JSFlow: Tracking Information Flow in JavaScript and its APIs

Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld
Chalmers University of Technology
{daniel.hedin, arnar.birgisson, bello, andrei}@chalmers.se

ABSTRACT

JavaScript drives the evolution of the web into a powerful application platform. Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services built up from third-party code. Such code provides a range of facilities from helper utilities (such as jQuery) to readily available services (such as Google Analytics and Tynt). Script inclusion poses a challenge of ensuring that the integrated third-party code respects security and privacy.

This paper presents *JSFlow*, a security-enhanced JavaScript interpreter for fine-grained tracking of information flow. We show how to resolve practical challenges for enforcing information-flow policies for the full JavaScript language, as well as tracking information in the presence of libraries, as provided by browser APIs. The interpreter is itself written in JavaScript, which enables deployment as a browser extension. Our experiments with the extension provide in-depth understanding of information manipulation by third-party scripts such as Google Analytics. We find that different sites intended to provide similar services effectuate rather different security policies for the user's sensitive information: some ensure it does not leave the browser, others share it with the originating server, while yet others freely propagate it to third parties.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*information flow controls*; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

Keywords

JavaScript, information flow, dynamic analysis

1. INTRODUCTION

Increasingly, web applications combine services from different providers. The script inclusion mechanism routinely turns barebone web pages into full-fledged services, often utilizing third-party code. Such code provides a range of facilities from utility libraries (such as jQuery) to readily available services (such as Google Analytics and Tynt). Even stand-alone services such as Google Docs, Microsoft Office 365, and DropBox offer integration into other services. Thus, the web is gradually being transformed into an application platform for integration of services from different providers.

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2554909.

Motivation: Securing JavaScript At the heart of this lies JavaScript. When a user visits a web page, even a simple one like a loan calculator or a newspaper website, JavaScript code from different sources is downloaded into the user's browser and run with the same privileges as if the code came from the web page itself. This opens up for abusing the trust, either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by an attacker. A recent empirical study [32] of script inclusion reports high reliance on third-party scripts. As an example, it shows how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains.

This poses a challenge: how do we guarantee that the integrated third-party code respects the security and privacy of web applications? At the same time, the business model of many of the online service providers is to give away a service for free while gathering information about their users and their behavior in order to, e.g., sell user profiles or provide targeted ads. How do we draw a line between legitimate information gathering and unsolicited user tracking?

Background: State of the art in securing JavaScript Today's browsers enforce the *same-origin policy (SOP)* in order to limit access between scripts from different Internet domains. SOP offers an all-or-nothing choice when including a script: either isolation, when the script is loaded in an iframe, or full integration, when the script is included in the document via a script tag. SOP prevents direct communication with non-origin domains but allows indirect communication. For example, sensitive information from the user can be sent to a third-party domain as part of an image request.

Although loading a script in an iframe provides secure isolation, it severely limits the integration of the loaded code with the main application. Thus, the iframe-based solution is a good fit for isolated services such as context-independent ads, but it is not adequate for context-sensitive ads, statistics, utility libraries, and other services that require tighter integration.

Loading a script via a script tag provides full privileges for the included script and, hence, opens up for possible attacks. The state of the art in research on JavaScript-based secure composition [35] consists of a number of approaches ranging from isolation of components to their full integration. Clearly, as much isolation as possible for any given application is a sound rationale in line with the principle of least privilege [37]. However, there are scenarios where isolation incapacitates the functionality of the application.

As an example, consider a loan calculator website. The calculator requires sensitive input from the user, such as the monthly income or the total loan amount. Clearly, the

application must have access to the sensitive input for proper functionality. At the same time, the application must be constrained in what it can do with the sensitive information. If the user has a business relationship with the service provider, it might be reasonable for this information to be sent back to the provider but remain confidential otherwise. However, if this is not the case, it is more reasonable that sensitive information does not leave the browser. How do we ensure that these kinds of fine-grained policies are enforced?

As another example, consider a third-party service on a newspaper site. The service appends the link to the original article whenever the user copies a piece of text from it. The application must have access to the selected text for proper functionality (there is no way of appending to the clipboard - the data must be read, modified and written back). However, with the current state of the art in web security, any third-party code can always send information to its own originating server, e.g. for tracking. When this is undesired, how do we guarantee that this trust is not abused to leak sensitive information to the third party?

Unfortunately, access control is not sufficient to guarantee information security in these examples. Both the loan calculator and newspaper code must be given the sensitive information in order to provide the intended service. The *usage* of sensitive information needs to be tracked *after* access to it has been granted.

Goal: Securing information flow in the browser Those scenarios motivate the need of information-flow control to track how information is used by such services. Intuitively, an information-flow analysis tracks how information flows through the program under execution. By identifying sources of sensitive information, an information-flow analysis can limit what the service may do with information from those sources, e.g., ensuring that the information does not leave the browser, or is not sent to a third party.

The importance of tracking information flow in the browser has been pointed out previously, e.g., [24, 41]. Further, several empirical studies [39, 18, 14, 12] (discussed in detail in Section 6) provide clear evidence that privacy and security attacks in JavaScript code are a real threat. The focus of these studies is *breadth*: trying to analyze thousands of pages against simple policies.

Complementary to previous work, our goal is to provide a practical mechanism for fine-grained enforcement of secure information flow for JavaScript. This paper takes a significant step towards this goal with the implementation of a proof-of-concept interpreter for JavaScript that dynamically tracks information flow. The interpreter is implemented in JavaScript which allows us to effectively replace the JavaScript engine of Firefox through the means of a browser extension. This has enabled us to perform experiments on real web-pages using the interpreter to execute the scripts on the pages, and allowed us to pursue two important subgoals: (i) a study of possibilities and limitations of dynamic information-flow enforcement, and (ii) an in-*depth* understanding of information flow in practical JavaScript.

In addition, while progress has been made at understanding the foundational tensions between static and dynamic information-flow control [34], practical trade-offs have not been explored. Hence, we set off to explore the practical trade-offs related to the permissiveness of the analysis.

Our work focuses on dynamic enforcement, since JavaScript is a highly dynamic language with features such as dynamic objects and dynamic code evaluation. This dynamism puts severe limitations on static analysis methods [38]. These limitations are of particular concern when it is desirable to pinpoint the source of insecurity. While static analysis can be reasonable for empirical studies with simple security policies, the situation is different for more complex policies. Because dynamic tracking has access to precise information about the program state in a given run, we show that it is more appropriate for in-depth understanding of information flow in JavaScript. The starting point for our work is a formalization of information-flow tracking for a a core of JavaScript [16]. This formalization provides sound information-flow tracking for a language with records and exceptions.

The implementation is intended to investigate the suitability of dynamic information-flow control, and lay the ground for a full scale extension of the JavaScript runtime in browsers. A high-performance monitor would ideally be integrated in an existing JavaScript runtime, but they are fast moving targets and focused on advanced performance optimizations. For this reason we have instead chosen to implement our prototype in JavaScript. We believe that our JavaScript implementation finds a sweetspot between implementation effort and usability for research purposes. Thus, performance optimization is a non-goal in the scope of the current work (while a worthwhile direction for future work).

Implementing the monitor in JavaScript allows for flexibility in the deployment. In addition to the possibility of deploying via a browser extension, the interpreter can also be deployed by a proxy, as a suffix proxy or as a security library. We have explored the different architectures for inlining security monitors in web applications in a separate study [25]. Further, in addition to being used to enforce secure information flow on the client side, our implementation can be used by developers as a security testing tool, e.g., during the integration of third-party libraries. This can provide developers with detailed analysis of information flows on custom fine-grained policies, beyond the analysis from empirical studies [39, 18, 14, 12] on simple policies.

Challenges: JavaScript and libraries The above goal leads us to the following three concrete challenges. The first challenge is covering the *full non-strict JavaScript* language, as described by the ECMA-262 (v.5) standard [10]. Our work draws on the sound analysis for the core of JavaScript [16], and so the challenge is whether the rich security label mechanism and key concepts such as read and write contexts scale to the full language.

The second challenge is covering *libraries*, both JavaScript's builtin objects as well as ones provided by browser APIs. The *Document Object Model (DOM)* API, a standard interface for JavaScript to interact with the browser, is particularly complex. This challenge is substantial due to the stateful nature of the DOM tree. Attempts to provide "security signatures" to the API result in missing security-critical side effects in the DOM. The challenge lies in designing a more comprehensive approach.

The third challenge is implementing the JavaScript interpreter in JavaScript. This allows the interpreter to be deployed as a Firefox extension by leveraging the ideas of Zaphod [30]. The interpreter keeps track of the security labels and, whenever possible, it reuses the native JavaScript engine and standard libraries for the actual functionality.

This paper This paper presents *JSFlow*, an information-flow interpreter for full non-strict ECMA-262 (v.5). JSFlow is itself implemented in JavaScript. This enables the use of

JSFlow as a Firefox extension, *Snowfox*, as well as on the server side, e.g., by running on top of *node.js* [19].

The interpreter passes all standard compliant non-strict tests in the SpiderMonkey test suite [29] passed by SpiderMonkey and V8. In addition to the core language we have implemented extensive stateful information-flow models for the standard API, e.g., Object, and Array, and the API present in a browser environment, including the DOM, navigator, location and XMLHttpRequest. Section 3 and 4 report on the challenges on the language and libraries, respectively.

To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

Addressing the first subgoal, we report on the implementation of the interpreter and our experience in modeling native libraries, i.e. the builtin ECMA-262 objects and the DOM. A distinction is made between *shallow* and *deep* models, which represent different trade-offs between reimplementing native code and maintaining a model of its information flow. Further, Section 5 also reports practical trade-offs of dynamic information-flow enforcement.

With respect to the second subgoal, we report on our experience with JSFlow/Snowfox for in-depth understanding of existing flows in web pages. Rather than experimenting with a large number of web sites for simple policies (as done in previous work [39, 18, 14, 12]), we focus on in-depth analysis of two case studies. These are presented in Section 5. The case studies show that different sites intended to provide similar service (a loan calculator) enforce rather different security policies for possibly sensitive user input. Some ensure it does not leave the browser, others share it only with the originating server, while yet others freely share it with third party services. The empirical knowledge gained from running such case studies on actual web pages and libraries has been invaluable in order to understand the possibilities and limitations of dynamic information-flow tracking, and to set the directions for future research.

2. DYNAMIC INFORMATION FLOW

Dynamic information-flow analysis is similar to dynamic type analysis. Each value is labeled with a security label representing the confidentiality of the value. By default information is considered public unless it originates from the user, e.g., information read from an input field, in which case it is labeled *user*. The default labeling can be overridden by explicit annotations in the HTML document. At runtime, the labels are updated to reflect the computation effects and checked to ensure that the computation adheres to some security policy. It is common to use a lattice as the set of labels. In this work we use the subset-lattice over sets of strings.

There are several compelling reasons for using a dynamic analysis for JavaScript over a static one. Features like dynamic code evaluation, dynamic typing, and dynamic object modification limit the accuracy of static analysis. For instance, consider static analysis of the common operation of indexing an object `e1[e2]`, which requires precise information about the objects that $e1$ may evaluate to, as well the values that $e2$ can result in. Both are related to the problem of doing precise alias analysis for languages like JavaScript, which is a well-known hard problem [21]. Similar challenges are posed by *eval* and dynamic object modification such as creating and deleting new object properties.

From now on, in all the examples of the paper, assume that `h` refers to a confidential (or *secret*) value, and that the initial values of all other variables are non-confidential (or *public*). An information-flow analysis must take both *explicit* and *implicit flows* into account. Explicit flows happen in direct assignments, as in `var l = h;`. Since `l` depends on the value of `h`, the monitor labels `l` as secret as well. Implicit flows arise through the control flow of the program, as in `var l=0; if (h) l=1;`. In this case the resulting value of `l` also depends on the value of `h`. In order to handle implicit flows, a security label associated with the control flow is introduced, called the *program counter label*, or *pc* for short. The *pc* reflects the confidentiality of guard expressions controlling branch points in the program, and governs side effects in their branches by preventing modification of less confidential values. For soundness, security violations force execution to halt [16]. Note that halting the program might introduce information leaks via its termination behavior, which is consistent with *termination-insensitive* noninterference [40, 36]. In a sequential batch-job setting, termination leaks are limited to one bit, since they cannot be amplified without restarting the program [2]. In a reactive setting, buffering output helps controlling the leakage bandwidth [33].

3. TACKLING FULL JAVASCRIPT

The interpreter extends the work of Hedin and Sabelfeld [16] to the full non-strict ECMA-262 (v.5) standard (referred to as *the standard* henceforth). Although strict mode may simplify information-flow tracking, its adoption is rather limited. Further, since it does not introduce obstacles for information-flow security and it is possible to run strict code using non-strict semantics, we have opted not to support it. Hedin and Sabelfeld model a core of JavaScript including `eval`, exceptions, higher-order functions and the *Function* object, the somewhat uncommon semantics of JavaScript variables and scoping, including variable hoisting and the `with` statement, and a representative selection of the statements and expressions of JavaScript. Below we report on the most interesting contributions of our extension: full support for functions, accessor properties, labeled statements, and the *pc* handling. Section 4 focuses on the extension made to provide full API support. A prototype implementation of the interpreter and its test suite are available at online [15].

Functions Function support is extended to the full standard, including function hoisting and proper creation of an *arguments* object. Further, to allow for free use of `return` we introduce a *return label*. This label is similar to the exception label [16], in that it defines an upper bound on the control contexts in which `return` can be executed. Consider this example, which returns from a secret control context.

```
l = true;
function f() {
  if (h) { return 1; }
  l = false;
}
```

For the `return` to be allowed, the return label must be secret before entering the conditional. The assignment `l = false` is then in a secret control context, correctly preventing it if `l` is public. The return label of a function is initialized to the *pc* of the calling context. Changing the return label can be facilitated by program annotations, see Section 5.

Labeled statements JavaScript contains labeled statements and allows jumping to them using `break` and `continue`.

As with conditional statements, such transfer of control may result in implicit flows. We handle this by associating labeled statements with a security label, in addition to their usual program label. The security label defines an upper bound on the control contexts in which jumping to the statement is allowed. The security label is also a part of the control context for the labeled statement itself. In the example below, the security label associated with $L1$ is part of the control context of the do/while loop:

```
l=true;
L1: do {
  if (h) { continue L1; }
  l=false;
  } while(0);
```

To allow `continue L1` to execute, the label of $L1$ must be secret, causing the whole code to be run with a secret control context. The labels in the above program are not security labels, but standard statement labels. Unlabeled statements and unlabeled `break` and `continue` are assigned default statement labels. When execution reaches a labeled statement the associated statement labels inherits the security label from the current *pc*, where it remains unless changed with some form of program annotations, see Section 5.

Accessor properties JSFlow supports accessor properties, as described by the standard. Accessor properties allow overriding property reads and writes with user functions. When reading, the return value of the *getter* function of the property is returned, and similarly, writing to a property invokes the *setter* function associated with the property. Section 4 shows how getters and setters can be used in complicated interplay with libraries, opening the door for non-obvious information flows.

The *pc* handling Implicit flows may be caused by more than the control flow of the program itself. When the interpreter branches internally on security labeled values, the control flow inside the interpreter may result in implicit flows. A ubiquitous example of this is *implicit type conversions*: Many expressions, statements and API functions of JavaScript convert their parameters to primitive types. JavaScript objects may override this conversion process with user functions. To appreciate the complexity of the interaction between internal control flow and program control flow, consider the following example which attempts to copy the value of h into l.

```
l = false;
x = { valueOf : function ()
              { return h ? {} : 1; },
      toString : function()
              { l = true; return 1; } };
h = x + 1;
```

The addition operation tries to convert its parameters to primitive values. For an object, the conversion first invokes its `valueOf` method, if present. If this returns a primitive value, it is used. If not, the `toString` method is invoked instead. In the example, `valueOf` is chosen so that `toString` is invoked only when h is true, which must therefore be reflected in its control context.

We solve this by replacing the notion of a *program pc* with a *pc stack* and whenever the *interpreter* branches on a security labeled value, its label is pushed onto the *pc* stack, where it remains until execution reaches a join point in the interpreter. Any side effects are governed by the join of all labels on the *pc* stack. Note that the novelty of this approach is not in the use of a stack for storing the *pc* (e.g., [22]), but the fact that it is the interpreter-internal information flow that is pushed onto the stack. Since all direct and indirect information flows in the interpreted program are induced by direct or indirect information flows in the interpreter, this generalizes and subsumes the standard notion of a program *pc*. For example, just as the type of a value decides which conversion methods to call, so does the value of the guard of a conditional statement decide which subprogram to execute. In both cases, the label of the value is pushed. See the implementation of `ToString` in Section 4.2 for an example of how the *pc* stack is used in the implementation.

4. LIBRARIES

Scripts on a web page typically rely on a rich set of libraries and APIs provided by the JavaScript runtime environment and the browser. While we could reimplement all of JavaScript's builtins, it is more reasonable to defer as much work as possible to the underlying JavaScript engine in which the interpreter itself runs. For browser APIs such as the DOM, doing this is necessary: DOM manipulations must be performed on the actual DOM for them to have an effect on the rendered page. In such cases we must *model* the information flow that those APIs incur and enforce information-flow policies before invoking them. In general, such modeling is useful for any kind of library code we wish to invoke, but not track information flow during its execution in detail.

The problem of modeling information flow in libraries is largely unexplored. Statically, libraries are typically handled by giving some form of boundary types to the interface of the library [31]. The precision and permissiveness of the enforcement of information-flow policies then depends on the expressive power of such boundary types. In this work, we have instead developed dynamic models that make use of actual runtime values to increase their precision.

4.1 Information-flow models

In designing an information-flow model for a library, its precision must be balanced with its complexity and usability. Increased precision allows more permissive enforcement, but typically adds to the complexity of using the library. For example, the user may need to supply security annotations, or otherwise be aware of the model itself. This results in a system that is harder to use and understand. On the other hand, being too imprecise risks having the enforcement reject too many secure programs, thus losing permissiveness. This also places a burden on the programmer, as she will have to work around the false positives.

Shallow vs. deep models We categorize information-flow models for libraries into two different types: *shallow* models and *deep* models.

Shallow models describe the operations on labels and label state in terms of the boundary values and types of the parameters to the library function, whereas deep models may compute internal, intermediate values such as private attributes of objects or local variables inside library code. The model may perform or replicate a part of the computation done by the library, in order to obtain a more precise model.

In the remainder of this section we discuss key insights gained from modeling the builtin JavaScript API, as well as browser APIs. In particular, we give examples where deep models are necessary to yield useful precision.

4.2 JavaScript builtin API

In addition to the core language, JSFlow implements the builtin API defined by the standard. The models of the standard API range from simple shallow models to more advanced deep models. Below we use String and Array to illustrate shallow and deep models, respectively.

String object The String object acts as a wrapper around a primitive string providing a number of useful operations, including accessing a character by index. We model the internal label state of String objects with a single label, matching the security model of its primitive string.

All the methods of String objects are essentially shallow models: After converting the parameters, they are passed to the corresponding native method. The `slice` method described below is a typical representative of String methods. `slice` takes two indices and returns the slice of the string between them. We highlight the implementation of `slice` by means of examples.

First, consider the following program, which passes an object to `slice` whose `valueOf` method will return a secret.

```
ix = { valueOf : function() { return h; }};
var l = '0123456789'.slice(ix,ix+1);
```

This example tries to exploit the conversion to numbers that `slice` performs on its arguments, which invokes `valueOf` when they are objects. In order to model this flow properly, the security label of the value returned by `valueOf` must be taken into account in the result of `slice`. In the example above, `slice` would return a secret value.

In addition, it is important that the security context of the calls made by `slice` include the labels of the indices. Otherwise, side effects in `valueOf` may leak. In the following example, assume that `ix` is secret and chosen to be either an object or a number depending on h.

```
var ix; var l = false;
if (h) {
  ix = { valueOf :
         function() { l = true; return 0; } }
} else { ix = 0; }
'0123456789'.slice(ix,ix+1);
```

Here, the security context of the call to `valueOf` when converting `ix` must reflect the label of `ix`. We ensure this in the internal functions `ToString` and `ToInteger`, used by `slice`. For example, the actual security context increase occurs in `ToString` on line 5, where the argument label is pushed onto the *pc* stack:

```
1 function ToString(x) {
2   if (typeof x.value !== 'object') {
3     return new Value(String(x.value),x.label);
4   }
5   monitor.context.pushPC(x.label);
6     var primValue = ToPrimitive(x, 'string');
7   monitor.context.popPC();
8   return new Value(String(primValue.value), x.label);
9 }
```

Array Array objects are list-like objects that map numerical indices to values. Arrays have a special link between the `length` property and the mapped values: Writing an element past the length of the array will increase the length property accordingly, and decreasing the length property will remove elements from the end of the array. We model the internal state of the array as an ordinary object, while catering for the connection between the `length` property and the indices. Arrays are mutable and equipped with methods for performing different operations on the elements of the array in different orders. This allows for complicated interplays with accessor properties, which calls for the use of deep models. Consider, for instance, the following example.

```
x = [h]; l = false;
Object.defineProperty(x,1,
  { get : function() { l = true; return 0}});
x.every(function (x) { return x; });
```

The `every` method of arrays invokes a function on each element, until either the list is exhausted or it returns a value convertible to *false*. Since all values are convertible to one of *true* or *false*, knowing that an element with index greater than 0 is read reveals that the function returned a *true*-convertible value for all lower indices. By populating an array with a secret followed by a getter, the "truthiness" of the secret could be observed. In the example, the first element contains the secret boolean `h` and the second element is a getter that sets `l` to *true*. Since the getter is only invoked in case h is *true*, this effectively copies h to l. For this reason, a shallow model cannot be used. Each successive call of the iterator function must be called in the accumulated context of the previous results, which is not possible if we simply delegate the computation to the primitive `every` method. Instead, we must use a deep model, illustrated below with an excerpt of the inner loop of `every`.

```
1 var testResult = fn.Call(_this, [kValue, k, O]);
2 var b = conversion.ToBoolean(testResult);
3 monitor.context.labels.pc.lubWith(b.label);
4 label.lubWith(b.label);
5
6 if (!b.value) {
7   monitor.context.popPC();
8   return new Value(false,label);
9 }
```

Note how line 2 converts the result of the function to a boolean, how line 3 uses the label of the result to accumulatively increase the top of the *pc* stack, and how line 4 accumulates the label used for the returned value at line 8.

4.3 Browser APIs

The execution environment provided by browsers is an extension of the builtin JavaScript environment. To a certain extent, what is provided is browser specific, while some parts are standardized by the World Wide Web Consortium (W3C). Although many of the extensions are fairly straightforward to model from an information-flow perspective, offering similar challenges as the standard library, a notable exception is the implementation of the Document Object Model (DOM) API [17]. The DOM is a standard describing how to represent and interact with HTML documents as objects. The DOM is a central data structure to all web applications. A large part of a web application typically deals with shuttling data to and from the DOM and responding to events generated by the DOM as the user interacts with it. Tracking information flows to and from the DOM is thus vital to having information-flow tracking that is useful for real web applications. However, in addition to acting as a data structure, the DOM also provides a rich set of behaviors. In particular, several features of the DOM force information-flow models to be *non-local*, i.e., operations on a certain element in the tree may require updates to the model of other elements in the tree. A prime example of such a feature is *live collections*.

Non-local models: live collections The DOM standard [17] specifies a number of methods for querying the

DOM for a collection of certain elements. Collections are represented as objects that behave much like arrays, with one big exception: as the DOM is modified, collections update to reflect the current state of the DOM.

For example, `getElementsByName` returns a live collection of elements with a particular name attribute. If a script changes the name attribute of an element in the page, the corresponding live collections automatically reflect the change. To appreciate this, consider the following example based on a web page containing a *div* element with id and name 'A'.

```
<div id='A' name='A'></div>
```

When the following code is executed in the context of this page, it encodes the value of h in the length of the live collection returned by `getElementsByName`.

```
c = document.getElementsByName('A');
if (h) { document.getElementById('A').name = 'B'; }
```

This is achieved by conditionally changing the name of the *div* from 'A' to 'B'. Initially, the collection stored in c has length 1, since there is one element in the document named 'A'. After the name change, however, the collection contains no elements. Importantly, this is done without any direct interaction with the collection itself; only the *div* element is referenced and modified.

The security model of live collections must interact with the model for the DOM tree, making it non-local. We keep in each tree node a map from queries generating live collections, such as `getElementsByName('A')`, to the label representing how that node's subtree affects that query.

Going back to the above example, the document (the root node of the DOM tree) maintains a map from names to labels. If this map associates 'A' with public, the interpreter will stop execution on the attempted name change, since it would be observable on existing public live collections. On the other hand, if this map associates 'A' with secret, the name change is allowed. In this case, however, any live collection affected is already considered secret.

Live collections are just one example of non-local behavior in the DOM. For another example, several DOM elements expose properties that are actually computed from state stored elsewhere in the DOM. For instance, a form element exposes values of nested input fields as properties on the form element itself. Also, some element attributes, which are DOM nodes of their own, are exposed as properties on the containing element. The security model of DOM nodes must properly model these cases and label the result appropriately. If we blindly accessed the properties in the underlying API, we could return secret values stripped of their security label.

5. EVALUATION

This section starts by reporting on two types of experiments, one to explore different policies for user data and the other to govern user tracking by third-party scripts. We then go on to discuss general security considerations, trade-offs for dynamic enforcement and going beyond dynamic analysis.

For the case studies we created *Snowfox*, a Firefox extension that uses JSFlow as the execution engine for web pages. Snowfox is based on Zaphod [30] and turns off Firefox's native JavaScript engine. Instead, the extension traverses the page as it loads and executes scripts using JSFlow.

This provides a proof-of-concept implementation that allows us to study the suitability of dynamic information flow on actual JavaScript code. When an information-flow policy is violated, the extension can respond in various ways, such as simply logging the leak, silently blocking offensive HTTP requests or stopping script execution altogether.

While performance is not a goal for this paper, we found that the speed of JSFlow did not hinder us in manually interacting with web pages. Compared to a fully JITed JavaScript engine, JSFlow is slower by two orders of magnitude on the tested pages.

User input processing We have evaluated the interpreter on several web applications that calculate loan payments, given input provided by the user. Such applications do not rely on external data. Running the interpreter under different policies reveals some security-relevant differences between applications and demonstrates our interpreter's ability to enforce them on real JavaScript code. As a baseline policy, we use a stricter version of SOP, where communication via requests such as creating image and script tags is not allowed if it involves information derived from user inputs.

We found three main classes of loan calculators: (i) Scripts that do all calculations in the browser: no data is submitted anywhere. (ii) Scripts that submit user data to the original host for processing, but not to third parties. (iii) Scripts that submit data to a third party, e.g., for collecting statistics, or allow third-party scripts to access user data. At the time of writing, example web pages for each class are (i) `http://www.halifax.co.uk/loans/loan-calculator/` and `http://www.asksasha.com/loan-interest-calculator.html`; (ii) `http://www.tdcanadatrust.com/loanpaymentcalc.form`; and (iii) `http://mlcalc.com/`. A calculator of the first class works under a policy that allows no data to leave the browser. On the other hand, the second class needs to send data to its origin server, but still works under the strict SOP policy. The third class requires a more liberal policy.

Web pages commonly use Google Analytics to analyze traffic. To do so, the web page loads a script provided by that service. This script triggers an image request conveying tracking information to Google. As long as no data about user input is contained in the request, scripts of type (i) and (ii) can still use Google Analytics under our interpreter. A calculator in class (iii) that tries to log user inputs, or any derived values, to Google Analytics is prevented from doing so by our interpreter. Flows are correctly tracked inside the Google Analytics script, and the interpreter does not allow creating an image element with such data in the source URL.

One of the sites in our tests, `mlcalc.com`, did send user inputs to Google Analytics, but indirectly. The user inputs were first submitted to `mlcalc.com`, but the following page included JavaScript code that logged them to Google Analytics. The flow in this case was essentially server-side, so some server-side support is needed to track them. Our monitor supports upgrade annotations, i.e. a server can explicitly label some of its data as being derived from sensitive user inputs. We note that JSFlow can also be run on the server side, e.g. via *node.js*, for an end-to-end solution.

When the host is not trusted for user data, a still stricter policy can be utilized, namely that no user data should leave the browser. Under this policy the interpreter correctly stops even the submission of a form. This still allows calculators of type (i), which compute everything client-side.

Behavior tracking via JavaScript Users are often unaware of information sent from their browser, e.g., for tracking purposes. As an example, the service Tynt offers a script to inject links back to the including website, into content

copied to the system clipboard. This service is used on popular web sites such as the Financial Times (`www.ft.com`). As the clipboard API in JavaScript does not provide append functionality, this script relies on having read access to the copied data, constituting a source of information. However, transparent to the user, the script also creates a request via an image to `tynt.com`, constituting an information sink, where it logs the copied data, together with a tracking cookie unique to the user across different websites using Tynt.

Our implementation supports all the necessary browser APIs used by Tynt's script and is able to detect this behavior. Since the selection is chosen by the user, the data copied is considered secret. When the script attempts to communicate this data back to Tynt, the interpreter detects the leak and throws a security error.

Security considerations At the core of JSFlow is a formalization of information-flow tracking for a language with records and exceptions [16]. The formalization includes a dynamic type system for a core of JavaScript that has been proven sound. Much of the extension to full JavaScript is via sound primitive constructions from [16], while extensions not expressed in sound primitive constructions are built using a small number of core principles also used in [16].

For now, the correctness of JSFlow is only verified using testing. This is true both for the functional correctness as well as for the soundness of the information flow. However, the connection between the interpreter-internal information flow and the information flow of the interpreted language provides an indication of a potential way of verifying the implementation of JSFlow using a specialized static type system. The basic idea is that any direct or indirect flow in the interpreted language is manifested as an direct or indirect flow in the interpreter. By connecting the representation of labeled values to a type system tailor made for checking JSFlow it would be possible to make sure that all such flows were properly taken into account.

Trade-offs for dynamic enforcement A key question is whether it is possible to implement an interpreter with enough precision for the enforcement to be usable and permissive. Our interpreter indicates that tracking flows in real-world applications is feasible. JavaScript is a flexible language, and provides many implicit ways for information to flow, in particular when combined with the rich APIs of the browser environment. Our interpreter successfully addresses such features, while being reasonably precise to allow real scripts to maintain their utility.

However, dynamic enforcement comes at a price of inherent limitations. In particular, there are limits on sound and precise propagation of labels under secret control [34], leading to a common restriction of enforcement known as *no sensitive upgrade* [43, 3]. We found that legacy scripts sometimes encounter such situations, even if they do not always leak confidential data. In such scripts, upgrade statements must be injected [5]. For two of the scripts used in our experiments, we have done so manually via a proxy. In particular, two upgrade annotations were added to Google Analytics and three to Tynt.

We also discovered two cases where the interpreter detected flows that originated from a nullity check on user input. Since those checks were performed relatively early in the execution, they resulted in much derived data to be labeled as secret. However, in both cases, we found the checks did not leak, since no action of the user can cause the checked values to be null. The checks were only safeguards to prevent failures due to browser differences or bugs in the scripts. We thus added declassification annotations to allow these benign flows.

Beyond dynamic analysis In summary, purely dynamic analysis is a reasonable fit for tracking information in real-world examples. However, under some circumstances, we have manually aided the analysis with upgrade annotations, in order to increase the accuracy of the dynamic approach. Although only a few annotations were needed in the case studies under consideration, it is still desirable to have automatic support for generating the annotations.

One promising approach is to use a hybrid analysis, where a static information flow analysis is used to approximate the locations in need of upgrade before entering a secret context. A limited form of such an analysis is already present in JSFlow. The analysis statically approximates the need to upgrade variables, which reduces the amount of upgrades that have to be inserted manually. We expect a full hybrid analysis to perform well on actual code and intend to investigate this in future work.

6. RELATED WORK

Web tracking is subject to much debate, involving both policy and technology aspects. We refer to Mayer and Mitchell [26] for the state of the art. In this space, the Do Not Track initiative, currently being standardized by World Wide Web Consortium (W3C), is worth pointing out. Supported by most modern browsers, Do Not Track is implemented as an HTTP header that signals to the server that the user prefers not to be tracked. However, there are no guarantees that the server (or third-party code it includes) honors the preference.

While there is much work on safe sub-languages, e.g, Caja [28], ADSafe [7], Gatekeeper [13], and work on refined access control for JavaScript, as in, e.g. ConScript [27] and JSand [1], we recall our motivation from Section 1 for the need of information-flow control beyond access control. In the rest, we focus on information-flow tracking for JavaScript.

Vogt et al. [39] modify the source code of the Firefox browser to include a hybrid information-flow tracker. However, their experiments show that it is often desirable for JavaScript code to leak some information outside the domain of origin: they identify 30 domains such as `google-analytics.com` that should be allowed *some* leaks. Their solution is to white-list these domains, and therefore allow *any* leaks to these domains, opening up possibilities for laundering.

Mozilla's ongoing project FlowSafe [11] aims at giving Firefox runtime information-flow tracking, with dynamic information-flow reference monitoring [3] at its core. Our coverage of JavaScript and its APIs provides a base for fulfilling the promise of FlowSafe in practice.

Chugh et al. [6] present a hybrid approach to handling dynamic execution. Their work is staged where a dynamic residual is statically computed in the first stage, and checked at runtime in the second stage.

Yip et al. [42] present a security system, BFlow, which tracks information flow within the browser between frames. In order to protect confidential data in a frame, the frame cannot simultaneously hold data marked as confidential and data marked as public. BFlow not only focuses on the client-side but also on the server-side in order to prevent attacks that move data back and forth between client and server.

Mash-IF, by Li et al. [23], is an information-flow tracker for client-side mashups. With policies defined in terms of DOM

objects, the enforcement mechanism is a static analysis for a subset of JavaScript and treats as blackboxes the language constructs outside this subset. Executions are monitored by a reference monitor that allows deriving declassification rules from detected information flows. An advantage of this approach is fine-grained control at the level of individual DOM objects. At the same time, the imprecision of the static analysis leads to both false positives and negatives, opening up for attackers to bypass the security mechanism.

Extending the browser always carries the risk of security flaws in the extension. To this end, Dhawan and Ganapathy [9] develop Sabre, a system for tracking the flow of JavaScript objects as they are passed through the browser subsystems. The goal is to prevent malicious extensions from breaking confidentiality. Bandhakavi, et al. [4] propose a static analysis tool, VEX, for analyzing Firefox extensions for security vulnerabilities.

Jang et al. [18] focus on privacy attacks: cookie stealing, location hijacking, history sniffing, and behavior tracking. Similar to Chugh et al. [6], the analysis is based on code rewriting that inlines checks for data produced from sensitive sources not to flow into public sinks. They detect a number of attacks present in popular web sites, both in custom code and in third-party libraries.

Guarnieri et al. [14] present Actarus, a static taint analysis for JavaScript. An empirical study with around 10,000 pages from popular web sites exposes vulnerabilities related to injection, cross-site scripting, and unvalidated redirects and forwards. Taint analysis focuses on explicit flows, leaving implicit flows out of scope.

Just et al. [20] develop a hybrid analysis for a subset of JavaScript. A combination of dynamic tracking and intra-procedural static analysis allows capturing both explicit and implicit flows. However, the static analysis in this work does not treat implicit flows due to exceptions.

De Groef et al. [12] present *FlowFox*, a Firefox extension based on *secure multi-execution* [8]. Multi-execution runs the original program at different security levels and carefully synchronizes communication among them. Multi-execution provides information-flow security by design since the run that computes public input only gets access to public input. At the same time, secure multi-execution does not *track* information flow in the original program: instead, it silently converts the execution to be independent of secrets. This makes it less suitable for in-depth understanding of information manipulation by JavaScript code. Instead, the empirical study by De Groef et al. focuses on studying the deviation in user experience when browsing with FlowFox and Firefox in the presence of simple policies.

The empirical studies above [39, 18, 14, 12] provide clear evidence that privacy and security attacks in JavaScript code are a real threat. As mentioned earlier, the focus is the *breadth*: trying to analyze thousands of pages against simple policies. Complementary to this, our goal is in-*depth* studies of information flow in critical third-party code (which, like Google Analytics, might be well used by a large number of pages). Hence, the focus on dynamic enforcement, based on a sound core [16], and the careful approach in fine-tuning the security policies to match application-specific security goals.

As mentioned before, the starting point for our work is a formalization of information-flow tracking for a a core of JavaScript [16]. This formalization provides sound information-flow tracking for a language with records and exceptions. Recently, we have explored different architectures for inlining security monitors in web applications [25]. The architectures allow deploying any monitors, implemented in JavaScript in the form of security-enhanced JavaScript interpreters, under different architectures. We have investigated the pros and cons of deployment as a browser extension, as a proxy, as a service, and an integrator-driven deployment. The pros and cons reveal security trade-offs and usability trade-offs. We have shown how to instantiate the general deployment approach with the JSFlow monitor.

7. CONCLUSIONS AND FUTURE WORK

We have presented JSFlow, a security-enhanced JavaScript interpreter, written in JavaScript. To the best of our knowledge, this is the first implementation of dynamic information-flow enforcement for such a large platform as JavaScript together with stateful information-flow models for its standard execution environment.

In line with our goals, we have demonstrated that JSFlow enables in-depth understanding of information flow in practical JavaScript, including third-party code such as Google Analytics, jQuery, and Tynt.

We have conducted a practical study of possibilities and limitations of dynamic information-flow enforcement. The study has identified the trade-offs of static and dynamic security enforcement. The trade-offs provide a roadmap for further work in the area. Hybrid information-flow tracking is a particularly promising direction, where we have the possibility to combine the best of static and dynamic analysis.

A major feature of our interpreter is tracking information flow in the presence of libraries. We have demonstrated how to model the libraries, as provided by browser APIs, by a combination of shallow and deep modeling. Our findings lead to possibilities of generalization: future work will pursue automatic generation of library models from abstract functional specifications.

Acknowledgements This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research is supported in part by this Google Fellowship.

8. REFERENCES

[1] Agten, P., Acker, S. V., Brondsema, Y., Phung, P. H., Desmet, L., and Piessens, F. JSand: complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC* (2012), R. H. Zakon, Ed., ACM, pp. 1–10.

[2] Askarov, A., Hunt, S., Sabelfeld, A., and Sands, D. Termination-insensitive noninterference leaks more than just a bit. In *Proc. ESORICS* (Oct. 2008), vol. 5283 of *LNCS*, Springer-Verlag, pp. 333–348.

[3] Austin, T. H., and Flanagan, C. Efficient purely-dynamic information flow analysis. In *Proc. ACM PLAS* (June 2009).

[4] Bandhakavi, S., Tiku, N., Pittman, W., King, S. T., Madhusudan, P., and Winslett, M. Vetting browser extensions for security vulnerabilities with vex. *Commun. ACM 54*, 9 (2011), 91–99.

[5] Birgisson, A., Hedin, D., and Sabelfeld, A. Boosting the permissiveness of dynamic information-flow tracking by testing. In *ESORICS* (2012), S. Foresti, M. Yung, and F. Martinelli, Eds.,

vol. 7459 of *Lecture Notes in Computer Science*, Springer, pp. 55–72.

[6] Chugh, R., Meister, J. A., Jhala, R., and Lerner, S. Staged information flow for JavasCript. In *PLDI* (2009), M. Hind and A. Diwan, Eds., ACM, pp. 50–62.

[7] Crockford, D. Making JavaScript Safe for Advertising. adsafe.org, 2009.

[8] Devriese, D., and Piessens, F. Non-interference through secure multi-execution. In *SSP* (May 2010).

[9] Dhawan, M., and Ganapathy, V. Analyzing information flow in javascript-based browser extensions. In *ACSAC* (2009), IEEE Computer Society, pp. 382–391.

[10] ECMA International. ECMAScript Language Specification, 2009. Version 5.

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

[12] Groef, W. D., Devriese, D., Nikiforakis, N., and Piessens, F. Flowfox: a web browser with flexible and precise information flow control. In *ACM CCS* (2012).

[13] Guarnieri, S., and Livshits, B. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proc. USENIX security* (USA, 2009), SSYM'09, USENIX Association.

[14] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the world wide web from vulnerable JavaScript. In *ISSTA* (2011), M. B. Dwyer and F. Tip, Eds., ACM, pp. 177–187.

[15] Hedin, D., Bello, L., Birgisson, A., and Sabelfeld, A. JSFlow. Software release. Located at `http://chalmerslbs.bitbucket.org/jsflow`, Sept. 2013.

[16] Hedin, D., and Sabelfeld, A. Information-flow security for a core of JavaScript. In *Proc. IEEE CSF* (June 2012), pp. 3–18.

[17] Hors, A. L., and Hegaret, P. L. Document Object Model Level 3 Core Specification. Tech. rep., The World Wide Web Consortium, 2004.

[18] Jang, D., Jhala, R., Lerner, S., and Shacham, H. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM CCS* (Oct. 2010), pp. 270–283.

[19] Joyent, Inc. Node.js. `http://nodejs.org/`.

[20] Just, S., Cleary, A., Shirley, B., and Hammer, C. Information Flow Analysis for JavaScript. In *Proc. ACM PLASTIC* (USA, 2011), ACM, pp. 9–18.

[21] Landi, W. Undecidability of static analysis. *ACM LOPLAS 1*, 4 (Dec. 1992), 323–337.

[22] Le Guernic, G., Banerjee, A., Jensen, T., and Schmidt, D. Automata-based confidentiality monitoring. In *Proc. ASIAN* (2006), vol. 4435 of *LNCS*, Springer-Verlag.

[23] Li, Z., Zhang, K., and Wang, X. Mash-IF: Practical information-flow control within client-side mashups. In *DSN* (2010), pp. 251–260.

[24] Magazinius, J., Askarov, A., and Sabelfeld, A. A lattice-based approach to mashup security. In *Proc. ACM ASIACCS* (Apr. 2010).

[25] Magazinius, J., Hedin, D., and Sabelfeld, A. Architectures for inlining security monitors in web applications. In *ESSoS* (2014), Lecture Notes in Computer Science, Springer.

[26] Mayer, J. R., and Mitchell, J. C. Third-party web tracking: Policy and technology. In *IEEE SP* (2012), IEEE Computer Society, pp. 413–427.

[27] Meyerovich, L. A., and Livshits, V. B. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *IEEE SP* (2010), IEEE Computer Society, pp. 481–496.

[28] Miller, M., Samuel, M., Laurie, B., Awad, I., and Stay, M. Caja: Safe active content in sanitized JavaScript, 2008.

[29] Mozilla Developer Network. SpiderMonkey – Running Automated JavaScript Tests. `https://developer.mozilla.org/en-US/docs/SpiderMonkey/Running_Automated_JavaScript_Tests`, 2011.

[30] Mozilla Labs. Zaphod add-on for the Firefox browser. `http://mozillalabs.com/zaphod`, 2011.

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

[32] Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., Piessens, F., and Vigna, G. You are what you include: large-scale evaluation of remote javascript inclusions. In *ACM CCS* (Oct. 2012), pp. 736–747.

[33] Rafnsson, W., and Sabelfeld, A. Limiting information leakage in event-based communication. In *Proc. ACM PLAS* (USA, 2011), ACM, pp. 4:1–4:16.

[34] Russo, A., and Sabelfeld, A. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE CSF* (July 2010), pp. 186–199.

[35] Ryck, P. D., Decat, M., Desmet, L., Piessens, F., and Joose, W. Security of web mashups: a survey. In *NORDSEC* (2010), LNCS.

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

[37] Saltzer, J. H., and Schroeder, M. D. The protection of information in computer systems. *Proc. of the IEEE 63*, 9 (Sept. 1975), 1278–1308.

[38] Taly, A., Erlingsson, U., Miller, M., Mitchell, J., and Nagra, J. Automated analysis of security-critical JavaScript APIs. In *Proc. IEEE SP* (May 2011).

[39] Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proc. NDSS* (Feb. 2007).

[40] Volpano, D., Smith, G., and Irvine, C. A sound type system for secure flow analysis. *J. Computer Security 4*, 3 (1996), 167–187.

[41] Yang, E., Stefan, D., Mitchell, J., Mazières, D., Marchenko, P., and Karp, B. Toward principled browser security. In *Proc.HotOS* (2013).

[42] Yip, A., Narula, N., Krohn, M., and Morris, R. Privacy-preserving browser-side scripting with bflow. In *EuroSys* (USA, 2009), ACM, pp. 233–246.

[43] Zdancewic, S. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.