Flexible Access Control Policies with Delimited Histories and Revocation

Christian Hammer, Gregor Richards, Suresh Jagannathan, Jan Vitek
S3 Lab, Department of Computer Science
Purdue University, West Lafayette, IN

ABSTRACT

Providing security guarantees for software systems built out of untrusted components requires the ability to enforce fine-grained access control policies. This is evident in Web 2.0 applications where JavaScript code from different origins is often combined on a single page, leading to well-known vulnerabilities. We present a security infrastructure which allows users and content providers to specify access control policies over subsets of JavaScript execution traces and reversion to a safe state if a violation is detected. The proposal is evaluated in the context of a production browser where security principals are based on the browser's same origin policy. Simple security policies can be shown to prevent real attacks without imposing drastic restrictions on legacy applications. We have evaluated our infrastructure with two non-trivial policies on 50 of the Alexa top websites with no changes to the legacy JavaScript code and measured the performance overheads of our instrumentation.

1. INTRODUCTION

Many popular Web 2.0 applications mix content from different sources such as news articles coming from an online newspaper, a search bar provided by a search engine, and advertisements served by a commercial partner. The behavior of the web site depends on *all* of its parts working, especially so if it is financed by ads. Yet, not all parts are equally trusted. Typically, the main content provider is held to a higher standard than the embedded third-party elements. A number of well publicized attacks have shown that ads and third-party components can introduce vulnerabilities in the overall application. Readers of the New York Times online version were subject to a scareware attack originating from code provided by a previously trustworthy ad agency.[1] Similarly, several German newspapers were attacked by malware from a legitimate advertisement service which delegated some ads to second-tier agencies. Other sources of attacks come from extension facilities built-in some popular web sites. Facebook, for example, encourages third party extensions to be delivered as JavaScript plugins. Taxonomies of these attacks are emerging [19]. Attacks such as *cross site scripting, cookie stealing, location hijacking, clickjacking, history sniffing* and *behavior tracking* are being catalogued, and the field is rich and varied.[2]

What makes JavaScript particularly challenging is that applications that run in a single client-side browser are composed on the fly, their source code is assembled from different sources and run in the same environment with little isolation. Moreover JavaScript is an incredibly dynamic language. Text can be turned into executable code at any time and very few properties can be statically guaranteed [31]. Recent studies of real-world JavaScript behavior conclusively demonstrate that dynamic features are widely used [28, 32]. Web browsers offer two lines of defense for end-users: The first is a sandbox that protects the operating system from JavaScript code. The second is known as the *same origin policy* (SOP); this policy segregates components into trust domains based on their origin (i.e., a combination of host name, port and protocol) and enforces access control restriction on elements of the web page with a different origin. However, the SOP is not uniformly applied to all resources (e.g. images may come from different origins, potentially leaking information in their URLs), and it is too coarse-grained. While the SOP prevents scripts in one frame from accessing content in another, many web sites choose not to use frames as this form of isolation is highly restrictive.

The majority of attempts to strengthen the security of Web 2.0 applications crucially rely on limiting the dynamism of JavaScript, either through static analysis techniques which reject programs that do not meet certain static criteria or by defining a subset of the language that is easier to verify [12, 13, 23–26]. These approaches have been adopted by the industry as exemplified by Facebook JavaScript (FBJS), Yahoo's AdSafe, or Google Caja. However, these techniques can be circumvented and can miss attacks due to peculiarities and leniencies of different browsers' JavaScript parsers, and they are so restrictive that many valid legacy JavaScript scripts would be rejected.

In this paper, we propose a novel security mechanism, *delimited histories with revocation*, as basis for defining access control policies that have at their disposal a partial view of the computation's history. Access control decisions are

[1] `www.nytimes.com/2009/09/15/technology/internet/15adco.html` and `www.h-online.com/security/features/Tracking-down-malware-949079.html`.

[2] See `www.webappsec.org/projects/threat` and `www.owasp.org/index.php/Category:Attack`.

made on the basis of the policy's security state and the operations performed within the delimited history. If a policy violation is detected, the history can be revoked, returning the program to the heap state preceding the history's start, i.e. before the incriminated script was executed. Delimiting histories allows our technique to scale. While JavaScript pages can generate millions of events, our histories are typically shorter, and fit well within the computation model underlying Web 2.0 applications: once the history of actions of an untrusted code fragment is validated, the history can be discarded. The advantage of the proposed approach is that policies can reason about the impact of an operation within a delimited scope. Effectively giving a view of the outcome of a sequence of operations and thus allowing for "optimistic" policies. Consider storing a secret in a public field of an object. This could very well be safe if the modification is subsequently overwritten and replaced by the field's original value. Traditional access control policies would have to reject the first write, we can postpone the decision and observe if this is indeed a leak. While policies of interest could stretch all the way to dynamic information flow tracking [7], we focus on access control and simple forms of tainting.

We realize our proposal within the JavaScript programming language, targeting Web applications. A consequence of this choice is that we must deal with all of the idiosyncrasies of modern web browsers and define a security model that could, conceivably, be adopted and deployed without disrupting the entire ecosystem. Our main design constraint was backwards compatibility with the web. Existing JavaScript security infrastructures tend to be very restrictive as they reject programs that use dynamic language features. Refactoring code to pass through a filter or use a subset of the language is usually possible, but time consuming. Our goal was to be able to write policies such that only code that actually violates at runtime them needs to be modified. Our secondary goal was to demonstrate acceptable performance. While many of the overheads of a proof-of-concept implementation can be optimized away, massive slowdown would make adoption unlikely. We address this by an in-browser implementation and a careful selection of the properties being recorded. Our implementation leverages the same origin policy as the basis for defining principals. We can thus integrate and strengthen existing SOP-based browser policies. To avoid invasive changes to the JavaScript source, trust boundaries need not be declared explicitly. Instead, we treat invocation of a JavaScript code fragment from a different origin as a potentially untrusted operation. Our system provides support for policy composition and a flexible selection mechanism for policies as a negotiation between content provider and end user.

This paper makes the following contributions:

- *A novel security infrastructure:* Access control decisions for untrusted code are based on delimited histories. Revocation can restore the program to a consistent state. The enforcable security policies with this model are a superset of [34] as revocation allows access decisions based on future events.

- *Support of existing JavaScript browser security mechanisms:* All JavaScript objects are owned by a principal. Ownership is integrated with the browser's same origin principle for backwards compatibility with Web 2.0 applications. Code owned by an untrusted principal is executed in a controlled environment, but the code has full access to the containing page. This ensures compatibility with existing code.

- *Deep browser integration:* Our system was implemented in the WebKit library[3] for e.g. the Safari 5 browser. We instrument all means to create scripts in the browser at runtime, so if untrusted code creates another script we add its security principal to the new script as well. Additionally, we treat the `eval` function as untrusted and always monitor it.

- *Flexible policies:* Our security policies allow enforcement of semantic properties based on the notion of security principals attached to JavaScript objects, rather than mere syntactic properties like method or variable names that previous approaches generally rely on.

- *Empirical Evaluation:* We validated our approach on 50 real web sites and two representative policies. The results suggest that our model is a good fit for securing web ad content and third-party extensions, as legacy code is mostly unhindered by the security policies, with between 66% and 82% of sites being fully functional. Our policies have successfully prevented dangerous operations performed by third-party code. The observed performance overheads were between 11% and 106%.

Limitations: This paper only addresses security issues related to the JavaScript language excluding covert channels and implicit flows. Attacks based on other browser technologies, the browser's layout engine, plugins, social engineering etc. are not in the scope of this work. We assume that the host page is trustworthy and do not prevent potentially malicious behavior stemming from the host, including most forms of cross site scripting. Other research has targeted these issues and future efforts will take some of these into consideration.

2. JAVASCRIPT AND SECURITY

JavaScript is an object-oriented language influenced by the Self language [35]. It has no classes, adopting a prototype-based programming model, and is extremely dynamic. Objects can be (and, as shown in [32], are) modified in arbitrary ways after their creation. Moreover, text can be turned into executable code by the `eval` function (which is more frequently used than we would like [31]). A JavaScript object is a set of properties, a mutable map from strings to values. A property that evaluates to a closure and is called using the context of its parent object plays the role of a method in Java. Each object has a prototype, which refers to another object. As a result, it is difficult to constrain the behavior of any given object, as either it or any of its prototypes could be modified at any time. A JavaScript program running in a browser executes in an event-driven fashion. The browser fires events in response to end-user interactions such as cursor movements and clicks, timer events, networks replies, and other pre-defined browser happenings. Each event may trigger the execution of a JavaScript function. When the function returns, the system is able to handle the next event. The timing and order of events depend on the particular browser, network latency, timer accuracy,

[3] `www.webkit.org`

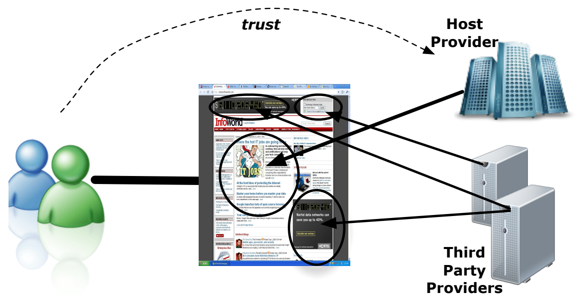

Figure 1: Web applications are made up of components of multiple origins. End-users typically trust the main provider, but do not have a relationship with third-party apps and ad providers. The browser's same origin policy attempts to isolate the different components of a web page.

and other environmental factors. The JavaScript code interacts with the browser, the network, and in a very limited way, persistent storage through a set of native operations that have browser-specific semantics.

2.1 Threat Model

Web browsers have standardized on the notion of same origin policy to isolate content from different providers.[4] Figure 1 illustrates the situation where a single web page is built out of a mixture of trusted and untrusted components kept at bay by the browser's enforcement of the SOP. Web pages are served by a host provider. Each party has its goals. The host provider's interests are to retain the user's trust and to maximize ad revenues. Users want access to the content while restricting, as much as possible, the behavior of ads and other untrusted elements. For the purposes of this paper, we focus on threats originating from third party scripts such as ads and widgets that are embedded in an otherwise trusted page. While there are plugins that block undesired content, this practice hurts web pages' funding, and it is also based on syntactic properties (a blacklist of resources not be loaded) instead of a semantic security policy.

A typical attack is easy to construct. Imagine a host which uses a third-party ad service. The ads are loaded by including a dynamically-created script into the host page. If the ad service is malicious or has been corrupted then it may do much more than simply display ads. For instance, in this scenario there is no built-in security mechanism in the browser which prevents the script from installing a handler for keypress events and silently intercepting and logging everything the user types, such as login credentials for the host site. Although the same origin policy is intended to prevent the script from then communicating this data to an untrusted host, there are ways to work around it, such as encoding the logged data into a source URL for an image tag.

2.2 Related Work

The work presented here builds on and combines ideas that are presented in the literature. History-based access control [1] extends the stack inspection security model of Java and CLR to include a history of methods called to perform access control decisions. We build on their insight and record a wider selection of operations. Inline reference monitors [11, 34] dynamically enforce a security policy by monitoring system execution with security automata. The design of our policy language is informed by Polymer [4]. While the notion of revocation is inspired by research on transactional memory [16], we avoid the transactional memory terminology because delimited histories depart from transactions in that they do not guarantee isolation (side-effects can performed by native methods), they do not perform conflict detection (JavaScript is single threaded), they do not re-execute aborted transactions (an abort is a security violation), and record very different sets of operations. Security applications of transactional ideas were proposed before. Dhawan et al. [10] outlined an API where policies can inspect read and write logs. The approach requires explicit insertion of transactional boundaries and, as there is no notion of ownership, callback functions installed by a transaction need to be wrapped manually into a new transaction with the same policy. In a Java implementation the approach had overheads of up to 10.8× [5]. Birgisson et al. [6] base enforcement of authorization policies in concurrent programs on ideas from transaction memory to eliminate race conditions related to security checks. Rudys and Wallach [33] proposed transactional rollback as a way to implement safe termination of misbehaving codelets in Java. Speculative execution [21] takes a similar approach (speculation and rollback) but is built on top of a binary rewriting tool and reports slowdowns of up to 3000×.

Improving the security of JavaScript programs on modern web browsers has attracted much interest. Many problems come from the pervasive sharing of data and code in JavaScript. While a browser could be viewed as an operating system for web applications, unlike a traditional OS it is quite common to execute components with no clear boundaries. This leads to, amongst others, Confused Deputy attacks [3], where a trusted program unknowingly exercises its authority to perform an action on the behest of an adversary. Many authors have proposed to draw stronger boundaries between components [12, 17, 30]. Library-based approaches tried to limit the interface between components [9]. Intrusion detection techniques were proposed to detecting misbehavior [14,15,36]. Considerable effort was invested in restricting the behavior of JavaScript programs. The highly dynamic nature of the language steered research towards a combination of filtering and rewriting [23]. Static analysis can filter programs before execution, while rewriting is used to inline reference monitors [26,27,29,37]. BrowserShield [29] underlines the difficulty of detecting malicious scripts. Differences between JavaScript parsers led to false positives or missed attacks. Carefully crafted subsets of JavaScript [23,25] allow the seperation of programs into statically verifiable components and others that must be checked at run-time. This was refined into the staged information flow verifier of [8] which reduced the false positive rate down to 33%. Defining two subsets of JavaScript, a trivially statically analyzable one and a run-time checked subset, reduced the false positive rate to 22% [13]. As detecting malicious scripts remains tricky, some authors have advocated extending browsers. Security policies can, for instance, by embedded in web pages [20] with a 15% slowdown for a native implementation, and 10x slowdowns for a purely JavaScript implementation. However, it was shown that neither white- or blacklisting approaches are imprevious to attacks [2]. Off-loading the

[4]`http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`

execution of untrusted code to another JavaScript engine is another alternative to impose isolation. This has been show to have 69% overhead [22] with the drawback of preventing many of valid interactions present in legacy code.

3. DELIMITED HISTORIES WITH REVOCATION

The security infrastructure proposed in this paper applies security policies to controlled, delimited portions of the execution of a program.

3.1 Model

We start with an abstract definition of our security infrastructure.

Definition 1. *The execution state of web application consists of the state of a JavaScript engine P and an environment E. A step of execution is captured by a transition relation $P|E \xrightarrow{\alpha} P'|E'$ where α is a label.*

The set of labels, given in Figure 2 is split into those representing actions initiated by the environment, α^E, either events (for some set of events ranged by n) or returns from calls to native functions; and those representing actions performed by the JavaScript engine, α^P, which include, at least, function calls (including calls to eval) and returns, properties reads and writes, object allocation, and calls to native functions.

α^E	EVT n	External event of type n
	REP v	Return value v from an external call
α^P	APP f v	Call function f with arguments v
	RET v	Return value v from a call
	GET v p v1	Member v.p is v1
	SET v p v1 v2	Update v.p from v1 to v2
	NEW f v	Create an object with constructor f
	INV f v	Invoke external operation f

Figure 2: Event labels.

Definition 2 (Trace). *A trace $T = \alpha_1 :: \cdots :: \alpha_n$ corresponds to an execution $P|E \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} P'|E'$. We write $P|E \vdash T, P'|E'$ when execution of a configuration $P|E$ yields trace T and configuration $P'|E'$.*

In the proposed framework, security policies are applied at the end of untrusted computations, at *decision points*, and, potentially, at every invocation of a native function, *suspension points*. Programs are run with a pair of policies $(\mathcal{P}_S, \mathcal{P}_D)$ such that $\mathcal{P}_S$ is applied at suspension points, i.e. when the trace is T :: INV f v, and $\mathcal{P}_D$ is applied at decision points. Policy decisions are based on the state of the computation P and a subset of trace T which we call a *delimited history*, denoted $T||_R$. We represent the outcome of applying a security policy as a label α^S in the set OK, REV.

Definition 3 (Policy application). *Given a policy $\mathcal{P}$ and a computation $P|E \vdash T, P'|E'$ with a delimited history $T||_R$, applying the policy can yield either $P|E \vdash T :: OK, P' \mid E'$ if $\mathcal{P}(P', T||_R) = OK$, or $P|E \vdash T :: REV, P \mid E'$ if $\mathcal{P}(P', T||_R) = REV$.*

If the policy returns REV, we say that the delimited history $T||_R$ has been revoked. This has the side effect of reverting the JavaScript state of the computation P' to its original state P, the state before the call was made. None of the changes to the memory and internal state of the JavaScript program are retained. On the other hand, the environment E' is not rolled-back (as there is no practical way to, e.g., undo network traffic). A policy may use suspension points to prevent external effects from happening at the cost of having to make an access control decision early.

The start of a delimited history and its matching decision point is an implementation choice (in our case, it is tied to the principal on whose behalf a script is run). The contents of the delimited history $T||_R$ are also implementation specific, retaining the full trace T is likely to be impractical for performance reasons, however the exact subset depends on the properties of interest. For pragmatic reasons we do not consider nested histories in our model, though this may be revisited in future work.

Delimited history with revocation allows policies that cannot be enforced by traditional inline reference monitors [11, 34] (IRMs). IRMs apply a security policy $\mathcal{P}$ before the execution of each event and terminate the program if the policy is violated. Decisions cannot depend on future events. In contrast, revocation lets us base access control decisions on events that are yet to happen: Having seen only a prefix T of the execution, whose last event α might violate security, we do not need to decide at that point whether the prefix T is benign or malign. Instead, we speculatively execute to the next decision point. Only then do we need to check the policy, and revocation ensures that no side-effects of α are visible if a policy violation is detected.

3.2 JavaScript Integration

We now describe the design choices made to integrate delimited histories with web applications. An implementation must choose when to start delimited histories and where to place decision points.

Modern web browsers support a *same origin policy* which defines a notion of principal based on three components of a web page – the application layer protocol, the domain name, and the TCP port of the URL the file originated from. The JavaScript engine uses the SOP to make access control decisions. We extend and build on this. Let's define *host page* as the web page obtained from the URL in the location bar of the browser. It would be reasonable to start a delimited history whenever code originating from any page other than the host page is executed. One could simply start recording when the browser encounters a <script> tag from a different origin and place a decision point at the matching end tag. Unfortunately many ads install callbacks (via setTimeout or an event handler) or install functions in the global object which can be called by the host page's code. Moreover, any eval may, potentially, take as an argument a string created by untrusted code. To handle those cases, we define a notion of ownership.

Definition 4 (Ownership). *Every JavaScript object is associated with an ownership record o, which is a quadruple* `(protocol, domain, port, eval)` *where the last element is either the empty string or the string passed to eval.*

In JavaScript, functions and scopes are objects, thus they are naturally tagged with an ownership record. The default ownership record is that of the host page.

Definition 5 (Recording). *For a page with default ownership record $o = (p, d, r, \text{""})$, recording of a delimited history $T||_R$ starts at a call* APP f v *if no recording is in progress and either of the following holds (a) the owner o' of function* f *is not o, (b) the previous label was a* EVT script *o' and $o' \neq o$, (c) the function* f *is* eval*, and $o' = (p, d, r, v)$. The history is said to be owned by o'. A decision point occurs at the matching return unless the history was revoked at a suspension. Every object created during recording is tagged with the ownership record of the history, o'.*

A delimited history starts at either of script tag, invocation of function object (either directly or through an event handler), or eval. The owner of the function is used to tag all objects created while the history is active. Ownership is invariant during the course of execution, so calling into another function with a different owner does not affect the tag associated by objects that the function creates.

The amount of information recorded in $T||_R$ can be tuned, but we will suppose that at least the following are available: $last(T||_R)$ returns the last label in the trace which, in case of a suspension point is of the form INV f v, $rd(T||_R)$ returns the ordered sequence of property accesses GET v w p v1, $fwr(T||_R)$ returns the ordered sequence of *first* writes SET v p v1 v2. *fwr* is defined so that only the first write to v.p is recorded. This is the minimum information required for suspension policies to make an access control decision (*last*), for privacy policies (*rd*), and for revocation (*fwr*). GET and SET are used for both property accesses on objects and variable access within scopes. These notions are conflated in JavaScript due to the fact that the global scope is also an object, window, and as a result there is little distinction between the two.

Some fine points: When reading a property, such as x.foo, lookup starts with the object referenced by x, but may potentially traverse the prototype chain. In a label GET v w p v1, v is the target of the property lookup, and w is the object where the property was found. The object NONE is used when the property was not found. For writes SET v p v1 v2, JavaScript semantics does *not* specify prototype traversal, i.e. the only object that may change due to a SET is v. There are three cases to consider: (a) the property p is found in v and its old value v1 is updated to v2, (b) the property did not exist in v, the property will be added to the object (this case is denoted by v1 = NONE), (c) the property was deleted from v by the JavaScript delete operation (v2 = NONE).

Revocation of a history entails going through *fwr* and reverting properties to their original values and, in the process, creating and deleting properties as appropriate.

Native code introduces suspension points. There are two cases to consider: functions for which the browser provides a native implementation, and access to properties which may be rerouted to a native getter or setter method. In both cases control escapes and irrevocable actions may be performed. For example, setting the src property of an image tag in the DOM will also trigger the download of the image at that URL. For pragmatic reasons the implementation maintains a whitelist of functions that are deemed safe and do not introduce a suspension point. Any other native function will invoke the suspension policy.

In both the SOP and our delimited history model, each frame is considered an independent entity, and so code running within it is owned by the frame's origin, and not the original page's origin. The SOP already protects inter-frame communication; as such, it is not important to apply the policy to the frame itself, only to code from further origins loaded within it.

4. SECURITY POLICIES

We now turn to the policies that can be expressed in our infrastructure. In the current implementation policies are written in C++ (the implementation language of WebKit). A library of policies is linked dynamically to the instrumented browser at startup. Using C++ implies that policies can be written so as to be relatively efficient and that they are protected from malicious JavaScript code. To improve readability we present policies in idealized pseudo-code.

4.1 Policy API

Our infrastructure includes a number of simple data structures used by security policies. Pseudo-code for these can be found in Appendix B. The class Owner encodes the ownership information required by our variant of the SOP. The enumeration Suggestion contains four values: IGNORE to denote cases where a history is irrelevant to a policy, OK if a history abides by a policy, REVOKE for policy violation which should be revoked, and FIX when the policy was able to take corrective actions. The abstract class Op has subclasses for all operations recorded in a history, these include GetOp, SetOp, CallOp, DownloadOp among others. A delimited history is an instance of the History class which holds a suggestion (by default IGNORE) and has methods to return the last operation in the history, an array of read operations, an array of first writes, a method to replace an operation, and method originalValue(op) which, for a Get or Set operation, will return the value of the field at the start of the delimited history.

A security policy is represented at runtime by an instance of a subclass of Policy created at page load time and reclaimed when the page is destroyed. Every policy has an owner, a reference to the Owner record describing the origin of the code being executed, set at the start of a history and unchanged until its the end. Policy classes have three main methods, querySuspend(), queryEnd(), and accept(). querySuspend() is called at each suspension point and is passed a history and the reason for suspension. It must reply by returning a (possibly modified) history with a suggestion. queryEnd() is called when the end of a history is reached. If the suggestion made by the policy was different from IGNORE, method accept() will be called by the infrastructure with the final history and suggestion to let the policy clean up its state. Policies are composed by building policy combinator objects with sub-policies embedded within them. We will now proceed with some examples of policies that have been implemented for our experiments.

4.1.1 Controlling changes to the Global Object

In JavaScript the global object is a dumping ground for variables defined at the top level and a communication channel between scripts. Our first example is a policy that imposes restrictions on untrusted scripts, they will be allowed to extend the global object, but not to change existing values or objects of a different owner. While this does not prevent breaking the host page, it does prevent subverting the host's functionality. When the host uses feature detection, the ad could install a new function tricking the host into executing it. However, the function would still be run in a delimited history due to its ownership. The policy in Figure 3 extends the base policy class by defining a queryEnd() method which iterates over the operations in the write set of the history (line 3). For each of the write operations it tests if the owner of the object differs from the owner of the history (line 4). If the write updates an existing property in the global object (lines 5-6), then the suggestion is set to REVOKE. The querySuspend function is not distinct from queryEnd, as the writes need to be checked in the same way.

```
1  class AddOnly : Policy {
2    fun queryEnd(history) {
3      foreach op in history.writes()
4        if (differentOwner(op) &&
5            (!op.target.isGlobalObject() ||
6             history.originalValue(op) != NONE)))
7          history.suggestion = REVOKE;
8      return history;
9    }
10 }
```

Figure 3: Restricting updates of global object properties that are owned by another source.

An alternative implementation of this policy could be to reset modified fields to their original value. This requires changing the target of the op in memory, and adding a write to that effect in the history. This would be a case where the suggestion should be FIX.

4.1.2 Hygenic Policy

One might want to adjust the add-only policy above to allow writes as long the original value is restored before the decision point. The policy in Figure 4 checks if all properties of objects that are not owned by the history's owner are restored to their original values by the end of the history. This policy is an example that semantically looks into the future to assess whether an event needs to be prevented.

```
class SameValue : Policy {
  fun querySuspend(history, op) {}
  fun queryEnd(history) {
    foreach op in history.writes()
      if (differentOwner(op) &&
            op.value() != history.originalValue(op))
            history.suggestion = REVOKE;
    return history;
  }
}
```

Figure 4: Ensuring that values of objects belonging to other owners are returned to their original state.

4.1.3 Send After Read Restriction

The lifetime of a policy is tied to that of the page encompassing possibly multiple invocations of untrusted operations. The policy can retain some security state across multiple suspension points and across multiple histories (for the same page). We illustrate the need for retaining state with a policy that prevents leakage of confidential information via HTTP requests. Pseudocode of this policy appears in Figure 5. The policy aims to prevent private data from being leaked. It disallows events which transmit data over the Internet after read events have been performed on data owned by another principal. It also prevents transmission after enabling event listeners, as knowing when events fire is a leak of potentially-private information. The policy maintains some internal security state. The variable pos tracks how much of the history has already been checked, which is updated at every query. The default behavior of querySuspend() is to call queryEnd(). The pos variable is reset by accept(), which is only called if one of the queries returned something other than IGNORE. The hasread variable is a bit of security state that is retained across different histories. This prevents a two stage attack where one script (running with one history) reads information and the second script (running in another history) performs a HTTP request leaking that value. By retaining state across histories we can ensure that the policy will remember that some script has read protected data and prevent the send.

```
class SendAfterRead : Policy {
  var hasread = false;
  var violation = false
  var pos = 0;
  fun queryEnd(history) {
   foreach op in truncate(history.ops(), pos) {
      if (op.isGetOp()) {
        hasread |= differentOwner(op);
      } else if (op.isCallOp()) {
        if (op.isNative() &&
            op.name.is("addEventListener"))
          hasread = true;
      } else if (op.isDownloadOp()) {
        violation |= hasread;
      }
    }
    pos = history.ops().length();
    if (violation) history.suggestion=REVOKE;
    else history.suggestion=OK;
    return history;
  }
  fun accept(history){violation=false; pos=0;}}
```

Figure 5: Preventing sending of data after potentially-private data has been read.

4.2 White Listing

Many sites use secondary servers for storing static content, including JavaScript. For instance, youtube.com's JavaScript is hosted on ytimg.com. For these sites, the same origin policy is too stringent as it would treat the secondary server as untrusted. The Whitelist class is a combinator which modifies the owner of subpolicies so as to give the same owner to all scripts coming from secondary servers. Figure 6 shows a simplified version of the whitelist combinator. The setOwner method is overridden to tag an object with the main hostname instead of the name of a secondary server. The list of secondary servers is passed to the constructor in the parameter list. findPrimary returns the name of the primary server if the current host is in the list of secondary servers.

4.3 Breadth of Security Policies

A survey of policies found in the research literature appeared in the ConScript paper [26]. We discuss whether these poli-

```
class Whitelist : Policy {
  var p, list;
  Whitelist(name, args, list) {
    p=makePolicy(name,args);
    this.list=list;}
  fun setOwner(o) {
    var o2=findPrimary(o)
    super.setOwner(o2);
    p.setOwner(o2);}
  fun queryEnd(h) {return p.queryEnd(h);}
  fun querySuspend(h) { return p.querySuspend(h);}
  fun accept(h){ p.accept(h);}
}
```

Figure 6: A simple Whitelist combinator.

cies fit into our framework. Their first policy disallows all scripts. It is interesting to observe that the related policy that disallows inline scripts (i.e. scripts embedded in attributes of html tags) doesn't fit our SOP-based model, as inline scripts would have the same owner as the host code and thus are not monitored. One could of course change our notion of principal, but this would likely cause more mismatches with legacy code. Most of the policies collected in [26] are syntactic. For example, all policies that restrict access to potentially hazardous methods (maybe involving some form of black- or whitelisting) can be trivially implemented as a policy in our framework. At suspension points the policy needs to check whether the method to be called matches a given signature, whether the arguments have certain properties like a given type, and whether the signature and/or arguments are valid according some black- or whitelist. For methods which do not trigger suspension points, blacklisting can be implemented by checking the history at a decision point. A strength of our approach is that while we can handle syntactic properties, we also can reason about the side-effects that untrusted scripts may induce and their potential for putting the trusted host code's security at risk. Therefore, the policies presented in this section have a different quality than those summarized in [26]. As the next section will elaborate, many real attacks can be prevented with a handful of simple policies.

5. EMPIRICAL EVALUATION

We evaluate our proposed security infrastructure along several dimensions. Based on an implementation in a production browser, we start by considering real attacks, then we run sample policies on realistic web sites and lastly we measure performance and scalability.

5.1 Implementation

We implemented our system in the open source rendering engine WebKit, which is used in the Safari browser. To that end, we modified 29 files, 20 in the JavaScriptCore package, which implements the core JavaScript interpreter and is independent of the actual browser integration, and 9 in WebCore, which accounts for the browser-specific JavaScript like the DOM. We added 588 lines of code to these 29 files. Apart from that, new classes added 4697 LOC. The biggest change was in the Interpreter class, which we instrumented to intercept all relevant events. In particular, we instrumented all opcodes where object properties are accessed (read, store and delete), or objects are created. We also intercept the beginning and ending of script execution, calls, returns, and abrupt termination due to exceptions. The call stack is the basis for determining the policy's decision points. An event is forwarded to a filter class that decides whether history needs to recorded based on the call stack and the owners of involved functions. In case we are recording a delimited history, events are relayed to a bookkeeping class that handles all policy-independent tasks. Newly created objects are tagged with the owner of the current history (whereas the owner of objects created outside a history defaults to the owner of the host page). The owner of the current history and the associated global object are recorded for that purpose. Furthermore, the bookkeeping class records operations including the read and write sets and maintains a list of policies to be checked. For each call to a function, the bookkeeping class determines whether this function has a native implementation, and if so, whether that function is contained in a whitelist of side-effect free functions. If not, a suspension point is triggered and we iterate over all installed polices to query if the call is allowed or needs to be prevented. In the latter case, a flag is returned to the interpreter to not call the native function and abort execution of the script. For getter and setter methods invoked as a result of property access we adopt a different strategy[5]. When calling into a setter or getter function results in reaching native code where a side-effect like network access or database storage is about to take place, we generate a synthetic download or storage event and pass that to the filter before triggering the side-effect. If we are currently in a delimited history these events are suspension points that will be passed along to the policy, which then decides whether the side-effect is permitted or not. In the latter case the native code will basically throw an exception instead of executing the side-effect that is subsequently caught at the invocation point of the setter or getter.

Revocation is a rather tricky business as there are two separate call stacks maintained by the interpreter, and both need to be popped to the point where history recording started. First, the interpreter maintains a stack modeling the JavaScript call stack which needs to be reverted to the level when the history started, before restoring the program counter and resuming execution from there. However, some native calls like eval alter the underlying C++ call stack in addition to the JavaScript call stack. Therefore, when a call frame contains WebKit's HostCallFrameFlag, which signifies the invocation of the interpreter loop function, we need to return from that C++ function and resume unrolling of the JavaScript call stack of the interpreter loop of the previous C++ call frame.

Once the system determines that a script is going to terminate, be it normally or due to an exception or violation of a security policy, it triggers the decision point check in the bookkeeping class. This iterates over all policies checking whether a violation has happened in the recorded history. If a policy signals a violation, all writes in the write set are rolled-back according to JavaScript semantics, any thrown exception is caught and the undefined value is returned. While the return value will be ignored for histories

[5] In WebKit, a native getter or setter may be called for optimization purposes, but in the majority of the cases a standard interpreter function is called, which makes it impossible to distinguish the external code.

started by a script tag, eval expressions might use the return value for further computations, which must be prevented for aborted scripts.

5.2 Attack Vectors

5.2.1 *Samy worm*

The Samy worm[6] is a Cross Site Scripting/Cross-site request forgery (CSRF) worm developed to propagate through the MySpace social networking site. While MySpace filtered the HTML that users add to their sites, the worm exploited holes in the filtering process to inject code into the profile of each person that viewed an infected page. The main hole in the filtering was that some browsers allow JavaScript within CSS tags. MySpace tried to prevent this but failed due the rather lenient JavaScript parsers found in many browsers. For instance, splitting a keyword (such as javascript) across line boundaries as shown in the following code snippet was enough to defeat MySpace's filters:

```
<div id=mycode style="background:url('java
script:eval(document.all.mycode.expr)')"...>
```

The injection mechanism relies on eval since both single and double quotes are already taken. Writing any meaningful JavaScript code inside the style tag is difficult. But it suffices to add the text of the attack to a property of the div tag, called expr, and access this property in the eval expression. The attack itself consists of reading several pieces of information from e.g. the document's location and the document itself and using them as parameters to subsequent AJAX requests that inject the malicious code into the profile of a viewer. The key steps involved: (1) eval('document.body.inne' + 'rHTML') to access the content of the website in a way that circumvented the filter mechanism; (2) redirect from profile.myspace.com to www.myspace.com, as the SOP would block AJAX calls; (3) html.indexOf('Mytoken') to access the random hash from a pre-POST page for the subsequent AJAX request; and (4) sending an AJAX request.

While our approach does not primarily target code injection attacks, it can prevent this CSRF attack. We are sandboxing JavaScript code originating from an eval expression and any other means to dynamically create code. Thus the SendAfterRead policy prevents reading private data on the page (like Mytoken) and subsequently sending the AJAX request. Blocking the AJAX request prohibits CSRF attacks as they rely on the requests being sent out with the user's credentials. We validated this claim experimentally by embedding the Samy worm in a test page and running it with the SendAfterRead policy. When running in an uninstrumented Safari browser, the page containing the Samy worm sends out all AJAX requests necessary to infect the viewer of the profile. With our system, this AJAX request is prevented as the script had read the document location. This read is considered private data, so the policy prevents subsequent Internet requests.

5.2.2 *History Sniffing*

It took more than 10 years[7] until an attack known as *history sniffing* was addressed by browser manufacturers. History sniffing infers the browser's history through the style of links (visited links are displayed in a different color). While newer browsers are immune to the basic form of history sniffing (including the CSS-only variant), this attack is being used in practice [19] and it has been shown that it can be abused very effectively, detecting as many as 30,000 links per second [18]. The SendAfterRead policy would prevent most sniffing attacks. We will return to this with an in depth example in Section 5.3

5.2.3 *Key-logging*

If third party scripts run unmonitored they may even install key- or mouse-loggers threatening user security. For example, a keylogger might attempt to intercept login credentials or other sensitive data provided to the host page. Figure 7 shows an example keylogger that sends out the log periodically via its reportLog function. The SendAfterRead policy prevents these kinds of attacks, as installing an event handler is considered by the policy to be another reason to reject send events. This is because knowing when events fire is a leak of potentially-private information.

```
window.addEventListener("keypress",
  function(event) {
    log.push(event.which);
    if (log.length >= 1024) reportLog(log);},
false);
```

Figure 7: Attack by keylogger in untrusted code.

5.2.4 *Storage*

Recently, a new attack against user privacy has been proposed based on replicating user tracking data in a set of storage mechanisms other than traditional cookies. In particular, HTML5 proposes three more storage containers apart from cookies: Session and local storage and database storage. Session storage is accessible to any page from the same site opened in that window, while local storage spans multiple windows and lasts beyond the current session. Both of these mechanisms provide a key/value storage interface to JavaScript. Even more powerful is the database storage mechanism that provides an SQL interface persisting multiple sessions. All these storage mechanisms represent side-effects that potentially threaten security, as data stored in one session can be accessed and e.g. sent out over the Internet at a later time, which would invalidate policies like NoReadSend. An extension of the SendAfterRead policy in Figure 5 where StorageEvents trigger a violation after reading (in analogy to DownloadEvents) prevents third-party code from changing storage of the host page, as well as reading that storage and sending out information later on.

5.3 Case Study: Sniffles

We present a real attack found on zaycev.net, a file and news sharing site in Russia (#1390 on the Alexa list). The host page loads advertisements from a third-party ad server, but included along with the ads is a history-sniffing attack which determines how many of a list of sites the user has visited in the past. It works by setting CSS styles for visited links, then checking if they are active on links that the attacker is interested in. The code reads data it should not have access to (the style of links) and sends the information to a foreign web server. Figure 8 is a reduced version of the attack seen on the site.

[6] http://namb.la/popular/tech.html

[7] https://bugzilla.mozilla.org/show_bug.cgi?id=57351

```
function ucv(c, d) {
    var a = document.createElement("a");
    a.href = c; d.appendChild(d);
    return (a.style.color == "#ff0000");
}
var d = document.createElement("div"), seen = [];
addStyle(d, "a:visited { color: #ff0000 }");
if (ucv("qwe.ru", d)) seen.push("qwe.ru");
//... same for other servers
seen = seen.join(",");
// send seen to the ad server
```

Figure 8: Excerpt of a history-sniffing attack

Although the SendAfterRead policy could prevent a history-sniffing attack, we devised an extension to the add-only policy which furthermore restricts what the foreign ad is allowed to read, and what functions it is allowed to call. The functions it is allowed to call are those which add to the page, instead of reading data from the page (e.g. the data that must be read to sniff history), which is consistent with the add-only policy.

```
class Sniffles : Policy {
  fun queryEnd(history) {
    foreach op in history.writes()
      if (differentOwner(op) &&
          (!op.target.isGlobalObject() ||
           history.originalValue(op) != NONE)))
        history.suggestion = REVOKE;
    foreach op in history.reads()
      if (differentOwner(op) &&
          (!op.target.isGlobalObject()) &&
          (!op.target.isHTMLDOMObject()))
        history.suggestion = REVOKE;
    foreach op in history.calls()
      if (differentOwner(op) &&
          (!isWhitelistedFunction(op.target, op.func)))
        history.suggestion = REVOKE;
    return history;}}
```

Figure 9: Restricting reads and calls.

The ad is allowed to read from the global state and even from the DOM, but not from the CSS style or from text nodes in the DOM. For certain targets it is only allowed to call certain functions: for document: write, getElementById, createElement, getElementsByTagName; for window: setTimeout, parseInt, open, encodeURIComponent, escape; for other DOM elements: setAttribute, appendChild; all functions are ok for Date; none for all other targets. These functions are sufficient for a conventionally-written ad script to add its advertisement to the page, and in fact the history-sniffing ad in question conforms to this policy if the history-sniffing attack itself is removed. Because none of these functions read information from a page, only adding information, they are a reasonably privacy-preserving subset of the DOM API. Using document.write is fine in our system as scripts that are thus created will be monitored by the security policy, as well. Further restrictions, like allowing access to a DOM node only if that was intended by the host [22] are possible. This is important for targeted ads that are allowed to process public parts of the page but not the sensitive information.

5.4 Real-Site Behavior

We applied security policies to unmodified web sites to evaluate how restrictive the policies are for legacy code. As the previous example demonstrated, in general policies should be tailored to the particular use case and in some cases, the source code of the ads or host page may have to be modified. But is interesting to apply generic policies to unmodified code. We show the results of applying three policies to the top 50 web sites (as displayed on alexa.com on 3-Mar-2011).

We give detailed behavioral data on the top sites in Figure 12 in the Appendix. It is striking to observe the size of the scripts. None are small, ranging from 92KB to nearly 1MB of JavaScript code executed during our short runs. On average, 32 delimited histories were recorded per site, but that number and the behavior of the actual histories varies considerably between these sites. For each site, we kept track of the count of untrusted scripts executed, and thus the number of histories created by our infrastructure and of decision points where policies are evaluated. The number ranges between 3 and 135. Two websites wikipedia and craigslist did not load any untrusted code.

Inspection of the code revealed that Google does not load any third-party code, but twelve eval statements were executed. None was suspended or revoked. Manual inspection suggests that Google is only using eval to deserialize JSON objects which could be off-loaded to a dedicated parsing function and thus need not be monitored. Facebook uses some alternate domains to store static content, which we had to whitelist. While it ran no third-party code, delimited histories were created due to eval, none of which was revoked. Most of these codes do not explicitly read or write properties, three of the evals create and define objects via JSON and pass them back to the host. Since none of the objects owned by the host are changed in that process, all of these are allowed, as well. Yahoo structures its services into several subdomains and maintains a number of hosts for static content, which we whitelisted. We found frames displaying ads from different hosts. Some scripts were revoked due to updates to a field owned by the host. In a realistic setting the frame content provider would specify the hosts to be whitelisted so that their scripts would work as expected. Yahoo also loads scripts from Facebook. These scripts change the href property of a link not owned by the third-party script, which means that its side-effect are revoked at the next suspension point. Youtube needs whitelisting of a static content domain s.ytimg.com. With that in place, the page looks and feels like the original even though some advertisement services and third-party functionality are rejected. For example, we found that a script from 2mdn.net tries to install a global function isValid, however a function with that name had already been defined by the host code in the global scope. Therefore, allowing the untrusted code to install the function may put the functionality of the host in jeopardy; in the worst case a malicious script may subvert the security of the host code. Our policy rejects that script and reverts its side-effects.

We categorized subjectively the behavior of the site with security enabled and report on the results in Figure 10. We first applied, as a sanity check, the Empty policy which places no constraints on the code. Not surprisingly, no degradation of behavior could be observed. The SendAfterRead policy performed well: 82% of web pages were fully functional. In 14% of the cases, the policy worked as an ad blocker, terminat-

Policy	Functional	AdBlock	Partial	Broken
Empty	50	0	0	0
SendAfterRead	41	7	2	0
AddOnly	33	8	7	2

Figure 10: Categorization of 50 benchmarks. **Ads blocked:** The site worked, but its ads did not. Although acceptable to clients, it is not acceptable to servers, since ads are a revenue source. **Partially Functional:** Features of the site proper did not work or had reduced functionality. **Broken:** Site is mostly nonfunctional.

ing the execution of the ad without affecting the site. In 4% of the pages the site worked with some features missing. The AddOnly policy is slightly more restrictive. live.com is rendered non-functional as none of the content can be accessed, as is sina.com.cn in which all nontrivial functionality was disabled. 14% of sites have some missing features (most often auto-complete in a search bar) and 16% have some ads blocked. This shows that even with generic policies and unmodified code, our infrastructure is able to provide guarantee for more than 66% of the sites with no loss of functionality. In a realistic deployment one can expect customized policies and modification in the JavaScript code to yield significantly smaller false positive rates. For all runs we used a policy combinator to add the Whitelist policy to treat secondary servers as trusted.

5.5 Performance

To evaluate the performance, we arranged for our browser to load three web sites and to fire a deterministic sequence of events on each of these. We measured the execution time, both with and without our instrumentation. The sites were cached in a proxy to avoid inconsistency between runs and the JIT was not used. Five runs of each site for each mode were performed. The measurements were performed with the empty policy, as it has all of the instrumentation overhead needed to record policies. The machine had 2 3GHz dual-core Intel Xeon processors and 4GB 667MHz DDR2 RAM. Our instrumentation is based on WebKit revision 60027, and the same revision was used for the uninstrumented runs. WebKit was loaded into Safari 5.0.2. The three measured sites span the spectrum of history sizes: MSNBC (www.msnbc.msn.com) runs nearly all of its code through eval, and so 98.41% of the traceable events were run in a history, YouTube (www.youtube.com) substantially less so (only 0.62%), and Google Maps (maps.google.com) has no histories at all.

Figure 11 shows the results. MSNBC which is a worst case scenario with most of the execution being recorded, has an overhead slightly of 106%. The execution records 17 histories accounting for 424 suspensions, 117,328 read events and 22,924 write events. In most sites we expect the amount of controlled code to be much less than the amount of host code as Figure 12 suggests. YouTube and GMaps are more typical with 12% to 14% overheads. Our YouTube benchmark ran 22 transactions accounting for 28 suspensions, 705 read events and 773 write events, and our Google Maps benchmark ran 48 transactions accounting for 3 suspensions, 33 read events and 122 write events.

We believe that the performance overheads could be reduced by optimizing the recording code and, in the case of sites that use eval heavily, use the ownership information

	Instrumented		Uninstrumented		Over-
Site	Avg.	Std. dev.	Avg.	Std. dev.	head
MSNBC	77	0.50	37.2	1.50	106.9%
YouTube	145	2.35	128.2	1.64	13.1%
GMaps	222.0	2.35	199.2	1.48	11.4%

Figure 11: Runtimes in milliseconds of various sites with and without instrumentation.

associated to strings to avoid monitoring the evaluation of strings that are entirely created by the host.

6. CONCLUSIONS

This paper presented a security mechanism for monitoring untrusted code based on delimited histories with revocation. When security policies can reflect upon sequences of operations performed by a computation, it is possible to support semantic properties rather than the more limited syntactic checks of traditional rewriting or wrapping approaches. Most prominently, it is possible to write policies to detect side-effects that jeopardize confidentiality or integrity, like sending out private data to untrusted servers or updating sensitive data structures. The ability to revoke history and roll-back their side effects increase the expressive power of policies over and above what can be expressed with a pure inlined reference monitor.

We focused on an integration of this idea into JavaScript and the security model of modern web browsers. By extending the same origin policy and tagging objects as well as code with owners we obtain a robust notion of principal which can leverage to add our new security mechanism in a non-intrusive manner. Whenever control is transferred across an ownership boundary, security policies are activated. For that reason, trust boundaries become implicit. It is remarkable that we can apply policies to unmodified JavaScript code and have significant success with legacy code. Our evaluation in the WebKit JavaScript engine demonstrates its effectiveness in preventing realistic attack vectors like internet worms, as well as applicability and scalability to realistic web sites exemplified by the 50 most popular pages. Our overhead is competitive with rewriting mechanisms even without extensive optimization of the implementation.

We see several promising direction for future work. We intend to improve performance by integrating the monitoring mechanism with a trace-based just-in-time compiler and specializing the code generated to the policy. Thus the compiler can emit only the logging that is needed for the policy to make its access control decision. A better treatment of eval will greatly reduce the size of histories, especially in the case of pathological web sites that run entirely in eval. A second strand of work will investigate a mixture of static and dynamic checking to target information flow policies. In particular, we are interested in seeing if we can get a handle of quantitative flows by reflecting on histories. Lastly, we intend to investigate high-level declarative languages for specifying polices to allow for policy-carrying web pages.

APPENDIX

A. REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Network and Distributed System Security Symposium (NDSS)*, 2003.

[2] E. Athanasopoulos, V. Pappas, and E. P. Markatos. Code-injection attacks in browsers supporting policies. In *W2SP 2009: WEB 2.0 Security and Privacy*, 2009.

[3] A. Barth, C. Jackson, and W. Li. Attacks on javascript mashup communication. In *W2SP 2009: WEB 2.0 Security and Privacy*, 2009.

[4] L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Trans. Softw. Eng. Methodol.*, 18:9:1–9:43, 2009.

[5] A. Birgisson, M. Dhawan, Ú. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Conference on Computer and Communications Security (CCS)*, pp. 223–234, 2008.

[6] A. Birgisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *Conference on Computer and Communications Security*, (CCS), pp. 223–234, 2008.

[7] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Symposium on Operating Systems Principles*, (SOSP), pp. 31–44, 2007.

[8] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Conference on Programming language design and implementation (PLDI)*, pp. 50–62, 2009.

[9] F. De Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. Smash: secure component model for cross-domain mashups on unmodified browsers. In *Conference on World Wide Web (WWW)*, pp. 535–544, 2008.

[10] M. Dhawan, C.-c. Shan, and V. Ganapathy. The case for JavaScript transactions. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, pp. 1–7, 2010.

[11] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, Ithaca, NY, USA, January 2004.

[12] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer. Talking to strangers without taking their candy: isolating proxied content. In *Workshop on Social Network Systems (SocialNets)*, pp. 25–30, 2008.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, pp. 151–197, 2009.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Conference on World wide web (WWW)*, pp. 561–570, 2009.

[15] O. Hallaraker and G. Vigna. Detecting malicious JavaScript Code in Mozilla. In *Conference on Engineering of Complex Computer Systems (ICECCS)*, pp. 85–94, 2005.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *International Symposium on Computer architecture (ISCA)*, pp. 289–300, 1993.

[17] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: operating system abstractions for client mashups. In *Workshop on Hot topics in Operating Systems (HOTOS)*, pp. 16:1–16:7, 2007.

[18] A. Janc and L. Olejnik. Feasibility and real-world implications of web browser history detection. In *Workshop on Web 2.0 Security and Privacy*, 2010.

[19] 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 (CSS*, pp. 270–283, 2010.

[20] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *International conference on World Wide Web (WWW)*, pp. 601–610, 2007.

[21] M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX Annual Technical Conference*, pp. 219–232, 2007.

[22] M. T. Louw, K. T. Ganesh, and V. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium (SECURITY)*, 2010.

[23] S. Maffeis, J. Mitchell, and A. Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *Computer Security (ESORICS)*, pp. 505–522. 2009.

[24] S. Maffeis, J. C. Mitchell, and A. Taly. Run-time enforcement of secure JavaScript subsets. In *Workshop of Web 2.0 Security and Privacy (W2SP)*, 2009.

[25] S. Maffeis and A. Taly. Language-based isolation of untrusted JavaScript. In *Symposium on Computer Security Foundations (CSF)*, pp. 77 –91, 2009.

[26] L. A. Meyerovich and B. Livshits. ConScript: specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Symposium on Security and Privacy (S&P)*, pp. 481–496, 2010.

[27] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *International Symposium on Information, Computer, and Communications Security (ASIACCS)*, pp. 47–60, 2009.

[28] P. Ratanaworabhan, B. Livshits, and B. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Conference on Web Application Development*, June 2010.

[29] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. *ACM Trans. Web*, 1(3):11, 2007.

[30] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *European Conference on Computer Systems (EUROSYS)*, pp. 219–232, 2009

[31] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do – a large-scale study of the use of eval in JavaScript applications. ECOOP 2011.

[32] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pp. 1–12, 2010.

[33] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Conference on Dependable Systems and Networks (DSN)*, pp. 439–448, 2002.

[34] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000.

[35] D. Unger and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 227–242, Dec. 1987.

[36] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Conference on Computer and Communications Security (CCS)*, pp. 173–186, 2009.

[37] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Symposium on Principles of programming languages (POPL)*, pp. 237–249, 2007.

B. POLICY API

```
class Owner {
  var port, domain, protocol, evalstr;
}
enum Suggestion {
  IGNORE, OK, REVOKE, FIX
}
class History {
  var suggestion;
  fun last();
  fun reads();
  fun writes();
  fun ops();
  fun replace(oldOp, newOp);
  fun originalValue(op);
}
class Policy {
  var owner;
  Policy(args) {}
  fun setOwner(o) {owner=o;}
  fun querySuspend(history, op) {
    return queryEnd(history);
  }
  fun queryEnd(history) { return history;}
  fun accept(history) {}
}
```

Pseudo-code for the Policy APIs. Class Policy must be extended to define security policies. Class History is the delimited history recorded while executing an untrusted script.

C. EVALUATION STATISTICS

We recorded how many times the same behavior occurred (Count), the owner of the page or iframe (Host Origin), the owner the code (Script Origin), the reason why a the history is being recorded (Cause) which can be one of **e**val, **s**ource, and **f**unction with different owner, the number of suspension points and how many of those caused revocation (Suspend(Rvk)), the total number of revocations (Revoke), the size of the read and write set in the history (R/W). For space reason we do not list the entire ownership information but rather part of the URL. Note that for a given site the host for frames may differ from that site's URL and that host would still be a trusted host, as frames are protected by the SOP in the browser. The websites wikipedia and craigslist did not load any potentially untrusted code.

Count	Host / Script Origin	Cause	Suspend(Rvk)	Revoke	R/W
	`google`		(8 files, 92KB)		
12	google / google.com	e	1(0)	0	0/0
	`facebook`		(26 files, 765KB)		
12	facebook / facebook.com	e	1(0)	0	0/0
1	facebook / facebook.com	e	1(0)	0	0/89
1	facebook / facebook.com	e	1(0)	0	0/35
1	facebook / facebook.com	e	1(0)	0	0/9
	`yahoo`		(22 files, 953KB)		
2	yahoo / facebook.com	s	2(1)	1	30/61
2	doubleclick.net / 2mdn.net	s	0(0)	0	0/3
2	doubleclick.net / doubleclick	e	5(0)	0	17/3
2	tweetmeme / tweetmeme.com	s	7(1)	1	142/25
96	tweetmeme / tweetmeme.com	s	0(0)	0	0/0
1	advertising / advertising.com	s	4(0)	0	7/8
	`youtube`		(8 files, 321KB)		
7	youtube / youtube.com	e	1(0)	0	0/0
2	doubleclick.net / doubleclick	e	1(0)	0	1/1
2	youtube / 2mdn.net	s	6(0)	1	90/73
2	youtube / 2mdn.net	s	9(0)	0	143/72
118	youtube / 2mdn.net	f	2(0)	0	30/0
1	youtube / googletagservices	s	1(1)	1	15/9
1	youtube / googlesyndication	s	6(1)	1	71/31
1	youtube / googlesyndication	s	1(1)	1	35/14
1	youtube / google.com	s	1(1)	1	605/707
	`amazon`		(92 files, 1012KB)		
9	amazon / about:blank	s	0(0)	0	0/0
1	amazon / amazon.com	e	1(0)	1	2/24
1	amazon / amazon.com	e	1(0)	1	2/25
1	amazon / amazon.com	e	1(0)	1	2/26
1	amazon / amazon.com	e	1(0)	1	2/22
1	amazon / amazon.com	e	1(0)	0	0/2
	`wikipedia`		(27 files, 301KB)		
	`twitter`		(29 files, 629KB)		
1	twitter / google-analytics.com	s	0(0)	1	6/31
1	twitter / twitter.com	e	1(0)	0	0/1
1	twitter / twitter.com	e	1(0)	0	0/0
	`ebay`		(5 files, 239KB)		
1	ebay / ebay.com	e	1(0)	0	0/19
1	ebay / ebay.com	e	1(0)	1	106/46
7	ebay / ebay.com	e	1(0)	0	0/0
5	ebay / ebay.com	e	2(0)	1	2/1
5	ebay / ebay.com	e	2(0)	0	3/0
2	ebay / ebay.com	e	2(1)	1	26/4
2	yahoo / yimg.com	s	1(1)	1	28/2
6	ebay / ebay.com	e	1(0)	0	8/0
1	ebay / ebay.com	e	1(0)	0	7/0
1	interclick / interclick.com	s	1(1)	1	74/8
1	interclick / doubleverify.com	s	1(1)	1	47/9
1	ebay / promo.ebay.com	s	0(0)	0	0/0
2	ebay / ebay.com	e	1(0)	0	1/0
1	ebay / ebay.com	e	1(0)	0	6/0
	`blogger`		(19 files, 716KB)		
2	blogger / google-analytics.com	s	1(0)	1	385/330
1	google / google-analytics.com	s	1(0)	1	400/332
1	blogger / google.com	s	1(1)	1	16/27
	`craigslist`		(8 files, 112KB)		

Figure 12: Dynamic characteristics of untrusted code for each of the top 10 site.