Between Worlds: Securing Mixed JavaScript/ActionScript Multi-party Web Content

Phu H. Phung, *Senior Member, IEEE,* Maliheh Monshizadeh, *Student Member, IEEE,* Meera Sridhar, *Member, IEEE,* Kevin W. Hamlen, *Member, IEEE,* and V.N. Venkatakrishnan, *Member, IEEE*

Abstract—Mixed Flash and JavaScript content has become increasingly prevalent; its purveyance of dynamic features unique to each platform has popularized it for myriad web development projects. Although Flash and JavaScript security has been examined extensively, the security of untrusted content that combines both has received considerably less attention. This article considers this fusion in detail, outlining several practical scenarios that threaten the security of web applications. The severity of these attacks warrants the development of new techniques that address the security of Flash-JavaScript content considered as a whole, in contrast to prior solutions that have examined Flash or JavaScript security individually. Toward this end, the article presents FlashJaX, a cross-platform solution that enforces fine-grained, history-based policies that span both Flash and JavaScript. Using in-lined reference monitoring, FlashJaX safely embeds untrusted JavaScript and Flash content in web pages without modifying browser clients or using special plug-ins. The architecture of FlashJaX, its design and implementation, and a detailed security analysis are exposited. Experiments with advertisements from popular ad networks demonstrate that FlashJaX is transparent to policy-compliant advertisement content, yet blocks many common attack vectors that exploit the fusion of these web platforms.

Index Terms—Access Controls, ActionScript, Flash, In-lined Reference Monitors, JavaScript, Online Advertising, Scripting, Web Security

1 Introduction

JavaScript (JS) and Adobe ActionScript (AS) (the language for authoring Flash applet content) are two widely used platforms for developing web content. According to recent surveys on w3techs.com, 92% of *all websites* use JS and 23% of them use AS, demonstrating the popularity of these platforms for web development.

Due to these two platforms' popularity, much of today's web contains mixed JS-AS content—untrusted code that combines AS and JS. Such code is extensively used in interactive advertisements, embedded third-party videos, and plugins for content-management systems such as WordPress and Joomla. The popularity of such content stems in part from interactive and multimedia features that are uniquely available through each platform. Mixed AS-JS content leverages the benefits of both: the interactive features of JS for click-tracking and context customization, and the multimedia features of Flash for improving user experience.

Hosting sites that include such third-party content must deal with the security and privacy issues that such inclusions introduce. Major concerns include confidentiality of private client data (e.g., cookies), integrity of host- and user-owned content, and availability of hosting site services (e.g., ads must not deter users from visiting the site).

The prior literature includes extensive research on securing third-party web inclusions, but most solutions focus on JS content. These include transformation of untrusted code (e.g., [1], [2]), security reference monitors (e.g., [3], [4], [5], [6], [7], [8]), and safe subsets of JS (e.g. [9], [10], [11], [12], [13], [14]). To a lesser extent, there have also been efforts to secure Flash/AS inclusions [15]. But in spite of these efforts, the security of *mixed AS-JS content* is relatively less researched.

Meanwhile, the abuse of mixed AS-JS content for malicious campaigns constitutes a significant rising threat for content currently in circulation [16]. For example, a Gmail vulnerability allowed attackers to steal sessions by exploiting the AS-JS interface [17]. A WordPress attack (CVE-2012-3414) exploits vulnerable AS-to-JS interface calls. A recent study found that 64 of over 1000 top sites contain Flash applications vulnerable to JS XSS attacks [18]. (Our evaluation discusses other real-world attacks).

A deeper examination of these attacks reveals that any defense against attacks arising from AS-JS interactions must adopt a *holistic* view of the security-relevant events on both platforms. Prior work developed for JS or Flash has not been designed with this holistic perspective, and therefore does not satisfactorily address security issues arising from mixed AS-JS content. The problem of preventing malicious behaviors that exploit combined AS-JS technologies has therefore remained open.

Problem Scenario: To illustrate the security challenges outlined above, consider a page publisher P who supports her site via embedded advertisements purveyed by an ad network N. Publisher P trusts neither the ads (some of which may be malicious) nor N (which may fail to filter some malicious ads, and whose ad-loading code might contain exploitable vulnerabilities). To protect the integrity and reputation of her site and retain clientele, P wishes to protect her clients from this potential malicious content.

Unfortunately, P cannot assume that all her clients take all available steps to protect themselves from the dangers that malvertisements pose. For example, some clients probably

- *P.H. Phung is with U. Gothenburg and U. Illinois at Chicago.*
- *M. Monshizadeh and V.N. Venkatakrishnan are with U. Illinois at Chicago.*
- *K.W. Hamlen is with U. Texas at Dallas.*
- *M. Sridhar is with U. North Carolina at Charlotte. The work reported herein was performed while at U. Texas at Dallas.*

use un-patched browsers with known vulnerabilities. Finally, some of the policies P must enforce are specific to P's site or page content. For example, on a page that uses pop-up windows for legitimate navigation, P may wish to disallow all ad-generated pop-ups, which could fool clients with phishing attacks that impersonate the legitimate pop-ups. P wishes to protect her clients as much as possible given these realities.

To host ads, N requires P to copy N's JS ad-loading code onto her published pages. When this code is served to clients and executed, it dynamically modifies the hosting page within the browser to display dynamically chosen ads (implemented in JS, Flash, or both) served either by N or directly from advertisers. Since the ad-loading code requires dynamic read and write access to the hosting page, it must not be placed in a protected *iframe*, nor may it be enclosed in any page element that disallows scripting. Such measures effectively deactivate ads, depriving P of most or all ad revenue. Likewise, many ads heavily use Flash-JS interaction (e.g., for click-tracking, contextual customization, and multimedia); therefore P must not disable such interaction lest it block many legitimate ads, losing significant ad revenue.

Our Approach: FlashJaX provides publisher P a means to enforce custom security policies on untrusted third-party ad and ad network content without deactivating the critical functionalities, like scripting and JS-Flash interaction, required by most ads. To use FlashJaX, P adds a ⟨`script`⟩ tag near the top of her published pages, which dynamically loads the FlashJaX IRM on client browsers before any other scripts run. She also statically labels any trusted, protected page content (e.g., publisher-authored JS code or Flash objects) with the owning principal (expressed as a principal-identifying *html* class attribute). Unlabeled Flash and JS code is, by default, fully untrusted by FlashJaX. Finally, she may write page-specific policies (detailed in §4) that define the events and event-traces that each principal may exhibit. To secure untrusted Flash content, P also hosts or accesses a trusted ad-proxy service that dynamically installs the FlashJaX IRM into untrusted Flash ads served to clients.

At runtime, the FlashJaX IRM dynamically monitors all untrusted JS and Flash code executed on client machines to enforce P's policies. As an example of such monitoring, consider the pop-up prevention policy mentioned above, which prohibits ad principals from exhibiting pop-ups but permits trusted publisher code from doing so. Pop-ups are implemented via a limited collection of JS Document Object Model (DOM) and Flash runtime API services. FlashJaX monitors these services by intercepting calls and checking the impending operation against the acting principal's policy. FlashJaX passes the call through to the browser's underlying JS/Flash VM only if the principal's policy permits it.

To track the current principal, FlashJaX enforces history-based policies that constrain dynamically generated code and the events it exhibits. For example, a Flash ad owned by principal A that dynamically generates JS code that creates a new script within a region of the page owned by principal B must be successfully monitored by FlashJaX and constrained by policy A, not B. Such dynamic script generation is extremely common; almost all real-world ads and ad networks perform many layers of dynamic script generation and *html* tree manipulation as they execute. Therefore, monitoring and constraining history-based policies (i.e., those that constrain event histories rather than just individual events in isolation) over dynamically generated, cross-platform code is a critical challenge addressed by our framework.

The remainder of the paper is organized as follows: Section 2 sketches some attack scenarios that motivate securing the AS-JS interface. Section 3 outlines the FlashJaX architecture and technical approach. Design and implementation details are described in §4, and a security analysis is summarized in §5. Section 6 evaluates the implementation in terms of effectiveness, compatibility, and performance. Related work is discussed in §7. Section 8 discusses the relevance of FlashJaX to web security, and §9 concludes.

2 AS-JS Interface Attacks

This section describes the AS-JS interface, and details several motivating attack scenarios that exploit it.

2.1 The AS-JS interface

AS-JS interaction is implemented by the *call* and *addCallback* methods of Flash's `ExternalInterface` runtime class. AS calls JS method $f(a_1, \ldots, a_n)$ by invoking $call(f, a_1, \ldots, a_n)$, where f is a string that is *passed uncensored to the JS VM and evaluated as JS code* to obtain a JS function reference, and where arguments $a_1, \ldots, a_n$ are passed as values. The evaluation of f as JS code at global scope is a root of many vulnerabilities in AS-JS cross-language scripts. To permit JS to call AS, the AS code may invoke $addCallback(n, c)$, which registers AS function closure c as callable by JS under the pseudonym n (a name that is added to the JS namespace of the html object that embeds the AS script). Closure c may return a value, which is marshaled and passed by value back to the JS caller. Together, these facilitate two-way communication between AS and JS.

Security for this interface is provided by the `allowScriptAccess` property of the `object` and `embed` tags of the embedding page, which may be set to `always` (full access), `sameDomain` (same origin access), or `never` (none). Same origin access is the default. Additionally, by default JS may only call an AS closure registered with *addCallback* if the caller and callee originate from the same domain. AS callees may adjust this restriction using the *allowDomain* method of the Flash runtime's `Security` class.

While useful in some settings, these security features are too coarse to distinguish malicious from non-malicious behavior in many contexts. Disallowing all AS-JS interaction or limiting it to same origin access breaks a large percentage of legitimate advertisement scripts. Therefore many ads and publishers resort to allowing all access, inviting attacks.

The following subsections introduce several attacks that exploit the AS-JS interface. Such attacks can only be prevented by defenses that span both domains. While the examples focus on AS-JS interface attacks, FlashJaX also prevents attacks launched purely from JS or AS. However, to highlight the novelty of our system over prior works that can only guard each platform in isolation, we focus our discussion here on attacks that involve the interface.

2.2 Threat Model

Publishers such as Gmail that display third-party content on client browsers are exposed to a wide variety of threats. It is therefore important to clarify our threat model, specifically on the nature of the protections we offer and the threats that are outside our scope.

In-scope threats. Our broad goal in this effort is to equip publishers with the ability to place restrictions on third-party content. Publishers need this ability, for instance, to ensure that third party content would not cause harm by compromising the integrity of first-party content. For instance, we would like to empower a publisher such as NY Times to place restrictions on third party ads in their ability to modify site-owned content. We also would like to give publishers the ability to use our framework to enforce content confidentiality policies. For instance, a publisher like Gmail can enforce a policy that prohibits ads that read email messages from subsequently communicating with any other untrusted principals.

Although the system we describe in this paper is capable of enforcing a broad range of content restriction policies, we primarily discuss its relation to the new attack surface explored in this paper—the AS-JS interface—for reasons of novelty and to explore this vector in depth. The remainder of this section motivates the importance of this threat vector with a discussion of attacks.

Out-of-scope threats. We omit threats for which publishers can readily deploy strong protections based on prior work, or for which appropriate policies are client browser-specific and therefore not amenable to specification or enforcement by publishers. Such threats include behavioral tracking attacks that abuse cookies (which clients can address by configuring their browsers' cookie policies to desired privacy levels), cross-site request forgery (CSRF), or attacks through side channels (such as visited links or timing channels).

2.3 Motivating attack scenarios

Attack #1: Circumvention of SOP

The AS and JS VMs both enforce *Same Origin Policies* (SOPs) that prohibit cross-domain interactions. However, AS and JS SOPs have slightly different semantics [19] due to their differing computation models, and these can presently be exploited to circumvent SOP on either side.

For example, a malicious Flash ad can circumvent AS's SOP to contact a victim third-party site by dynamically crafting a malicious JS script and passing it to the JS VM via Flash's external interface. The malicious JS script accesses the browser's DOM API to create a new ⟨`script`⟩ node (e.g., using `appendChild` or `document.write`). This new node has a `src` attribute whose URL references the third-party victim site. The URL can additionally contain information passed from the AS applet to the third-party site. The new node is not subject to AS's SOP, so it successfully contacts the remote site and retrieves the result, which is communicated back to the AS side using the external interface. The attacker thereby escapes AS's SOP to perform two-way communication with the victim, which can be exploited to launch click forgery, resource theft, or flooding attacks.

This malicious behavior cannot be recognized by single-platform detection on either the AS or JS side, since AS permits (and ads regularly use) AS-to-JS communication, and JS permits (and ads regularly use) dynamic script generation. A cross-platform solution is required to link these two steps together and detect the SOP violation.

Attack #2: Malicious Payload Injection From Flash

Heap-spraying is a form of code injection attack that first allocates large regions of malicious payload code into a victim VM's heap, and then exploits a control-flow hijack vulnerability (e.g., buffer overflow) to branch to the injected payload. Address space randomization and other protections prevent attackers from reliably learning the addresses of these injected payloads, but if the payload is large enough and has enough entry points, a randomly corrupted control-transfer targets it with high probability.

Since some vulnerabilities are previously unknown (i.e., zero-day), signature-matching malware protections often attempt to detect the payload injector instead, because it is larger and easier to identify using monitoring mechanisms. However, malware authors have been frustrating these defense efforts by using cross-language heap-spraying attacks [20]. In this scenario, the attacker implements AS code that sprays the JS VM's address space. The exploit is then implemented separately in JS. Identification of such attacks requires cross-platform solutions that can piece together the two separate halves of the attack implementation.

Attack #3: Cross-Principal Resource Abuse

Publishers often embed ads from multiple ad networks. This exposes the publisher and ad network to attacks from the (possibly less trustworthy) ads hosted by another network if those ads abuse AS-JS interaction to hijack shared DOM resources or functions exposed by victim scripts.

Although Flash scripts may control access to their exposed functions, such as by calling `allowDomain`(⟨*domains*⟩) to admit only JS callers from ⟨*domains*⟩, the coarse granularity of these facilities makes it extremely common for ad developers to use them imprudently, such as by supplying wildcard "$\star$" for ⟨*domains*⟩, which permits universal access [21]. This makes the AS functions accessible in the JS global scope, allowing them to be invoked by all untrusted JS code.

Hosting sites cannot effectively filter ads by the quality of their underlying implementations, so inevitably some vulnerable ads become embedded in the served pages on the client side, exposing the clients to attack. For example, a malicious JS advertisement, even if sandboxed in the JS domain, can call such exported functions. This affords the ad illegitimate access to DOM objects if the exposed AS functions access or manipulate those objects in the DOM. Prevention of this attack requires the ability to attribute principals to actions across the AS-JS interface.

The above scenarios illustrate the need for cross-language monitoring. It is clear that JS sandboxing methods alone cannot prevent the attacks in scenarios #1–3. These scenarios involve the AS-JS boundary, which is typically outside the scope of approaches aimed at sandboxing purely JS or AS code. The next section describes how FlashJaX's architecture prevents these malicious scenarios.

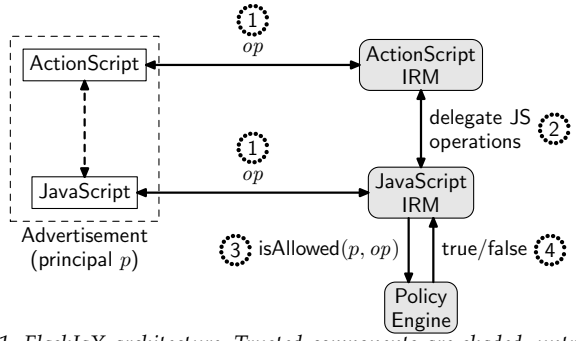

Fig. 1: FlashJaX architecture. Trusted components are shaded; untrusted (monitored) components are unshaded.

3 Architecture

3.1 Overview

FlashJaX affords publishers a fine-grained mechanism to safely embed untrusted JS and AS content in their web pages. To avoid modifying the client browser or VMs (which would introduce significant deployment barriers), we adopt an in-lined reference monitoring approach. *In-lined Reference Monitors* (IRMs) [22] modify untrusted code to enforce security policies from the inside. The resulting code is *self-monitoring*, and can therefore be safely executed on standard browsers and VMs without additional client-side monitoring.

FlashJaX's IRM consists of JS and AS code introduced by the embedding page. The IRM mediates security-relevant events exhibited on the client, permitting or denying them based on a provider-specified policy.

A naïve design implements separate IRMs for JS and AS; however, this approach has many drawbacks. To enforce policies involving a global event history, separate IRMs must ensure that their security states are synchronized at every decision point. This raises difficult race condition and TOCTTOU vulnerability challenges, and impairs performance.

To avoid this, FlashJaX centralizes security state-tracking to the JS half of the IRM, and implements an AS side that shifts the significant policy decisions to the JS side. This is efficient because most security-relevant AS events include AS-JS communication as a sub-component; the IRM therefore couples its AS-JS communications atop these existing ones to avoid unnecessary context-switches.

Figure 1 summarizes the resulting architecture. The FlashJaX components (shaded) consist of JS and AS event mediators, and a JS policy engine. The former intercept events from untrusted JS and AS, respectively, whereas the latter tracks event history and makes policy decisions.

Step 1 of the figure depicts the exhibition of a security-relevant event op by the untrusted JS or AS code, which is intercepted by the IRM. If the event occurs on the AS side, the AS IRM implementation consults the JS side in step 2. The JS-side IRM intercepts the event or AS-to-JS communication and consults a principal-specific policy in step 3. The policy engine updates the security state and yields a true/false answer in step 4, causing the operation to be permitted or suppressed.

As an example, an embedded AS ad might exhibit an op that spawns JS code that tries to overwrite the publisher's DOM. In a typical browser environment, there is nothing to prevent a malicious ad from successfully attacking its embedding page in this way. However, on a page equipped with FlashJaX, the ad's JS code is intercepted by the IRM and executed at a lower privilege level than the publisher. When the unprivileged write-operation is intercepted, the policy engine determines that the acting principal lacks write-access to publisher-owned content, and suppresses the operation.

3.2 Technical approach

The example above illustrates three essential capabilities of the IRM: It must (1) protect its programming and other publisher-provided page content from harm, (2) guard access to all security-relevant operations, and (3) attribute guarded events and page content to acting/owning principals. We henceforth refer to these three capabilities as IRM *tamper-proofing* (i.e., *integrity enforcement*), *complete mediation*, and *principal-tracking*, respectively. In addition, to enforce multi-principal, history-based policies, the IRM must track both principal-specific and global security states. This section discusses our technical approach to achieving these goals.

FlashJaX implements a JS/AS cross-language IRM that constrains untrusted script access to the DOM API—functions and data properties that JS scripts access to manipulate the page and browser. AS code cannot access the DOM directly; instead, it submits strings to the JS VM via Flash's external interface, which are executed as JS code at global scope. The heart of our IRM is therefore a JS-side implementation that guards access to the DOM and tracks security state, while the AS half redirects external interface accesses to the JS half.

In addition to tamper-proofing and complete mediation, which are established challenges for any IRM, our enforcement of multi-principal policies introduces significant challenges associated with principal-tracking. Accurate principal-tracking is challenging because modern ad scripts are highly dynamic, performing many layers of event-driven runtime code generation as they execute. Solutions that conservatively reject or lose principal information for dynamic code are therefore impractical because they break most ads.

We now describe each of these capabilities of FlashJaX at a high level. Section 4 discusses implementation details.

3.2.1 Tamper-proofing

Tamper-proofing ensures that the IRM's internals are unavailable to untrusted content. This is enforced differently at the JS and AS layers as described below.

At the JS layer, tamper-proofing is achieved by placing most of the IRM's implementation inside an anonymous JS function closure, as illustrated in Listing 1. Declarations beginning with the **var** keyword are strictly local to the closure's scope, and therefore cannot be accessed by JS code outside that scope unless the local scope explicitly exports global aliases to them. This enables the IRM to enforce a protected interface for its internal implementation.

A similar approach suffices to tamper-proof the AS half of the IRM. The majority of the AS IRM is implemented as a sealed, final, monitor class in a dedicated namespace. AS type-safety and object encapsulation therefore prevent untrusted code from accessing the monitor class's private members.

```
(function(){                          // begin local scope
  var principal = "bottom";           // protected principal-tracking var
  getPrincipal = function(){          // export global get-accessor
    return principal; };
  var wrap_window = function(w) { // wrap security-relevant op
    var o_open = w.open;
    w.open = function() {
      if (isAllowed(principal, "open", arguments))
        return wrap_window(o_open.apply(this, arguments));
      else return null; }
    return w; }
  wrap_window(window);
})();                                 // close and execute local scope
```

Listing 1: A tamper-proof local scope.

3.2.2 JavaScript Mediation

FlashJaX mediates DOM API events by wrapping them with guard functions that consult the policy before forwarding the request to the DOM. To achieve complete mediation, the IRM assigns wrappers to all aliases of these security-relevant functions before any untrusted code runs. Aliases include static names and properties of dynamically created window objects (e.g., `frame` and `iframe`). Although some static aliases are browser-specific, all aliases of a given security-relevant operation can typically be captured by wrapping the properties of a single root object atop the JS prototype inheritance chain [23]. The wrapper assignment code is placed first on the page so that it is guaranteed to run prior to any untrusted code. Dynamically generated aliases are captured by mediating all DOM functions that can generate window objects, and wrapping any fresh aliases they introduce before returning control to untrusted code.

Data property accesses are guarded using JS *setters* and *getters*, which trigger specified handler code whenever an operation would otherwise read or write a given property.

3.2.3 ActionScript Mediation

The AS half of our IRM guards AS-to-JS control-flows by statically in-lining an external interface wrapper class into untrusted AS code at the binary level. FlashJaX's binary rewriter automatically, statically replaces all bytecode operations that access the external interface with ones that access the wrapper class instead. This affords the wrapper class complete mediation of all AS-to-JS flows.

Static identification of class member references can be complicated by the fact that AS binaries frequently generate references dynamically (e.g., from strings). Malicious code can use such dynamic generation to obfuscate references, concealing them from static analyses.

To avoid these complications, our rewriter therefore guards references to the external interface's *namespace* rather than its classes or members. The namespace part of a reference is almost never generated dynamically. (The only AS mechanism for doing so requires a static reference, making it statically identifiable.) This approach greatly reduces the amount of in-lined code, improving performance and providing a natural resistance to reference obfuscation attacks.

Thus, all JS events invoked by AS are labeled with the originating principal and mediated by the JS IRM, so that the policy engine can apply the correct policy for each principal. For example, to block attack scenario #1 (Flash circumvention of SOP), the AS IRM labels each AS-to-JS communication with the acting principal, allowing the JS IRM to enforce a whitelist policy that maps each principal to the domains it may access.

Rewriting AS binaries changes their origins; but this is not a problem because, as discussed in §2, communicating Flash applets can opt-out of SOP enforcement whenever the sender and receiver agree. This allows the IRM to enforce a different SOP that constrains communications as if the applets had their original origins. If an ad must open direct communication channels back to the advertiser's server, the advertiser can unobtrusively accommodate this via a cross-domain policy [24]. (Note that this is transparent to ad networks, since their communications with ads are facilitated by network-served JS code, not advertiser-authored AS code.)

3.2.4 Principal Tracking and Event Attribution

On pages with multiple ads, each ad principal is governed by a distinct policy. Enforcing such *multi-principal policies* is necessary to prevent scenarios such as cross-principal resource abuse (scenario #3 of §2).

FlashJaX therefore deploys multi-principal tracking and event attribution as follows. Whenever trusted code (e.g., the page publisher content) introduces untrusted code (e.g., by loading an ad), the untrusted code is launched using the IRM's `runAs` method, which defines and maintains the code's principal in a protected shadow stack. The shadow stack stores a list of principal identifiers, one for each `runAs` frame on the JS VM's call stack. The IRM's `runAs` method is the only means by which the privilege level changes, and is strictly local to the IRM; untrusted code may not call it directly. The policy manager can read the shadow stack to identify the principal responsible for each event exhibited by the code, and thus apply a principal-specific policy.

Dynamic runtime code generation is a great challenge for principal tracking. FlashJaX addresses this by catching all runtime code generation channels and wrapping them in new calls to `runAs`. We discuss this in more detail in §4.3.

3.2.5 Policy Engine

FlashJaX enforces publisher-specified policies on third party content, and therefore requires a policy language that supports the following.

Multi-principal, cross-language policies: Attack scenario #1 of §2 entails a sequence of inter-principal, cross-language communications. To enforce such policies, FlashJaX must track policy-relevant actions within and across both the AS and JS platforms, and attribute actions to acting principals.

Stateful policies: Scenario #1 also has the characteristic that although each step is permitted in isolation, the full sequence of steps is impermissible. Such policies cannot be expressed as a static list of access control rules [25], [26], [27]; an adequate policy language must capture the evolving security state of the system. Such policies are most commonly expressed as finite state automata (FSAs) [22], [26], [28].

Fine-grained control: Many security-relevant page resources are stored within a single, monolithic data structure—the document tree. To guard such resources, the policy language must support fine-grained controls. For example, publishers may restrict a principal's access to entire element subtrees, individual elements, or both.

Custom policies: We also require first-class support for publishers to author custom policies based on emerging threats. Although a fixed list of rules might support the specific policy instances described in this article, it will not generalize to the needs of all publishers and to future threats. For example, to block heap-sprays (scenario #2 of §2), publishers may need to write custom content-filtering predicates that mine binary data for newly discovered malware indicators.

To meet these requirements, FlashJaX expresses policies as FSAs that track security state based on past events. The FSAs recognize languages of permissible *traces*, where a trace is a sequence of principal-event pairs, and each event is a DOM operation parameterized by its argument values. Our policies are powerful enough to detect and prevent a wide range of attacks, including the attack scenarios described earlier. A detailed exposition of policy specifications and their expressiveness is provided in §4.4.

4 IMPLEMENTATION

This section describes implementation details of FlashJaX that have been briefly introduced in the previous section.

4.1 JavaScript Wrappers

FlashJaX implements wrappers to mediate DOM API access. Listing 1 illustrates a wrapper that guards the `window.open` DOM function, which creates a pop-up window, by assigning `window.open` $= f$, where f is a function that creates the requested window if and only if the current principal's policy permits it. Thereafter, all calls to `window.open` call wrapper f instead.

Naïve JS wrapper implementations are known to be vulnerable to a variety of attacks, including *prototype poisoning* and *caller-chain abuse* [4], [7], [23]. FlashJaX therefore employs secure wrapper implementations advanced by prior work [23]. In summary, these safe wrappers:

1) wrap all aliases of each security-relevant operation,
2) coerce all untrusted inputs to expected types, and
3) only call securely stored copies of JS API methods (e.g., those of *Function* and *Array*), which are carefully protected from attacker corruption.

FlashJaX augments these secure wrappers to additionally mediate events from AS, and to precisely attribute events to the correct principal, even if the event-exhibiting code was generated dynamically. As a result, FlashJaX is tamper-proof against common exploits such as those described in [7].

For data property access mediation, FlashJaX leverages ECMAScript 5's `defineProperty` function [29, §15.2.3.6] to define setters and getters for a given property.[1] The getters and setters read and store values to protected, locally-scoped, principal-specific copies of each guarded property. The guarded properties are set *non-configurable* so that untrusted JS code cannot remove or change the guards. Global variables introduced by scripts are similarly protected from abuse by other scripts by adding non-configurable getters and setters to such variables during privilege-changes (i.e., within `runAs` from §3.2.4).

A special approach is required to adequately guard the DOM's `document.cookie` property, for which writes have the side-effect of creating or modifying browser cookies that may persist across sessions, and reads yield lists of previously written cookies (possibly some from prior sessions). FlashJaX employs two browser-dependent techniques to protect cookies: On browsers that support cloning of the `document` node (e.g., FireFox and IE), FlashJaX creates a local, protected copy of `document`, which the IRM's wrappers henceforth exclusively access to safely read and write cookies. On browsers that implement cookie facilities as browser-specific getters and setters of `document.cookie` (e.g., Opera), FlashJaX creates local, protected copies of these getters and setters to mediate access to them.

In both cases, FlashJaX adds custom getters and setters to the global `document.cookie` property to provide filtered, principal-specific views of the cookie store for each untrusted principal. (The trusted hosting origin's access is not filtered.) This confines each untrusted principal's cookie accesses to its own cookies.

One browser we tested (Chrome) currently admits neither approach due to a known browser bug,[2] preventing us from protecting cookies on that browser. However, once this bug is fixed, FlashJaX's cookie-protection is expected to be compatible with all major browsers.

4.2 ActionScript Rewriter

Our AS binary rewriter automatically in-lines wrappers around all AS-to-JS flows within AS bytecode applets. The in-lined wrapper class redirects all such flows to a JS method named `fromAS` exposed by the JS IRM. For example, a JS call originally of the form $f(a_1, \ldots, a_n)$ is translated by the wrapper into a JS call of the form $\texttt{fromAS}(id, s, f, a_1, \ldots, a_n)$, where id identifies the principal, s is a one-time secret (discussed below), f is a JS expression identifying the callee, and $a_1, \ldots, a_n$ are the arguments to f. The `fromAS` method then executes $f(a_1, \ldots, a_n)$ at privilege id.

Impersonation Attack & Defense. The `fromAS` function must protect itself from impersonation attacks in which a malicious JS principal calls it with a false *id*. JS callees cannot reliably identify their callers; incoming calls are essentially anonymous. Therefore, the `fromAS` implementation calls-back the AS applet from which each incoming AS call claims to originate, asking it to confirm the call. The AS-side IRM confirms by validating secret s. Secret s is freshly chosen for each AS-to-JS call, exists only for the lifetime of the confirmation process (just a few AS/JS instructions), is temporarily stored on the AS side in a private field, and exists on the JS

1. Safari does not currently comply with this part of the ECMAScript 5 standard, preventing protection of data properties on Safari. However, the rest of the DOM remains protected.

2. http://code.google.com/p/chromium/issues/detail?id=45277

```
var shadowStack = [ ]; // Implement a shadow stack as a list.

// Other code may read (but not write) the current principal.
thisPrincipal = function(){
  return (shadowStack.length < 1) ? "bottom" :
         shadowStack[shadowStack.length − 1];
}

// Execute closure f at a specified privilege level.
var runAs = function(principal,f) {
  shadowStack.push(principal);
  f.apply = js.Function.apply;
  var r = f.apply(this,
    js.Array.prototype.slice.call(arguments,2));
  shadowStack.pop();
  flush_write(principal);
  if (typeof r !== "undefined") return r;
}
```

Listing 2: Shadow stack code. Object `js` stores original native JS objects. Exception-handling is not shown.

side only as a local argument to `fromAS`. This keeps it safe from interception during the limited window when it is valid.

4.3 Principal Tracking and Event Attribution

Listing 2 sketches FlashJaX's shadow stack implementation, by which it tracks principals.

Principal-tracking Algorithm. To execute an untrusted function f at privilege level p, the IRM invokes `runAs`(p, f), which pushes principal identifier p onto the shadow stack, runs f to completion, pops p off the shadow stack, and returns the result. Note that since f is a closure with its own context, calling f within the lexical scope of the monitor does not give it access to anything in the IRM's local scope. Its scope is whatever context it had at creation.

As f executes, it may exhibit security-relevant events, which are intercepted by the IRM. The IRM's guards consult `thisPrincipal()` to determine the principal to whom each event should be attributed. Based on the result, a principal-specific policy is then consulted to determine whether to grant or deny each event.

The (trusted) embedding page may label static code f with a principal identifier p, causing the IRM to execute f at privilege p. Trusted (non-ad), static code is therefore typically labeled with identifier *top* ($\top$), which grants full privileges. Untrusted, static code for ads is labeled with ad principals so that it executes with lesser privileges. Unlabeled code runs with *bottom* ($\bot$) privileges by default—i.e., the intersection of all privileges granted to all the principals.

Dynamically Generated Code. As callee f runs, it might modify the page, such as by adding new elements with event-handlers containing code. The DOM provides several mechanisms for dynamic page modification (e.g., `Node.appendChild`), all of which are monitored by FlashJaX. No special monitoring is required for `eval`, since the code it generates inherits the context of the `eval`, preserving the shadow stack. Thus, FlashJaX handles all dynamically generated code channels. To illustrate, we here consider the most common and most general one: `document.write`.

```
1  var flush_write = function(principal){
2    var i = document.createElement("ins");
3    i.innerHTML = write_buffer[principal];
4    write_buffer[principal] = "";

6    foreach element e within i do {
7      // Enclose handlers in principal-preserving closures.
8      foreach attribute a of e do
9        if (typeof e.a == "function") {
10         var oldHandler = e.a;
11         e.a = function() {
12           var r = runAs(principal, oldHandler);
13           if (typeof r !== "undefined") return r; };
14       }

16     // Execute scripts at generating principal's privileges.
17     if (e is a ⟨script⟩ element) {
18       var newScript = makeFunction(e.textContent);
19       e.textContent = "";
20       runAs(principal, newScript);
21     }

23     // Wrap any fresh aliases of security-relevant functions.
24     if (e is a ⟨frame⟩ or ⟨iframe⟩ element) {
25       wrap_window(e.contentWindow);
26       wrap_document(e.contentWindow.document);
27     }
28   }

30   i.owner = principal;
31   document.lastChild.appendChild(i); // Append i to page.
32 }
```

Listing 3: Wrapping dynamically-generated code.

Operation `document.write`(s) pushes string s directly onto the head of the browser's input stream during page-loading. Browsers execute scripts as soon as they are parsed during the page-loading process, so these dynamic scripts run sometime after the generating script writes them but before the page is fully loaded. (The exact time of execution is browser-specific.) Ads depend on this behavior, so it is important to support and preserve it.

To do so, FlashJaX intercepts and buffers strings passed to `document.write` (by storing them in the `write_buffer` variable in Listing 3) without immediately committing them to the page. Once f completes, `runAs` calls the algorithm sketched in Listing 3 to parse these buffered strings, label the resulting HTML and JS code with the contributing principal's identifier, and commit it to the page. To avoid writing our own parser, we use a trick: Assigning to the `innerHTML` property in line 3 leverages the browser's built-in parser to convert the string into an HTML tree stored in the body of an ⟨`ins`⟩ node object.

Listing 3 replaces all code in the new content with closures that recursively call `runAs`, so they will run at the proper privilege level when triggered. For example, lines 11–13 replace event-handler `e.a` with such a closure. The JS closure semantics guarantee that when this closure is executed, `principal` will equal the principal that generated the code, and `oldHandler` will execute at its original scope (not the IRM's scope). Thus, dynamically contributed code inherits the privileges of its creator.

Line 18 processes JS code contributed in the body of a dynamically generated ⟨`script`⟩ element. IRM subrou-

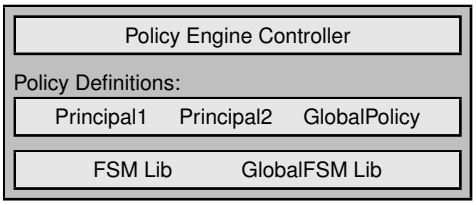

Fig. 2: Policy enforcement system architecture

```
Ctrl.checkPolicy = function(principal, event, obj, flags){
  localFSM = Ctrl.policies[principal][FSM_CTRL];
  return localFSM.checkPolicy(event, obj, flags) &&
     globalFSM.checkPolicy(principal, event, obj, flags);
}
```

Listing 4: Policy engine controller

tine `makeFunction` (implementation not shown) uses JS's `Function` constructor to convert its string argument into a closure that can be called by `runAs`. Closures created with `Function` always have global lexical scope, and therefore safely exclude the IRM's local scope. The new closure is executed immediately, since that is how most browsers treat dynamically contributed scripts.

In addition to the local ⟨`script`⟩ content handled by Listing 3, the full FlashJaX implementation also handles remote scripts (specified as a URL in a `src` attribute) by loading them through a proxy via `XMLHttpRequest` and processing the resulting string as a local script. This step is omitted from the listing for brevity.

Finally, any dynamically generated window objects introduce fresh, unguarded aliases to security-relevant operations protected by the IRM, and are therefore wrapped with suitable guards by lines 24–27.

4.4 Policy Definition and Enforcement

FlashJaX's policy engine is implemented in three layers of JS as depicted in Fig. 2. The bottom layer provides two library classes, *FSM* and *GlobalFSM*, which are built on the FSMJS library [30]. They provide tools for defining and accessing policy files within the policy engine.

The next layer defines global and per-principal policies. In a typical policy file, page publishers define security states, the initial state, forbidden states (i.e., those rejected by the security automaton) and the transition relation. There is typically one policy file per principal, plus a global policy file that constrains all untrusted principals and their interactions.

The third layer is the *Policy Engine Controller*, illustrated in Listing 4, which interfaces the policy engine to the monitor. Publishers assign policies by adding policy class instances to the `Ctrl.policies` array in the controller. At runtime, the controller calls `checkPolicy` to test whether the global FSM and acting principal's local FSM accept the impending event. If so, the controller updates the FSM states; otherwise it rejects.

This design accommodates history-based, stateful policies over events exhibited by multiple principals. Events include API calls with various arguments (e.g., DOM objects), and global variable accesses. Some expressive policy examples are illustrated below.

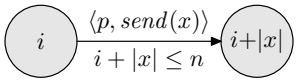

Fig. 3: A local FSA for a policy preventing heap-sprays.

Formal Description. FlashJaX defines and enforces safety policies expressible as security automata [25] or edit automata [26] that intervene by suppressing policy-violating events. (Other interventions are possible, but we have found suppression to be the most useful and practical for our policies.) Formally, a FlashJaX policy is a quadruplet $\langle P, E, S, G\rangle$ where P is the universe of principal identifiers, E is the universe of events, $S : P \to RE$ is a mapping from principals $p \in P$ to regular expressions over alphabet $\{p\} \times E$, and G is a regular expression over alphabet $P \times E$. Regular expression $S(p)$ specifies the language of permissible traces for principal p, and G specifies the language of globally permissible traces. The system-wide policy is therefore given by regular expression $\bigcap_{p\in P} S(p) \cap G$. Intuitively, the policy identifies the set $S(p)$ of event sequences that each individual principal p may exhibit, and an additional set G that all untrusted principals as a collective may exhibit.

Policy Example. Fig. 3 shows a policy that prevents cross-platform heap-spraying attacks (scenario #2 of §2). Such attacks conceal themselves by implementing the spray in AS so that it is not visible to JS analysis tools. The sprayed payload is then passed across the AS-JS boundary, allowing malicious JS code to branch to the payload via a JS-side exploit not visible to AS analysis tools.

The FSA in Fig. 3 prevents such behavior by tracking the cumulative size of data passed from AS to JS by each untrusted principal p. When the cumulative transmission size reaches bound n, future transmissions are rejected. The policy therefore conservatively rejects applets that pass suspiciously large quantities of data from AS to JS. Our experience is that only malicious ads exhibit such behavior, but a more refined policy could additionally apply malware detection heuristics to the passed payloads to support non-malicious ads that pass large quantities of legitimate data to JS.

The FSA for this policy consists of $n + 1$ states, where n is the maximum cumulative transmission bound. (The number of states is not an implementation burden, since all $n + 1$ can be expressed as a single integer whose values range from 0 to n.) For brevity, we draw the FSA using the notation of extended finite automata (XFAs) [31] in Fig. 3.

A global policy can likewise be defined to limit the total cumulative transmission size of all principals by using $\star$ (denoting any principal) on the edges. This blocks heap spraying through collusion.

Multi-principal Policies. FlashJaX's label-based attestation (§4.3) facilitates enforcement of some sophisticated write-protection policies, which can be leveraged to block cross-principal resource abuses (e.g., scenario #3 of §2). Fig. 4 shows an example with three principals: an ad network p_1, and two ads p_2 and p_3 served by the network. Event $read(e, p)$ denotes a read from element e labeled p. The label is assigned dynamically by the IRM's attestation mechanism. The FSA on the left allows p_2 to read p_1's data and its own data, but not p_3's data. Similarly, the FSA on the right

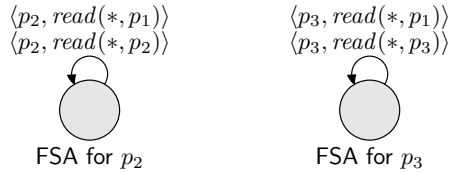

Fig. 4: Local FSAs for a policy that permits ads p_2 and p_3 to read data owned by ad network p_1 but not data owned by each other.

```
ElementWhitelist = ['p1', 'p2','p3'];
fsm = new FSM();
t.init = function(){
  fsm.setState( 's0'); fsm.setInitialState( 's0' );
  fsm.setState( 's1'); fsm.setFinalState( 's1' );
  fsm.setEdge( 's0', 's1', 'readDOM', 'p3');
  currentState = fsm.getInitial();
}
```

Listing 5: Policy definition for p_2 in Fig. 4

prohibits p_3 from reading p_2. Thus, ads may consult the ad network but not each other. Wildcard $*$ is used to denote edge labels ranging over all principals and event arguments ranging over all values. Self-edges for events that do not affect the security state are not shown.

Listing 5 shows the policy definition for p_2's FSA, where $s1$ is the final state that rejects p_2's attempts to read p_3.

Other Policy Examples. Using this policy language, we designed and implemented policies that address several attack scenarios, including the three attack scenarios described in §2, which abuse AS-JS interactions. These are described below. As mentioned earlier, these attacks cannot be prevented by mechanisms in JS or AS alone. Other policies are discussed in §6.3.

To stop Flash circumvention of SOP (scenario #1), FlashJaX enforces a principal-based whitelist policy: *Each principal may only communicate with sites defined in a whitelist*. FlashJaX's principal-tracking and event attribution mechanisms attribute all JS code called from AS. Therefore, FlashJaX identifies whether the JS event originates from an AS principal, and applies an appropriate policy. The policy enforces SOP by only permitting communications with whitelisted sites.

To inhibit cross-language heap sprays (scenario #2), FlashJaX enforces a multi-principal, history-based, resource-bound policy: *The cumulative AS-JS data transmission by each principal may not exceed a per-principal bound defined by the policy, and the total transmission by all principals may not exceed a global bound defined by the policy*. The size of transmissions by each AS principal is tallied by the policy engine. If it exceeds the limit, FlashJaX destroys the violating Flash object by removing it from the page to prevent the attack.

To block cross-principal resource abuse (scenario #3), FlashJaX enforces principal-based access control policies: *Each principal may only access particular page elements*. The legitimate accesses for each principal are defined by a whitelist of DOM objects. The IRM monitors all DOM tree accesses and disallows accesses that originate from unauthorized principals.

5 Security Analysis

FlashJaX enforces *rewrite-enforceable safety policies* [27]—i.e., trace properties that stipulate that some observable, decidable "bad thing" (possibly contingent upon the history of past events) must not happen. Security-relevant events consist of JS API calls and member accesses, parameterized by their arguments and a principal identifier. Prior work has shown that such policies can be formalized as *aspect-oriented security automata* [28]. Principals are defined by the embedding page, which provides a trusted mapping from untrusted scripts to principal identifiers. Dynamically generated scripts inherit the identifier of the code that generates them.

The IRM's ability to enforce these policies is contingent upon its ability to (1) maintain IRM integrity (i.e., tamper-proofing), (2) completely mediate security-relevant events, and (3) accurately attribute events to principals. The enforcement strategy for each of these goals forms the foundation for enforcing the next, as depicted in Figure 5. Each tier of security is described below.

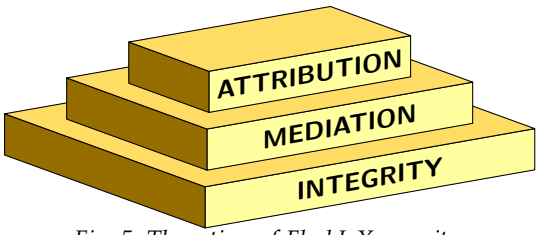

Fig. 5: Three tiers of FlashJaX security.

IRM Integrity follows from two core language features: lexical scoping on the JS side, and object encapsulation (type-safety) on the AS side. That is, on the JS side, all security-critical data and code are stored within the local scope of an anonymous JS closure. This prevents any outside access except via accessors explicitly exported as global variables. This accessor collection constitutes the protected interface to the IRM. Similarly, on the AS side, all security-critical data and code are stored as private members of a final, sealed AS class. Integrity of the AS portion of the IRM therefore follows from the type-safety and object encapsulation guarantees of the AS bytecode language.

Complete Mediation of JS API calls is achieved by moving all security-relevant API method pointers inside the protected lexical scope before any untrusted code runs. For a given security-relevant API method, FlashJaX systematically explores and wraps all its aliases, including static names and dynamic aliases (§3.2.2). Furthermore, FlashJaX also wraps all channels generating JS code at runtime (§4.3). Thus, IRM integrity implies complete mediation of these events; once they are inside the local scope, they can only be accessed via the protected IRM interface.

Mediation of data property accesses is via non-configurable JS getters and setters, whose complete mediation is guaranteed by the JS VM [29, §8.7]. It is impossible for untrusted code to change or delete the properties of wrapped objects.[3] Complete mediation on the AS side is achieved by statically rewriting all references to the `flash.external`,

3. In earlier versions of Mozilla browsers, deleting a wrapped object could silently restore the original object [3]; however, this is no longer possible with ECMAScript 5's non-configurable feature [29].

`flash.net`, and `flash.utils` namespaces (except those within the trusted IRM class) before any AS code runs. This makes it impossible for any untrusted AS code to acquire a direct reference to any external interface member; all JS accesses therefore use the AS IRM.

Accurate Event Attribution follows from complete mediation of security-relevant events, which include all page- and code-write operations. The IRM's write-mediation labels all dynamically written content with the authoring principal. JS code is labeled by dynamically replacing it with a closure that preserves the principal. Thus, when it runs, it inherits the privileges of its author.

For this labeling to succeed, the IRM must account for all possible locations where JS code can be dynamically submitted and stored. For example, if the JS `setTimeout` method is inadvertently omitted from the list of mediated methods, scripts could use it to escape the labeling mechanism and run unlabeled code. Since in practice the JS API has a broad, browser-specific, and ever evolving surface, we consider this to be the most attackable portion of our system. To make FlashJaX robust against such omissions, unlabeled code therefore always runs at the lowest privilege level (defined as the intersection of permissions granted to all principals in the system). Thus, a principal-tracking failure could lead to conservative rejection, but never a policy violation.

Correctness of the guard code that enforces each principal's policy is facilitated by our choice of an automaton-based policy formalism whose semantics, expressive power, and correct implementation are extremely well-established in the literature (cf., [22], [26], [27], [28], [32]). Our implementation leverages these solid design principles for high assurance.

6 Evaluation

6.1 Code and Experiment Settings

The core JS IRM is a 300-line static script atop the hosting page that wraps DOM functions before untrusted code runs. The wrappers consist of about 600 more lines that mediate security-relevant events, including dynamic writes, by consulting the policy engine. The policy engine implements FSMs using an adaptation of the `fsmjs` library [30] (about 9K LOC). Each individual FSM-controller contributes less than 100 LOC.

Our AS binary rewriter is a small ($<$ 1K SLOC) standalone Java application that uses no external libraries except the Java standard libraries. It injects the wrapper class (about 700 bytes of pre-compiled AS bytecode) and redirects all external interface references to the injected wrapper methods. Rewriting is fast; the median rewriting time is 0.62ms/K (averaged across 57 Flash ads on a 2.93 GHz, Intel quad core desktop running 64-bit JDK 1.7.0 atop Windows 7 SP1 with 4 GB ram), and rewriting increases the binary size of ads by just 1.24%.

FlashJaX code and experiments described in this section are available at http://securemashups.net/flashjax. (The site does not collect any information regarding its visitors.)

TABLE 1: Attack scenarios and FlashJaX prevention

Attacks	Policy applied	FlashJaX prevents?
AS Circum- vention of SOP	Principal-specific whitelist	✓
Cross-language Heap-spraying	Principal-specific and history-based	✓
Cross-Principal Resource Abuse	Principal-specific access control	✓
Wrapper vulnerabilities	Wrapping all aliases	✓
Confidentiality and integrity	Principal-specific & fine-grained access control	✓
Ad-specific	Principal-specific & fine-grained access control	✓

6.2 Compatibility

Our compatibility experiments test whether FlashJaX preserves existing, policy-adherent content in JS, AS, and mixed AS-JS ads. We performed two sets of experiments to test a cross-section of ads from various sources:

First, we deployed FlashJaX with ad scripts from four popular ad networks: Google AdSense, Yahoo! Network, Microsoft Media Network, and Clicksor. The first three of these were among the top 15 networks in U.S. market reach in April 2012, with market reach of 92.2%, 80.3%, and 76.9%, respectively [33]. We ran these ads with and without FlashJaX to observe their rendering results. All render correctly with no visible distinctions introduced by monitoring. No user interactions were visibly affected. This shows that our prototype can be deployed with real-world ad networks without loss of functionality.

Second, we tested our AS binary rewriter on 57 Flash ads harvested from browsing sessions on popular browsers over several weeks, intended to reflect ads observed by typical users. Of the 57 ads, 32 interact with JS using Flash's external interface to perform tasks such as cookie manipulation, pop-up creation, click tracking, or information exchange with JS-side ad network support code. Our AS rewriter successfully injected IRMs into all 57 samples.

6.3 Security

To evaluate FlashJaX's resilience against attacks, FlashJaX was deployed and tested against several malicious JS and AS programs. These include real-world attacks reported on CVE (http://cve.mitre.org), the attack scenarios introduced throughout the paper, and other attacks related to wrapper corruption, confidentiality, integrity, and ad-specific attacks. Each experiment was conducted by first running the malicious code without FlashJaX to verify that the attack is successful. Then the same script was run with FlashJaX to test whether it is blocked. The attacks and defenses are categorized and described below, and summarized in Table 1.

6.3.1 Real-world attacks

We studied two recent real-world attacks reported on CVE: CVE-2012-3414 ("XSS vulnerability in SWFUpload 2.2.0.1 and earlier") and CVE-2012-2904 ("XSS vulnerability in LongTail JW Player 5.9").

CVE-2012-3414 is a vulnerability in Wordpress 3.3.2 that allows reflected XSS via a Flash parameter derived from

user input. The attacker can inject arbitrary JS code by passing it to the applet as a malicious URL string, resulting in execution of the injected code at the privileges of the hosting page. However, with FlashJaX added to the content produced by WordPress, the attack fails. The JS code injected by the attacker is labeled and executed with the lower privilege of the untrusted Flash, disallowing attacker access to protected JS functions and page content.

CVE-2012-2904 is a vulnerability in LongTail JW Player 5.9, which is active on over one million web sites. Exploits inject script text as a parameter of the Flash, executing the payload at the privileges of the hosting page. FlashJaX, however, successfully labels the injected code with the untrusted Flash principal, causing it to execute at the lower (untrusted) privilege level and denying it access to publisher-protected resources. Thus, all prohibited JS operations in the payload are suppressed by the monitor, foiling the attack.

6.3.2 Simulated Attacks

Attack scenarios. We implemented and validated the policies discussed in §4.4. These policies address all three attack scenarios described in §2, preventing the attacks.

Wrapper Attacks. We implemented wrapper attacks identified by prior works [4], [7], [23], which defeat naïve JS wrapper implementations by abusing static aliases, dynamic aliases, and caller-chains. When successful, the attacks pop up an unmediated alert box. All the attacks failed since FlashJaX wraps these channels.

Script injection. FlashJaX does not prohibit script injections; it downgrades them to an untrusted privilege level so that they cannot perform policy-violating actions. We tested all script injection channels, including remote script files, script code, event handlers via `document.write`, `eval`, and script inclusion via `appendChild` and `insertBefore`. The experiments show that our principal-tracking mechanism attributes correct privileges to all the dynamically-generated code. We note here again that scripts with an unidentified principal run with lowest privileges, and therefore never violate any principal's policy.

Confidentiality and integrity attacks. These attacks steal or modify sensitive data of the hosting page, such as cookies and protected content. To evaluate these attacks, we deployed a web email page with a fine-grained access control policy that prohibits ads from reading the contact list or changing the email content. Ads that try to do so are successfully blocked by FlashJaX in the experiment.

Cookie protection. FlashJaX does not prohibit cookie access, but each principal may only read and write its own cookies. Malicious code that attempts to steal cookies belonging to another principal was evaluated and found to be unsuccessful.

Ad-specific attacks. We tested numerous attacks specific to web ads, including *clickjacking*, *oversized/arbitrary ad positioning*, and *resource abuse*. Each is described below.

Clickjacking attacks create an invisible `iframe` that injects a remote page with an invisible click button [34]. FlashJaX prevents this by enforcing a policy that disallows creating invisible `iframe`s.

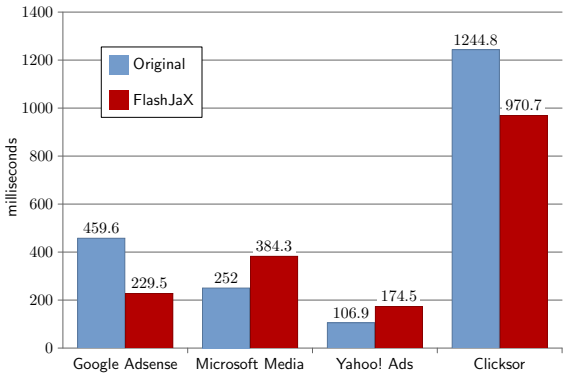

Fig. 6: Rendering overheads of unmonitored vs. FlashJaX-monitored ads.

Malicious ads often generate content that is larger than the maximum allowed by the ad network, or that is positioned inappropriately on the page (e.g., covering other content). These actions are prevented by FlashJaX by placing the ad in a fixed-sized ⟨`div`⟩ element whose size it write-protects. The policy additionally forces the offset of any ad-generated content to 0×0 and write-protects the offset, preventing the ad from popping up misplaced or mis-sized dynamic content.

In addition, we enforced other fine-grained policies that disallow or limit calls, and that filter call arguments to a whitelist of API methods that are frequently targets of resource abuse attacks. These include pop-up creators like `window.alert` and `window.open`. FlashJaX correctly prevented these resource abuse attacks.

6.4 Performance

Macro-benchmarks. Figure 6 evaluates the performance overhead by measuring the total render time to load pages with and without FlashJaX. The test machine is a 1.6 GHz AMD Athlon Neo MV-40 Processor laptop with 2 GB RAM running Chrome 19.0.1084.52m on Windows 7.

The observed rendering overhead varies widely based on the content from various ad networks. For Microsoft Media Network and Yahoo! Ads, the additional overhead is around 55%. However, for Google Adsense and Clicksor, we consistently observe rendering times that are actually faster with FlashJaX than without. We investigated this and found that Microsoft and Yahoo! generate Flash content, whereas Google and Clicksor generate `iframe`s that make heavy use of runtime-generated JS content. Our buffering of dynamic write operations (see §4.3) improves the performance of these dynamic writes, speeding rendering.

Micro-benchmarks. We additionally performed a set of microbenchmark experiments that measure the overhead of five monitored JS operations called from AS. Each test ran a tight loop for 1000 iterations, and we averaged the results over five trials. Reading and writing of JS properties (e.g., `document.cookie`) was tested using `eval`, since the AS-JS interface only supports JS method access.

Table 2 shows the slow-down ratio of the rewritten Flash without (column 2) and with (column 3) JS-side monitoring. The table shows a 3.07–4.47 times overhead for AS-JS boundary communications. This range compares favorably with similar microbenchmarks reported by related works

TABLE 2: FlashJaX micro-benchmarks measuring the ratio of the runtimes of FlashJaX-rewritten Flashes to originals without (column 2) and with (column 3) JS-side monitoring.

Operation	Rewritten Flash	FlashJaX
`document.appendChild`	3.52	3.59
`document.getElementById`	4.47	5.17
`toString()`	4.26	4.47
`eval("document.cookie='test'")`	3.67	5.91
`eval("document.cookie")`	3.07	6.33

(e.g., overheads of 1–324 times [35], or 0.09–19.54 times [3]). The inclusion of the JS IRM (column 3) results in a small additional overhead that is similar, except for the two eval tests. The overhead is higher for the eval tests because each iteration invokes the JS IRM twice (once for the `eval` and once for its content).

7 Related Work

Behavioral sandboxing. Our FlashJaX framework adopts a reference monitor approach, which monitors the behavior of web pages to detect and prevent attacks. There are a number of such methods in the recent literature [3], [5], [6], [7], [13], [36], [37], [38]. These works explore many subtle scenarios that arise when considering security issues in JS. For instance, JSand [36] isolates untrusted JS by loading it into a sandbox environment that can only interact with a virtual DOM. Thus, the policy definition and enforcement are implemented in a virtual DOM implementation. In contrast, FlashJaX keeps track of principals for untrusted scripts within a shadow stack in order to enforce an appropriate policy at runtime. FlashJaX can therefore handle JS script actions from AS while JSand cannot, since the latter requires full source codes of untrusted scripts. Virtual Browser (VB) [37] mediates third-party JS accesses to the browser via a virtual browser expressed in JS. The implementation is a variant of a security reference monitor. Unlike FlashJaX, VB does not support multi-principal or fine-grained policies for multi-party web applications, and does not support Flash content.

Isolating third-party content into (often invisible) iframes and providing a mechanism for cross-domain communication is an alternative approach to constraining untrusted scripts. Examples include Adjail [39], Webjail [40], and Subspace [41]. This technique is unsuitable for Flash content for performance reasons—transporting Flash content through browser-supported communication channels is prohibitively slow.

Configurable Origin Policy (COP) [42] is a recent proposal that allows web developers to associate web pages with a security principal via a configurable ID in the browser, so that web applications having a common ID are treated as same-origin even when hosted from different domains, such as `gmail.com` vs. `docs.google.com`. This clean-slate approach is a promising one in the design space of browser security. In contrast to a clean-slate approach such as COP, FlashJaX follows a design that is compatible with today's browsers and Flash interpreters. In general, since these methods only focus on the JS side, they cannot prevent attacks exploiting JS-AS interactions.

Similarly, there are several protection methods focusing on privacy and behavioral targeting, such as Privads [43], Adnostic [44], and RePriv [45], which address user privacy issues from behavioral targeting. These rely on specialized, in-browser systems that support contextual placement of ads while preventing behavioral profiling of users. In contrast, our work mainly focuses on a different, publisher-centric problem of protecting confidentiality and integrity of publisher and user-owned content. Our work is also aimed at providing compatibility with existing ad networks and browsers.

Restricting content languages. There have been a number of works in the area of JS analysis that restrict content from untrusted sources to provide security protections [9], [10], [11], [12], [14], [46]. These works focus on limiting the JS language features that untrusted scripts may use. Only those language features that are statically deterministic and amenable to analysis are allowed. Since these methods restrict content at a language level, they do not impose the runtime penalty of reference monitors. In the cases of FBJS [9] and ADsafe [47], untrusted scripts are confined to an access-controlled DOM interface, which incurs some overhead but affords additional control.

The disadvantage of a restricted JS subset is that many ads are unlikely to conform to it, and will therefore require re-development. In contrast, FlashJaX neither imposes the burden of new languages nor places restrictions on JS language features used in ad scripts. The only effort required from a publisher that incorporates FlashJaX is to specify policies that reflect site security practices.

Code transformation approaches. Many recent works have transformed untrusted JS code to interpose runtime policy enforcement checks [1], [2], [8], [35], [48], [49]. These works cover many diverse attack vectors by which third-party content may subvert the checks. Since these works are aimed at general JS security, they do not consider the security of the JS-AS interface and attacks that target this interface.

Browser-enforced protection. A modified browser can be instructed to enforce security policies, as illustrated by BEEP [50], CoreScript [51], End-to-End Web Application Security [52], Content Security Policies [53], and ConScript [4]. Other works, such as AdSentry [54], JCShadow [55], ESCUDO [56], and Tahoma [57], have taken this approach to prevent attacks by untrusted content. The main advantage of this approach is that it can enforce fine-grained policies with low overhead. However, the primary drawback is that today's browsers do not agree on a standard for publisher-browser collaboration, leaving a large void in the near-term for protecting users from malicious third-party content.

Safety of ActionScript content. Jang et al. [24] point out the pervasive nature of misconfigured AS content, particularly with reference to cross-domain policies. Ford et al. [58] describe a malware identification approach for Flash ads.

The work that is closest to ours is FIRM [15], which uses an IRM approach for prevention of Flash-related attacks. FIRM is strictly limited to AS mediation, whereas FlashJaX tackles a much broader class, that of mixed AS-JS content. As a result, our monitor is able to address a much broader class of attack vectors that target JavaScript as well as ActionScript (as discussed in §2), especially those that exploit the interface

boundary. Since FIRM focuses purely on AS-side monitoring, it adopts a less conservative threat model that assumes that some parts of the JS namespace can be read-protected from adversaries. This relaxed model admits a capability-based approach, which FIRM implements using secret tokens that are maintained by the reference monitor. In contrast, FlashJaX's threat model acknowledges that protection of secrets in a JS environment is hard. There are many different ways through which an attacker can get *read* access to the JS namespace (cf., [59]) in order to gain access to secret tokens. We therefore conservatively assume that adversaries may have the ability to read the complete JS namespace, and therefore developed a more robust approach whose security is argued in §5.

8 Discussion

This section discusses the relevance of FlashJaX to the larger landscape of web application security.

Context and Relevance. A plethora of threats are faced by web applications today; the most common include script injection attacks, heap spraying, drive-by downloads, UI spoofing, and clickjacking. Extensive research in both server- and browser-side defenses seek to mitigate these threats. The work presented in this paper exposes a relatively unexplored threat vector (compared to the threats mentioned above, which have been well explored).

We have also developed a systematic defense for this threat using the principled approach of IRMs. Our work can be seen as a defense that sits in conjunction with existing browser defenses, including those for JS (e.g., [4]), XSS attacks, and heap-spraying. FlashJaX strengthens those defenses by adding protection against a significant attack vector that these defenses do not address.

Other related browser plugins. Recent surveys indicate that Flash is the most commonly used browser plugin.[4] FlashJaX provides a systematic way to enforce security on Flash-JS content. Similar content can be authored in other plugins, such as Java and Silverlight. Our work could be extended to Silverlight via similar IRM-based techniques that have been used for .NET binary rewriting [60].

Deployment. Research efforts such as FlashJaX point out that the nature of attack surfaces will continue to evolve as browsers evolve to support new features. As a result, the nature of policies that security engineers want to enforce is continuously evolving as well, and there will always be a need to enforce policies that current browsers do not universally support. FlashJaX's approach to security through IRM enforcement allows for a principled defense mechanism that can be flexibly adapted to address future threats, while remaining compatible with existing browsers.

9 Conclusion

In this paper, we presented FlashJaX, a solution for enforcing security policies on third-party mixed JS/AS web content using an IRM approach. FlashJaX allows publishers to define and enforce fine-grained, multi-principal access policies on JS-AS third party content and runtime-generated code. Moreover, it can be easily deployed in practice without requiring browser modification. Experiments show that FlashJaX is effective in preventing attacks related to AS-JS communication, and its lightweight IRM approach exhibits low overhead for mediations. It is also compatible with advertisements from leading ad networks.

4. http://www.statowl.com/plugin_overview.php

Acknowledgments

This research was supported in part by NSF grants 1065134, 1065216, and 1054629, 1069311, 1065537 and by an international postdoc grant from the Swedish Research Council (VR).

References

[1] Google Caja, "Compiler for making third-party HTML, CSS, and JavaScript safe for embedding," http://code.google.com/p/google-caja, 2007.
[2] Microsoft Live Labs, "Microsoft web sandbox," http://websandbox.livelabs.com, 2009.
[3] P. H. Phung, D. Sands, and A. Chudnov, "Lightweight self-protecting JavaScript," in *Proc. ACM Sym. Information, Computer and Communications Security*, 2009, pp. 47–60.
[4] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proc. IEEE Sym. Security & Privacy*, 2010, pp. 481–496.
[5] T. V. Cutsem and M. S. Miller, "Proxies: Design principles for robust object-oriented intercession APIs," in *Proc. Dynamic Languages Sym.*, 2010, pp. 59–72.
[6] M. Heiderich, T. Frosch, and T. Holz, "IceShield: Detection and mitigation of malicious websites with a frozen DOM," in *Proc. Int. Sym. Recent Advances in Intrusion Detection*, 2011.
[7] L. A. Meyerovich, A. P. Felt, and M. S. Miller, "Object views: Fine-grained sharing in browsers," in *Proc. Int. Conf. World Wide Web*, 2010, pp. 721–730.
[8] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *Proc. SIGACT-SIGPLAN Sym. Principles of Programming Languages*, 2007, pp. 237–249.
[9] Facebook Developers, "JavaScript SDK," http://developers.facebook.com/docs/reference/javascript, 2010.
[10] S. Maffeis and A. Taly, "Language-based isolation of untrusted JavaScript," in *Proc. IEEE Computer Security Foundations Sym.*, 2009, pp. 77–91.
[11] S. Maffeis, J. C. Mitchell, and A. Taly, "Run-time enforcement of secure JavaScript subsets," in *Proc. IEEE Work. Web 2.0 Security & Privacy*, 2009.
[12] M. Finifter, J. Weinberger, and A. Barth, "Preventing capability leaks in secure JavaScript subsets," in *Proc. Network and Distributed System Security Sym.*, 2010.
[13] A. Taly, Ú. Erlingsson, J. C. Mitchell, M. S. Miller, and J. Nagra, "Automated analysis of security-critical JavaScript APIs," in *Proc. IEEE Sym. Security & Privacy*, 2011, pp. 363–378.
[14] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi, "ADsafety: Type-based verification of JavaScript sandboxing," in *Proc. USENIX Security Sym.*, 2011.
[15] Z. Li and X. Wang, "FIRM: Capability-based inline mediation of Flash behaviors," in *Proc. Annual Computer Security Applications Conf.*, 2010, pp. 181–190.
[16] M86 Security, "Security labs report: January–June 2010 recap," M86 Security, Tech. Rep., July 2010.
[17] Y. Amit, "Cross-site scripting through Flash in Gmail based services," http://blog.watchfire.com/wfblog/2010/03/cross-site-scripting-through-flash-in-gmail-based-services.html, Mar. 2010.
[18] S. V. Acker, N. Nikiforakis, L. Desmet, W. Joosen, and F. Piessens, "FlashOver: Automated discovery of cross-site scripting vulnerabilities in rich internet applications," in *Proc. ACM Sym. Information, Computer and Communications Security*, 2012, pp. 12–13.
[19] M. Zalewski, "Browser security handbook," http://code.google.com/p/browsersec/, 2011.
[20] J. Wolf, "Heap spraying with ActionScript: Why turning off JavaScript won't help this time," FireEye Malware Intelligence Lab, July 2009, http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html.

[21] E. Elrom, "Top security threats to Flash/Flex applications and how to avoid them," http://www.slideshare.net/eladnyc/top-security-threats-to-flashflex-applications-and-how-to-avoid-them-4873308, July 2010.
[22] F. B. Schneider, "Enforceable security policies," *ACM Trans. Information and System Security*, vol. 3, no. 1, pp. 30–50, 2000.
[23] J. Magazinius, P. H. Phung, and D. Sands, "Safe wrappers and sane policies for self protecting JavaScript," in *Proc. Nordic Conf. Secure IT Systems*, 2010, pp. 239–255.
[24] D. Jang, A. Venkataraman, G. M. Sawka, and H. Shacham, "Analyzing the cross-domain policies of Flash applications," in *Proc. IEEE Work. Web 2.0 Security & Privacy*, 2011.
[25] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987.
[26] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: Enforcement mechanisms for run-time security policies," *Int. J. Information Security*, vol. 4, no. 1–2, pp. 2–16, 2005.
[27] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Computability classes for enforcement mechanisms," *ACM Trans. Programming Languages and Systems*, vol. 28, no. 1, pp. 175–205, 2006.
[28] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in *Proc. ACM Work. Programming Languages and Analysis for Security*, 2008, pp. 11–20.
[29] ECMA International, *ECMAScript Language Specification (ECMA-262)*, 5th ed. Geneva, Switzerland: ECMA, June 2011.
[30] G. Jähnig, "fsmjs: A finite state machine library in JavaScript," http://code.google.com/p/fsmjs, 2010.
[31] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc. IEEE Sym. Security & Privacy*, 2008, pp. 187–201.
[32] F. Martinell, "Through modeling to synthesis of security automata," in *Proc. Int. Work. Security and Trust Management*, 2006, pp. 31–46.
[33] comScore, "comScore media metrix ranks top 50 U.S. web properties for April 2012," http://www.comscore.com/Press_Events/Press_Releases/2012/5/comScore_Media_Metrix_Ranks_Top_50_U.S._Web_Properties_for_April_2012, April 2012.
[34] R. Hansen and J. Grossman, "Clickjacking," *SecTheory Internet Security*, September 2008.
[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," in *Proc. USENIX Sym. Operating Systems Design and Implementation*, 2006, pp. 61–74.
[36] P. Agten, S. V. Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens, "JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications," in *Proc. Annual Computer Security Applications Conf.*, 2012, pp. 1–10.
[37] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual Browser: A virtualized browser to sandbox third-party JavaScripts with enhanced security," in *Proc. ACM Sym. Information, Computer and Communications Security*, 2012, pp. 8–9.
[38] P. H. Phung and L. Desmet, "A two-tier sandbox architecture for untrusted JavaScript," in *Proc. Work. JavaScript Tools*, 2012, pp. 1–10.
[39] M. T. Louw, K. T. Ganesh, and V. N. Venkatakrishnan, "AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements," in *Proc. USENIX Security Sym.*, 2010.
[40] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen, "WebJail: Least-privilege integration of third-party components in web mashups," in *Proc. Annual Computer Security Applications Conf.*, 2011, pp. 307–316.
[41] C. Jackson and H. J. Wang, "Subspace: Secure cross-domain communication for web mashups," in *Proc. Int. Conf. World Wide Web*, 2007, pp. 611–620.
[42] Y. Cao, V. Rastogi, Z. Li, Y. Chen, and A. Moshchuk, "Redefining web browser principals with a configurable origin policy," in *Proc. Annual IEEE/IFIP Int. Conf. Dependable Systems and Networks*, 2013, pp. 1–12.
[43] S. Guha, B. Cheng, A. Reznichenko, H. Haddadi, and P. Francis, "Privad: Rearchitecting online advertising for privacy," Max Planck Institute for Software Systems, Kaiserslautern-Saarbrücken, Germany, Tech. Rep. MPI-SWS-2009-004, October 2009.
[44] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas, "Adnostic: Privacy preserving targeted advertising," in *Proc. Network and Distributed System Security Sym.*, 2010.
[45] M. Fredrikson and B. Livshits, "RePriv: Re-imagining content personalization and in-browser privacy," in *Proc. IEEE Sym. Security & Privacy*, 2011, pp. 131–146.
[46] S. Guarnieri and B. Livshits, "GATEKEEPER: Mostly static enforcement of security and reliability policies for JavaScript code," in *Proc. USENIX Security Sym.*, 2009, pp. 151–168.
[47] D. Crockford, "ADsafe," http://www.adsafe.org, 2007.
[48] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "JavaScript instrumentation in practice," in *Proc. Asian Sym. Programming Languages And Systems*, 2008, pp. 326–341.
[49] Z. Li, T. Yi, Y. Cao, V. Rastogi, Y. Chen, B. Liu, and C. Sbisa, "WebShield: Enabling various web defense techniques without client side modifications," in *Proc. Network and Distributed System Security Sym.*, 2011.
[50] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *Proc. Int. Conf. World Wide Web*, 2007, pp. 601–610.
[51] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *Proc. SIGACT-SIGPLAN Sym. Principles of Programming Languages*, 2007, pp. 237–249.
[52] Ú. Erlingsson, B. Livshits, and Y. Xie, "End-to-end web application security," in *Proc. Work. Hot Topics in Operating Systems*, 2007.
[53] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proc. Int. Conf. World Wide Web*, 2010, pp. 921–930.
[54] X. Dong, M. Tran, Z. Liang, and X. Jiang, "AdSentry: Comprehensive and flexible confinement of JavaScript-based advertisements," in *Proc. Annual Computer Security Applications Conf.*, 2011, pp. 297–306.
[55] K. Patil, X. Dong, X. Li, Z. Liang, and X. Jiang, "Towards fine-grained access control in JavaScript contexts," in *Proc. Int. Conf. Distributed Computing Systems*, 2011, pp. 720–729.
[56] K. Jayaraman, W. Du, B. Rajagopalan, and S. J. Chapin, "ESCUDO: A fine-grained protection model for web browsers," in *Proc. IEEE Int. Conf. Distributed Computing Systems*, 2010, pp. 231–240.
[57] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen, "A safety-oriented platform for web applications," in *Proc. IEEE Sym. Security & Privacy*, 2006, pp. 350–364.
[58] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and detecting malicious Flash advertisements," in *Proc. Annual Computer Security Applications Conf.*, 2009, pp. 363–372.
[59] S. Maffeis, J. C. Mitchell, and A. Taly, "Isolating JavaScript with filters, rewriting, and wrappers," in *Proc. European Conf. Research in Computer Security*, 2009, pp. 505–522.
[60] K. W. Hamlen, G. Morrisett, and F. B. Schneider, "Certified in-lined reference monitoring on .NET," in *Proc. ACM SIGPLAN Work. Programming Languages and Analysis for Security*, 2006, pp. 7–16.

Dr. Phu H. Phung is a Research Associate at the University of Illinois at Chicago, employed by the University of Gothenburg, Sweden. His research directions include web application security, runtime policy enforcement for untrusted software, and policy enforcement for cloud-based sustainability governance platforms.

Maliheh Monshizadeh is a PhD student at the Department of Computer Science, University of Illinois at Chicago. Her research interests lie in Web application security, vulnerability analysis, and programming languages.

Dr. Meera Sridhar is an Assistant Professor in the Department of Software and Information Systems at The University of North Carolina at Charlotte. She received her BS and MS degrees in Computer Science from Carnegie Mellon University, and her PhD in Computer Science from The University of Texas at Dallas. Her research interests span language-based and systems security, and formal methods.

Dr. Kevin W. Hamlen is an Associate Professor of Computer Science at The University of Texas of Dallas whose research applies programming language and compiler theory to software security problems. He received his BS degree in mathematics and computer science from Carnegie Mellon, and his MS and PhD in computer science from Cornell University.

Dr. V.N. "Venkat" Venkatakrishnan's research encompasses several aspects of software security for web, mobile and desktop platforms. His specific interests include vulnerability analysis, attack prevention and damage mitigation. He is currently an Associate Professor of Computer Science at the University of Illinois at Chicago.