Spectator: Detection and Containment of JavaScript Worms

Benjamin Livshits *Weidong Cui*

Microsoft Research

Abstract

Recent popularity of interactive AJAX-based Web 2.0 applications has given rise to a new breed of security threats: JavaScript worms. In this paper we propose Spectator, the first automatic detection and containment solution for JavaScript worms. Spectator performs distributed data tainting by observing and tagging the traffic between the browser and the Web application. When a piece of data propagates too far, a worm is reported. To prevent worm propagation, subsequent upload attempts performed by the same worm are blocked. Spectator is able to detect fast and slow moving, monomorphic and polymorphic worms with a low rate of false positives. In addition to our detection and containment solution, we propose a range of deployment models for Spectator, ranging from simple intranet-wide deployments to a scalable load-balancing scheme appropriate for large Web sites.

In this paper we demonstrate the effectiveness and efficiency of Spectator through both large-scale simulations as well as a case study that observes the behavior of a real-life JavaScript worm propagating across a social networking site. Based on our case study, we believe that Spectator is able to detect all JavaScript worms released to date while maintaining a low detection overhead for a range of workloads.

1 Introduction

Web applications have been a prime target for application-level security attacks for several years. A number of attack techniques, including SQL injections, cross-site scripting, path traversal, cross-site request forgery, HTTP splitting, etc. have emerged, and recent surveys have shown that the majority of Web sites in common use contain at least one Web application security vulnerability [38, 42]. In fact, in the last several years, Web application vulnerabilities have become significantly more common than vulnerabilities enabled by unsafe programming languages such as buffer overruns and format string violations [39].

While Web application vulnerabilities have been around for some time and a range of solutions have been developed [15, 17, 20, 22, 24, 29, 44], the recent popularity of interactive AJAX-based Web 2.0 applications has given rise to a new and considerably more destructive breed of security threats: JavaScript worms [11, 13]. JavaScript worms are enabled by cross-site scripting vulnerabilities in Web applications. While cross-site scripting vulnerabilities have been a common problem in Web based-applications for some time, their threat is now significantly amplified with the advent of AJAX technology. AJAX allows HTTP requests to be issued by the browser on behalf of the user. It is no longer necessary to trick the user into clicking on a link, as the appropriate HTTP request to the server can just be manufactured by the worm at runtime. This functionality can and has been cleverly exploited by hackers to create self-propagating JavaScript malware.

1.1 The Samy Worm

The first and probably the most infamous JavaScript worm is the Samy worm released on MySpace.com, a social networking site in 2005 [35]. By exploiting a cross-site scripting vulnerability in the MySpace site, the worm added close to a million users to the worm author's "friends" list. According to MySpace site maintainers, the worm caused an explosion in the number of entries in the friends list across the site, eventually leading to resource exhaustion. Two days after the attack the site was still struggling to serve requests at a normal pace.

The Samy worm gets its name from the MySpace login of its creator. Initially, the malicious piece of JavaScript (referred to as the *payload*) was manually placed in Samy's own MySpace profile page, making it infected. Each round of subsequent worm propagation consists of the following two steps:

1. **Download**: A visitor downloads an infected profile and automatically executes the JavaScript payload. This adds Samy as the viewer's "friend" and also adds the text *but most of all, samy is my hero* to the viewer's profile. Normally, this series of steps would be done through `GET` and `POST` HTTP requests manually performed by the user by clicking on various links and buttons embedded in MySpace pages. In this case, all of these steps are done in the background without the viewer's knowledge.
2. **Propagation**: The payload is extracted from the contents of the profile being viewed and then added to the viewer's profile.

Note that the enabling characteristic of a JavaScript worm is the AJAX propagation step: unlike "old-style" Web applications, AJAX allows requests to the server to be done in the background without user's knowledge. Without AJAX, a worm such as Samy would be nearly impossible. Also observe that worm propagation happens among properly authenticated MySpace users because only authenticated users have the ability to save the payload in their profiles.

1.2 Overview of the Problem

While Samy is a relatively benign proof-of-concept worm, the impact of JavaScript worms is likely to grow in the future. There are some signs pointing to that already: another MySpace worm released in December 2006 steals user passwords by replacing links on user's profile site with spoofed HTML made to appear like login forms [6]. The stolen credentials were subsequently hijacked for the purpose of sending spam. Similarly, Yamanner, a recent Yahoo! Mail worm, propagated through the Webmail system affecting close to 200,000 users by sending emails with embedded JavaScript to everyone in the current user's address book [4]. Harvested emails were then transmitted to a remote server to be used for spamming. Additional information on eight JavaScript worms detected in the wild so far is summarized in our technical report [21]. Interested readers are also referred to original vulnerability reports [4–6, 26, 33–35].

The impact of JavaScript worms will likely increase if attackers shift their attention to sites such as `ebay.com`, `epinions.com`, `buy.com`, or `amazon.com`, all of which provide community features such as submitting product or retailer reviews. The financial impact of stolen credentials in such a case could be much greater than it was for MySpace, especially if vulnerability identification is done with the aid of cross-site scripting vulnerability cataloging sites such as `xssed.com` [30]. Today cross-site scripting vulnerabilities are routinely exploited to allow the attacker to steal the credentials of a small group of users for financial gain. Self-propagating code amplifies this problem far beyond its current scale. It is therefore important to develop a detection scheme for JavaScript worms before they become commonplace.

A comprehensive detection solution for JavaScript worms presents a tough challenge. The server-side Web application has no way of distinguishing a benign HTTP request performed by a user from one that is performed by a worm using AJAX. An attractive alternative to server-side detection would be to have an entirely client-side solution. Similarly, however, the browser has no way of distinguishing the origin of a piece of JavaScript: benign JavaScript embedded in a page for reasons of functionality is treated the same way as the payload of a worm. Filtering solutions proposed so far that rely on worm signatures to stop their propagation [37] are ineffective when it comes to polymorphic or obfuscated payloads, which are easy to create in JavaScript; in fact many worms detected so far are indeed obfuscated. Moreover, overly strict filters may cause false positives, leading to user frustration if they are unable to access their own data on a popular Web site.

1.3 Paper Contributions

This paper describes Spectator, a system for detecting and containing JavaScript worms, and makes the following contributions:

- Spectator is the first practical solution to the problem of detecting and containment of JavaScript worms. Spectator is also insensitive to the worm propagation speed; it can deal with rapid zero-day worm attacks as well as worms that disguise their presence with slow propagation. Spectator is insensitive of what the JavaScript code looks like and does not rely on signatures of any sort; therefore it is able to detect polymorphic worms or worms that use other executable content such as VBScript or JavaScript embedded in Flash or other executable content.
- We propose a scalable detection solution that adds a small constant-time overhead to the end-to-end latency of an HTTP request no matter how many requests have been considered by Spectator. With this detection approach, Spectator is able to detect all worms that have been found in the wild thus far.
- Our low-overhead approximate detection algorithm is mostly conservative, meaning that for the majority of practical workloads it will not miss a worm if there is one, although false positives may be possible. However, simulations we have performed show that false positives are unlikely with our detection scheme.

- We propose multiple deployment models for Spectator: the Spectator proxy can be used as a server-side proxy or as a browser proxy running in front of a large client base such as a large Intranet site. For large services such as MySpace, we describe how Spectator can be deployed in a load-balanced setting. Load balancing enables Spectator to store historical data going far back without running out of space and also improves the Spectator throughput.
- We evaluate Spectator is several settings, including a large-scale simulation setup as well as a real-life case study using a JavaScript worm that we developed for a popular open-source social networking application deployed in a controlled environment.

1.4 Paper Organization

The rest of the paper is organized as follows. Section 2 describes the overall architecture of Spectator. We formally describe our worm detection algorithm and Spectator implementation in Sections 3 and 4, respectively. Section 5 describes the experiments and case studies we performed. Section 6 discusses Spectator design choices, tradeoffs, and threats to the validity of our approach. Finally, Sections 7 and 8 summarize related work and provide our conclusions.

2 Spectator Design Overview

This section provides an overview of Spectator architecture and design assumptions. Section 3 gives a formal description of our worm detection algorithm.

2.1 Spectator Overview

A recent study concluded that over 90% of Web applications are vulnerable to some form of security attack, including 80% vulnerable to cross-site scripting [42]. Cross-site scripting, which is at the root of JavaScript worms, is commonly identified as the most prevalent Web application vulnerability.

While it is widely recognized that secure programming is the best defense against application-level vulnerabilities, developing fully secure applications remains a difficult challenge in practice. For example, while MySpace was doing a pretty good job filtering well-formed JavaScript, it failed to filter out instances of `java\nscript`, which are interpreted as legal script in Internet Explorer and some versions of Safari. Despite best intentions, insecure applications inevitably get deployed on widely used Web sites.

The goal of Spectator is to protect Web site users from the adverse effects of worm propagation *after* the server has failed to discover or patch a vulnerability in a timely manner. The essence of the Spectator approach is to *tag* or mark HTTP requests and responses so that copying of the content across a range of pages in a worm-like manner can be detected. Note that JavaScript worms are radically different from "regular" worms in that they are centralized: they typically affect a single Web site or a small group of sites (the same-origin policy of JavaScript makes it difficult to develop worms that propagate across multiple servers).

Spectator consists of an HTTP proxy inspecting the traffic between the user's browser and a Web server in order to detect malicious patterns of JavaScript code propagation. Our tagging scheme described in Section 4 is a form of distributed tainting: whenever content that contains HTML is uploaded to the server, Spectator modifies it to attach a tag invisible to the end-user. The tag is preserved on the server and is contained in the HTML downloaded by subsequent requests. Spectator injects client-side support so that tags are reliably propagated on the client side and cannot be removed by worms aware of our tagging scheme. Client-side support relies on HTTP-only cookies and does not require specialized plug-ins or browser modifications, thus removing the barrier to client-side adoption.

Worm detection at the Spectator proxy works by looking for long propagation chains. Our detection algorithm is designed to scale to propagation graphs consisting of thousands of nodes with minimal overhead on every request. Whenever a long propagation chain is detected, Spectator disallows further uploads that are caused by that chain, thereby containing further worm propagation.

The Spectator detection algorithm is designed to detect propagation activity that affects multiple users. With every HTML upload, we also record the IP address of the user issuing the request. The IP address is used as an approximation of user identity. We keep track of IP addresses so that a user repeatedly updating their profile is not flagged as worm. If multiple users share an IP address, such as users within an intranet, this may cause false negatives. If the same user connects from different IP addresses, false positives might result. Worm detection relies on sufficiently many users adopting Spectator. However, since Spectator relies on no additional client-side support, it can be deployed almost instantaneously to a multitude of users.

2.2 Spectator Architecture

To make the discussion above more concrete, a diagram of Spectator's architecture is shown in Figure 1. Whenever a user attempts to download a page containing Spectator tags previously injected there by Spectator, the following steps are taken, as shown in the figure:

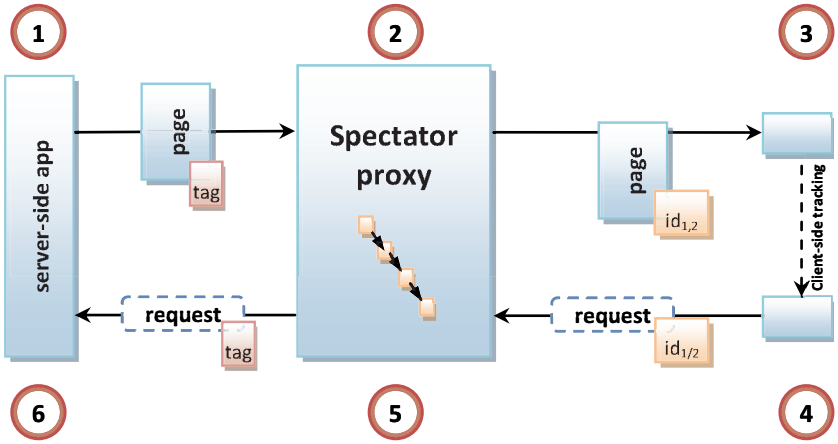

Figure 1: Spectator architecture

1. The tagged page is retrieved from the server.
2. The Spectator proxy examines the page. If the page contains tags, a new session ID is created and associated with the list of tags in the page. The tags are stripped from the page and are never seen by the browser or any malicious content executing therein.
3. The modified page augmented with the session ID stored in a cookie (referred to below as "Spectator cookie") is passed to the browser.

Whenever an upload containing HTML is observed, the following steps are taken:

4. If a Spectator cookie is found on the client, it is automatically sent to Spectator by the browser (the cookie is the result of a previous download in step 3).
5. If the request has HTTP content, a new *tag*—a number uniquely identifying the upload—is created by the Spectator proxy. If the request has a valid session ID contained in a Spectator cookie attached to the request, the list of tags it corresponds to is looked up and, for every tag, an edge between the old and the new tags are added to the propagation graph to represent tag causality. The request is not propagated further if the detection algorithm decides that the request is part of worm propagation.
6. Finally, the request augmented with the newly created tag is uploaded and stored at the server.

The Spectator worm detection algorithm relies on the following properties that guarantee that we can observe and record the propagation of a piece of data during its entire "round trip", captured by steps 1–6 above, thereby enabling taint tracking. The properties described below give the information required to formally reason about the Spectator algorithm. A detailed discussion of how Spectator ensures that these properties hold is delayed until Section 4.

Property 1: Reliable HTML input detection. *We can detect user input that may contain HTML and mark it as tainted. Additionally, we can mark suspicious user input without disturbing server-side application logic so that the mark propagates to the user.*

Property 2: Reliable client-side tag propagation. *Browser can propagate taint tags from an HTTP response to a subsequent request issued by the browser.*

3 Worm Detection Algorithm

This section describes our worm detection algorithm. Section 3.1 formalizes the notion of a worm and Section 3.2 talks about our detection algorithm. Finally, Section 3.3 discusses worm containment.

3.1 Propagation Graph Representation

We introduce the notion of a *propagation graph* that is updated whenever new tags are inserted. Each node of the graph corresponds to a tag and edges represent causality edges. Each node carries with it the IP address of the client the tag originates from.

Definition 1. *A* tag *is a long integer uniquely identifying an HTML upload to Spectator.*

Definition 2. *A* causality edge *is a tuple of tag-IP address pairs* $\langle (t_1, ip_1), (t_2, ip_2) \rangle$ *representing the fact that* t_2 *requested by* ip_2 *originated from a page requested by* ip_1 *that has* t_1 *associated with it.*

Definition 3. *Propagation graph* $G = \langle \mathcal{V}, \mathcal{E} \rangle$, *where vertices* $\mathcal{V}$ *is a set of tag-IP pairs*

$\{(t_1, ip_1), (t_2, ip_2), \dots\}$ *and* $\mathcal{E}$ *is the set of causality edges between them.*

Definition 4. *The* distance *between two nodes* N_1 *and* N_2*, denoted as* $|N_1, N_2|$*, in a propagation graph* G *is the smallest number of unique IP addresses on any path connecting* N_1 *and* N_2*.*

Definition 5. Diameter *of a propagation graph* G*, denoted as* $\mathcal{D}(G)$*, is the maximum distance between any two nodes in* G*.*

Definition 6. *We say that* G contains a worm *if* $\mathcal{D}(G)$ *exceeds a user-provided threshold* d*.*

Note that the propagation graph is acyclic. While it is possible to have node sharing, caused by a page with two tags generating a new onehaving a cycle in the propagation graph is impossible, as it would indicate a tag caused by another one that was created chronologically later. Ideally, we want to perform worm detection on the fly, whenever a new upload request is observed by Spectator. When a new edge is added to the propagation graph G, we check to see if the diameter of updated graph G now exceeds the user-defined threshold d.

The issue that complicates the design of an efficient algorithm is that we need to keep track of the set of unique IP addresses encountered on the current path from a root of the DAG. Unfortunately, computing this set every time an edge is added is exponential in the graph size in the worst case. Storing the smallest set of unique IP addresses at every node requires $O(n^2)$ space in the worst case: consider the case of a singly-linked list where every node has a different IP address. Even if we store these sets at every node, the computation of the IP address list at a node that has more than one predecessor still requires an exponential amount of work, as we need to consider all ways to traverse the graph to find the path with the smallest number of unique IP addresses. Our goal is to have a worm detection algorithm that is as efficient as possible. Since we want to be able to detect slow-propagating worms, we cannot afford to remove old tags from the propagation graph. Therefore, the algorithm has to scale to hundreds of thousands of nodes, representing tags inserted over a period of days or weeks.

3.2 Incremental Approximate Algorithm

In this section we describe an iterative algorithm for detecting when a newly added propagation graph edge indicates the propagation of a worm. As we will demonstrate later, the approximation algorithm is mostly conservative, meaning that if there is a worm, in most cases, the approximation approach will detect it *no later* than the precise one.

3.2.1 Data Representation

The graph G_A maintained by our algorithm is a forest approximating the propagation graph G. Whenever node sharing is introduced, one of the predecessors is removed to maintain the single-parent property. Furthermore, to make the insertion algorithm more efficient, some of the nodes of the graph are designated as *storage stations*; storage stations accelerate the insertion operation in practice by allowing to "hop" towards a root of the forest without visiting every node on the path.

We use the following representation for our approximate algorithm. $PREV(N)$ points to the nearest storage station on its path to the root or `null` if N is the root. Every node N in G_A has a set of IP addresses $IPS(N)$ associated with it. The number of IP addresses stored at a node is at most c, where c is a user-configured parameter. At every node N we maintain a depth value denoted as $DEPTH(N)$, which is an approximation of the number of unique IP addresses on the path from N to the root. Whenever the $DEPTH$ value exceeds the user-defined threshold d, we raise an alarm.

3.2.2 Worm Detection

For space reasons, detailed pseudo-code for the insertion algorithm that describes the details of data structure manipulation is given in our technical report [21]. Here we summarize the essence of the insertion algorithm. Whenever a new causality edge from node $parent$ to node $child$ is added to G_A:

1. If $parent$ is the only predecessor of $child$ in G_A, we walk up the tree branch and find all storage stations on the current tree branch. We copy $IPS(parent)$ into $IPS(child)$ and then add $child$'s IP if it is not found by the search. In the latter case, $DEPTH(child)$ value is incremented. If the size of $IPS(child)$ reaches threshold c, we designate $child$ as a storage station.
2. If $child$ has two predecessors in G_A, we compare $DEPTH$ values stored at the two predecessors, select the larger one, and remove the other edge from the graph, restoring non-sharing. After that we follow step 1 above. Note that the predecessors do not have to belong to the same tree. However, after the insertion is complete, $child$ will be a member of a single tree.

Observe that the the *maximum* $DEPTH$ value computed by this algorithm is exactly $\mathcal{D}(G_A)$ because the maximum distance in G_A is that between a node and a root.

Notice that the approach described in this section is essentially a greedy algorithm: in the presence of multiple parents, it chooses the parent that it believes will result in

$$IPS(N) = \begin{cases} \text{IP addresses on the path from } N \text{ to } PREV(N) \textit{ not contained} \text{ in any other } IPS \text{ sets of nodes between } N \text{ and the root} & \textit{if } PREV(N) \neq \texttt{null} \\ \text{IP addresses on the path from } N \text{ to the root} & \textit{if } PREV(N) = \texttt{null} \end{cases}$$

Figure 2: Definition of IPS.

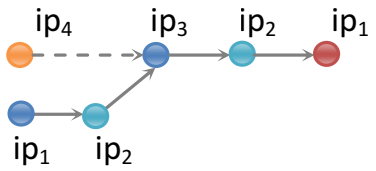

Figure 3: Propagation graph for which the approximate algorithm *under*-approximates the diameter value

higher overall diameter of the final approximation graph G_A. Of course, the advantage of the approximate algorithm is that it avoids the worst case exponential blow-up. However, without the benefit of knowing future insertion operations, the greedy algorithm may yield a lower diameter, potentially leading to false negatives. While this has never happened in our experiments, one such example is described below.

Example 1. Consider the propagation graph displayed in Figure 3. Suppose we first insert the two nodes on the bottom left with IPs ip_1 and ip_2 and then the node with ip_4. When we add ip_3 to the graph, the approximation algorithm will decide to remove the newly created edge (showed as dashed) because doing so will result in a greater diameter. However, the greedy algorithm makes a *suboptimal* decision: when nodes on the right with IPs ip_2 and ip_1 are added, the resulting diameter will be 3, not 4 as it would be with the precise approach. □

3.2.3 Incremental Algorithm Complexity

Maintaining an approximation allows us to obtain a very modest time and space bounds on new edge insertion, as shown below. Discussion of how our approximate detection algorithm performs in practice is postponed until Section 5.

Insertion Time Complexity. The complexity of the algorithm at every insertion is as follows: for a graph G_A with n nodes, we consider d/c storage stations at the most. Since storage stations having non-overlapping lists of IP addresses, having *more* storage stations on a path from a root of G_A would mean that we have over d IPs in total on that particular path, which should have been detected as a worm. At every storage station, we perform an $O(1)$ average time containment check. So, as a result, our approximate insertion algorithm takes $O(1)$ time on average.

Space Complexity. We store $O(n)$ IP addresses at the storage stations distributed throughout the propagation graph G_A. This is easy to see in the worst case of every IP address in the graph being unique. The union of all IP lists stored at all storage stations will be the set of all graph nodes. Additionally, we store IP addresses at the nodes *between* subsequent storage stations. In the worst case, every storage station contains c nodes and we store $1 + 2 + \cdots + c - 1 = c \cdot (c-1)/2$ IP addresses, which preserves the total space requirement of $O(n)$. More precisely, with at most n/c storage stations, we store approximately

$$\frac{c^2}{2} \times \frac{n}{c} = \frac{1}{2} \cdot c \cdot n$$

IP addresses. Note that in practice storage stations allow insertion operations to run faster because instead of visiting every node on the path from the root, we can instead "hop" to the next storage station, as demonstrated by the d/c bound. However, using storage stations also results in more storage space being taken up as shown by the $1/2 \cdot c \cdot n$ bound. Adjusting parameter c allows us to explore this space-time trade-off: bigger c results in faster insertion times, but also requires more storage.

Worm Containment Complexity. When a worm is detected, we walk the tree that the worm has infected and mark all of its nodes as such. This takes $O(n)$ time because in the worst case we have to visit and mark all nodes in the tree. The same bound holds for when we mark nodes in a tree as false positives.

3.3 Worm Containment

Whenever the depth of the newly added node exceeds detection threshold d, we mark the entire tree containing the new edge as infected. To do so, we maintain an additional status at every leaf. Whenever a tree is deemed infected by our algorithm, we propagate the infected status to every tree node. Subsequently, all uploads that are caused by nodes within that tree are disallowed until there is a message from the server saying that it is safe to do so.

When the server fixes the vulnerability that makes the worm possible, it needs to notify the Spectator proxy, at which point the proxy will remove the entire tree containing the new edge from the proxy. If the server deems the

vulnerability reported by Spectator to be a false positive, we never subsequently report activity caused by nodes in this tree as a worm. To do so, we set the node status for each tree node as a false positive and check the node status before reporting a worm.

4 Spectator Implementation

Distributed tainting in Spectator is accomplished by augmenting both upload requests to insert tracking tags and download requests to inject tracking cookies and JavaScript.

4.1 Tag Propagation in the Browser

To track content propagation on the client side, the Spectator proxy maintains a local *session* for every page that passes through it. Ideally, this functionality would be supported by the browser natively; in fact, if browsers supported *per-page cookies*, that is, cookies that expire once the current page is unloaded, this would be enough to precisely track causality on the client side. Since such cookies are not supported, we use a combination of standard per-session browser cookies and injected JavaScript that runs whenever the current page is unloaded to accomplish the same goal.

4.1.1 Client-Side Causality Tracking

Whenever a new page is sent by the Spectator proxy to the browser, a new *session tuple* $\langle id_1, id_2 \rangle$ is generated, consisting of two long integer values, which are randomized 128-bit integers, whose values cannot be easily guessed by the attacker. Our client-side support consists of two parts:

HTTP-only Spectator cookie in the browser. We augment every server response passing through Spectator with an HTTP-only cookie containing id_1. The fact that the session ID is contained in an HTTP-only cookie means that it cannot be snooped on by malicious JavaScript running within the browser, assuming the browser correctly implements the HTTP-only attribute. For a page originating from server D, the domain of the session ID cookie is set to D, so it is passed back to Spectator on every request to D, allowing us to perform causality tracking as described above.

Ideally, we would like to have a per-page cookie that expires as soon as the page is unloaded. Unfortunately, there is no support for such cookies. So, we use session cookies that expire after the browser is closed, which may not happen for a while. So, if the user visits site D served by Spectator, then visits site E, and then returns to D, the Spectator cookie would still be sent to Spectator by the browser.

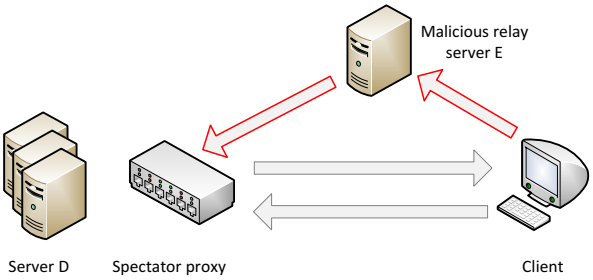

Figure 4: Relaying user requests through a malicious server

Injected client-side JavaScript to signal page unloads. In order to terminate a propagation link that would be created between the two unrelated requests to server D, we inject client-side JavaScript into every file that Spectator sends to the browser. Furthermore, before passing the page to the client, within Spectator we add an `unload` event handler, which sends an `XmlHttpRequest` to the special URL `__spectator__` to "close" or invalidate the current session, so that subsequent requests with the same id_1 are ignored. The `__spectator__` URL does not exist on server D: it is just a way to communicate with Spectator while including id_1 created for server D (notice that it is not necessary to pass the session ID as a parameter to Spectator, as the session ID cookie will be included in the request as well).

Injected client-side code is shown in Figure 5. To make it so that malicious JavaScript code cannot remove the `unload` handler, we mediate access to functions `window.attachEvent` and `window.detachEvent` as suggested in the BEEP system [16] by injecting the JavaScript shown in Figure 6 at the very top of each page served by Spectator. Furthermore, we also store id_2 is a private member of class `handler` [8]; this way it is not part of the `handler.unload` function source code and cannot be accessed with a call to `toString`. To prevent id_2 from being accessed through DOM traversal, the original `script` blocks defining the `unload` handler and containing the numerical value of id_2 embedded verbatim is subsequently removed through a call to `removeChild` in the next `script` block, similar to what is suggested by Meschkat [25].

4.1.2 Attacks Against Client-Side Tracking

While the basic client-side support is relatively simple to implement, there are two types of potential attacks against our client-side scheme to address, as described below.

Worm Relaying. First, the attacker might attempt to break a propagation chain by *losing* the session ID contained in the browser, a technique we refer to as *worm*

```
<script id="remove-me">
   if (window.attachEvent) {
      var handler = function(id) {
          var id2 = id;
          this.unload = function() {
                   var xhr = new ActiveXObject("MSXML2.XMLHTTP.3.0");
                   xhr.open("POST", "http://www.D.com/__spectator__&" + id2, true);
                   xhr.send(null); // send message to D just before unloading
               }
       };
       // embed id_2 verbatim and create an unload handler
       window.attachEvent("unload", (new handler(<id_2>)).unload);
   }
</script>

<script>
    // remove the previous script block from the DOM
    var script_block = document.getElementById("remove-me");
    script_block.getParentNode().removeChild(script_block);
</script>
```

Figure 5: Intercepting page `unload` events in JavaScript

relaying. Suppose we have a page in the browser loaded from server D. The attacker may programmatically direct the browser to a different server E, which would in turn connect to D. Server E in this attack might be set-up solely for the sole purpose of relaying requests to server D, as shown in Figure 4. Notice that since the session ID cookie will *not* be sent to E and its value cannot be examined. We introduce a simple restriction to make the Spectator proxy redirect all accesses to D that do *not* contain a session ID cookie to the top-level D URL such as `www.D.com`. In fact, it is quite common to disallow access, especially programmatic access through AJAX RPC calls, to an inside URL of large site such as `www.yahoo.com` by clients that do not already have an established cookie. With this restriction, E will be unable to relay requests on behalf of the user.

Tampering with `unload` events. To make it so that malicious JavaScript code cannot remove the `unload` handler or trigger its own unload handler after Spectator's, we mediate access to function `window.detachEvent` by injecting the JavaScript shown in Figure 6 at the very top of each page served by Spectator. If malicious script attempts to send the `unload` event to Spectator prematurely in an effort to break the propagation chain, we will receive more that one `unload` event per session. When a sufficient number of duplicate `unload` events is seen, we raise an alarm for the server, requiring a manual inspection. It is still possible for an attacker to try to cause false positives by making sure that the `unload` event will *never be sent*. This can be accomplished by crashing the browser by exploring a browser bug or trying to exhaust browser resources by opening new windows. However, this behavior is sufficiently conspicuous to the end-user to prompt a security investigation and is thus not a good candidate for inclusion within a worm.

```
<script>
window.attachEvent = function(sEvent, fpNotify) {
    if (sEvent == "unload") return;
    window.attachEvent(sEvent, fpNotify);
}

window.detachEvent = function(sEvent, fpNotify) {
    if (sEvent == "unload") return;
    window.detachEvent(sEvent, fpNotify);
}
</script>
```

Figure 6: Disallow adding or removing `unload` event handlers

Opening a new window. Note that opening a new window will not help an attacker break causality chains. If they try to perform a malicious upload before the `unload` event in the original window is sent to the proxy, Spectator will add a causality link for the upload. Fetching a new page with no tags before the malicious upload will not help an attacker evade Spectator because this clean page and the original page share the same HTTP-only cookie. As such, Spectator will think that the upload is caused by either of the sessions corresponding to that cookie. This is because Spectator's approximation approach selects the parent node with a larger depth when a node has multiple predecessors.

4.2 Tagging Upload Traffic and Server-Side Support for Spectator

The primary goal of server-side support is to embed Spectator tags into suspicious data uploaded to a protected Web server in a transparent and persistent manner so that (1) the tags will not interfere with the Web

server's application logic; and (2) the embedded tags will be propagated together with the data when the latter is requested from the Web server. To achieve these goals of transparency and persistence, we need to be able to reliably detect suspicious data to embed Spectator tags into uploaded input. Next, we discuss our solutions to the challenges of transparency and persistence that do *not* require any support on the part of the Web server.

Data uploads are suspicious if they may contain embedded JavaScript. However, for a cross-site scripting attack to be successful, this JavaScript is usually surrounded with some HTML tags. The basic idea of detecting suspicious data is to detect the *presence* of HTML-style content in the uploaded data. Of course, such uploads represent a minority in most applications, which means that Spectator only needs to tag and track a small portion of all requests. Spectator detects suspicious data by searching for opening matching pairs of HTML tags $<$`tag attribute1` $=$ ... `attribute2` $=$ ...$>$ and $<$/`tag`$>$. Since many servers may require the uploaded data to be URL- or HTML-encoded, Spectator also attempts to decode the uploaded data using these encodings before attempting the pattern-matching.

Spectator embeds a tag immediately preceding the first opening > for each matching pair of HTML tags. (Note that if the original data is URL encoded, Spectator will re-encode the tagged output as well.) To illustrate how tag insertion works, consider an HTTP request containing parameter

```
<div><b onclick="javascript:alert(...)">...</b></div>
```

This parameter will be transformed by Spectator into a request containing

```
<div spectator_tag=56><b
    onclick="javascript:alert(...)"
    spectator_tag=56>...</b>
</div>
```

We tested this scheme with several real-world web servers chosen from a cross-site scripting vulnerability listing site `xssed.com`. For vulnerable servers that reflect user input verbatim, this scheme works well as expected. Our further investigations into three popular Webmail sites, Hotmail, Yahoo Mail, and Gmail, have shown this scheme did not work because the Spectator tags were stripped by the Web servers. While this is difficult to ascertain, our hypothesis is that these sites use a whitelist of allowed HTML attributes.

To handle Web sites that may attempt to strip Spectator tags, we propose an alternative approach. In this new scheme, Spectator embeds tags directly into the actual content surrounded by HTML tags. For example $<$ `b` $>$ `hello world`... $<$ /`b` $>$ will be transformed by Spectator into a request containing $<$ `b` $>$ `spectator_tag` $=$ `56hello world`... $<$ /`b` $>$ We tested this scheme with the three Webmail sites above and found that it works for all of them. However, there is a possibility that such tags may interfere with Web server's application logic. For example, if the length of the actual content is explicitly specified in the data, this tagging scheme will affect data consistency. Unfortunately, while our approach to decode and augment the uploaded traffic works for the sites we have experimented with, in the worst case, the server may choose an entirely new way to encode uploaded parameters. In this case, properly identifying and tagging HTML uploads will require server-side cooperation.

5 Experimental Evaluation

An experimental evaluation of Spectator poses a formidable challenge. Since we do not have access to Web sites on which real-life worms have been released, worm outbreaks are virtually impossible to replicate. Even if we were able to capture a set of server access logs, we still need to be able to replay the user activity that caused them. Real-life access patterns leading to worm propagation are, however, hard to capture and replay. Therefore, our approach is to do a large-scale simulation as well as a small-scale real-world study.

Large-scale simulations. We created OurSpace, a simple Web application that conceptually mimics the functionality of MySpace and similar social networking sites on which worms have been detected, but without the complexity of real sites. OurSpace is able to load and store data associated with a particular user given that user's ID. For faster access, data associated with the user is stored in an in-memory database with which OurSpace communicates. With the help of OurSpace, we have experimented with various access patterns that we feel reflect access patterns of a real-life site under attack.

A real-life case study. It is difficult to experiment with real-life released worms as we discussed earlier. Ideally, we want to have the following features for our experimental setup: (1) a real-life popular widely-deployed Web application or a popular Web site; (2) a running JavaScript worm; (3) users running widely used browsers; and (4) multiple users observed over a period of time. To make sure that our ideas work well in a practical setting, we performed a series of experiments against Siteframe, an open-source content management system that supports blogging features [3]. On a high level, Siteframe is similar to MySpace: a user can post to his own blog, respond to other people's posts, add other users as friends, etc. We used Siteframe "Beaumont", version 5.0.0B2, build 538, because it allows HTML tags in blog posts and does not adequately filter uploaded content for JavaScript.

For our experiments, both OurSpace and Siteface were deployed on a Pentium 4 3.6 Ghz machine with 3 GB of memory machine running Windows XP with Apache 2.2 Web server installed. We ran the Spectator proxy as an application listening to HTTP traffic on a local host. The Spectator proxy is implemented on top of AjaxScope, a flexible HTML and JavaScript rewriting framework [18, 19]. The Spectator proxy implementation consists of 3,200 lines of C# code. For ease of deployment, we have configured our HTTP client to forward requests to the port the Spectator proxy listens on; the proxy subsequently forwards requests to the appropriate server, although other deployment strategies are possible, as discussed in Section 6.1.

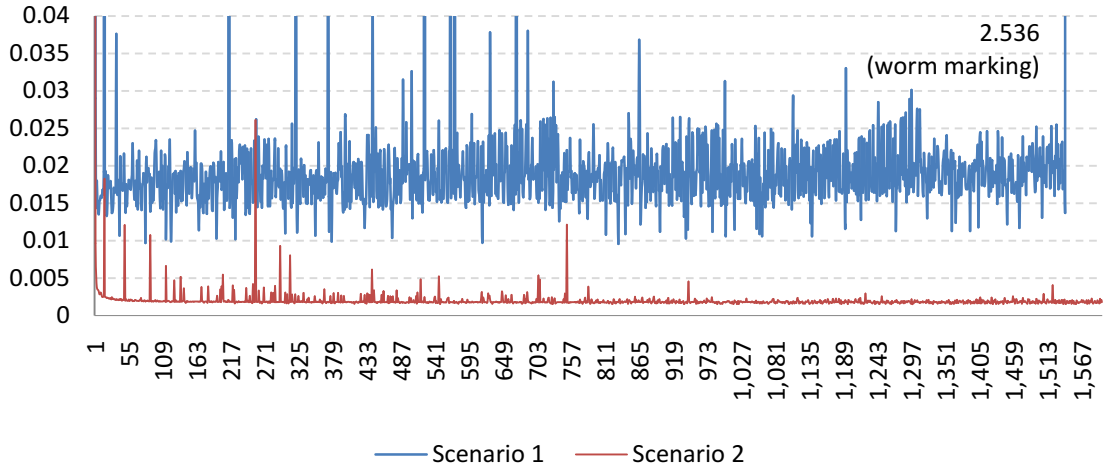

Figure 7: Propagation graph maintenance overhead, in ms, for Scenarios 1 (top) and 2 (bottom)

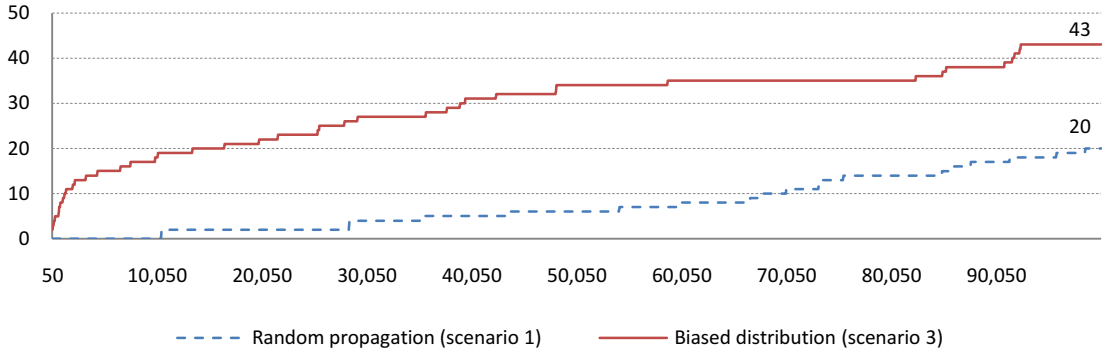

Figure 8: $\mathcal{D}(\mathcal{G}_{\mathcal{A}})$ values for random propagation (Scenario 1) vs biased distribution (Scenario 3)

5.1 OurSpace: Simulating User Behavior

In this section we describe access patterns that we believe to be representative of normal use.

Scenario 1: Worm outbreak (random topology). We have a pool of N users, each with a separate home page served by the Web application. Initially, user 1 wakes up and introduces malicious JavaScript into his profile page. At every step, a random user wakes up and visits a random page. If the visited page is infected, the user infects himself by embedding the content of the visited page into his home page. This simplified propagation model assumes worm traffic is the only HTML traffic that gets tagged. Regular non-HTML uploads do not lead to propagation edges being created. Note that this scenario can be easily augmented so that a user may view pages of multiple other users, thereby introducing sharing in the resulting propagation graph.

Scenario 2: A single long blog entry. We have a pool of N users that access the same blog page one after another. After user k accesses the page, he reads the previous $k-1$ posts and then proceeds to create and uploads an HTML post that contains the previous posts and a new HTML post; the total diameter of the resulting graph is 2.

Scenario 3: A model of worm propagation (power law connectivity). To reflect the fact that some users are much more connected and active than others, in Scenario 3 we bias user selection towards users with a smaller ID using the power law. Most of the time, the user selection process heavily biases the selection towards users with a small ID. This bias reduces the set of users most actively participating in worm propagation, leading to "taller" trees being created.

5.2 Overhead and Scalability

To estimate the overhead, we experimented with Scenario 1 to determine how the approximate algorithm insertion time fluctuates as more nodes are added to the graph. Figure 7 shows insertion times for Scenario 1 with the detection threshold d set to 20. The x-axis corresponds to the tag being inserted; the y-axis shows the insertion time in milliseconds. The entire run took about 15 minutes with a total of 1,543 nodes inserted.

The most important observation about this graph is that the tag insertion latency is pretty much constant, hovering around .01 to .02 ms for Scenario 1 and close to .002 ms for Scenario 2. The insertion time for the second scenario is considerably lower, since the resulting approximation graph G_A

is much simpler: it contains a root directly connected to every other node and the maximum depth is 2. Since our proxy is implemented in C#, a language that uses garbage collection, there are few spikes in the graph due to garbage collection cycles. Also, the initial insertions take longer since the data structures are being established. Moreover, once the worm is detected at $d = 20$ for tag 1,543, there is another peak when all the nodes of the tree are traversed and marked.

5.3 Effectiveness of Detection

One observation that does not bode well for our detection approach with a random topology is that it takes a long time to reach a non-trivial depth. This is because the forest constructed with our approximation algorithm usually consists of set of shallow trees. It is highly unlikely to have a long narrow trace that would be detected as a worm before all the previous layers of the tree are filled up. However, we feel that the topology of worm propagation is hardly random. While researchers have tried to model worm propagation in the past, we are not aware of any work that models the propagation of JavaScript worms. We believe that JavaScript worms are similar to email worms in the way they spread. Propagation of JavaScript worms also tends to parallel social connections, which follow a set of well-studied patterns. Connectivity distribution is typically highly non-uniform, with a small set of popular users with a long tail of infrequent or defunct users. Similar observations have been made with respect to World Wide Web [1] and social network connectivity [2].

To properly assess the effectiveness of our approximation approach, we use Scenario 3, which we believe to be more representative of real-life topology. The simulation works as follows: initially, user 1's page is tainted with a piece of malicious JavaScript. At each step of the simulation, a user wakes up and chooses a page to view. The ID of the user to wake up and to view is chosen using the power law distribution. Viewing this page will create an edge in the propagation graph from the tag corresponding to the page selected for viewing to the newly created tag of the user that was awoken.

In propagation graphs generated using Scenario 3, once worm propagation reaches a well-connected node, it will tend to create much longer propagation chains involving that node and its friends. Figure 8 shows the diameter of G_A on the y-axis as more nodes are added up to 100,000 nodes for Scenarios 1 and 3, as shown on the x-axis. Observe that the diameter grows more rapidly in the case of selecting users from a biased distribution, as fewer nodes will be actively involved in propagation and shallow trees are less likely. This result indicates that in a real-life large-scale setting, which is likely to be similar to Scenario 3, our worm detection scheme is effective.

5.4 Precision of the Detection Algorithm

Note that as discussed in Section 3.3, the approximate algorithm detects the worm before the precise one in most cases. In fact, we have not encountered instances of when the approximate algorithm produces false negatives. However, a legitimate question is how much *earlier* is the worm detected with the approximate algorithm. If the approximate strategy is too eager

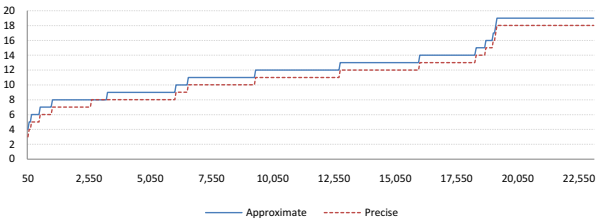

Figure 9: Approximate vs precise $\mathcal{D}(G_A)$ values

to flag a worm, it will result in too many false positives to be practical. Whether that happens depends on the structure of the graph and the amount of sharing it has.

In order to gauge the detection speed obtained with the approximate scheme as opposed to the precise one we used simulations of Scenario 3 to generate a variety of random propagation graphs. We measured the diameter of the resulting propagation graph, as obtained from the precise and approximate methods. Figure 9 shows how $\mathcal{D}(G)$ and $\mathcal{D}(G_A)$ values, shown on the y-axis differ as more nodes are inserted, as shown on the x-axis for one such simulation. The differences between the two strategies are small, which means that we are not likely to suffer from false alarms caused by premature detection in practice, assuming a sufficiently high detection threshold. Furthermore, the approximation algorithm is always conservative in this simulation, over-approximating the diameter value.

5.5 Case Study: Siteframe Worm

For our experiments we have developed a proof-of-concept worm that propagates across a locally installed Siteframe site (The entire code of the Siteframe worm is presented in our technical report [21].). Conceptually our worm is very similar to how the Adultspace worm [32] works: the JavaScript payload is stored on an external server. At each propagation step, a new blog page is created, with a link to the worm payload embedded in it. This allows the worm to load the payload from the server repeatedly on every access. Whenever somebody visits the page, the worm executes and proceeds to create a new entry on the viewer's own blog that contains a link to the payload. To make our experiment a little easier to control, infection is triggered by the user clicking on an HTML `<DIV>` element. In real-life infection would probably occur on every page load.

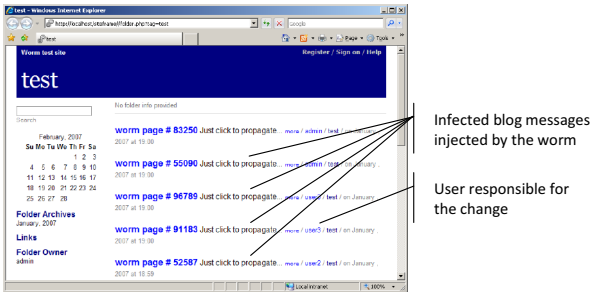

Figure 10: The main Siteframe site page after worm propagation

The worm does not check if a particular user has already been infected.

For our case study we created a total of five users on a fresh Siteframe site. Each user performed various activity on the site, leading to one or more worm propagation steps. The diameter of the resulting propagation graph was 5. To give a sense of the effects of worm propagation in this case, a screen shot of the resulting top-level page of the Siteframe site is shown in Figure 10. While small in scale, the Siteframe worm experiment has *significantly enhanced* our trust in the effectiveness of Spectator. Because the Siteframe worm was modeled after worms previously released in the wild, we believe that Spectator would have detected those worms.

6 Discussion

This section is organized as follows. Section 6.1 presents different deployment models for Spectator and Section 6.2 addresses threats to the validity of our approach.

6.1 Deployment Models for Spectator

Spectator works in both small-scale environments with servers that do not have a lot of activity and also with servers that have thousands of active users. We envision the following deployment models.

Server-side Deployment. Server-side deployment is the easiest way to protect an existing Web site from JavaScript worms using Spectator. Deploying the Spectator proxy in front of the server or servers that the site resides on allows the proxy to monitor all client-server interaction for that site and detect worms faster than it would in the case of being deployed elsewhere on the network and seeing only a portion of the total browser-server traffic. This model has the advantage of simplifying worm reporting, since the server is responsible for Spectator proxy maintenance.

Intranet-wide Deployment. Intranet deployment can be used to protect users within an organization, such as a university or a small enterprise against worm outbreaks. In many cases, these environments are already protected by firewalls and the Spectator proxy can be easily integrated within that infrastructure. Of course, worm detection in this kind of deployment is only possible if sufficiently many intranet users get infected. However, in the case of intranet deployment, the same proxy can be used to prevent worms propagating on a wide variety of sites without changes to our detection or tagging approaches.

A technical issue with client-side deployment is the use of SSL connections, which are not handled by the Spectator proxy. However, SSL sessions are frequently only used for initial authentication in Web applications and it is easy to set up one's browser to redirect requests to the Spectator proxy for non-SSL connections only. For server-side deployment though, the proxy can be placed before the SSL connection.

Large-scale Deployment. For large-scale server-side deployment, we may implement Spectator as part of the site's load balancer. Load balancing is a strategy used by most large-scale services such as MySpace or Live Spaces. When dealing with multiple proxies, our approach is to distribute different trees in the forest G_A across the different proxy servers. The load balancer considers the source node of the edge being added to decide which proxy to redirect the request to. To avoid maintaining explicit state at the load-balancer, such as a lookup map that maps the parent tag to the proxy server containing that tree, our strategy is to assign the tag number *after* node insertion, based on which proxy it goes into. For instance, the last 5 bits of the tag may encode the number of the proxy to pass the request to. In the case of a node having more than one parent, we choose between two parents, based on the parent's depth as described in Section 3. When a proxy server is full and a new edge, whose parent resides on that proxy server is inserted, we migrate the newly inserted node to a different proxy server as a new tree. However, instead of the initial depth of 1, the depth of the root node for that tree is computed through our standard computation strategy.

While this deployment strategy closely matches the setup of large sites, an added advantage is the fact that we no longer have to store the entire forest in memory of a single proxy. A similar distributed strategy may be adopted for intranet-wide client-side deployment. Distributed deployment has the following important benefit: an attacker might try to avoid detection by flooding Spectator with HTML uploads, leading to memory exhaustion, and then unleashing a worm. Distributed deployment prevents this possibility.

6.2 Threats to Validity

The first and foremost concern for us when designing Spectator was limiting the number of false positives, while not introducing any false negatives. Recall that we require reliable HTML input detection and marking (see Property 1 in Section 2.2). Violations of this required property will foil our attempt to tag uploaded HTML and track its propagation, resulting in false negatives. However, Property 1 holds for all worms detected in the wild so far, as described in our technical report [21], and we believe that Spectator would have successfully detected them all. Still, potential for false positives remains, although without long-term studies involving large-scale data collection it is hard to say whether false positives will actually be reported in practice. Furthermore, it is possible for a group of benign users to perform the same activity a worm would run automatically. With a low detection threshold, the following manual worm-like activity is likely to be regarded as worm outbreaks.

Chain email in HTML format. As long as forwarding preserves HTML formatting, including Spectator tags, this activity will be flagged as a worm. Spectator has difficulty distinguishing this manual process from an automatic one such as the propagation of the Yamanner worm [4].

A long blog post. Similarly to a piece of chain mail, while a precise detection algorithm will an not flag an excessively long blog post as a worm, the approximate algorithm will.

To avoid false positives, site administrators can set the detection thresholds higher. For instance, 500 is a reasonable detection threshold for Webmail systems and 1,000 is very conservative for blogging sites. As always, there is a trade-off between the possibility of false positives and the promptness of real worm

detection. However, a worm aware of our detection threshold may attempt to stop its propagation short of it [23].

7 Related Work

While to the best of our knowledge there has not been a solution proposed to JavaScript worms, there are several related areas of security research as described below.

7.1 Worm Detection

Since 2001, Internet worm outbreaks have caused severe damage that affected tens of millions of individuals and hundreds of thousands of organizations. This prompted much research on detecting and containing worms. However, most of the effort thus far has focused on worms that exploit vulnerabilities caused by unsafe programming languages, such as buffer overruns. Many techniques have been developed, including honeypots [9, 14, 41], dynamic analysis [7, 28], network traffic analysis [27, 36, 43], and worm propagation behavior [10, 45]. Our work is primarily related to research in the latter category. Xiong [45] proposes an attachment chain tracing scheme that detects email worm propagation by identifying the existence of transmission chains in the network. The requirement for monitoring multiple email servers limits the practicality of this scheme. Spectator, on the other hand, can observe all relevant traffic if deployed on the serve side.

Ellis *et al.* [10] propose to detect unknown worms by recognizing uncommon worm-like behavior, including (1) sending similar data from one machine to the next, (2) tree-like propagation and reconnaissance, and (3) changing a server into a client. However, it is unclear how the behavioral approach can be deployed in practice because it is difficult to collect necessary information. We use a very basic worm-like behavior — long propagation chains — to detect JavaScript worms. Unlike Internet worms, JavaScript worms usually propagate inside the same Web domain. Spectator proposes an effective approach to achieve centralized monitoring, enabling worm detection. Our approach that only counts *unique* IP addresses on a propagation path is similar to looking at the propagation tree breadth in addition to its depth.

7.2 Server-side XSS Protection

There has been much interest in static and runtime protection techniques to improve the security posture of Web applications. Static analysis allows the developer to avoid issues such as cross-site scripting before the application goes into production. Runtime analysis allows exploit prevention and recovery. The WebSSARI project pioneered this line of research [15]. WebSSARI uses combined unsound static and dynamic analysis in the context of analyzing PHP programs. WebSSARI has successfully been applied to find many SQL injection and cross-site scripting vulnerabilities in PHP code. Several projects that came after WebSSARI improve on the quality of static analysis for PHP [17, 44]. The Griffin project proposes a scalable and precise sound static and runtime analysis techniques for finding security vulnerabilities in large Java applications [22, 24]. Based on a vulnerability description, both a static checker and a runtime instrumentation is generated. Static analysis is also used to drastically reduce the runtime overhead in most cases. The runtime system allows vulnerability recovery by applying user-provided sanitizers on execution paths that lack them. Several other runtime systems for taint tracking have been proposed as well, including Haldar *et al.* for Java [12] and Pietraszek *et al.* [31] and Nguyen-Tuong *et al.* for PHP [29].

7.3 Client-side Vulnerability Prevention

Noxes, a browser-based security extension, is designed to protect against information leakage from the user's environment while requiring minimal user interaction and customization effort [20]. Information leakage is a frequent side-effect of cross-site scripting attacks; e.g., the act of sending a cookie to an unknown URL will be detected and the user will be prompted. While effective at blocking regular cross-site scripting attacks, Noxes is generally helpless when it comes to data that is transmitted to a presumably trusted side without user's knowledge, such as it would be in the case of a JavaScript worm. In [40], Vogt *et al.* propose to prevent XSS on the client side by tracking the flow of sensitive information inside the web browser using dynamic data flow and static analysis. The main issues with their solution are the number of false alarms and how an average user can decide if an alarm is false.

8 Conclusions

This paper presents Spectator, the first practical detection and containment solution for JavaScript worms. The essence of the Spectator approach is to observe and examine the traffic between a Web application and its users, looking for worm-like long propagation chains. We have implemented and evaluated the Spectator solution on a number of large-scale simulations and also performed a case study involving a real JavaScript worm propagating across a social networking site. Our experiments confirm that Spectator is an effective and scalable, low-overhead worm detection solution.

Acknowledgments

We would like to express our profound gratitude to Karl Chen, Emre Kıcıman, David Molnar, Berend-Jan "SkyLined" Wever, and others for their help in refining the ideas contained here and last-minute proof-reading help. Special thanks go to Úlfar Erlingsson and Martin Johns for their suggestions on how to implement client-side support. We are also grateful to the anonymous reviewers as well as our shepherd Sam King.

References

[1] L. A. Adamic, B. A. Huberman;, A. Barab'asi, R. Albert, H. Jeong, and G. Bianconi;. Power-law distribution of the world wide web. *Science*, 287(5461):2115a+, Mar. 2000.

[2] R. L. Breiger. *Dynamic Social Network Modeling and Analysis*. National Academies Press, 2004.

[3] G. Campbell. Siteframe: a lightweight content-management system. `http://siteframe.org/`.

[4] E. Chien. Malicious Yahooligans. http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf, Aug. 2006.

[5] S. Corporation. Acts.spaceflash. http://www.symantec.com/security_response/writeup.jsp?docid=2006-071811-3819-99&tabid=2, July 2006.

[6] S. Corporation. JS.Qspace worm. http://www.symantec.com/enterprise/security_response/writeup.jsp?docid=2006-120313-2523-99&tabid=2, Dec. 2006.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of Symposium on the Operating Systems Principles*, Oct. 2005.

[8] D. Crockford. Private members in JavaScript. http://www.crockford.com/javascript/private.html, 2001.

[9] W. Cui, V. Paxson, and N. Weaver. GQ: realizing a system to catch worms in a quarter million places. Technical Report TR-06-004, ICSI, Sept. 2006.

[10] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A behavioral approach to worm detection. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, October 2004.

[11] J. Grossman. Cross-site scripting worms and viruses: the impending threat and the best defense. http://www.whitehatsec.com/downloads/WHXSSThreats.pdf, Apr. 2006.

[12] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, Dec. 2005.

[13] B. Hoffman. Analysis of Web application worms and viruses. http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Hoffman/BH-Fed-06-Hoffman-up.pdf, 2006.

[14] Honeynet. The honeynet project. http://www.honeynet.org/.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web application code by static analysis and runtime protection. In *Proceedings of the Conference on World Wide Web*, pages 40–52, May 2004.

[16] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. Technical report, Department of Computer Science, University of Maryland, 2006.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities (short paper). In *Proceddings of the Symposium on Security and Privacy*, May 2006.

[18] E. Kiciman and B. Livshits. AjaxScope: a platform for remotely monitoring the client-side behavior of Web 2.0 applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[19] E. Kiciman and H. J. Wang. Live monitoring: using adaptive instrumentation and analysis to debug and maintain Web applications. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[20] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the Symposium on Applied Computing*, Apr. 2006.

[21] B. Livshits and W. Cui. Spectator: Detection and containment of JavaScript worms. Technical Report MSR-TR-2007-55, Microsoft Research, 2007.

[22] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2005.

[23] J. Ma, G. M. Voelker, and S. Savage. Self-stopping worms. In *Proceedings of the ACM Workshop on Rapid malcode*, pages 12–21, 2005.

[24] M. Martin, B. Livshits, and M. S. Lam. SecuriFly: Runtime vulnerability protection for Web applications. Technical report, Stanford University, Oct. 2006.

[25] S. Meschkat. JSON RPC: Cross site scripting and client side Web services. In *23rd Chaos Communication Congress*, Dec. 2006.

[26] M. Murphy. Xanga hit by script worm. http://blogs.securiteam.com/index.php/archives/166, Dec. 2005.

[27] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, May 2005.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of Network and Distributed System Security Symposium*, February 2005.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening Web applications using precise tainting. In *Proceedings of the IFIP International Information Security Conference*, June 2005.

[30] P. Petkov. The generic XSS worm. http://www.gnucitizen.org/blog/the-generic-xss-worm, June 2007.

[31] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the Recent Advances in Intrusion Detection*, Sept. 2005.

[32] RSnake. Adultspace XSS worm. http://ha.ckers.org/blog/20061214/adultspace-xss-worm/, Dec. 2006.

[33] RSnake. Semi reflective XSS worm hits Gaiaonline.com. http://ha.ckers.org/blog/20070104/semi-reflective-xss-worm-hits-gaiaonlinecom/, Jan. 2007.

[34] RSnake. U-dominion.com XSS worm. http://ha.ckers.org/blog/20061214/adultspace-xss-worm/, Jan. 2007.

[35] Samy. The Samy worm. http://namb.la/popular/, Oct. 2005.

[36] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[37] SonicWALL. SonicWALL deploys protection against MySpace worm. http://sonic-wall.blogspot.com/2006/12/sonicwall-deploys-protection-against.html, Dec. 2006.

[38] M. Surf and A. Shulman. How safe is it out there? http://www.imperva.com/download.asp?id=23, 2004.

[39] Symantec Corporation. Symantec Internet security threat report, volume X, Sept. 2006.

[40] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, and C. Kruegel. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium*, pages 1–2, 2007.

[41] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2005.

[42] WebCohort, Inc. Only 10% of Web applications are secured against common hacking techniques. http://www.imperva.com/company/news/2004-feb-02.html, 2004.

[43] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. Technical Report HPL-2002-172, HP Labs Bristol, 2002.

[44] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 271–286, Aug. 2006.

[45] J. Xiong. ACT: Attachment chain tracing scheme for email virus detection and control. In *Proceedings of the Second ACM Workshop on Rapid Malcode (WORM)*, Oct. 2004.