}` Elements

The `domnode` policy element expands the TPML's expressiveness by refining the information security rules for the DOM node objects. This semantic is hard to capture by using the `object` policy element, since most (if not all) DOM nodes are initialized by the Web browser by parsing the HTML page. The text of this policy element has the same syntax and value range as the text of the `object` element described above. It defines the default information security rule for all the properties and attributes of DOM node objects, unless otherwise explicitly specified by the children `domproperty` or `domattribute` policy elements.

These two children policy elements of `domnode` further refine the information security rules for the properties or attributes of the DOM node. The `id` attribute of the `domproperty` policy element uniquely identifies the DOM property in question (*e.g.,* the `innerHTML` property). The text of these policy elements have the same syntax, value range and semantics as the text of the `object` element described above.

Example Policy – Restricting Dynamic Code Injection: It is possible to introduce dynamic JavaScript code on-the-fly. This practice is strongly discouraged by many parties [14]. The following example policy can restrict this behavior by preventing one commonly used mechanism for this purpose:

```
<principal id="*">
  Allow
  <domnode>
    (AllowRead,AllowWrite)
    <domproperty id="innerHTML">
      (AllowRead,DenyWrite)
    </domproperty>
  </domnode>
</principal>
```

Listing 4: Restricting Dynamic Code Injection

4.2 Policy Adherence Assessment

Once the trust policy has been specified, external libraries must be evaluated against the policy. ToMaTo uses static code analysis as its policy adherence assessment engine. The static code analysis is designed to be *sound*, in which we consider all possible cases for condition breach, function overload, and array access through index variables. Our analyzer is based on the WALA static analysis framework developed by IBM [23]. Overall, the assessment process has four steps:

1. *Pre-processing*: The mashup source code is first processed by a HTML parser to extract all the JavaScript code used. The core JavaScript definition and browser built-in JavaScript APIs are also included to form a complete code space. All the JavaScript code is then passed into a JavaScript interpreter, which will translate the source program into its abstract syntax tree (AST) representation. By traversing through the AST, the source program is further translated into an intermediate representation (IR) that is designed to take the static single assignment (SSA) form. SSA is a well-studied instruction form to facilitate static code analysis [4]. The whole translation process preserves the semantic of the source program. During this translation process, two other tasks are conducted in parallel: (1) every IR instruction is associated with the original source program position information (*i.e.,* the corresponding source JavaScript library file and concrete line numbers); (2) the prototype chain information of the source program are conservatively tracked and recorded to facilitate later analysis steps.

2. *Evaluating* `function` *policy element*: To evaluate the `function` policy element, a call graph is constructed by analyzing the IR. Every node in the call graph corresponds to a JavaScript function and every directed edge `(x[i],f)` corresponds to the function in-

vocation from the caller node `x` to the callee node `f` at instruction `i`. Besides the default function invocation syntax `function(arguments)`, we also include `function.call` and `function.apply` as the syntax indicator of function call to construct the call graph. Assuming a `function` policy element forbids the invocation of function `f` by external library `e`, this policy element is violated, *if and only if*, the node corresponding to function `f` has at least one in-edge `(x[i],f)`, where `x[i]` is associated with `e` as its source program position. This results in the triggering of a *policy violation warning*, which includes information about the violated policy element and the original program position information of instruction `i`.

3. *Evaluating* `object` *policy element*: We begin by conducting an Andersen-style pointer analysis [1] to identify the alias information of all the variables. We denote the set of aliases of a variable `v` (including `v` itself) as `Alias(v)`. A context-sensitive information flow solver is used to track information flow through the entire mashup application. The information flow solver is based on the tabulation-based analysis proposed in [21]. Assuming an `object` policy element forbids write access to a variable `v` by external library `e`, this policy element is violated, *if and only if*, there is at least one instruction `i` that flows information from the external library`e` into any variable in set `Alias(v)`. In contrast, assuming an `object` policy element forbids read access of variable `v` by external library `e`, this policy element is violated, *if and only if*, there is at least one instruction `i` from the external library `e` that uses the information from any variable in set `Alias(v)`. A violation of the policy results in a warning being triggered, as in the previous step.

4. *Evaluating* `dom{node,property,attribute}` *policy elements*: Since DOM nodes, properties, and attributes are operated by the JavaScript DOM APIs provided by the browser, we use the evaluation process of the `function` and the `object` policy elements as building blocks to evaluate the DOM related policies. To do so, we first identify three variable sets as shown in Table 1. We then translate the `domproperty`, `domattribute`, and `domnode` policy elements into policy sets using the basic `function` and `object` elements as shown in Table 2, 3, and 4, respectively, for the same external library `e` in question. The violation of any policy in the policy set will result in a warning being triggered, as in the previous step.

Besides policy violation warnings, the static code analyzer of ToMaTo also tracks mechanisms that can be used to dynamically introduce JavaScript code during the mashup execution phase. Specifically tracked are (1) invocations of function `eval`, `setTimeout`, `setInterval`, `document.write` and `document.writeln`; and (2) write access of DOM property `innerHTML`. A *code injection warning* is triggered, for the source code segment that matches either of these two rules.

Table 1: Variable Sets Used for DOM-related Policy Element Evaluation

Set Name	Elements
`DomNode`	The return variable of function `item`, `getElementById`, `getElementByName`, `getElement`, `ByTagName`, and `cloneNode`.
`DomNodeString`	The return variable of function `v.toString`, where `v` is in the `DomNode` set.
`DomAttribute`	The return variable of function `v.getAttribute`, where `v` is in the `DomNode` set.

Table 2: `domproperty` Policy Element Translation

Rule	Translation
`DenyWrite` for DOM property `p`	`object DenyWrite` policy for variable `v.p`, where `v` is in the `DomNode` set.
`DenyRead` for DOM property `p`	`object DenyRead` policy for variable `v.p`, where `v` is in the `DomNode` set.

Table 3: `domattribute` Policy Element Translation

Rule	Translation
`DenyWrite` for DOM attributes	`object DenyWrite` policy for variable `v.attribute`, where `v` is in the `DomNode` set.
	`function DenyInvoke` policy for function `v.setAttribute` and function `v.removeAttribute`, where `v` is in the `DomNode` set.
`DenyRead` for DOM attributes	`object DenyRead` policy for every variable in `DomAttr` set and variable `v.attribute`, where `v` is in the `DomNode` set.

Table 4: `domnode` Policy Element Translation

Rule	Translation
`DenyWrite` for DOM nodes	`object DenyWrite` policy for variable `v.childNodes`, where `v` is in `DomNode` set.
	`function DenyInvoke` policy for function `v.normalize`, `v.insertBefore`, `v.appendChild`, `v.removeChild`, and `v.replaceChild`, where `v` is in the `DomNode` set.
	`domproperty DenyWrite` policy for every DOM property, which is further translated.
	`domattribute DenyWrite` policy for DOM attributes, which is further translated.
`DenyRead` for DOM nodes	`object DenyRead` policy for variable `v` and `v.childNodes`, where `v` is in the `DomNode` set.
	`object DenyRead` policy for variable `s` in the variable set `DomNodeString`.
	`domproperty DenyRead` policy for every DOM property, which is further translated.
	`domattribute DenyRead` policy for DOM attributes, which is further translated.

4.3 Policy Violation Handling

If any policy violation warning is triggered, the mashup developer can choose to handle the potential trust issue using one of the following three approaches:

1. The mashup developer could use alternative JavaScript libraries with similar functionality to replace the policy violating libraries. If the alternative libraries can satisfy the trust policy, the potential trust violation is then avoided. This approach takes advantage of the mashup programming paradigm, which is modular in nature. The disadvantage of this approach is the overhead of programming the prototype mashup application using the alternative libraries and re-running the ToMaTo workflow. Alternatively, the mashup developer could revisit the trust policy to see whether the current policy is too strict to achieve the trust requirement. By appropriately relaxing the trust policies that are too demanding or unnecessary, the corresponding violation warnings can be reduced.

2. The mashup developer could choose to skip the execution of the policy violating code segment to preserve both functionality and trust. For example, the mashup developer may specify a policy that forbids external libraries from triggering alert messages or pop-up windows by invoking function `alert` or `window.open`. The ToMaTo tool will pinpoint the concrete source code position where these policies are violated. Using this information, the mashup developer can use a local copy of the problematic library with the policy violating code carefully removed, thus addressing the problem.

3. The mashup developer could design application-specific countermeasures to address the potential trust violations. For example, the mashup developer may specify policies that forbid external libraries to gain read access to the `cookie` of the mashup application. If such policy violations are identified, the mashup developer might consider encrypting the information stored in the `cookie`. This approach is application dependent and requires the mashup developer to select the best countermeasures according to the mashup application and its trust requirements.

Note that, ToMaTo, given the static nature of the policy assessment process, cannot evaluate dynamically generated code for policy adherence. It therefore possible that such dynamically generated code might introduce new policy violations that are not observed during the mashup design phase. As the result, code injection warnings are used to identify such instances within the external libraries. We are currently working on extending our policy adherence capabilities to address such dynamic code generation scenarios.

5. RELATED WORK

Guaranteeing the behavior of a JavaScript program has received much attention in recent years. Studies in the literature address this issue from three main aspects:

(1) *JavaScript Program Analysis*: Pointer analysis techniques of JavaScript program have been proposed in [10, 11, 16]. These techniques make different trade-offs between analysis precision and performance overhead. In [10], the authors also applied their technique for Web widget portals (*e.g.,* `live.com`) to perform sanity checks before allowing new widgets to be published. In [2], a *staged* approach has been proposed to reduce the performance overhead of information flow analysis for JavaScript programs. This technique requires JavaScript program developers to collaborate with application end-users to split up the workload. The major difference between these works and ToMaTo is that our ystem *combines* the use of these individual analysis techniques to evaluate the high-level trust semantics of JavaScript code mashup applications. Each of these individual techniques can be used with ToMaTo to further improve its precision and performance.

(2) *Debugging Client-side Web Application Code*: In [15], the authors propose a proxy service that positions itself between Web applications and end-users to facilitate JavaScript program debugging and testing. This proxy service works transparently and adopts a code rewriting technique. The authors demonstrated that many commonly performed debugging and testing tasks can be easily handled by their proxy service. FLAX [22] is a development tool designed for JavaScript programmers to find code injection bugs in their JavaScript applications. FLAX assumes end users and remote Web servers might be malicious and will try to inject and execute arbitrary JavaScript code, breaking the integrity of the original JavaScript programs. FLAX combines static code analysis and black-box testing techniques to identify such flaws and trigger alerts to JavaScript programmers. Recently, Google also release a Chrome browser extension that helps JavaScript programmers to monitor and debug insecure practices commonly found in JavaScript code [6]. Work in this category focuses mainly on general Web programming issues. It therefore does not address the specific trust problems of code mashup, causing by the reuse of third-party libraries.

(3) *Enhancing Web Application Security*: In [17], a sandbox mechanism, called AdJail, was proposed to enforce information security on third-party advertisements. By placing ads in a shadow `iframe` tag and taking advantage of the same-origin-policy of Web browsers, ads cannot access any information on the host page by default. Further, by monitoring the operations performed by the ad script, AdJail mimics these operations on the host page, if and only if they are allowed by the predefined security policies of the host page owners. Although similar to the design object of our work, AdJail is much more focused on the web advertisement application domain. ToMaTo, on the other hand, is designed to work with general code mashup applications.

Finally, it should be noted that in the realm of web security much work has been done to understand the Web-based malware problem [3, 18, 19, 20]. However, these address issues orthogonal to our work on mashups, where the focus is on protecting against the trust issues that this programming paradigm might introduce.

6. CONCLUSION

In this paper we presented ToMaTo, a development tool for creating trustworthy JavaScript code mashups. By allowing mashup developers to specify customized and fine-grained trust policy, the trust requirements on external code libraries are explicitly captured. The policy evaluation process is sound, which conservatively triggers warnings to pinpoint source code segments with policy violations or code-

injection risks. By properly using the warning information, mashup developers can plan ahead and handle trust risks before publishing their code mashup applications.

Currently, we are actively working on the performance optimization of ToMaTo and preparing for an open-source release to the mashup development community. In the future, we would like to extend the capabilities of ToMaTo into the execution phase of code mashups. By obtaining run-time information of mashup executions, we plan to improve the expressiveness of the trust policy language used by ToMaTo and its policy adherence evaluation capabilities. The ultimate goal is to make ToMaTo a complete trust management solution that covers the entire trustworthy mashup life cycle.

7. REFERENCES

[1] Lars Ole Andersen. Program analysis and specialization for the c programming language. In *USENIX Technical Conference*, 1994.

[2] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[3] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[5] Document Object Model, 2005. `http://www.w3.org/DOM/`.

[6] DOM Snitch, Chrome Reconnaissance Tool. `https://code.google.com/p/domsnitch/`.

[7] Standard ECMA-262, 5th Edition, 2009. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`.

[8] Google AdSense API. `http://code.google.com/apis/adsense/`.

[9] Google Maps API Family. `http://code.google.com/apis/maps/`.

[10] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[11] Dongseok Jang and Kwang-Moo Choe. Points-to analysis for javascript. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 1930–1937, New York, NY, USA, 2009. ACM.

[12] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in javascript web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 270–283, New York, NY, USA, 2010. ACM.

[13] jQuery JavaScript Library. `http://www.jquery.com/`.

[14] JSLint: The JavaScript Code Quality Tool. `http://www.jslint.com/`.

[15] Emre Kiciman and Benjamin Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 17–30, New York, NY, USA, 2007. ACM.

[16] Benjamin Livshits and Salvatore Guarnieri. Gulfstream: Incremental static analysis for streaming javascript applications, 2010.

[17] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. Adjail: practical enforcement of confidentiality and integrity policies on web advertisements. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 24–24, Berkeley, CA, USA, 2010. USENIX Association.

[18] Long Lu, Vinod Yegneswaran, Phillip Porras, and Wenke Lee. Blade: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 440–450, New York, NY, USA, 2010. ACM.

[19] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iframes point to us. In *Proceedings of USENIX Security Symposium*, pages 1–16, 2008.

[20] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: vulnerability-driven filtering of dynamic html. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 61–74, Berkeley, CA, USA, 2006. USENIX Association.

[21] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[22] P Saxena, S Hanna, P Poosankam, and D Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *In 17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[23] WALA - T.J. Watson Libraries for Analysis. `http://wala.sourceforge.net/`.

[24] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 961–970, New York, NY, USA, 2009. ACM.

[25] YUI Library. `http://developer.yahoo.com/yui/`.