Security Signature Inference for JavaScript-based Browser Addons

Vineeth Kashyap   Ben Hardekopf
University of California Santa Barbara
{vineeth,benh}@cs.ucsb.edu

ABSTRACT

JavaScript-based browser addons are a tempting target for malicious developers—addons have high privileges and ready access to a browser user's confidential information, and they have none of the usual sandboxing or other security restrictions used for client-side webpage JavaScript. Therefore, vetting third-party addons is important both for addon users and for the browser providers that host official addon repositories. The current state-of-the-art vetting methodology is manual and ad-hoc, which makes the vetting process difficult, tedious, and error-prone.

In this paper, we propose a method to help automate this vetting process. We describe a novel notion of *addon security signatures*, which provide detailed information about an addon's information flows and API usage, along with a novel static analysis to automatically infer these signatures from the addon code. We implement our analysis and empirically evaluate it on a benchmark suite consisting of ten real browser addons taken from the official Mozilla addon repository. Our results show that our analysis is practical and useful for vetting browser addons.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Security, Verification

Keywords

Static Analysis, JavaScript, Browser Addons

1. INTRODUCTION

The web-browser addon framework is a powerful and popular mechanism for extending browser behavior—thousands of third-party developers are creating addons, and browser users have downloaded billions of copies [1]. These addons have almost complete access to a user's information: browser history, cookies, passwords, clipboard, geo-location, mouse and keyboard actions, the local filesystem, and more. Malicious addons are trivially easy to write, and yet can be difficult to detect. Thus, vetting third-party addons is critical both for users (whose information is at risk) and for browser providers (whose reputations are at risk). However, the current vetting process for addons submitted to official addon repositories is mostly manual and completely ad-hoc.

Our goal is to help automate this vetting process by creating an analysis to automatically infer *security signatures* for JavaScript-based browser addons. A security signature captures both (1) information flows between interesting sources and sinks, for example, *from* the current browser URL *to* a network message; and (2) interesting API usage, for example, to detect deprecated or unsafe APIs. API usage inference is treated as a special case of information flow—in essence, can any information potentially flow to a use of that API. This signature inference analysis can be used by official addon repositories upon addon submission (and also by third-party developers prior to submission) to detect potential security problems, thus reducing the vetting burden and increasing addon security.

1.1 Key Challenges

We must address three key challenges to enable security analysis of browser addons:

1. **Flexible Security Policies**: Naively, we might expect to use a standard information flow analysis [34] to establish the security of an addon. For such an analysis, we would use a security lattice to specify a security policy describing allowable information flows, and report any information flows in the addon that violate the specified policy. Unfortunately, there is no single security policy (and hence no single security lattice) that is suitable for all addons. Whether an addon's information flow is secure or not depends largely on that addon's intended purpose. For example, the current URL being browsed by the user should usually be private. However, if an addon's intended purpose is send URLs over the network to an URL shortener service, then this information flow is expected and allowed. There are many other examples of information flows that would usually be considered insecure, but that are allowable given the intended purpose of the addon. Thus, we need a more flexible solution than traditional information flow analysis.

2. **Classifying Information Flows**: Traditional information flow analysis simply reports whether a *leak* (a flow violating the given security policy) might occur. However, this information alone is not useful for our purpose—there are many possible ways for information flow to happen, with varying levels of importance and concern. We must be able to classify the information flows to aid the addon vetter in their task and enable them to understand exactly what the addon is doing. This requires a more discriminating analysis than traditional information flow.

3. **Inferring Network Domains**: A large part of addon security concerns the network domains that the addon communicates with. In JavaScript, these domains are created and passed around in the form of strings. It requires careful and precise analysis to recover the actual network domains from these strings.

1.2 Our Contributions

To meet these challenges, we present the following contributions:

1. **Annotated Program Dependence Graph**: We base our analysis on the Program Dependence Graph (PDG) [19]. Defining and implementing a PDG for JavaScript is novel; moreover, we extend the classic definition with a novel set of graph annotations that allow us to classify various information flows according to their natures. We use the annotated PDG specifically for information flow in this work, but it can be more generally useful, e.g., for program slicing, code obfuscation, code compression, and various code optimizations. The annotated PDG definition and construction algorithm are described in Section 3.

2. **Security Signatures**: To accomodate the fluid nature of addon security policies, we develop a novel notion of *addon security signatures.* Rather than attempting to enforce a specific policy, instead we infer interesting flows and API usages and present them to the vetter, allowing them to compare the inferred signature against the addon description to decide whether the addon should be accepted. We define what constitutes a signature and how to construct a signature from the annotated PDG. The definition of security signatures and their construction are described in Section 4.

3. **Prefix String Analysis**: Inferring network domains from strings requires precise analysis, but that analysis must also remain tractable. We have defined a sweet-spot in this space by developing a prefix string analysis that is precise enough to compute most of the statically-determinable network domains while still retaining practical performance. This analysis is described in Section 5.

4. **Evaluation**: Finally, we evaluate the usefulness and practicality of our work by inferring security signatures for a set of ten real browser addons taken from the Mozilla Firefox offical addon repository. The evaluation and results are described in Section 6.

2. BACKGROUND

In this section we provide necessary background information on addons, as well as illustrative examples (taken from real addons) of how addons can violate user privacy.

Addon Security Context.

Modern web-browsers offer the ability to extend browser behavior with user-installed *addons* (also called *extensions*). Addons[1] are written in JavaScript by third-party developers; they have much higher privileges than client-side JavaScript programs, and they are *not* subject to the sandboxing and other security restrictions that exist for client-side programs. Proof-of-concept malicious addons have been developed that demonstrate how easily such privileges can be misused [2, 3], and other researchers have demonstrated that even non-malicious addons can be exploited to break security [33, 11]. These are not just theoretical problems; for example, the Mozilla vetting team has seen a number of submitted addons that contain malicious code copied from these published exploits [8].

Addon Execution.

Addons use an event-driven programming model: they continuously execute a loop responding to *events* such as mouse movement and clicks, keyboard entry, page loads, network responses, timeouts, etc. When the browser first starts up the addon code is fully evaluated, during which a set of event handlers are registered. Then the addon enters an loop in which the following two steps are executed infinitely often: (1) if the event queue is not empty, then an event is pulled off the event queue; (2) if there is an event handler corresponding to the given event, then the handler is invoked and evaluated to completion. More event handlers can be registered and existing handlers can be de-registered during the event handling phase.

Addon Vetting.

The current addon vetting process for official addon repositories employs volunteers who manually inspect addon code. There are no fully documented or precisely specified security policies, rather, the vetters look for "code smells". Any addon that does not pass the sniff test is rejected. Dynamic code injection is particularly discouraged, given the difficulty in statically determining what the dynamically-injected code will do. This fact is encouraging for static analysis, because unlike client-side JavaScript (which uses `eval` and related APIs heavily) we can safely disallow addons from using dynamic code. Our analysis reports any potential use of these restricted APIs.

Privacy Leaks.

An addon's elevated privileges make it trivial to leak private user information. We concern ourselves with two kinds of information flows: *explicit flows* (due to data dependencies) and *implicit flows* (due to control dependencies). Timing and termination flows are beyond the scope of this work. We give two examples derived from actual information flows discovered by our analysis in real addons that have been downloaded millions of times. In these exam-

[1]Extensions to browsers written in native code are referred to as browser plugins, and they are not the focus of our work.

ples, the current URL being browsed by the user is accessed by the addon via `content.location.href`, and the call `XHRWrapper(publicServer)` sets up a cross site request to the network domain `publicServer`. Consider the code:

```
function ajax(params) {
  var data = params["data"];
  request = XHRWrapper(publicServer);
  request.send("url is: " + data);
}
ajax({ data: content.location.href });
```

Here, the current URL is used to construct the `data` field of an object literal passed as an argument to `ajax`. The function `ajax` creates a network request to `publicServer` over which the `data` field of its formal parameter is sent, thus explicitly leaking the private URL information. Now consider the code:

```
window.addEventListener("load", check, false);
function check(e) {
 var seen = false;
 if (content.location.href == "sensitive.com")
   seen = true;
 var request = XHRWrapper(publicServer);
 request.send(seen);
}
```

Here `check` is registered as an event handler for page load events, thus, whenever the user loads a new page `check` is executed. `check` sets `seen` to true only if the current URL is `sensitive.com`, and then sends `seen` over the network to `publicServer`. This code implicitly leaks private information about whether the user visits `sensitive.com`.

These are just two—certainly non-exhaustive—ways in which privacy can be breached by addons. Our main goal in this work is to develop a static analysis for JavaScript addons that can reliably and precisely detect these kinds of privacy leaks, as well as distinguish between various kinds of leaks.

3. ANNOTATED PDGS FOR JAVASCRIPT

A *Program Dependence Graph* (PDG) [19, 10, 7] is an explicit representation of a program's data and control dependencies. We use an novel extended variant of PDGs as a basis for our security signature inference (described in Section 4). The relation between information flow and program dependencies has been noted before (e.g., Abadi et al [6]) and has previously been exploited for information flow analysis of Java bytecode [22]. Our novel contributions are (1) defining PDG construction for JavaScript; and (2) a set of annotations for the PDG that allow us to classify the various types of information flows found in a program.

We assume we are given a base analysis for JavaScript that is flow- and context-sensitive and computes a reduced product of pointer analysis (what objects a reference may point to), string analysis (what set of strings a value may represent), and control-flow analysis (what functions a call may refer to). Any such base analysis can be used for our technique; two existing analyses that meet these requirements are JSAI [30] and TAJS [27]. From this information we compute the following as input to our PDG construction:

1. A context-sensitive interprocedural control flow graph (CFG), with one node per statement per context.
2. Read and write sets for each statement under each context, consisting of the set of variables and the set of (object, property) pairs that the statement may read from or write to. JavaScript uses computable property accesses, i.e., an object property name is a string that can be computed at runtime, unlike languages such as Java where object fields are statically known. Therefore, the object properties in the read/write sets are actually abstract strings (elements from the abstract string domain used in the base analysis) representing potentially multiple possible concrete property names.

In the rest of this section, we explain how to use this information to construct an annotated PDG. We first define the annotated PDG, then describe the two stages of PDG construction: constructing the annotated data dependence graph (DDG) and constructing the annotated control dependence graph (CDG).

3.1 Defining the Annotated PDG

A classic PDG is a graph (V, E) such that $v \in V$ are the program statements and there is an edge $v_1 \rightarrow v_2 \in E$ if there is a data or control dependence from v_1 to v_2. Statement v_2 is *data dependent* on statement v_1 if v_1 writes to a location in memory, v_2 reads from that location in memory, and the value read by v_2 could potentially be the value written by v_1. Statement v_2 is *control dependent* on statement v_1 if the execution of v_1 controls the number of times that v_2 is executed (e.g., v_1 is the guard of a conditional and v_2 is contained in one branch of that conditional). Information can flow from statement v_1 to statement v_2 if there is a path in the PDG from v_1 to v_2.

In order to classify information flows, we annotate the edges of the PDG to denote how each particular edge was derived from the program. We can broadly classify edges based on whether they correspond to data or control dependencies, but an even finer granularity of classification is useful. We describe and motivate the different possible annotations here.

Data Dependence Annotations.

We can classify data dependence edges as *strong* or *weak*. A strong data dependence arises between v_1 and v_2 if v_1 writes to a single memory location, v_2 definitely reads from that exact same memory location, and the value it reads is definitely the value written by v_1. A weak data dependence arises between v_1 and v_2 if either v_2 only possibly reads from the same memory location as v_1 writes to, or if that memory location was possibly over-written by another value during the execution between v_1 and v_2. The idea behind this classification is that information flow along strong data dependence edges is more likely to be interesting/relevant than that along weak data dependence edges.

Control Dependence Annotations.

We can classify control dependence edges as *local* or *non-local.* We can further subdivide non-local control edges into *explicit* or *implicit.* A local control edge arises from structured local control flow, such as conditionals or loops; all other control edges are classified to be non-local. An explicit non-local control edge arises from explicit (i.e., syntactically visible) control-flow jumps in the code, such as a `break` or `continue` instruction inside a loop, or an exception thrown using the `throw` instruction, or returning from a function using a `return` instruction. An implicit non-local control edge arises from implicit (i.e., syntactically invisible) exceptions that can be thrown by various JavaScript instructions (e.g., accessing a property of the `undefined` value, or attempting to call a non-function). It is useful to distinguish these categories; for example, consider line 20 in Figure 1, and suppose that the analysis infers `obj` to be potentially `undefined` at this line. Since this statement may raise an implicit exception, the statement at line 21 and all the statements that follow inside the try block are control dependent on the conditional on line 19 (because the conditional evaluation of statement 19 dictates whether or not these statements execute), causing many additional edges to be added to the PDG. Most of these additional edges are a gross over-approximation of the actual control-flow during program execution. Thus, information flow along local control edges is likely to be more interesting/relevant than that along non-local control edges, and information flow along explicit non-local control edges are in turn likely to be more interesting/relevant than that along implicit non-local control edges.

Amplified Control.

Finally, we can also classify control edges (independently from the classifications above) as *amplified* or *unamplified*. An amplified control edge is contained within a cycle of the CFG, whereas an unamplified control edge is not. This is interesting for information flow because an unamplified control edge can convey at most one bit of information (i.e., whether a statement is executed or not), whereas an amplified control edge could potentially convey an arbitrary number of bits of information (one for each iteration of the loop or recursive call).

Annotation Grammar.

From these various classifications, we define the following annotation grammar:

$$\begin{aligned} ann \in Annotation &::= data \mid control \\ data \in DataDep &::= \mathsf{data}_{strong} \mid \mathsf{data}_{weak} \\ control \in CtrlDep &::= ctrl \mid ctrl^{amp} \\ ctrl \in Ctrl &::= \mathsf{local} \mid \mathsf{nonloc}_{exp} \mid \mathsf{nonloc}_{imp} \end{aligned}$$

The annotated PDG is then a graph (V, E) such that $v \in V$ are the program statements and there is an edge $v_1 \xrightarrow{ann} v_2 \in E$ if there is a data or control dependence from v_1 to v_2 that matches the criteria of annotation *ann*. The remaining subsections describe how we construct the PDG and assign the appropriate annotations to its edges.

3.2 Constructing the Annotated DDG

The first phase of PDG construction creates the Data Dependence Graph, which contains all of the data dependence edges of the eventual PDG. In JavaScript, data dependencies arise from reads and writes to variables and to object properties. For statement v, let $ReadVar(v)$ be the set of variables that v can read from, $ReadProp(v)$ be the set of (object, property) pairs that v can read from, $WriteVar(v)$ be the set of variables that v can write to, and $WriteProp(v)$ be the set of (object, property) pairs that v can write to; these sets are computed from the base analysis described earlier. Dynamically adding, updating, or deleting a property are all considered object property writes. Recall that the properties in these (object, property) pairs are actually abstract strings representing possibly multiple concrete property names.

Each element of these sets is qualified as *strong* (a definite read or write) or *weak* (a possible read or write). Definite reads/writes occur for a variable when its associated abstract memory location is guaranteed to correspond to a single concrete memory location. Definite reads/writes occur for a (object, property) pair when a similar criterion holds for the object *and* the property abstract string corresponds to a single, exact concrete string. Note that definite writes correspond to strong updates in static analysis, and thus write sets that are qualified to be strong are singleton sets. We use normal set intersection for the $ReadVar(\cdot)$ and $WriteVar(\cdot)$ sets, but for the $ReadProp(\cdot)$ and $WriteProp(\cdot)$ sets we must define a new set intersection operator that accounts for the abstract string property names (which abstractly represent sets of concrete strings). We define the operator $\Cap$ as: $S_1 \Cap S_2 = \{(obj, prop) \mid (obj, prop_1) \in S_1,\ (obj, prop_2) \in S_2,\ prop = prop_1 \sqcap prop_2,\ prop \neq \bot\}$.

There is a DDG edge $v_1 \xrightarrow{\mathsf{data}_{strong}} v_2$ if there is a CFG path from v_1 to v_2 and both of the following conditions hold:

- $WriteVar(v_1) \cap ReadVar(v_2) = \{var\}$ and var is strong in both sets, or $WriteProp(v_1) \cap ReadProp(v_2) = \{(obj, prop)\}$ and $(obj, prop)$ is strong in both sets. In other words, v_2 definitely reads from the memory location written by v_1.
- There is no statement v_3 along any path from v_1 to v_2 such that $WriteVar(v_1) \cap WriteVar(v_3) \neq \emptyset$ or $WriteProp(v_1) \Cap WriteProp(v_3) \neq \emptyset$, i.e., the value read by v_2 is definitely the value written by v_1.

There is a DDG edge $v_1 \xrightarrow{\mathsf{data}_{weak}} v_2$ if there is a CFG path from v_1 to v_2, there is not an edge $v_1 \xrightarrow{\mathsf{data}_{strong}} v_2$, and both of the following conditions hold:

- $WriteVar(v_1) \cap ReadVar(v_2) \neq \emptyset$ or $WriteProp(v_1) \Cap ReadProp(v_2) \neq \emptyset$. In other words, v_2 possibly reads from the memory location written by v_1.
- There at least one path from v_1 to v_2 such that for any statement v_3 on that path, $WriteVar(v_1) \cap WriteVar(v_3)$ is empty or contains only weak elements and $WriteProp(v_1) \Cap WriteProp(v_3)$ is empty or contains only weak elements. In other words, the value read by v_2 is possibly the value written by v_1.

Figure 1 and Figure 2 give an example program and the associated PDG to illustrate these points. The edge $1 \xrightarrow{\mathsf{data}_{strong}} 2$ exists because we can determine definitely that the call argument at line 2 refers to the (object, property) pair created at line 1. The edge $1 \xrightarrow{\mathsf{data}_{weak}} 3$ exists because (assuming the analysis cannot exactly determine the return value of `getString`) we don't know which property of the object defined at line 1 is being accessed.

3.3 Constructing the Annotated CDG

The final phase of PDG construction creates the Control Dependence Graph (CDG); the PDG is the union of the DDG and CDG. The CDG is constructed using standard

```
1   var data = { url: doc.loc };
2   send(data.url);
3   send(data[getString()]);
4   func();
5   if (doc.loc == "secret.com")
6       send(null);
7   var arr = ["covert.com", "priv.com"/*,...*/];
8   var i = 0, count = 0;
9   while(arr[i] && doc.loc != arr[i]) {
10     i++;
11     count++; } // end while
12  send(count);
13  try {
14      if (doc.loc != "hush-hush.com")
15          throw "irrelevant";
16      send(null);
17  } catch(x) {};
18  try {
19      if (doc.loc != "mystic.com")
20          obj.prop = 1;
21      send(null);
22      /* ..... */
23  } catch(x) {}
```

Figure 1: An example program to show the various annotations of the PDG. We assume the following for this example: `doc.loc` is the current browser url; the `send` method sends it arguments over the network; the base analysis infers `obj` to either reference an object or `null`; `func` is inferred to be either a callable function or `undefined`; and the call to `getString()` returns an unknown string.

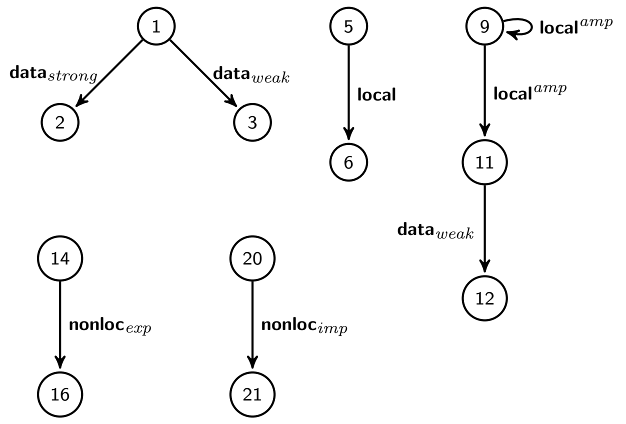

Figure 2: A subset of the annotated PDG for the example program in Figure 1, to illustrate the interesting edges and nodes.

techniques [19], but we stage its construction in order to properly annotate the CDG edges. We also omit from the CDG all edges due to uncaught exceptions (for example, in Figure 1, we omit edges due to a potential implicit exception at line 4). If we included those edges, then for all statements that may throw an exception outside of a try/catch block we would need an edge to every other reachable statement in the CFG. For our purposes omitting these edges is sound because uncaught exceptions result in termination, and we are not considering termination leaks in our security analysis.

We construct the annotated CDG in four stages in the following order:

1. Create a pruned CFG by removing all edges arising from non-local control-flow (i.e., exceptions and jumps). Compute CDG_1 from the pruned CFG using standard techniques, and annotate all edges with $\textsf{local}$.

2. Create another pruned CFG from the original CFG by removing all non-local control-flow edges arising from implicit exceptions. Compute CDG_2 from this pruned CFG, subtract any edges present in CDG_1, and annotate all remaining edges with $\textsf{nonloc}_{exp}$.

3. Compute CDG_3 from the full CFG, subtract any edges present in CDG_1 or CDG_2, and annotate all remaining edges with $\textsf{nonloc}_{imp}$.

4. Update all three CDGs so that any annotation *ctrl* for an edge whose source node is contained within a CFG cycle is updated to ctrl^{amp}. The final CDG is $\text{CDG}_1 \cup \text{CDG}_2 \cup \text{CDG}_3$.

When creating a pruned CFG some nodes may become unreachable from the CFG entry node; we add a new edge in the pruned CFG from the entry to any such node before computing the CDG.

In the previous example, the edge $5 \xrightarrow{\textsf{local}} 6$ exists because line 6's execution depends on line 5 but there is no loop, and $9 \xrightarrow{\textsf{local}^{amp}} 11$ exists because line 11's execution depends on line 9 and there is a containing loop. Line 16's execution is control dependent on line 14, because along its `true` branch, the explicit non local control flow at line 15 can cause line 16 to not execute. Hence the edge $14 \xrightarrow{\textsf{nonloc}_{exp}} 16$. Line 20 can potentially throw an implicit exception, because the base analysis is assumed to infer `obj` to either be a reference an object or `null`. Hence the edge $20 \xrightarrow{\textsf{nonloc}_{imp}} 21$.

4. GENERATING SECURITY SIGNATURES

From the annotated PDG described in the previous section, we can infer interesting information flows to report to the addon vetter and classify them according to types based on the annotations. In this section we describe the form of the signature and how we infer signatures from the annotated PDG.

4.1 Description of Security Signatures

Figure 3 gives the formal description of a security signature. A signature consists of zero or more entries, where each entry describes either a particular information flow from an interesting source to an interesting sink, or an interesting API usage. API usage is a special case of information flow that indicates there exists some source (interesting or not) that may flow to an instance of that API. The set of interesting sources, sinks, and APIs is given to the analysis; in our implementation we have used the sources, sinks, and APIs considered interesting by the Mozilla vetting team (where the interesting APIs include various script injection APIs such as `Services.scriptloader` and various deprecated APIs), but they are easily configurable if desired. The **send** sink (corresponding to a network send using `XMLHttpRequest`) takes a parameter indicating the network

$$\begin{aligned} sign \in \mathit{Signature} &::= \overrightarrow{entry} \\ entry \in \mathit{Entry} &::= src \xrightarrow{type} sink \;\mid\; sink \\ type \in \mathit{FlowType} &::= \mathbf{type1} \;\mid\; \ldots \;\mid\; \mathbf{type8} \\ src \in \mathit{Source} &::= \mathbf{url} \;\mid\; \mathbf{key} \;\mid\; \mathbf{geoloc} \;\mid\; \ldots \\ sink \in \mathit{Sink} &::= \mathbf{send}(\mathit{Pre}) \;\mid\; \mathbf{scriptloadr} \;\mid\; \ldots \end{aligned}$$

Figure 3: Grammar for a security signature *sign*. *Pre* is the prefix string domain described in Section 5; it is used to indicate the network domain being communicated with. We give a subset of the complete list of interesting sources and sinks. The eight flow types are described in the text and Figure 4.

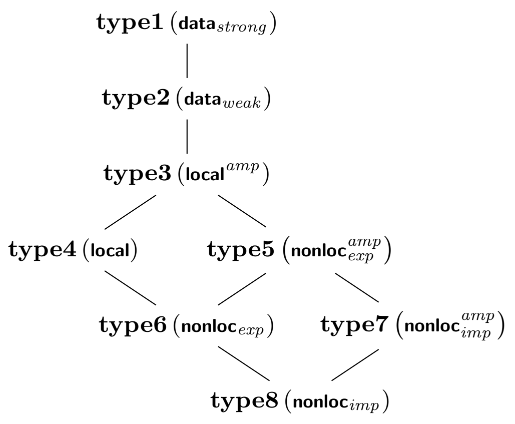
$\mathbf{type1}$ (data_{strong})
$\mathbf{type2}$ (data_{weak})
$\mathbf{type3}$ (local^{amp})
$\mathbf{type4}$ (local)
$\mathbf{type5}$ ($\mathsf{nonloc}^{amp}_{exp}$)
$\mathbf{type6}$ (nonloc_{exp})
$\mathbf{type7}$ ($\mathsf{nonloc}^{amp}_{imp}$)
$\mathbf{type8}$ (nonloc_{imp})

Figure 4: Flow types ordered in a lattice of perceived strength. Higher in the lattice indicates a more important type of flow. Each flow type is associated with an annotation from the PDG. A flow has a given type if there is a path from source to sink using only PDG edges annotated with any annotation at a level equal or higher in the lattice.

domain being communicated with. Each information flow entry also has one of eight types, described further below.

An information flow between source and sink is derived from a path in the PDG from the source to the sink. The type of flow is derived from the annotations on the PDG edges along that path. We order the flow types by the kinds of edges (i.e., edges with particular annotations) we allow the associated flow to traverse in the PDG: the more kinds of edges allowed, the weaker the flow type. We have structured the set of flow types into a lattice, pictured in Figure 4. Each flow type is associated with an annotation from *Annotation*; the meaning is that a flow of a given type only traverses PDG edges annotated with the given annotation or some annotation at a higher level in the lattice. This lattice is based on our perceived strength of the type of flow—obtained by manually examining the commonly intended and commonly accidental kinds of flows. This lattice is the one we use in our analysis, but the lattice is independently configurable to accommodate changes in perceived strength of the flow types.

Consider the examples below to better interpret the various flow types in the lattice in the Figure 4. The strongest flow type, **type1**, is assigned to information flows that only traverse PDG edges annotated with data_{strong}. The **type4** flow type is assigned to information flows that only traverse PDG edges annotated with local, local^{amp}, data_{weak}, or data_{strong}. The weakest flow type, **type8**, is assigned to information flows that traverse any kind of PDG edge. One can think of a particular flow type as corresponding to a sub-graph of the PDG containing only the allowed kinds of edges; an information flow is assigned that flow type if (1) there is a path from the source to the sink contained in that sub-graph; and (2) there is not a path from the source to the sink in the sub-graph of any higher flow type.

4.2 Inferring Signatures

Given an annotated PDG, we must infer signatures of the form described above. Inferring the API usage part of a signature (i.e., is there any information flow to an interesting API) is straightforward: if there a reachable call statement in the CFG whose call expression is data dependent on any node (including itself) with a $\mathit{ReadProp}(\cdot)$ set containing a designated interesting sink *snk*, then the *snk* API may be used. Note that for inferring API usage we consider all call expressions that are data dependent on reads to APIs because functions can be copied and passed around in JavaScript. Inferring the information flow entries of the signature is more involved; the rest of this subsection explains how this is done.

We wish to characterize the set of paths between interesting sources and sinks with a flow type. For each (source, sink) pair there is a set of paths between them in the PDG; we need to compute the strongest flow type(s) possible that are consistent with that set of paths and their edge annotations (because some flow types are noncomparable in strength, there may not be a single strongest flow type). To describe this computation, we first define two helper functions:

$$\begin{aligned} \mathit{extend} &: (\mathit{FlowType} \times \mathit{Annotation}) \to \mathit{FlowType} \\ \mathit{max} &: \mathcal{P}(\mathit{FlowType}) \to \mathcal{P}(\mathit{FlowType}) \end{aligned}$$

The *extend* function takes a flow type t and extends it with an annotation *ann*—the function returns the strongest flow type t' which includes all the edge annotations corresponding to the flow type t as well as *ann*. For example, $\mathit{extend}(\mathbf{type4}, \mathsf{nonloc}^{amp}_{exp}) = \mathbf{type6}$, and $\mathit{extend}(\mathsf{local}^{amp}, \mathsf{nonloc}^{amp}_{exp}) = \mathbf{type5}$. The *max* function takes a set of flow types and returns the strongest flow types in that set (again, since there are noncomparable flow types there may not be a single strongest flow type in the set). For example, $\mathit{max}(\{\mathbf{type4}, \mathbf{type5}, \mathbf{type6}\}) = \{\mathbf{type4}, \mathbf{type5}\}$.

For each source we will compute a set of flow types for each statement in the PDG reachable from that source; the final set of flow types are taken from the statements corresponding to interesting sinks. Let $\mathit{FlowType}(v)$ be the set of flow types assigned to statement v, and initialize $\mathit{FlowType}(v) = \{\mathbf{type1}\}$ for all statements v. Then compute the fix point over all v of the following equation:

$$\mathit{FlowType}(v) = \max\left(\bigcup_{\substack{v' \xrightarrow{ann} v \in E \\ t \in \mathit{FlowType}(v')}} \{\mathit{extend}(t, ann)\} \right)$$

Intuitively, $\mathit{FlowType}(v)$ gives the strongest set of flow types using which the source under consideration can reach v. To compute this, we look at all the predecessors v' of v, and extend of the flow types computed at v' with the edge annotation *ann* between v' and v, and keep only the strongest flow types amongst these. Because we consider all the predecessors v' of v and edges between v' and v, we account for all the possible paths from the source to v. Due to the presence of cycles in PDG, we compute a fixpoint of these equations.

Consider the following PDG example to illustrate the flow type equation. Let the PDG include the edges $v_1 \xrightarrow{\textsf{nonloc}^{amp}_{exp}} v_3$ and $v_2 \xrightarrow{\textsf{nonloc}^{amp}_{exp}} v_3$, with $FlowType(v_1) = \{\textbf{type4}, \textbf{type5}\}$ and $FlowType(v_2) = \{\textbf{type3}\}$. To compute $FlowType(v_3)$, we first extend the flow types at predecessors v_1 and v_2 with the corresponding edge annotations, and take their union to obtain $\{\textbf{type6}, \textbf{type5}\}$. We then pick the strongest flow types from these to obtain $FlowType(v_3) = \{\textbf{type5}\}$.

We compute the above fixpoint for the various statements with respect to each interesting source in turn; the signature is created by taking the flow types at each interesting sink. If for source *src* the sink *snk* has flow types $\{type_1, type_2\}$, then the signature contains the entries $src \xrightarrow{type_1} snk$ and $src \xrightarrow{type_2} snk$.

5. INFERRING NETWORK DOMAINS

The most common way in which addons communicate with network domains is to create a network request object `XMLHttpRequest` and pass it a string that contains the desired URL. To generate precise signatures, our analysis should statically infer as many of these URL strings as possible. However, a string constant analysis (analogous to the traditional integer constant analysis) is insufficient to determine many of these strings. Often an addon will communicate with the same domain, but dynamically extend that domain's URL with different suffixes, e.g., different arguments to the same web application. Consider the following code which exemplifies a common pattern found in addons:

```
var baseURL = "www.example.com/req?";
if (...) baseURL += "name"; else baseURL += "age";
// communicate with baseURL
```

A string constant analysis would infer `baseURL` to be an unknown string after the conditional. Our insight is that, for inferring the network domain contained in the string, we only need the URL's prefix rather than the entire URL; e.g., in the example above we need to infer only the base domain `www.example.com/req?` and not the two URLs constructed from that base domain.

Therefore, we augment the base JavaScript analysis (which uses a constant string analysis) with a *prefix string* analysis in order to infer these network domain prefixes. Our abstract prefix string domain is similar in concept to the prefix domain described by Costantini et al. [15], except that we also track exact strings whenever possible—because we use the same string domain for inferring URLs as well as object properties, this is an important distinction for precision. We describe our abstract prefix string domain and one example abstract string operation for that domain, string concatenation. The complete prefix domain formalization and proof sketches of soundness are contained in the supplemental materials.[2]

The prefix string abstract domain is a lattice $\mathcal{L}^\sharp_p = (Pre, \sqsubseteq, \sqcup, \sqcap)$. Let $\preceq$ mean string prefix and let $\oplus$ mean the greatest common prefix; then:

- *Pre* is a set of (string, boolean) pairs augmented with a bottom element: $(str, b) \in Pre = (String \times Boolean) \cup \{\bot\}$, such that $b = \textsf{true}$ means *str* is an exact string and $b = \textsf{false}$ means *str* is a prefix of an unknown string.
- The bottom of the lattice $\bot$ represents an uninitialized string value, and the top of the lattice $\top = (\epsilon, \textsf{false})$ represents all possible strings.
- $\bot \sqsubseteq (str, b) \sqsubseteq \top$ for all $(str, b) \in Pre$, and $(str_1, b_1) \sqsubseteq (str_2, b_2)$ iff either $b_2 = \textsf{false}$ and $str_2 \preceq str_1$, or $b_1 = \textsf{true}, b_2 = \textsf{true}$, and $str_1 = str_2$
- $(str_1, b_1) \sqcup (str_2, b_2) =$
$$\begin{cases} (str_1, b_1) & \text{if } str_1 = str_2, b_1 = b_2 = \textsf{true} \\ (str_1 \oplus str_2, \textsf{false}) & \text{otherwise} \end{cases}$$
- $(str_1, b_1) \sqcap (str_2, b_2) =$
$$\begin{cases} (str_1, b_1) & \text{if } b_2 = \textsf{false},\ str_2 \preceq str_1 \\ (str_2, b_2) & \text{if } b_1 = \textsf{false},\ str_1 \preceq str_2 \\ \bot & \text{otherwise} \end{cases}$$

The lattice is noetherian, i.e., it meets the finite ascending chain condition. We describe the abstract string concatenation operation $+$ on the prefix domain as a representative example of the set of required abstract operations. Let X be any element of $\mathcal{L}^\sharp_p$; then:

- $\bot + X = X + \bot = \bot$
- $(str_1, \textsf{true}) + (str_2, b_2) = (str_1 \cdot str_2, b_2)$
- $(str_1, \textsf{false}) + (str_2, b_2) = (str_1, \textsf{false})$

6. EVALUATION

In this section we first briefly describe our analysis implementation and our benchmarks and experimental methodology; we then describe and discuss our evaluation results.

6.1 Implementation

We implement our signature inference analysis on top of JSAI [30], a flow- and context-sensitive abstract interpreter for JavaScript. JSAI, and hence our analysis, is implemented in Scala. The analysis is performed in three passes: (1) use JSAI to compute the CFG and read/write sets; (2) construct the annotated PDG as described in Section 3; and (3) infer the signature as described in Section 4.

We extend JSAI in two ways for our analysis. First, we augment JSAI's abstract string domain with the prefix string domain described in Section 5. Second, we extend JSAI to handle browser-embedded code: we provide

[2]Available under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`.

manually-written stubs for the native APIs (e.g., DOM and XPCOM APIs) used by our benchmarks, and simulate the addon event-handling loop by adding a loop at the end of the addon that non-deterministically executes all registered event handlers. Our implementation is available under the `Downloads` link at `http://www.cs.ucsb.edu/~pllab`.

6.2 Benchmarks and Methodology

Our benchmark suite consists of real addons taken from the Mozilla addon repository [4]. All of these addons were vetted manually by Mozilla before being added to the repository, and have been present in the repository for years. Table 1 lists the addons, their intended purpose, their size, and the number of times they have been downloaded (as an indication of their popularity). The size is given as the number of AST nodes parsed by Rhino [5], a more accurate representation than number of lines of code. All of these addons, along with a set of tests showing various kinds of information flows, are bundled with our implementation.

For expository purposes, we classify the addons into three categories based on each addon's summary submitted by its developer:

Category A: Addons intended to explicitly send the current URL information to a specified domain. For example, `LivePageRank`, which sends the active URL over the network to find out its page rank.

Category B: Addons intended to implicitly send information about the current URL or user key presses to a specified domain. For example, `YoutubeDownloader` will check whether the current URL is in fact `youtube.com` before attempting to download a video.

Category C: Addons intended to communicate with a specified domain, but without sending any interesting information. For example, `Chess.comNotifier` will communicate with `chess.com` to find out whose turn it is to play. These addons exemplify API usage discovery, using network communication as the API of interest.

In order to check the precision of our inferred signatures, we first manually write a signature for each addon based on its developer-provided summary (this is done *before* we automatically infer any signatures). We can then use the manual signatures to compare against the automatically inferred signatures: if the inferred signatures are weaker (allow more flows) than the manual signature, it indicates either a false positive or a misleading addon summary. We give an example manual signature for one addon in each category:

- `LivePageRank` (A): **url** $\xrightarrow{\textbf{type1}}$ **send**(`toolbarqueries.google.com`). Rationale: its stated purpose is to display the page rank of the active URL, computed by sending the URL to `toolbarqueries.google.com`.

- `HyperTranslate` (B): **key** $\xrightarrow{\textbf{type3}}$ **send**(`translate.google.com`). Rationale: it translates selected text by using a web service, but only if the keys pressed by the user match its defined keyboard shortcuts. Thus, the addon can implicitly reveal information about key presses to the domain `translate.google.com`. Because the addon continuously listens for key presses, this information flow can be amplified.

- `Chess.comNotifier` (C): **send**(`chess.com`). Rationale: it does not reveal information about any interesting sources over the network, but it does communicate with `chess.com` about game status.

We also measure the time taken by the analysis to infer signatures for each benchmark. Our main purpose is to show that the analysis time is reasonable; our prototype implementation is written with emphasis on correctness rather than performance, and there are multiple opportunities for improving the performance of our implementation. We divide the time taken into three phases:

Phase 1 (P1): time taken by the base analysis to compute information assumed as input to our annotated PDG construction.

Phase 2 (P2): time taken to construct the annotate PDG as described in Section 3.

Phase 3 (P3): time taken to convert the annotated PDG into a signature as described in Section 4.2.

To compute the timing results we run the analysis 11 times on each benchmark, discard the first result, and report the median of the remaining runs. The timing information is obtained on a Mac OS X 2.3 GHz Intel Core i7 machine with 8GB of RAM.

6.3 Results and Discussion

Table 2 summarizes the result of signature inference analysis on the benchmarks. For each addon, the analysis result is summarized as *pass* (the inferred signature matches the manual signature); *fail* (the inferred signature has more flows than the manual signature, and manual inspection determined they were false positives); or *leak* (the inferred signature has more flows and manual inspection determined they were real). The times are given separately for each analysis phase, as described in Section 6.2. The total time taken by the analysis for each of the addons is under one minute.

Five of the addons passed. Of the remainder, two failed and three had unintended leaks. We discuss the failures and leaks in more detail below.

Failed Addons.

The inferred signatures for `LessSpamPlease` and `VKVideoDownloader` fail simply because the analysis was not able to determine the exact network domain being communicated with. For example, VKVideoDownloader checks whether the current URL is one of three different video player domains, and communicates with the corresponding domain. Our prefix abstract string domain is not expressive enough to precisely represent all three domains, and hence infers the final domain to be unknown. It is worth noting that in the remaining eight out of the ten addons, our prefix string analysis can determine the exact domains with which the addons communicate. Both failed signatures had the correct information flow sources, sinks, and flow types; the only imprecision was in the network domain.

Leaky Addons.

`YoutubeDownloader` computes a video id taken directly from the current URL and sends it to `youtube.com`; this is a

Addon Name	**Listed Purpose**	**Category**	**Size**	**# of Downloads**
`LivePagerank`	Display PageRank for active URL	A	3,900	515,671
`LessSpamPlease`	Generates a reusable anonymous real mail address	A	3,696	194,604
`YoutubeDownloader`	Youtube video downloader	B	3,755	7,600,428
`VKVideoDownloader`	Downloads videos from sites	B	2,016	459,028
`HyperTranslate`	Translates selected text when key shorts are pressed	B	3,576	62,633
`Chess.comNotifier`	Notifies your turn on `chess.com`	C	1,079	2,402
`CoffeePodsDeals`	Indicates coffee pods for sale	C	1,670	1,158
`oDeskJobWatcher`	Indicates oDesk job opening	C	609	8,279
`PinPoints`	Save clips (addresses) from web text	C	2,146	7,042
`GoogleTransliterate`	Allows user to type in Indian languages	C	4,270	77,413

Table 1: Real addons from Mozilla addon repository [4] used as benchmarks for our evaluation. We manually sort addons into categories based on their behavior, the category descriptions are given in Section 6.2. The size of the benchmarks give the number of AST nodes parsed by Rhino [5]. We also indicate the number of times a particular addon has been downloaded.

Addon Name	**Result**	**Time Taken**(s)		
		P1	**P2**	**P3**
`LivePagerank`	pass	15.9	30.3	0.5
`LessSpamPlease`	*fail*	4.0	24.0	0.1
`YoutubeDownloader`	**leak**	13.2	22.4	0.2
`VKVideoDownloader`	*fail*	0.7	8.7	0.1
`HyperTranslate`	pass	9.6	30.9	0.3
`Chess.comNotifier`	pass	0.8	2.1	0.1
`CoffeePodsDeals`	pass	0.4	2.7	0.1
`oDeskJobWatcher`	pass	0.4	0.9	0.1
`PinPoints`	**leak**	3.6	16.9	0.1
`GoogleTransliterate`	**leak**	1.8	10.87	0.1

Table 2: Addon signature inference result summary. An addon is marked *pass* if the inferred signature has no more flows than the manual signature; *fail* if it has more flows and they are false positives; and *leak* if it has more flows and they are real. The last three columns indicate the time taken by the inference analysis, divided into three phases as outlined in Section 6.2. All times are given in seconds.

real explicit information flow. While this is probably an acceptable flow, it was not described in the developer's addon summary and hence was unexpected. `GoogleTransliterate` communicates with the transliterate web API only if the current URL is not `about:blank` (i.e., the empty page); this is an real implicit information flow, though again probably harmless. These examples highlight the usefulness of using security signatures rather than checking against a fixed policy: rather than a simple pass/fail result, the signature allows the addon vetter to easily determine what types of flows are present and whether they are acceptable or not.

`Pinpoints` is an interesting case. Besides communicating with `yourpinpoints.com` (as indicated in the developer summary), it also communicates with `maps.google.com`. It required careful reading of the extended addon description and the addon code to determine that this was actually intended behavior that should have been included in the addon summary (the addon uses information from the Google Maps API to improve the information it saves). This illustrates another benefit of our signature inference, by highlighting flows that are undocumented or only documented in the addon's fine print.

7. RELATED WORK

There have been a number of previous efforts targeting either information flow security, security analysis specific to JavaScript, or browser addon security. In this section we discuss those efforts most relevant to our own work.

Secure Information Flow.

There are decades of work on secure information flow; for details see the survey by Sabelfeld and Myers [34]. Most of this work is based on type systems. There is some existing work on using abstract interpretation [12, 16], however they do not target any language nearly as complex and difficult to analyze as JavaScript. Abadi et al. [6] establish a close connection between secure information flow and program slicing using dependencies. Hammer et al. [22] present an information flow analysis for Java bytecode using PDGs. They use a traditional lattice-based approach for their analysis, and apply it to a different language and domain than we do. They also do not attempt to distinguish between the different kinds of information flows.

Security Analysis for JavaScript.

There have been both static and dynamic (e.g., [23, 25]) approaches to JavaScript analysis; here we focus specifically on those that contain some static component (e.g., [24, 27, 9, 26, 36, 35]), as well as some security component. These analyses target client-side webpage JavaScript programs rather than JavaScript-based browser addons, which present different challenges and opportunities.

Justet al. [28] blend static and dynamic analyses; they track information flow dynamically as much as possible, but resort to static analysis to capture implicit flows. Because of dynamic tracking, their approach requires changes to the JavaScript runtime and incurs an average overhead of 150%.

Guarnieri and Livshits [20] define a statically analyzable subset of JavaScript and implement a tool to enforce certain security and reliability policies on JavaScript widgets. They use dynamic checks to make certain the executing widget code is within the defined subset language. Their security policy is not formally specified and it is not clear whether they handle only explicit flows or also track implicit flows.

Chugh et al. [14] propose a hybrid mechanism to check certain specific types of malicious information flow in client-side JavaScript. Since client-side JavaScript (unlike browser addons) are allowed to dynamically load new code, they cannot perform a whole-program analysis. Instead, their tool performs a static analysis on all available code and infers a set of dynamic checks necessary to enforce security. Their technique does not scale to more general information flow policies.

Keil and Theimann [31] propose a type-based dependency analysis for JavaScript, and formalize their analysis for a subset of JavaScript. Their analysis can be viewed as static counterpart to data tainting, and they build a tool over the TAJS [27] framework. While not a security analysis, they claim that their analysis could be used as a basis for investigating various security properties.

Browser Addon Security.

Browser addon security has also attracted much attention. Barth et al. [13] propose a new browser addon architecture (which is now adopted by the Chrome web browser) that reduces the attack surface of addons. They achieve this by separating out addons into components with different privileges and isolating the components by running them in different processes. While Chrome requires the addon to explicitly request access for different privileges, it does not perform any information-flow based reasoning to figure out what the addons do with accessible information and whether any confidential information is being leaked.

Guha et al. [21] describe IBEX, a framework to develop and verify secure browser addons. IBEX requires developers to write browser addons in a dependently-typed language called Fine. Their tool can statically check if addons conform to policies specified in a Datalog-like policy language, but only if the addons are written in Fine, requiring extensive developer effort.

Dhawan and Ganapathy [17] describe SABRE, a system that guards against Firefox addon security flaws by performing in-browser dynamic information flow tracking of JavaScript addons. SABRE requires extensive modifications to the browser and the execution-time cost of SABRE is high. Djeric and Goel [18] present another dynamic taint tracking analysis for Firefox addons which has similar characteristics. In contrast, we perform a static analysis of the addons; this means that there is no runtime cost and that reviewers can use their discretion to ignore warnings that turn out to be false positives.

Bandhakavi et al. [11] describe VEX, a static tool for highlighting potential security vulnerabilities in Firefox addons. VEX performs an unsound (by design) static taint analysis of JavaScript code (tracking only explicit leaks) with the intent of finding certain types of vulnerability bugs.

Beacon [29] is a static analysis tool to detect capability leaks in Firefox Jetpack extensions (which is a library of modules that makes writing addons much easier). While Beacon detects capability leaks between modules and over-privileged modules, their analysis is unsound by design, and cannot perform information-flow reasoning.

Lerner et. al. [32] present a type-system based approach to verify compliance of JavaScript-based addons with Private-Browsing mode. This requires some annotation effort and cannot perform information-flow reasoning.

8. CONCLUSION

Browser addons written using JavaScript are extremely popular, but they can be easily exploited by malicious developers. We develop a static analysis to automatically infer security signatures for browser addons. Security signatures summarize uses of security critical APIs, as well as interesting information flows augmented with how they occur in addons. These signatures can be used to understand the behavior of addons with regard to security much more easily than having to go through the entire addon source code manually. Inference of security signatures can be employed to automate addon vetting with very little manual intervention. In our evaluation, we demonstrate the usefulness of our strategy by applying our analysis to ten real browser addons from the official Mozilla addon repository.

9. ACKNOWLEDGMENTS

We thank Tommy Ashmore and Ben Wiedermann from Harvey Mudd College for insightful discussions regarding PDG construction, and Dave Herman and Matt Basta from Mozilla for discussions regarding browser addons. We also thank the anonymous reviewers of CGO for suggesting useful changes. The work was supported by NSF CCF-1319060 and CCF-1117165.

10. REFERENCES

[1] `https://blog.mozilla.org/blog/2012/07/26/firefox-add-ons-cross-more-than-3-billion-downloads/`.

[2] `http://azurit.elbiahosting.sk/ffsniff/`.

[3] `http://www.subhashdasyam.com/2011/04/mozilla-firefox-strategies-mozilla.html`.

[4] `https://addons.mozilla.org/en-US/firefox/`.

[5] `https://developer.mozilla.org/en-US/docs/Rhino`.

[6] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Symposium on Principles of Programming Languages*, 1999.

[7] M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, 2003.

[8] AMO Team at Mozilla. Personal Communication, 2011.

[9] C. Anderson, P. Giannini, and S. Drossopoulou. Towards Type Inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[10] T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, 1993.

[11] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting Browser Extensions for Security Vulnerabilities. In *USENIX Conference on Security*, 2010.

[12] R. Barbuti, C. Bernardeschi, and N. De Francesco. Abstract Interpretation of Operational Semantics for Secure Information Flow. *Inf. Process. Lett.*, 2002.

[13] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Annual Network & Distributed System Security Symposium*, 2010.

[14] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for Javascript. In *Conference on Programming Languages Design and Implementation*, 2009.

[15] G. Costantini, P. Ferrara, and A. Cortesi. Static analysis of string values. In *ICFEM*, 2011.

[16] N. De Francesco and L. Martini. Abstract interpretation to check secure information flow in programs with input-output security annotations. In *International Conference on Formal Aspects in Security and Trust*, 2006.

[17] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Annual Computer Security Applications Conference*, 2009.

[18] V. Djeric and A. Goel. Securing Script-based Extensibility in Web Browsers. In *USENIX Conference on Security*, 2010.

[19] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, July 1987.

[20] S. Guarnieri and B. Livshits. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code. In *USENIX Security Symposium*, 2009.

[21] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified Security for Browser Extensions. In *IEEE Symposium on Security and Privacy*, 2011.

[22] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Secur.*, Oct. 2009.

[23] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *IEEE Computer Security Foundations Symposium*, 2012.

[24] D. Jang and K.-M. Choe. Points-to analysis for javascript. In *ACM symposium on Applied Computing*, 2009.

[25] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Conference on Computer and Communications Security*, 2010.

[26] S. H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications. In *European Conference on Foundations of Software Engineering*, 2011.

[27] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *International Symposium on Static Analysis*, 2009.

[28] S. Just, A. Cleary, B. Shirley, and C. Hammer. Information flow analysis for javascript. In *International Workshop on Programming Language and Systems Technologies for Internet Clients*, 2011.

[29] R. Karim, M. Dhawan, V. Ganapathy, and C.-c. Shan. An analysis of the mozilla jetpack extension framework. In *European Conference on Object-Oriented Programming*, 2012.

[30] V. Kashyap, K. Dewey, E. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: Designing a Sound, Practical, and Extensible Static Analyzer for JavaScript. `http://www.cs.ucsb.edu/~pllab`, 2013. Available under the `Downloads` link.

[31] M. Keil and P. Thiemann. Type-based Dependency Analysis for JavaScript. In *ACM Workshop on Programming Languages and Analysis for Security*, 2013.

[32] B. S. Lerner, L. Elberty, N. Poole, and S. Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *ESORICS*, 2013.

[33] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. *Defcon 17*, 2009.

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

[35] A. Taly, U. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

[36] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript Instrumentation for Browser Security. In *Symposium on Principles of Programming Languages*, 2007.