Securing Web Applications

A Dissertation presented

by

Riccardo Pelizzi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2016

Stony Brook University

The Graduate School

Riccardo Pelizzi

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

R. Sekar - Dissertation Advisor
Professor, Department of Computer Science

Scott Stoller - Chairperson of Defense
Professor, Department of Computer Science

Nikolaos Nikiforakis - Committee Member
Professor, Department of Computer Science

William E. Robertson - External Committee Member
Professor, Department of Computer Science, Northeastern University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Securing Web Applications

by

Riccardo Pelizzi

Doctor of Philosophy

in

Computer Science

Stony Brook University

2016

Over the past decade, web application vulnerabilities have become far more common than vulnerabilities in conventional applications. To mitigate them, we approach the problem from two extremes: one that requires no changes to existing applications but is limited to a few well-defined vulnerability classes, and the second that provides a comprehensive solution but requires a re-thinking of web applications.

Our first approach mitigates specific vulnerabilities using policies that do not depend on the application logic, and thus require no developer involvement or effort. We target two of the most common high-profile vulnerabilities, namely, cross-site scripting (XSS) and cross-site request forgery (CSRF). The solutions we have developed are very effective, efficient, and represent significant advances over previous research in these area.

Unfortunately, some of the more subtle and complex vulnerabilities arise due to a lack of specification of security policies, and due to the ad-hoc way in which they are enforced within application code. We therefore propose a new way to develop web applications that separates and decouples security policy from application logic. Our proposal, called WebSheets, provides a simple and intuitive language for policy specification, based on the familiar spreadsheet paradigm. A spreadsheet model is natural because web appli-

cations typically operate on tabular data. As a result, we show that the logic of many simple web applications is nothing more than a specification of security policies, and hence a WebSheet security specification is all that is needed to realize them. This dissertation presents the WebSheet model, and describes proposed work aimed at developing and implementing the model, and demonstrating its ability to secure a range of significant web applications.

Dedicated to

My fiancée Lisa, who supported me through this journey and gave me the strength and the motivation to overcome all obstacles and finish

&

My parents Roberto and Grazia, who taught me to aspire to greatness and set the foundations for my success

Table of Contents

Acknowledgements

First and foremost, I want to thank my advisor, Prof. R. Sekar. He taught me everything I know about being an academic scholar. He guided me and supported me during this whole journey, never giving up on me, even when I was too stubborn or lazy to take his advice or learn from his teachings. After many years, I am hopeful that some of his knowledge, compassion and work ethic rubbed off on me, and that I am finishing the Ph.D. program as a better person. There were troubled, stressful times when I doubted that a Ph.D. was the best use of my time and energy, or that I had what it takes to finish, but I never doubted for a second that I had the best advisor I could ask for.

I must also thank him for providing continuous financial support – not once did I have to worry about funding, which allowed me to concentrate on my research full-time. Speaking of funding, I want to formally acknowledge the grants that have made my research possible: NSF grants CNS-0831298 and CNS-1319137, ONR grant N00014-07-1-0928 and AFOSR grant FA9550-09-1-0539.

I would also like to thank my dissertation committe for dedicating their time and providing constructive criticism. On the same note, I also want to thank Prof. Donald Porter, who did not end up in the final committee, but provided invaluable feedback on an earlier iteration of this document.

I must also thank all my colleagues and friends, who made working at Stony Brook an enjoyable experience. In particular, I want to thank my colleagues from the System Security Lab: Alireza, Tung, Niranjan, László, Peter, Mingwei, Rui and Nahid; my roommates from Terryville 228: Heraldo, Jon and Navid; my roommate Puneet; and my Italian friends: Giulia, Luisa and Andrea.

Last but not least, I want to thank my family and my fiancée for their unconditional love and support. Academic research can sometimes leave you feeling directionless, but I could always count on them to provide structure and purpose.

1 Introduction

The evolution of the World Wide Web from a limited set of static HTML pages to a vast network of interconnected dynamic web applications has drastically changed the vulnerability scenario. While traditional applications are written in low-level, unsafe languages such as C, web applications are written using high-level, type safe languages such as PHP and Python. The most common vulnerabilities found in programs written in unsafe languages are *memory errors*, such as buffer overflows. However, the threat of memory errors is largely a non-issue for web applications: they are plagued by a fundamentally different set of vulnerabilities, which must be studied and mitigated separately. As the web application model becomes more and more popular, replacing not only traditional server applications (e.g. POP3 Mail Servers, which have been replaced by Webmail portals such as GMail), but also desktop applications (e.g. Microsoft Office, which is now also sold as a SaaS web application), the ratio of memory errors to web vulnerabilities keeps shifting in favor of the latter. According to the Common Vulnerabilities and Exposures (CVE) list, web vulnerabilities are now more common than traditional memory errors [31]. Thus, mitigating web vulnerabilities, which are often the result of logic errors rather than bugs regarding low-level details such as the length of a buffer on the stack, is becoming increasingly more important.

Many of these vulnerabilities share the same root cause: while the awareness about web vulnerabilities has increased in the past decade, developers still follow the same unprincipled approach towards security: namely, the security policy of web applications is implemented using ad-hoc checks scattered throughout the codebase and mixed with the application logic, and vulnerabilities are addressed and fixed after they are found and reported. This unprincipled, reactive approach to security leads to "lock the barn door after the horse is stolen" situations, because between the time the vulnerability is first exploited and the time it is fixed, the attack has usually caused considerable damage. For example, the United States Office of Personnel Management (OPM) was the victim of a major data leak in 2014 that involved 21.5 million records, including highly sensitive data such as SSNs, fingerprints and security clearance information [97]. The leak has since been stopped, but the damage cannot be undone: requesting a new SSN is a tedious, ineffective process that damages the victim's credit history and requires proof of an *ongoing* misuse, which rules data breaches out as a valid

reason until the data is actually used for identity theft [13]. Worse, the attackers now possess fingerprint data that will forever identify these individuals. Consider the possibility that a US covert agent can now be identified by a foreign government using data from this leak.

Intuitively, the best solution to the problem of scattered security checks is to extract them by analyzing the informally specified policy and express them in a way that is not intertwined with the application logic. However, this clashes with the reality of web development; most web developers have limited experience with policy specification and are tasked with developing new features instead of improving the security of legacy code; it is very cumbersome for them to think about the security policy informally implemented by their checks, especially considering the time constraints they are subject to.

Given that developer's incentives and skills are not aligned with security, it is important to provide usable security for web applications. We tackle the problem from two different perspectives: firstly, we focus on providing defenses with the widest applicability, that is, defenses for the most common vulnerabilities that require no programmer effort; then, we envision a rearchitecting of how web application are written to provide broader security guarantees.

1.1 Black-Box Automatic Defenses

To maximize the practical impact of our research work, the first part of this dissertation focuses on defenses with the broadest applicability. In particular, we realized this goal through two main design decisions:

- *focus on the most common vulnerabilities*: some classes of vulnerabilities are more prevalent than others. Their popularity is a combination of many factors, such as how easy it is to write code free of such vulnerabilities, how low the barrier of entry is for new attackers, etc. Many security firms and non-profit initiatives (such as CVE) collect statistical data on real-world vulnerabilities and compile lists of top vulnerabilities. In particular, the Open Web Application Security Project (OWASP), a non-profit organization dedicated to educating the public on web application security, aggregates data from several sources and maintains a Top 10 list [133] of web application vulnerabilities. Note that vulnerabilities such as Cross-Site Scripting (XSS) and Cross-Site

Request Forgery (CSRF) are "staple vulnerabilities", appearing in every edition since 2007.

- *require no programmer effort*: although all vulnerabilities ultimately reflect a weakness in the security policy of a specific web application, some can be discovered and exploited with little knowledge of the web application internals, because the fault is largely independent of the rest of the security policy and application logic. This usually implies the presence of a recurring unsafe pattern, which security researchers can leverage to provide an automatic defense. We refer to these defenses as *black-box defenses*: because they are independent of the internals, their deployment and configuration effort are simple or non-existent.

In this dissertation we chose to focus on XSS and CSRF, two popular vulnerabilities which can both be addressed using black-box defenses. Both vulnerabilities are technically server-side vulnerablities, but successful attacks damage the integrity and confidentiality of end users' data. An XSS vulnerability allows an attacker to inject malicious script content in web pages returned to victim users, bypassing the basic isolation mechanism known as the Same-Origin Policy. Because the injected content runs with the website's full privileges on the victim's browser, the attacker can steal confidential information, impersonate the user (e.g. take over his online banking session [101]) or even use his browser to perform internal network scans and DDos attacks [44]. On the other hand, a CSRF vulnerability allows an attacker to abuse the credentials of the website's users, tricking the browser of the victim into performing an unauthorized request. For example, CSRF vulnerabilities have been exploited on banking sites to transfer money from the victim to the attacker's account [148]. Other entries from the aforementioned OWASP Top 10 list are either problematic to mitigate without developer input (e.g. *Sensitive Data Exposure*), are not relevant to web applications (e.g. Insecure Direct Object References), or have been tackled and mitigated by a great deal of previous work (e.g. SQL Injection (SQLI) and OS Command Injection [118, 5, 45, 49]).

To protect against these two vulnerabilities, this thesis presents XSSFilt and jCSRF, two defenses against XSS and CSRF respectively. In particular, XSSFilt proposes a new filter architecture that avoids the problems found in previous filters such as IE8's XSS filter and XSSAuditor, covering a greater number of XSS vectors; jCSRF protects existing web applications by auto-

matically rewriting outgoing HTTP responses, augmenting web pages with the ability to authenticate not only same-origin but also cross-origin requests.

With respect to the problems of ad-hoc checking that plague traditional web applications, these defenses provide a limited, yet significant improvement: for a specific vulnerability (and a subset of its attack vectors), they provide protection for the entire web application, including its corner cases and unfamiliar or untested features. Moreover, because these defenses do not rely on the underlying application logic or policy, when the web application is modified the defense is immediately effective on the new codebase, thus doing away with the need for maintenance of the defense itself.

1.2 Towards a principled approach

Unfortunately, black-box defenses have a major limitation: they are vulnerability-specific defenses that block attacks based on *how* the attacks bypass the security policy, not whether the policy is bypassed at all. This means that they protect against *some* ways to bypass the security policy informally expressed in the application logic, but not all of them. For example, an ideal XSS filter is supposed to prevent attackers from inserting unauthorized scripts, thus prohibiting an attacker from reading HTTP responses and DOM nodes that are meant for another user. However, most real-world XSS filters only defend against one specific way of inserting scripts, namely injection of reflected content into the HTML response. Unfortunately, this is not the only way information can leak out: there are other ways to inject script content (i.e. stored XSS, where there is no immediate relationship between request and response), and there are countless other ways to access the same piece of data that do not involve injection of unauthorized scripts altogether. For example, a SQLI attack can extract the data directly from the web application database without involving the victim's browser, or a bug in the authentication implementation may allow the attacker to impersonate a particular user without stealing his session cookies. It is hard to offer a black-box protection that covers against all these different vectors, especially considering that each vulnerability can be exploited in different ways.

To provide broader protection, it is beneficial to take a step back and approach the problem from a different perspective: although there are countless vulnerabilities representing even more attack vectors, perhaps there is a root cause to all these vulnerabilities? In that case, addressing it would prevent exploitation of any of them. If not a root cause, perhaps a common goal to

all these attacks? If so, addressing it would reduce or nullify the damage caused by successful exploits. Indeed, we believe that data theft is the end goal of most attacks in the wild. Small-scale data theft involves stealing the credentials of a specific user, to exploit his specific permissions or to steal his personal information (e.g. as a stepping stone for a spear phishing attack). On the other hand, large-scale data leaks involve *millions* of users, whose data is harvested for its resale value. The latter kind of attacks are steadily on the rise, almost quadrupling from 2011 to 2015 [117]. For example, 80 million records containing sensitive personal information such as SSNs were stolen from Anthem, the second largest insurance provider in the US [57].

Regardless of their scale, these are all attacks on the confidentiality of the data of each user, and stopping them brings us back to the core problem of web application development we already discussed: the security of the user's data does not have a first-class role, and the privacy policy is implicitly implemented using checks scattered throughout the application logic. This approach has many major drawbacks:

- *Hard to cover all cases.* Because enforcing a specific policy requires placing the same check many times throughout the codebase, *it is easy to forget one check and cause a vulnerability.* For example, phpMyAdmin requires 1409 checks for user input sanitization alone. Unsurprisingly, its developers forgot a handful of checks, resulting in vulnerabilities [145].

- *Lack of separation of concerns.* Large, complex projects benefit from division of labor: each developer is tasked with implementing or maintaining different pieces of the codebase. Nowadays, it is common to have a separate security team that handles the security concerns of the project. However, the existing approach *prevents policy and application logic developers from operating on different code that can be independently reviewed.*

- *Difficult to maintain.* Maintenance is even more problematic than the initial development itself: *if the informally specified policy must be modified* (either because of a bug fix or to accommodate for a new feature), *the developers must track down and update all checks.*

- *No least-privilege principle.* Since the application logic itself performs all checks and controls access to all of the data, it must therefore have

access itself to all the data: *there is no application of the least privilege principle*. If the ad-hoc policy has a bug that allows an attacker to execute custom code (or SQL query, Shell Command, etc.), there is no limit to the damage or the extent of the data leak. While this is standard practice in web application development, we note that OS developers would balk at the idea of trusting userspace applications (e.g. a Word Processor) to enforce system-wide policies such as checking for file permissions!

- *No formal verification or analysis.* Because the security policy is scattered throughout the codebase and and written in a high-level language with complex semantics, *it is virtually impossible to provide the developer with analysis tools to formally verify the policy*, or at least check for a restricted set of desirable security properties.

In this thesis, we devise a new approach that avoids these drawbacks. While we break away from traditional web application development, we have a chance to solve two important problems simultaneously:

- *The technical problem of bug-free policy development.* Traditional development practices force developers to convert the high-level policy they devised into low-level checks scattered throughout the application logic. This is an error prone process: as we have seen, the same checks need to be repeated into several code paths to avoid introducing a data leak.

- *The non-technical problem of conflicting security concerns.* Often, application developers and end users are fundamentally different actors. Although the former can provide a baseline privacy policy that works for common use cases, developers are mostly focused on implementing the core functionality of the application (i.e. the application logic), forgoing support for fine-grained policies. Moreover, the best way to avoid support calls is to enforce relaxed policies that avoid security failures. Instead, it is the end users who ultimately know which pieces of information are sensitive and who should be given access to them.

To understand what's at stake when solving the technical problem of policy errors, consider the recent Voter Database leak [39]: a security researcher discovered a publicly exposed database of information on 191 million voters

due to a web application misconfiguration. If developers could specify the privacy policy of voter records using a high-level policy framework that automatically performs the necessary low-level checks, it would be easy for them to prevent exposure to attackers. A sensible policy would expose the data only to allowed officials; in this case, the developers would understand the need to keep this information secret and the policy is cut and dry. However, other kinds of information require support for fine-grained policy that are controlled by the end user. In this case, the non-technical problem of mismatching security concerns plays a part. Consider a social network: while many users are privacy conscious, developers do not have a strong incentive to provide fine-grained privacy controls, as the success of most social networks depends on providing a simple interface that users leverage to share the most amount of data possible to the widest audience.

In this thesis, we start from a clean slate and envision a new web application development paradigm where users retain control of their data and can specify a privacy policy that follows the data throughout the web application. Support for high-level policy development has already been retrofitted into existing web application practices using Mandatory Access [100, 15] and in particular Decentralized Information-Flow Control [87, 145, 42] (DIFC). However, none of these works directly empower *end users* to provide their own policies. The new paradigm is based on the spreadsheet model: application logic and user data is entered in data tables, while the security policies are defined through permission tables. Both types of tables can contain not only immediate values, but also formulas which can refer to other table cells, just like an ordinary spreadsheet.

1.3 Contributions

This dissertation describes two black-box defenses, XSSFilt and jCSRF, and a new web application development paradigm focused on security, WebSheets. Below, we detail our contributions.

1.3.1 XSSFilt

We present XSSFilt, a client-side, browser-resident black-box defense against reflected and DOM-Based XSS attacks. XSS was a natural choice for our purposes, because it has consistently been among the top threats for web applications [133]. Previous work on automatic XSS defenses has mostly

focused on static HTML filtering [63, 118, 108], comparing pairs of HTTP requests and responses for injection of script content.

Performing XSS filtering on the HTML response has two major shortcomings: firstly, it is vulnerable to "browser quirks", non-standard parsing exceptions built into each browser's HTML parser. For example, Firefox considers `<script/hello src=...></script>` to be a valid `script` tag. Since the filter needs to correctly identify which parts of the response represent code that will be executed, non-standard behavior can be exploited to confuse the filter. Secondly, static filtering cannot protect against DOM-Based XSS attacks, a variant of XSS where the malicious content does not appear in the HTML response and is only introduced at runtime by existing JavaScript code. In this case, static filters cannot speculate on the runtime behavior of an HTTP response.

To provide immunity against browser quirks, XSSAuditor [7] proposed a new filter architecture that tightly integrates with Google Chrome's HTML parser, which guarantees consistent detection of script content in the HTML response. However, the architecture does not fully address the problem of DOM-Based attacks, because the majority of DOM-Based vectors never interact with the HTML parser. Moreover, the matching algorithm used by XSSAuditor does not account for arbitrary string transformations performed by the web application and partial injections, where malicious javascript content is injected into an existing script.

XSSFilt solves all the aforementioned problems: a new architecture provides complete coverage of DOM-Based XSS vectors and an approximate substring- matching algorithm is used to account for arbitrary string transformations and to cover the partial injection vector. Thanks to its improvements, XSSFilt is able to detect more attacks than other state-of-the-art black-box XSS defenses. The filter was eventually deployed in the Palemoon web browser, an open-source Firefox fork with over 500,000 users.

1.3.2 jCSRF

We also present jCSRF, a CSRF defense for Web 2.0 Applications. Although writing code free of CSRF vulnerabilities is conceptually simple, requiring only the use of a nonce for all sensitive requests, CSRF vulnerabilities are still common. Even to this day, CSRF vulnerabilities are regularly found in router firmware, which enable attackers to gain access to the router's internal network and possibly intercept its traffic [130].

Researchers have already proposed several CSRF defenses [66, 64, 28, 96, 143, 120]. However, they all suffer from limited applicability, because they either a) require access to the source code or knowledge of the web application internals (i.e. they are not black-box defenses), b) require modification to existing WWW standards (e.g., a new HTTP header), c) are incompatible with dynamically generated requests (e.g., `XMLHttpRequest`s) and d) cannot support certification of cross-origin requests.

jCSRF is a novel CSRF defense that does not suffer from any of the above drawbacks. It operates by interposing transparently on the communication between clients and servers, modifying HTML responses and adding a script that authenticates both same-origin and cross-origin requests. Our prototype implementation is a server-side proxy that requires no configuration, no changes to the web application and no special browser extensions or modifications on the browser.

1.3.3 WebSheets

Finally, this dissertation presents WebSheets, a new web application development paradigm based on the spreadsheet model that focuses on principled security. Instead of focusing on *black-box* defenses that require no programmer-effort, we take a complementary approach and design a new paradigm to provide security conscious developers the ability to specify security policies that are not scattered across the application logic.

The paradigm has two main goals: firstly, give security a first-class role, making it *easy* for developers to specify and maintain the policy for a web application; secondly, we want to empower users to keep control of their data: instead of signing off their data to the web application, putting its secrecy at the mercy of the web application developers, we let users specify a policy for their own data, which the runtime enforces as the data is manipulated by untrusted parties.

To achieve these two goals, we leveraged the spreadsheet model, augmenting it by associating a permission cell to each data cell and using Mandatory Access Control to automatically enforce the respective permission wherever data from a cell is used for a computation. The choice of basing WebSheets on the spreadsheet model is not arbitrary: if users are to specify their own policies, we need to provide a familiar interface for them.

Our prototype implementation is a Node.js application where users can create and share websheets.

Part I
Black-Box Defenses

2 XSSFilt: Protection, Usability and Improvements in Reflected XSS Filters

This section presents XSSFilt, our defense against reflected and DOM-Based XSS attacks. The reason for focusing on XSS is simple: XSS has consistently been among the top threats for web applications (#1 in the 2007 edition of the OWASP Top 10, #2 in the 2010 edition and #3 in the 2013 edition). In terms of raw numbers, it is one of the most commonly reported vulnerabilities: in 2011, XSS vulnerabilities accounted for 14.7% of all reported CVE vulnerabilities. `xssed`, a famous repository of XSS vulnerabilities, currently hosts more than 40000 individual vulnerabilities. Before, we dwelve into the specifics of our defense, we provide some background information about XSS attacks. In particular, we present a simple example and explain the difference between feflected, DOM-Based and stored XSS attacks.

2.1 Background on XSS Attacks

An XSS attack involves three entities: a *web-site* that has an XSS vulnerability, a legitimate *user* of this web-site, and the *attacker.* The attacker's goal is to be able to perform sensitive operations on the web-site using the credentials of the legitimate user.

Although an attacker is able to run his code on the user's browser, the same-origin policy (SOP) of the browser prevents his code from stealing the user's credentials, or observing any data exchanged between the user and the web-site. To overcome this restriction, the attacker needs to inject his code into a page returned by the web-site to the user. An XSS vulnerability in the web-site allows this to happen.

Exploiting an XSS vulnerability involves three steps. First, the attacker uses some means to deliver his malicious payload to the vulnerable web-site. Second, this payload is used by the web site during the course of generating a web page (henceforth called a *victim page*) sent to the user's browser. The left side of Figure 2 shows an example of a vulnerable page that uses a user-supplied parameter `id` to construct the `href` parameter via `document.write`, while the right side shows an example payload. In this case, the payload does not need to open a new script tag because it is already contained in one; rather, it closes the string where the parameter is supposed to be confined and writes additional JavaScript code.

If the web site is not XSS-vulnerable, it would either discard the malicious payload, or at least ensure that it does not contribute to code content in its output. However, if the site is vulnerable, then, in the third step, the user's browser would end up executing attacker-injected code in the page returned by the web site.

There are three approaches that an attacker can use to accomplish the first step:

- In a *stored XSS attack,* the injected code first gets stored on the web-site in a file or database, and is subsequently used by the web-site while constructing the victim page. For instance, consider a site that permits its subscribers to post comments. A vulnerability in this site may allow the attacker to post a comment that includes `<script>` tags. When this page is visited by the user, the attacker's comment, including his script, is included in the page returned to the user.

- In a *reflected XSS attack,* an attacker lures a user to the attacker's web page, or to click on a link in an email. At this point, the user's browser launches a GET (or, in some cases, POST) request with attacker-chosen parameter values. When a vulnerable web site uses these parameters in the construction of its response (e.g., it echoes them into the response page without adequate sanitization), the attacker's code is able to execute on this response page. The widespread prevalence of spam and scam emails (and web sites) make reflected XSS relatively easy, hence their popularity among attackers. For example, Figure 1 shows how a reflected attack can be carried out on a vulnerable website: maliciously crafted input can open a `script` node in the middle of the page and execute JavaScript code in the context of the web application. This code will thus have access to the domain cookies, and may send them to an external location controlled by the attacker.

- In a *DOM-Based XSS attack,* the attacker also tricks the user into performing a request to the vulnerable site containing a malicious payload. However, the attacker does not exploit a vulnerability in the server-side logic, and the malicious payload is not present at all in the response (or if present, it does not introduce script code). Instead, the attacker exploits a vulnerability in the JavaScript code executed by the browser: the DOM API exposes a handful of properties that can be controlled by the attacker (e.g. `document.location` and `window.name`), and if

Server Code

```
<h1>0 search results returned for <?=$_GET["term"];?></h1>
```

Malicious URL

```
http://a.com/search?term=<script>
  document.location='http://evil.com/' + document.cookie
  </script>
```

Figure 1: Reflected XSS Example

the attacker can trick existing benign code into using this information to create new JavaScript content (e.g. using an existing call to `eval`), he can execute his own code in the context of the web application.

In this thesis, we only address blocking reflected and DOM-Based XSS attacks. Blocking stored XSS attacks effectively requires cooperation from the web server, which defeats the purpose of offering automatic, black-box protection. For a survey of server-side or hybrid defenses that can protect against stored XSS attacks, refer to Section 2.7.

2.2 Limitations of existing filters

The increase in prevalence and severity of XSS attacks has motivated many security researchers to devise XSS defenses. Many of these efforts [63, 78, 10, 118] have focused on the server-side, and attempt to detect (or prevent) unauthorized scripts from being included in the server output. Modern web-browsers incorporate very complex logic to "fix" HTML syntax errors and hence provide an acceptable rendering of syntactically incorrect pages. Several researchers [62, 90, 136, 78] have eloquently argued that no server- side logic can accurately account for all such "browser quirks." As a result, hybrid approaches that combine client-side support with a primarily server-side XSS defense have been developed [136, 90, 122].

Since XSS is a server-side vulnerability, it seems natural to employ a server-side defense. Unfortunately, the party that is most directly affected by an XSS attack is a browser-user that accesses a vulnerable server. Consequently, there may not be enough of an incentive for some web sites to

implement XSS defenses — this is one reason why XSS vulnerabilities are so easy to find. Client-side protections are thus desirable, despite their limitation to reflected XSS.

Microsoft Internet Explorer 8 was the first browser to include a built-in XSS filter. However, the filter did not exploit its vantage point as a browser component to its fullest potential, performing the kind of low-level network interception that would also be possible with a client-side proxy. Because of this, its filtering logic is similar to other server-side reflected XSS filters that are implemented as server-side proxies [63, 118]: firstly, it intercepts HTTP requests that look suspicious; then, HTML responses to such requests are scanned for script content that may be derived from suspicious parameters, and this content is then "sanitized" to prevent its interpretation as a script.

Unfortunately, identification of unsafe content is very hard because of a browser's HTML parsing quirks. Researchers have shown [92] that there are several ways to bypass detection by IE8 filter. Worse, the sanitization technique used in IE8 could be exploited to perpetrate XSS attacks on some sites that weren't previously vulnerable [93]! In particular, the sanitization logic used to neutralize injected scripts caused some other part of the page that was previously interpreted as passive content to be interpreted as a script. Although the specific vulnerability reported in [93] has been fixed, their architecture makes it difficult to rule out that similar vulnerabilities still exist.

XSSAuditor [7] is Google Chrome's more recent XSS filter. Unlike IE8's filter, it employs a more involved architecture that avoids "browser quirks" problems by tightly integrating with the browser's HTML parser, examining new tags as the parser processes HTML content. If script content "resembles" a request submitted by the browser, XSSAuditor removes the tag before it is executed. This simple approach also avoids IE8's sanitization pitfall, since preventing the execution of a script does not change the interpretation of the rest of the page. Moreover, this new architecture can address a small subset of DOM-Based XSS vectors that feed new HTML content to the parser (e.g. `document.write`). However, although XSSAuditor overcomes the main drawbacks of IE8's XSS filter, it does not address the following problems:

- *Whole Vs partial script injection*: XSSAuditor is geared towards detecting the most common form of XSS, where an entire script is injected into a victim page. However, damage can also be effected by altering the structure of an existing script. Figure 2 shows an exam-

Server Code

```
<script>
document.write(
  '<a href="../plugin.php?passed_id=' +
  '<?=$_GET["id"];?>"></a>');
</script>
```

HTTP GET Parameters

```
id:  '); do_xss(); document.write('
```

Figure 2: An example server-side script (abstracted from the popular SquirrelMail web-based email program) with partial injection vulnerability (left) and a malicious parameter value to exploit it (right).

ple abstracted from the web application SquirrelMail, where a GET parameter named `id` is inserted into a `document.write` call in an existing script (left frame). Even though the intent of the developers is to dynamically write an anchor tag, the logic can be subverted to inject arbitrary JavaScript code. Note the similarity of the malicious input (right frame) with those used in SQL injections: first, the string argument previously opened by benign code is closed; then the payload is inserted; finally, tokens are inserted to synchronize the syntax with the rest of the benign script and thus avoid syntax errors. In nearly all cases (including the example presented), the vulnerability allows for arbitrary code injection into the existing script, thus being as severe as a whole script injection.

These sorts of vulnerabilities arise naturally in template-based web application development frameworks and in dynamic web applications in general. Our experimental results (see Section 2.6.3) demonstrate that partial injection vulnerabilities are common, accounting for 8% and 18% respectively in two collections of vulnerabilities. Moreover, as the first generation client-side defenses (against whole-script injection) get deployed, attackers are bound to try evading them through partial script injections.

- *Incomplete DOM-Based XSS vector coverage*: XSSAuditor's architec-

ture can only prevent attacks that inject malicious JavaScript content through the HTML parser. This obviously includes all reflected vectors. However, many DOM-Based vectors introduce new JavaScript code directly from a JavaScript execution context, bypassing the HTML parser. Section 2.6.1 shows that out of 9 DOM-Based vectors, XSSAuditor's architecture can only cover 2, and XSSAuditor's implementation only covers 1.

- *Accurate algorithms for detecting injections*: XSSAuditor uses an exact string matching algorithm to detect components of a web application's output that have been derived from the input request. Character encoding and sanitizations incorporated into typical web applications are performed before string matching. However, this approach does not handle application-specific sanitizations that may take place. A more systematic approach would rely on approximate string matching in order to (a) better cope with application- specific sanitizations, and (b) to more precisely identify the beginning and end of injected strings in the presence of sanitizations.

2.3 Overview of XSSFilt and Contributions

XSSFilt is a new client-side XSS defense that addresses the above-mentioned drawbacks of previous filters. In particular, this dissertation makes the following contributions:

- In Section 2.4.1, we present the architecture of XSSFilt, a browser-resident XSS defense. Unlike previous browser-resident defenses that all relied on exact string matching, XSSFilt uses approximate string matching. (See Section 2.4.2.) This enables our defense to cope with web applications that perform application-specfic sanitizations.

- In Section 2.4.3, we present a set of policies to detect XSS attacks. These policies detect attacks involving injection of whole scripts, or those occurring due to injection of parameters within scripts (partial injections).

- We present a discussion of the full range of attacks possible on XSSFilt, and ensure that our design can successfully defend against these attacks (see Section 2.6.5).

- In Section 2.6, we present a comparative study of the protection and usability of XSSFilt and XSSAuditor. In particular, our evaluation shows:
  - the importance of addressing partial script injections, which accounted for 8% of the 400 vulnerabilities we studied from the XSS repository `xssed.com`, and 18% of the 10K vulnerabilities we discovered on the web using an XSS vulnerability discovery tool that we built.
  - the benefits of using an approximate string matching algorithm over the exact matching algorithm employed by XSSAuditor: our false negatives were decreased five-fold due to approximate matching.
  - that false positives generated by XSSAuditor and XSSFilt are more likely to be symptoms of underlying injection vulnerabilities rather than mere annoyances: 85% of the false positives reported by XSSFilt were in fact caused by an underlying XSS vulnerability.
  - that XSSFilt has a performance overhead of about 2.5%, making it suitable for real-world deployment.

2.4 Design

This section describes the architecture of XSSFilt, the approximate substring matching algorithm used to match inputs and script content despite arbitrary transformations, and the policies that decide whether a match is an attack that must be prevented.

2.4.1 XSSFilt Overview

To illustrate XSSFilt's design, consider the sequence of operations that take place from the time the user's browser submits a request to a web server to the time the web page loads. The steps in this sequence are identified using numbers in Figure 3, and we describe them in more detail below.

In Step 1, the browser submits a request to a web site. This submission may be in response to a user clicking on a hyperlink in a web page or email, or the result of execution of scripts on a page that is currently being displayed.

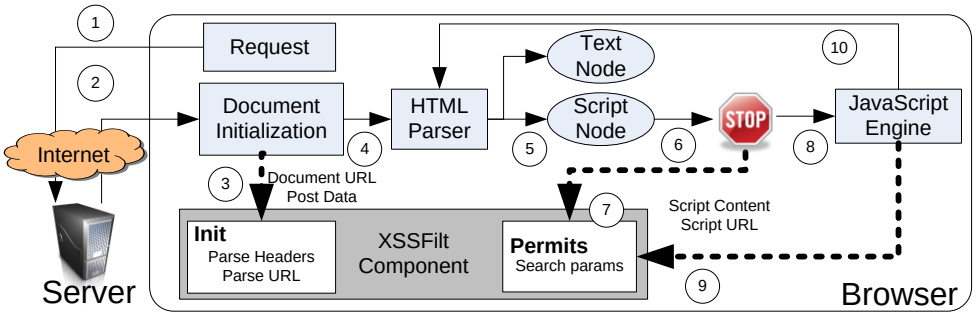

Figure 3: XSSFilt architecture

Either a GET or POST request may be used for submission, and it will include parameter data that is under the control of the web page or email containing the link.

In Step 2, the web site returns a response to the browser's request. This leads to Step 3, when a new document is created by the browser. In this step, the browser invokes the `Init` method of XSSFilt, providing information about the request submitted in Step 1. XSSFilt parses the URL and POST data of all the input parameters and converts them into a list of (name, value) pairs. This step will later enable XSSFilt to detect partial script injections. The filter then returns control to the browser so as to start rendering the page.

In Step 4, the web browser's internal HTML parser is used to parse the document received in Step 2. This causes the creation of various nodes in the document tree, including script and text nodes in Step 5. In Step 6, a script node would normally be sent to the JavaScript engine, but the browser intercepts the script and sends it to the `Permits` method of XSSFilt. At Step 7, XSSFilt uses an approximate substring matching algorithm to search for one or more of the GET/POST parameters inside the script. Any matching content is deemed *reflected* or *tainted*. Further details on this detection technique can be found in Section 2.4.2. If the tainted components of a script violate the policies described in Section 2.4.3 then the execution of the script is blocked. Otherwise, it is handed over to the JavaScript engine in Step 8.

Note that, during the execution of a script, new script content may be created and immediately executed using dynamic code primitives such as `eval`. Our architecture ensures that such newly created code is passed to

the `permit` operation of XSSFilt in Step 9, thus ensuring that dynamically created code is checked for XSS in the same manner as code included statically within `<script>` tags. Similarly, scripts execution can also cause the the creation of new HTML content, e.g., as a result of `document.write` or setting `innerHTML` attribute of some DOM nodes. In all these cases, the HTML parser will be invoked in Step 10, and Steps 5 through 8 will be repeated. Covering both Step 9 and Step 10 ensures complete DOM-Based XSS coverage.

2.4.2 Identifying Reflected Content

Detection of reflected content is a *taint analysis* problem, where the HTTP request supplied by the browser is a source of (tainted) attacker- controlled data, and each script that is either statically present in the HTML response or introduced by JavaScript at runtime is a sensitive sink. The architecture presented in Section 2.4.1 demonstrates how all sensitive sinks are covered, but does not prescribe how taint is propagated from sources to sinks. Server-side XSS defenses can rely on taint-tracking instrumentation on the web application code code for accurate tracking, but this is obviously not a possibility for a browser-resident defense. Thus, the only option is to infer possible taint by comparing the attacker controlled data with script content. A known limitation of such an approach is that if data goes through complex transformations, then there would be no match between the input and output and hence no taint can be inferred. Fortunately, the transformations used by most web applications seem to consist of character encodings and simple sanitizations (e.g. removing spaces). Reference [118] presents an approximate substring matching algorithm that can account for arbitrary changes up to a specified edit distance. In the context of web applications, where the matching process returns similar results as taint-tracking, it is helpful to consider the algorithm as a variant of taint-tracking; thus we refer to the algorithm as *taint- inference*.

The core of our taint-inference algorithm is the same as that of Reference [118]. However, since XSSFilt is embedded in a browser and Reference [118] describes a server-side proxy, there are differences in terms of identifying taint sources, recognizing tainted content, and a few additional optimizations.

1. The URL is parsed into a list of (name, value) parameters. The parameter name is used for reporting purposes, but is of no other interest

to XSSFilt. This decomposition into parameters is necessary to detect partial script injections. If the URL cannot be parsed properly, or if special characters are present in the URL path (or if they span more than one parameter), the entire path is also appended as a single parameter. This step ensures that the technique would not fail for applications that use non-standard parameter encoding, but instead will operate in a degraded mode where it can at least detect whole-script injection.

2. As an optimization, parameters whose content cannot possibly include JavaScript or HTML code are ignored. Specifically, we discard parameters shorter than 8 characters, and parameters containing only alphanumeric characters, underscores, spaces, dots and dashes. These characters are commonly used in benign URLs. When these parameter values are included in the returned page, the resulting content will not match the policies described in Section 2.4.3, but ignoring them will not cause attacks to be missed.

3. Before any inline script is executed, an approximate substring matching algorithm is used to establish a relationship between the parameters and the script. If the parameter is longer than the script, then the script is searched within the parameter, to detect *whole script injection*. On the other hand, if the script is longer than the parameter, then the parameter is searched within the script, to detect *partial script injection*.

   A similar check is performed before an external script is fetched for execution. If the script URL is longer than the parameter, then the parameter is searched within the URL to detect hijacking of existing external scripts, where the attacker is able to point them to a malicious domain. Otherwise the URL is searched within the parameter to account for *whole script injection* of an external script name.

Previous browser-resident techniques for XSS detection, including XSSAuditor [7] and noXSS [60] use *exact* substring matching rather than an *approximate substring matching* to identify reflected content. Another difference is that XSSAuditor does not parse parameters and hence it can only detect those cases where an entire script is injected, while XSSFilt can detect partial script injections as well. The main advantages of XSSAuditor's approach are:

- *Faster runtime performance:* exact substring matching has linear-time complexity, and thus better performance over the quadratic-time worst-case complexity of approximate matching.

- *Lower false positive rate:* This is because (a) exact matching is stricter than approximate matching, and (b) likelihood of coincidental matches for the entire script is smaller than that for any of its substrings.

Our approach, on the other hand, has complementary strengths:

- *Coping with application-specific sanitizations:* Approximate substring matching is better able to cope with application-specific sanitizations that may take place, e.g., when a '*' character is replaced by a space. In contrast, an exact matching algorithm will fail to match even if a single such substitution takes place. Our results in Section 2.6, as well the results of References [4] and [118] show that such application-specific sanitizations do occur in practice.

- *Partial script injections:* As described in the introduction (Figure 2), template-based web application frameworks create natural opportunities where an existing script could be modified by injecting a parameter value into its middle. In this case, there would not be a match for the whole script, and hence XSSAuditor would miss such injections. As we show in the evaluation section, such partial script injection vulnerabilities are relatively common.

Although the results in Reference [118] seem to indicate that the above benefits could be obtained without undue performance overheads or false positives, a more careful examination indicates that those results are not necessarily applicable for a client-side XSS defense:

- The false positive evaluation in Reference [118] was done in the context of SQL injection, specifically on simple web applications. In contrast, a browser-side XSS defense needs to avoid false positives on virtually all applications that have been deployed on the web.

- In terms of performance as well, the results in Reference [118] were obtained using a SQL injection data set. The volume of data subjected to approximate matching and policy checking are thus much smaller than that involved in the rendering of a web page, and hence performance constraints are more stringent for XSS-defense within a browser.

Thus, it was unknown, prior to this work, whether a client-side XSS defense can benefit from the strengths of approximate matching without incurring its drawbacks. Our evaluation answers this question affirmatively. Section 2.6 shows that XSSFilt benefits from the increased power of approximate matching, while minimizing its drawbacks.

2.4.3 XSS Policies

Section 2.4.2 presented an algorithm to compare the content of untrusted sources and sensitive sinks to emulate taint-tracking. If a match is found, the filter must decide whether to allow or prevent execution of the sink. This section details the policies involved in this decision.

Previous research on injection attacks on web applications showed that a few generic policies can detect a wide range of attacks. In particular, Su et al [129] proposed the *syntactic confinement policy* that confines tainted data to be entirely within certain types of nodes of a parse tree for the target language (e.g., SQL or JavaScript). A lexical confinement policy has been used successfully by others [118]. However, these works primarily targeted SQL injection, which is relatively simple. In contrast, XSS is more challenging due to the diversity of injection vectors and the many evasion techniques available to attackers. Below, we describe policies that address these difficulties in a systematic manner.

Policy for Inline Code

This policy is used for protecting against XSS attacks embedded in inline content. Specifically, the following types of content are addressed:

A. Inline code: This category includes code injected in the web page using one of the following mechanisms:

i. *Inline scripts:* Script content, enclosed between `<script>` and `</script>` tags

ii. *Event listeners:* Code enclosed in an event handler specification, e.g.,
`<a onclick="alert(...)">`

iii. *JavaScript URLs:* Code provided using JavaScript protocol, e.g.,
`<a href="javascript:alert(...)">`

iv. *Data URLs:* These provide a general mechanism to include inline data (e.g. text, images, HTML documents) using a URL. Because they support base64 encoding, it is important to perform taint-inference on the original base64 encoded string. For example, `<object data="data:text/html;base64,PHNjcmlwdD5hbGVydCgiSGVsbG8iKTs8L3NjcmlwdD4="></object>` inlines a page containing `<script>alert(1)</script>`.

Note that, as shown in Figure 3, the vectors covered by this policy can be both reflected (Step 4, feeding into Step 5) and DOM-Based (Step 10, feeding the attack back to Step 5).

B. Dynamically created code: New code may come into being when a value stored in a variable is `eval`'d or used in an operation such as `setTimeout`.

The simplest (and most restrictive) inline policy is one that prohibits any part of a script from being tainted. Unfortunately, this policy produces many false positives because it is common for scripts to contain data from HTTP parameters. For this reason, we implemented a lexical confinement policy that restricts tainted data to be contained entirely within a limited set of tokens. In practice, we discovered that the data injected is normally inside strings. All of the attacks in our dataset consisted of such injections within strings. We therefore specialized the policy to ensure that tainted data appears only within string literals, and does not extend before or after the literal.

Policy for External Code

This policy is enforced on external code that is specified by a URL, i.e., injection vector (C) described below.

C. External code: Code that is referenced by its name using one of the following mechanisms:

i. *External scripts:* Script name provided using a script tag, e.g., `<script src="xyz.js"></script>`

ii. *Base tags:* These can be used to achieve an effect similar to external script injection by implicitly changing the URL from where scripts in the document are loaded.

iii. *Objects:* Similar to external scripts, but embedded between `<object>` and `</object>` tags.

The following policy is applied to the name of external scripts or objects:

1. If the host portion of the URL is untainted, then the script is allowed. Note that an attacker cannot typically upload a malicious script onto a server controlled or trusted by a web application. For this reason, the attacker needs to control the host portion of the URL.

   Unlike the host component, our policy permits the path component of a URL to be tainted, since some web applications may derive script names from parameter values.

2. Even if the host portion of the URL is tainted, our policy permits the script if it is from the same origin. We use a relaxed same-origin check [28] which verifies if the registered domains of the URLs match. Thus, `www.google.com` is considered same-origin with `reader.google.com`.

3. Finally, if the tainted domain was previously involved in a check that was deemed safe, then it is allowed. Intuitively, XSSFilt assigns trust on a per-domain basis, and considers all requests from the same domain as trusted or untrusted.

Note that DOM-Based vectors are covered here as well. For example, a vulnerable script that performs
`document.write("<script src="evil.com/xss.js"></script>")`
will eventually trigger a check for external scripts (Step 10 → Step 5 → Step 6 from Figure 3)

2.5 Implementation

The first prototype of XSSFilt was implemented by modifying the Content Security Policy (CSP) [122] implementation in Firefox 4, which allowed us to leverage its existing interposition callbacks throughout the Firefox codebase. CSPs implement the `nsIContentSecurityPolicy` interface, which is used to a) check if the URL of external resources being loaded is included in a whitelist, and b) check whether inline scripts are enabled in the CSP for the current page, denying their execution regardless of their content. Since

XSSFilt needs to decide whether to allow or deny inline scripts based on their content, we modified the existing CSP callbacks to pass the script content to `permits*` calls where appropriate. We also added new callbacks for `base` elements and Data URLs. Note that Firefox's CSP implementation was a JavaScript prototype at the time; we implemented most of our high-level functionality in JavaScript, but, for performance, we implemented the taint-inference algorithm in a separate C++ XPCOM component. The results from Section 2.6 were collected using this implementation, which is available for download on the XSSFilt project page [107].

After the CSP-based prototype was developed, the author was hired by Mozilla Corporation for a summer internship, to investigate the possibility of merging the filter into the official Firefox codebase. For this purpose, the filter was reimplemented in a more principled fashion, providing its own interface and source files. To optimize its performance, we implemented the entire filter in C++. Our work resulted in a patch for Firefox 17 for bug #528661 [61]. Besides the open-source patch, we have released a patched Firefox binary that includes the filter [107].

2.5.1 Deployment on the PaleMoon browser

Unfortunately, the process got stuck in review phase, and the patch never made it into a stable release of Firefox. However, the main developer of a popular Firefox fork, PaleMoon [82], expressed his interest in late 2015 in integrating the patch into its codebase, which is based on Firefox 24 ESR. Besides dealing with minor bitrot (from Firefox 17 to 24), we added the following two features:

- *whitelisting for dynamically loaded scripts.* While compatibility testing of our reference implementation showed that XSSFilt is practical because false positives are few and far between, deploying it to a large userbase (∼500k users) requires more attention to the issue. In particular, beta testers discovered two false positives, one in Yahoo Mail and one in the Twitch player.

  The Yahoo mail false positive follows the *loader* pattern: a query parameter is used by a script on the page as a URL for an external script tag. While our experimental evaluation from Section 2.6.2 shows that many sites implement this pattern insecurely and allow an attacker to load a malicious script, our analysis of Yahoo's code did not uncover a

vulnerability. In this case, a regular expression securely limits the hostname to a handful of safe domains. To work around this false positive, we implemented the necessary logic and UI elements to whitelist domains that *loader*s use to host their javascript files. Whitelisting target domains instead of the loader itself is safer because, if a vulnerability is ever found in the loader script, the filter will still prevent malicious scripts from executing. Note that this is not a spurious flow caused by the use of taint-inference instead of taint-tracking, but rather an actual flow that only happens if the script URL is benign. Even filters that use real taint- tracking, such as Reference [126], would flag this (and most other *loader* implementations) as a false positive, because they cannot capture how the flow is present, yet safe.

The Twich player false positive follows the related *sandbox* pattern, which consists of an purposedly unsafe of the *loader* pattern: since the *loader* script runs on a third-party domain that contain no data worthy of an exploit, an attacker that exploits the *loader* vulnerability and loads his own content into the domain can cause no harm. For example, embedding the Twitch player into a site causes a third-party iframe from a CDN (*cdn.embedly.com*) to be embedded in the page. The frame contains a vulnerable loader script which uses a query parameter as a script URL, which can be straightforwardly modified to load a malicious script. However, the Same-Origin Policy restricts the attack surface to the CDN domain, which doesn't contain interesting credentials.

- *whitelisting for incompatible sites*: a handful of sites contain server-side logic that cause false positives even in the absence of an attack. For example, Google Translate takes the URL of the page to translate as a parameter and rewrites the translated page with an additional `BASE` tag so that relative links do not need to be rewritten. Because the `href` attribute of the `BASE` tag is often similar (or identical) to the URL of the translated site (e.g. `http://www.example.com` vs. `http://example.com/example.html`), the filter believes that the `BASE` tag has been injected. XSSAuditor does not recognize this as an attack, because it is more tightly integrated with the HTML parser: the filter can check for additional conditions in the parsing context, such as whether the match extends past the URL and includes the `href` attribute or the `BASE` tag. However, this also makes it harder for them to recognize DOM-Based XSS attacks that do not feed malicious data

back through the HTML parser (i.e. anything but `document.write` and `Element.innerHTML`). Because these sites are rare, although we added support for a whitelist, we did not add the UI to let users add new sites to the whitelist, which the PaleMoon staff plan to maintain themselves.

After addressing these issues, we successfully shipped the filter turned on by default in PaleMoon version 26 [82].

2.6 Evaluation and Comparison

In this section, we first evaluate and compare the protection and compatibility offered by XSSFilt and XSSAuditor, showing that XSSFilt protects against more attacks while not suffering from more false positives. Then, we demonstrate that partial script injections are a realistic threat; finally we evaluate XSSFilt's performance and discuss how our architecture covers all vectors.

Note that to test XSSAuditor without having to instrument another browser, we did not use the original implementation included in Google Chrome. Instead, we reimplemented a stricter version of its policies within our filter and then manually confirmed false positives and false negatives.

2.6.1 Protection Evaluation

To compare the protection offered by XSSFilt and XSSAuditor, we tested them against 4 sources of XSS attack data, most of which have been used in previous research.

xssed: `xssed.com` [37] contains reports of websites vulnerable to XSS, along with a URL for a sample attack. Since the dataset is very large, we randomly selected a subset of 400 recent, working attacks among these in order to estimate the effectiveness of our filter against real-world attacks.

cheatsheet: the `xssed` dataset is biased towards very simple attack payloads, since most of them simply inject a script tag. To assess the filter's protection for more complex attacks, we created a web page with multiple XSS vulnerabilities and tried attack vectors from the XSS Cheat Sheet [50], a well-known and oft-cited source for XSS filter circumvention techniques.

Dataset	XSSFilt	XSSAuditor
xssed	399/400	379/400
cheatsheet	20/20	18/20
dexterjs	25/25	23/25

Figure 4: Results for the `xssed`, `cheatsheet` and `dexterjs` datasets

dexterjs: the `xssed` and the `cheatsheet` datasets are both heavily biased towards reflected XSS. As a matter of fact, they contain no DOM-Based XSS attacks at all! To rectify this, we leveraged DexterJS's [99] dataset, a recently published DOM-Based XSS scanner. The authors have made a set of 820 attacks on 89 domains publicly available, but these have since been fixed. However, the authors also took snapshots of the vulnerable websites at the time of discovery using HTTrack and were kind enough to share the snapshots with us. Although only a subset of vulnerabilities can be replicated on a local server, we were able to find 25 distinct working vulnerabilities among them that worked on both Chrome and Firefox[1]

domxsswiki: like `xssed`, the `dexterjs` dataset is also biased towards simple vulnerabilities. Therefore, to evaluate the coverage of all DOM-XSS vectors, we used the `domxsswiki` [125] as a reference to identify 9 distinct DOM-Based vectors that work on Firefox, and generated 9 attacks employing these vectors.

To automatically test this large number of attacks, we modified XSSFilt to log XSS violations to a file and wrote a Firefox extension to automatically navigate the browser to all the URLs in the datasets.

Figure 4 summarizes the results for the first three datasets. XSSFilt successfully stopped all but one of the attacks from the `xssed` dataset and all attacks from the `cheatsheet` and `dexterjs` datasets. Its lone failure is attributed to a limitation of taint-inference previously explained: when

[1]The attacks actually only work on Chrome and IE, because the three major browsers have inconsistent behavior about URL fragment encoding: the fragment portion of `location.href` is returned *urlencoded* in Firefox, and *urldecoded* in Chrome and IE[134]. To make them work in Firefox, we leverage the instrumentation capabilities of Jate [135] and configure `location.href`'s and `document.URL`'s getters to decode their result before returning it.

Vector	XSSAuditor	XSSFilt
eval	No	**Yes**
Function	No	**Yes**
setTimeout	No	**Yes**
script.src	No	**Yes**
script.text	No	**Yes**
location	No	**Yes**
location.href	No	**Yes**
document.write	**Yes**	**Yes**
node.innerHTML	No	**Yes**

Figure 5: DOM-Based XSS Vector Coverage

the web application applies extensive string transformations, the matching algorithm might fail to find a relationship between the parameter and the content. In this specific example, the filter failed because the parameter:

```
alert("HaCkEd By N2n-HaCkEr     -   3rd@live");
```

was transformed into

```
alert("HaCkEd N2n-HaCkEr 3rd@live");
```

by the web application. Some (but not all) spaces and dashes had been deleted, along with the word "By". In these situations, no client-side filter can realistically be expected to detect the attack.

The 100% coverage on the `cheatsheet` dataset is not surprising: these attacks are designed to bypass server-side sanitization functions, which look for specific patterns in text and are vulnerable to browser quirks and unusual XSS vectors. Since this filter architecture is immune to browser quirks and covers all vectors uniformly, none of these attacks succeeded.

The table also shows 100% coverage on the `dexterjs` dataset. Together with the results from table 5, which show the result of the `domxsswiki` dataset, it provides support to our assertion that XSSFilt covers all DOM-Based XSS vectors.

XSSAuditor missed many more attacks in these datasets. Figure 6 shows the underlying causes:

Partial Script Injection: XSSAuditor does not detect this type of attack because, unlike XSSFilt, it does not perform URL parsing and substring

Dataset	Partial Script Injection	String Transformation	Unhandled DOM-Based Vectors
xssed	**16**	**5**	0
cheatsheet	**2**	0	0
dexterjs	0	0	**2**

Figure 6: XSSAuditor failures

matching, and cannot search for a substring of the URL in the middle of a script.

String Transformation: XSSAuditor relies on canonicalization to account for common string transformations in web applications. This approach can break when an uncommon transformation takes place. Taint-inference relies on approximate substring matching, which is more tolerant of exceptions.

Unhandled DOM-Based Vectors: Although the `dexterjs` dataset is biased towards the most popular DOM-Based XSS attack vector, `document.write`, the only one that XSSAuditor can detect, the dataset also contains two attacks that use different vectors, which XSSAuditor is unable to detect.

2.6.2 Compatibility Evaluation

Browser-resident reflected XSS defenses restrict the capabilities of browsers with respect to content found in input parameters, such as GET parameters from the querystring. As a result, they have the potential to break some web pages, and thus lead to compatibility problems.

Types of False Positives

False positives can be accidental, represent benign injections, or may be specifically induced by the attacker. We discuss these three cases separately.

Accidental False Positives: There is a small chance that JavaScript content is incorrectly associated with benign parameters, which can result in degraded user experience. For such an event to happen, two conditions must be met:

1. The taint-inference must report a match between an input parameter and JavaScript content or between an input parameter and an URL pointing to a JavaScript resource. Because taint-inference does not track taint explicitly inside the web application, similarities in input and output values can result in an accidental match.

2. The match must violate the policies discussed in Section 2.4.3; in particular, the matched string must contribute JavaScript content to the page.

Even though accidental matches can occur, these two conditions are unlikely to be met for the same piece of content, since JavaScript code represent a fraction of content of HTML pages, and user input does not commonly resemble script content. The results presented next support this assertion: accidental false positives are not responsible for any of the 8 false positives reported. No "random" matches were observed: all violations represent a real flow of information, which was restricted to benign values by a whitelist check. Unfortunately, a black-box filter (or a filter based on full-fledged taint-tracking, for that matter) cannot reason about the safety of these checks.

Benign Injections: XSS filters prevent injections under the assumption that all injections are malicious and that they can always cause harm, and XSSFilt is no exception. However, this assumption does not always hold. In rare cases, the policies from Section 2.4.3 can stop an injection that is intended by the web application developer. Although the results from this section show that developers often misuse this feature, a developer who is aware of the threat can support benign injections safely, and the XSS filter can get in the way and cause false positives. The two most common cases are the *loader* and the *sandbox* patterns: in the former, an input parameter is used to load an external script; if the location of the external script is unrestricted, the developer shot himself in the foot and the XSS filter would correctly prevent this unsafe behavior; however, if the location of the external file is checked against a whitelist or restricted to a benign host, then the web developer can safely inject the new script into the existing page, and the XSS filter would disrupt the pattern causing a real false positive. The latter pattern is implemented by executing the benign injection from a throwaway origin that contains no sensitive data on either the browser or the server. In this case, the filter would block the injection that the developer implemented

under the correct assumption that it would cause no harm, thus causing a false positive.

Induced False Positives: A drawback of all black-box approaches for taint computation is that the attacker can trick the filter into inferring a relationship between a spurious parameter and a script which is always present. Unlike accidental false positives, which can at most degrade the user browsing experience, or false negatives (the filter missing an actual XSS attack), which result in a browser no more insecure than a similar browser without any XSS defense, induced false positives can potentially lead to new security vulnerabilities that do not affect a browser without XSS defenses. For example, it may be possible to use induced false positives to disable a framebusting script, which is included in HTML pages to protect against clickjacking attacks. Reference [111] describes such an attack for IE8 and XSSAuditor, proposing a non-vulnerable framebusting script. Unfortunately, this is an inherent weakness of black-box XSS filters.

Tests

To estimate the compatibility of XSSFilt and compare it with XSSAuditor, we instrumented Firefox to log information about XSS checks while performing a crawl starting from a custom set of 120 URLs[2]. We developed our own crawler as a Firefox extension, which allowed us not only to support discovery of dynamically constructed links and forms, but also to check all the resources loaded by the web page (including scripts and advertisements inserted through DOM manipulation) for XSS violations. We also ran the test using XSSAuditor's policies.

Overall, both filters reported a moderate number of false positives. However, most of them were either due to a URL being supplied as a parameter and then used by an existing script to construct a new script tag (the aforementioned *loader* pattern), or by a parameter being directly passed to a string- to-code function such as `eval`. These practices would be safe if the application code checked the value against a whitelist of pre-approved URLs for the former case or JavaScript snippets for the latter, and the violation could be indeed considered a false positive. However, we found that out of 51 XSSFilt notifications, only 8 did such checks; the remaining violations were in fact due to vulnerable pages that could be subverted to load a script

[2]The set of URLs can be found at http://pastebin.com/kYqas9ae

Filter	XSSFilt	XSSAuditor
# of violations	8	6

Figure 7: Compatibility Comparison

from an arbitrary host or execute arbitrary code. This set of pages include important websites such as `wsj.com`, `weather.com` and `tripadvisor.com`. For this reason, we do not consider these scenarios as false positives, and we discounted them from the results shown in Figure 7.

2.6.3 Partial Injection Prevalence

Compared to XSSAuditor, XSSFilt is able to detect partial script injection vulnerabilities. Therefore, it is important to assess how prevalent these are. To estimate their prevalence, we used three different methods.

Partial Injections in `xssed.com` Data Set: Out of 400 real-world live XSS attacks, 4% were targeting partial injection vulnerabilities. We analyzed the rest of the vulnerable pages attacked through whole script injections to discover if they contained partial injection vulnerabilities as well, and we discovered that an additional 4% of pages are vulnerable, for a total of 8% of pages vulnerable to partial script injection.

Thus, even though the coverage against *attacks* on the xssed dataset for XSSAuditor was 95%, the actual coverage on *vulnerabilities* is lower at 91%. However, the size of this dataset is quite limited for the purpose of extrapolating statitics about the nature of XSS attacks in general. Moreover, the website does not review submissions and does not reward contributors for creative or complex attacks. For this reason, the dataset is biased towards simple vulnerabilities that can be discovered automatically.

Partial Injection Vulnerabilities in the Wild: We have developed a tool/scanner called gD0rk [105] to study the prevalence of XSS vulnerabilities in real-world sites. Although gD0rk was developed independently from XSSFilt, we believe that it is very helpful for assessing the prevalence of partial injection vulnerabilities:

- gD0rk analyzed a much larger collection of web sites as compared to `xssed.com` data, and hence provides a broader basis for drawing inferences about vulnerabilities in deployed sites.

- gD0rk uses a mechanical procedure for finding vulnerabilities, with no built-in bias for either whole or partial script injections.

gD0rk uses Google's advanced search capabilities to guide its crawl to candidate web sites that are more likely to be vulnerable, probes them for reflected content by modifying the URL, and examines the context in which the content is injected in the web pages returned to build an attack. The exact details of the tool are not important for the purposes of this dissertation, but we do want to note that it is sophisticated enough to a) understand the syntactic context in which the input parameter is found in the HTML output, to generate an appropriate payload, and b) detect and circumvent many sanitizations performed by web applications where possible. For example, if a GET parameter is reflected inside a JavaScript string in a `script` tag, the scanner attempts to write `"; payload(); //` through the filter to exploit the partial injection vulnerability. However, if the application sanitizes double quotes, the scanner attempts to close the script tag instead and open a new script node with `</script><script>payload();</script>`.

We ran gD0rk for one month and identified 272,051 vulnerable websites. For scalability and performance reasons, we did not validate the generated attacks for all these vulnerabilities. Instead, we used statistical sampling to estimate the fraction of these sites that were actually vulnerable. In particular, a random subset of 1000 vulnerabilities among these were selected, and then we were able to verify that 98% of the generated attacks worked on this subset. We then selected a random subset of 10000 vulnerable websites and used the scanner to identify the context of the vulnerable reflections. We found that 18% of these reflections were included within `script` tags or event handlers, and thus represent partial script injection vulnerabilities.

Dynamically Generated Scripts: Intuitively, the necessary requirement for a partial injection vulnerability is a script that is assembled dynamically from input parameters by the web application. We believe that it is reasonable to expect developers to fail to sanitize parameters which appear inside scripts just as often as they fail to sanitize them anywhere else in the page. Under this hyphothesis, the rate of pages that construct scripts dynamically is a good estimator for the ratio of partial injection vulnerabilities to whole script injection vulnerabilities. The benefit of this indirect approach over the previous one is that the dataset is not made out of vulnerable pages, which represents a skewed sample from mostly unpopular websites.

For this reason, we reused our Firefox crawler from Section 2.6.2 and

seeded it with the 1000 most popular websites according to the Alexa rankings. When the crawler processes a page with non-trivial HTTP parameters and detects that a parameter appears in a script, it substitutes the parameter value with a placeholder, requests the page with the newly constructed URL and then attempts to find the original value and the new placeholder in the response. If the placeholder is found in the same script and the original parameter value is not found, then the relationship between script and parameter is confirmed and the page is marked as containing a dynamic script. When we stopped the crawler, it had crawled a total of 35145 pages, of which 9% contained dynamically generated scripts. Given the strictness of the requirements, we believe this is a conservative estimate.

2.6.4 Performance Evaluation

Unlike the protection and compatibility evaluations from Section 2.6.1 and 2.6.2, the performance evaluation is focused on XSSFilt only. XSSAuditor's performance has already been evaluated in Reference [7].

Calculating the overhead imposed by the filter requires a dataset that represents real-world websites. This is because XSSFilt contains many optimizations that bypass policy enforcement if parameters do not contain special characters, if they are too short or if an external script is fetched on the same origin of the page.

The Mozilla codebase includes a performance test for regression testing: tp4 is an automated test that can be run on patches to the mozilla codebase, to estimate the overhead that these can impose on the browser. The test estimates the time required to load a set of predefined pages that are saved locally to produce consistent results over time. The overhead estimated by tp4 for XSSFilt is negligle. Unfortunately, since tp4 fetches homepages saved locally, it overestimates the effect of the filter's optimizations: requests don't have parameters to check, and all external scripts are from the same origin.

To produce more meaningful results, we measured the load times on the real- world dataset from Section 2.6.2 using `pageloader`, the Firefox extension that is used internally by tp4. We used an aggressive caching proxy to factor out network delay while transparently providing pageloader with remote resources. This way, we can avoid overestimating the speedup due to XSSFilt's optimizations. The test showed an overhead of 2.5%.

However, even though the dataset clearly triggered many XSSFilt checks, the figure is not necessarily representative of the overhead normally expe-

rienced by users, because this is a) heavily dependent on the amount of parameters in web applications and b) ultimately diluted when factoring in the delay involved with fetching a webpage off the network. For this reason, we used profiling data available from an ordinary user session consisting of 3000 unique pages. Since a web browser is a multithreaded execution environment, the overhead cannot be estimated by simply timing the calls to XSSFilt: the same call will take longer if the user is simultaneously watching a video on YouTube on a different tab. Therefore, the profiler logged the actual parameters of XSS checks, and given that the only expensive operation is approximate substring matching, we can perform the approximate string matching computations offline to estimate the time spent during XSS checks for each page load. This yields an average overhead of 0.5%, which shows that the overhead is almost negligible when factoring in network latency.

2.6.5 Security Analysis

In this section we identify possible attack vectors and strategies that may be used for an XSS attack, and argue how our design addresses these threats.

We expect an attacker to deploy the full range of techniques available to evade detection by XSSFilt. There are two logical steps involved in the operation of XSSFilt: (I) recognizing script content in the victim page, (II) identifying if this code is derived from request data. The following techniques may be used to defeat Step I:

1. *Exploit all of the previously mentioned vectors to inject code,* hoping that one or more of the vectors may not be (correctly) handled by XSSFilt. However, the filter architecture makes it simple to enforce complete mediation [113], since there are only a small number of code paths that call into the JavaScript engine.

2. *Exploit various browser parsing quirks* to prevent XSSFilt from recognizing one or more of the scripts in the victim page. Note, however, that browser quirks pose a problem for techniques that attempt to detect scripts by statically parsing HTML. In contrast, XSSFilt operates by intercepting scripts dispatched to the Javascript engine, and hence does not suffer from this problem.

3. *Exploit DOM-based attacks:* If the victim page uses a script to dynamically construct the page, e.g., by setting the `innerHTML` attribute, then

try to defer script injection until this time. This technique defeats filters that scan for scripts at the point the response page is received. However, XSSFilt's architecture ensures that all scripts, regardless of the time of their creation, are checked before their execution. Hence, XSSFilt is not fooled by DOM-based attacks.

4. *Exploit sanitization to modify the parse tree* and force the parser to interpret another part of the page as script. This vulnerability existed in Internet Explorer [93], whose filter deactivated script nodes by modifying the `<script>` tag. In contrast, XSSFilt's decision to block the execution of a script has no effect on how the rest of the page is parsed.

To defeat Step II, an attacker may use the following techniques:

5. *Employ partial rather than whole script injection:* We have already described how our taint inference implementation (Section 2.4.2), together with the policies described in Section 2.4.3, can detect partial injections.

6. *Employ character encodings* such as UTF-7 to throw off techniques for matching requests and responses. Note that by the time a browser interprets script content, it has already determined the character encoding to be used. Therefore XSSFilt can apply the corresponding decoding operation before the taint inference step, and thus thwarts this evasion technique.

7. *Exploit custom sanitizations performed by an application* to evade taint inference. As described before, XSSFilt uses approximate substring matching, which provides a degree of resilience against application-specific sanitizations employed by web applications. However, if a web application makes extensive use of non-standard character transformations, it may be possible to exploit them to evade XSSFilt. It seems unlikely that any purely client-side defense can address this evasion. Moreover, note that the attacker cannot induce such behavior on arbitrary applications — he can only exploit applications that already perform extensive, non-standard transformations.

8. *Employ second order attacks* that operate by injecting malicious parameters into links or forms contained in the victim web page. An XSS

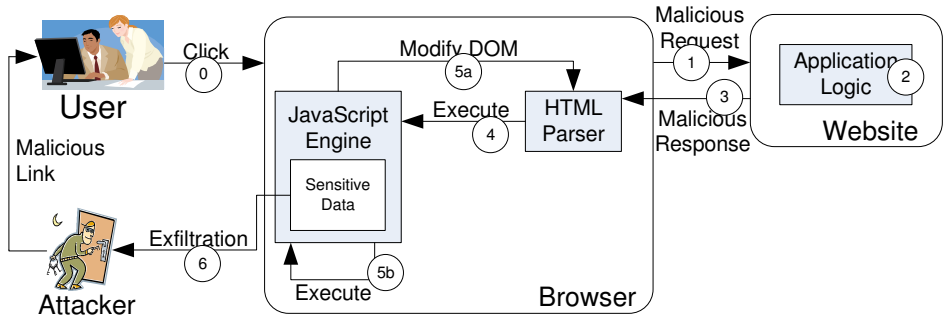

Figure 8: Steps of an XSS Attack

attack would be effected when these forms are subsequently submitted. Note, however, that XSSFilt will apply policies to these submissions as well, and hence detect second (or higher order) order attacks. IE has an exception for same-origin links and would be vulnerable to this attack.

2.7 Related Work

This section is divided into two parts: firstly, we identify the steps required to launch a successful XSS attack and organize XSS filters according to how they interact with an ongoing attack to prevent it (this section also includes defenses that published after XSSFilt). Secondly, we review new surveys and attacks that have advanced the state-of-the-art since our publication.

2.7.1 Systematization of XSS Filters

A successful reflected or DOM-Based XSS attack requires the following:

- a malicious parameter value sent to the server,
- creation or alteration of scripts included in the server page,
- actions taken by this script that compromise the victim.

XSS defenses protect against XSS attacks by stopping any of the three conditions mentioned above. Figure 8 shows a more detailed depiction of the process that leads to a successful XSS attack, splitting the process into 6

distinct steps. The purpose of this formalization is to organize XSS defenses by how they interact with an ongoing attack during these steps. The steps are:

Step 0 (Dissemination): The user's browser is tricked into performing the malicious request. In Figure 8, we assume the user clicked a malicious link supplied by a Spam email, but other possibilities exist; for example, malicious advertisements can automatically redirect the user and launch the attack[3]. We label this Step 0 because preventing this step is out of scope, and our threat model assumes that attackers always find a way to trick the user's browser into sending the malicious request.

Step 1 (Malicious Request): The browser sends an HTTP request containing the attack to the target web server. Because there is a confused deputy component to XSS, the browser or the request have no way to detect and communicate to the server that the request was unintended or outright malicious. However, a potential XSS attack can be stopped in this step using heuristics, assuming that the outgoing request contains something that looks like an XSS payload.

Step 2 (Faulty Application Logic): The application logic unsafely consumes untrusted input and builds an HTML response containing the attack. XSS attacks can be stopped during this step by performing appropriate sanitization on inputs, a deceptively difficult problem. Alternatively, some defenses might collect information during this step but wish to delay actual enforcement to Step 3 or Step 4, and either commmunicate to the client how inputs made their way into the output (i.e., marking certain sections of the page as untrusted), or, using a complementary approach, endorse certain parts of the output as trusted.

Step 3 (Malicious Response): The web server returns a malicious response to the browser. Reflected XSS attacks can be stopped at this step by checking if any input ends up in the HTML output and could be interpreted as a script. Note that Step 3 can be performed on the client-side (e.g. client-side proxy), transparently over the network

[3]Historically, the SOP has always considered redirecting the top frame a necessary nuisance and not a security vulnerability

(e.g. corporate interception proxy) or on the server-side (e.g. web application output filter). Note that DOM-Based XSS attacks are not observable at this step, because they only manifest themselves after the JavaScript engine consumes untrusted input in an unsafe way.

Step 4 (Malicious Script Execution): The malicious script contained in the response is parsed by the HTML parser and its content is sent over to the JS engine for execution. If the server sent trust information to the client along with the response, the trust level of the script can be checked here and XSS attacks can be stopped. If no trust information is present, the attack can be stopped by looking for the current script about to be executed in the input. Even if a script is benign, it might contain DOM-Based vulnerabilities, and an attack can happen during its execution (see 5a and 5b).

Step 5a (Vulnerable DOM Operation): A benign script can also modify the DOM, adding new script content (e.g. `document.write`). If the malicious script is inserted during this step, then this is defined as a DOM-Based attack. An XSS attack can be stopped here by blocking new script nodes; however, because new script content must eventually go through Step 4, defenses can simply rely on existing defenses on the execution path.

Step 5b (Malicious Dynamic Code Execution): A benign script can also add new script content directly (e.g. `eval`). If the malicious script is inserted during this step, this is also known as a DOM-Based attack. Unlike Step 5a, malicious scripts introduced from Step 5b do not go through Step 4, so XSS defenses must stop most DOM-Based attacks here.

Step 6 (Malicious Payload): At this point, we assume that the attacker is executing its own malicius JavaScript code on the user's browser. Not all is lost: an XSS defense can restrict the runtime behavior of the malicious script, preventing specific actions (e.g. exfiltrating cookies for session hijacking). Note however, that any runtime restrictions must apply to benign code as well, since malicious code is indistinguishable from benign code at this stage.

Even though each defense has its own unique twist on how to recognize

and stop XSS attacks, they can be organized into the following categories, according to where in Figure 8 they check for XSS attacks.

Preventing Malicious Requests (Step 1): A filter using this approach blocks, prevents or sanitizes HTTP requests that are deemded dangerous. The main drawback of this type of filter is that it must make a purely speculative decision about the nature of the URL with minimal information: because the request hasn't even been sent yet, the filter does not know how the web application uses the suspicious URL in the response (e.g. if is sanitized, or if it is present in the response at all!).

The only filter in this category is included in NoScript [43], a popular Firefox plugin which allows users to execute JavaScript only on trusted websites manually added to a whitelist. To detect suspicious content in outgoing URLs, its XSS filter relies on regular expressions. For example, if a URL contains a `script` tag in a parameter, the filter sends out a sanitized request without the tag. Since the filter cannot actually check if malicious data is actually present in the response, it can suffer from a higher rate of false positives compared to other approaches.

Sanitization routines (Step 2): While server-side sanitization is a necessary first line of defense and is not complicated for simple inputs (e.g. only allow alphanumeric characters), allowing complex inputs (e.g. WikiMedia, BB Code) while disallowing scripts is prone to errors.

Several works have highlighted how simple taint-tracking to verify that "blessed" sanitization functions are applied to untrusted inputs [67] is inadequate, because it puts too much trust into the correctness of sanitization functions [3, 115, 55]. ScriptGard [115] assumes the existence of sanitization functions that are safe *for a specific context* (e.g. writing inside an HTML attribute) and identifies two sanitization errors (besides the more obvious lack of sanitization altogether) that can lead to vulnerabilities: *context-mismatched sanitization*, where the wrong sanitization function is applied, and *inconsistent multiple sanitization*, where sanitization routines are composed incorrectly. During a training phase, ScriptGard uses taint-tracking to check if the right sanitization is applied to untrusted data flowing to a specific sink. The correct sanitization routine is then patched into the web application and performed at runtime.

Bek [55] describes a restricted language for writing sanitizers that allows static analysis of the sanitizer's behavior. For example, given a sanitization

function Bek can answer queries such as "can you construct an input that produces the following output?", to check if malicious strings can circumvent sanitization. The author's intuition is that, unlike application logic, sanitization functions do not need to be written in powerful languages that are hard to analyze. Instead, the authors argue that Bek is sufficiently expressive to model real-world code and yet restricted enough to be amenable to static analysis.

Blueprint [78] presents a server-side sanitization routine which converts HTML into JavaScript code that inserts the original HTML content into the page using the DOM API. The purpose of this transformation is to fix the parse tree of the page (i.e. its dynamic interpretation) on the server-side, detecting and filtering script content on the DOM representation and generating JavaScript code to reliably reconstruct the sanitized parse tree once the code is rendered by the browser. Unlike traditional server-side filters, this makes Blueprint immune to browser quirks.

Recognizing Injection into HTML (Step 3): A filter using this approach prevents XSS attacks in three (sub)steps, all of which are non- trivial and prone to bypasses and issues. In Step 3.1, the filter uses string matching to detect which parts of the HTTP response are made out of the inputs from the HTTP request; if the application performs input sanitization, the matching might fail. In Step 3.2, the filter must detect which parts of the HTML response will be parsed as script content by the browser, either using a similar parser or using other heuristics; any minimal difference in parsing behavior, even regarding non-standard recovery of errors (browser quirks) can lead to bypasses or false positives. In Step 3.3, if an input parameter is found in a script region of the HTML file, the script must be sanitized; an attack might survive sanitization, or the sanitization itself might change how the rest of the page is parsed, re-activating a broken attack. Also, note that filters using this approach cannot possibly deal with DOM-Based XSS attacks, since there is no DOM representation of the page at this stage.

XSSDS [63] describes two different server-side filters: an XSS filter for stored and reflected XSS attacks that builds a whitelist of scripts using training, and a reflected XSS filter based on string matching. To be more resilent against browser quirks, the filter performs step 3.2 using the Firefox parser, which is at least able to defeat browser quirks against Firefox clients; however, this parser adds significant overhead and cannot reliably handle browser quirks for other browsers.

Reference [142] is a server-side filter that uses taint-tracking to protect against a wider range of injection attacks (SQL Injection, OS Command Injection, etc). In the context of XSS, it greatly simplifies step 3.1, because taint information can be used to precisely identify which parts of the HTML response come untrusted inputs. However, full-fledged taint-tracking has higher overhead than string matching, and the binary or the source code must be transformed, possibly leading to a loss of reliability. The policies are based on *syntactic confinement*: tainted tokens of sensitive operations should not span multiple syntactical constructs. An example in the case of XSS is "no parts of the page originating from untrusted inputs shall close a tag and open a `script` tag". Reference [118] introduces taint- inference instead of taint-tracking, using similar policies.

XSS-GUARD [10] ports the approach of Candid [5] from SQL Injection to XSS: the application is instrumented to build an alternative "shadow" response along with the ordinary one; instead of using untrusted parameters to assemble the response, the application logic of the shadow page uses dummy inputs. Once both responses have been built, XSS-GUARD verifies that every script present in the real page is also present in the alternative page. Although XSS-GUARD is a server-side defense, the idea of sending a dummy request along with the original request for XSS protection has been used on the client-side as well in Reference [58].

Internet Explorer 8 [108] was the first XSS filter to be integrated by default in a major browser. Despite being part of the browser itself, it actually works like a client-side proxy and thus belongs firmly in this category. Compared to XSSDS and Reference [118], IE8's filter performs step 3.2 and 3.3 in a faster and simplistic way: from the untrusted inputs the filter creates regular expressions of possibly malicious injections using heuristics; the regular expressions are then used to find scripts in the HTML output. The filter's goal is to provide a usable protection for ordinary users, thwarting basic attacks without incurring false positives.

Recognizing Injection into the DOM (Step 4): A filter using this approach is tightly integrated with a specific browser (in particular, its HTML parser) and can stop execution of new JavaScript code by manipulating the AST while it is generated by the parser, removing the offending nodes. Filters from this category prevent XSS attacks in two substeps, which are related to those found in from the previous category. Step 4.1 is equivalent to Step 3.1, and the same caveat about sanitization apply here as well. Step 4.2 is

equivalent to step 3.3, but it is much easier: the filter can prevent execution of scripts content simply by removing the offending AST node, sidestepping the complex step of sanitizing HTML code. Note that there is no equivalent for Step 3.2: identifying which parts of the HTML response are parsed as script is now done "for free" by the browser's parser instead of by the filter, defeating all browser quirks. However, as shown in Figure 8, filters from this category automatically cover Step 5a through Step 4, but not 5b, stopping only a subset of DOM-Based attacks.

XSSAuditor [7] is the name of the XSS filter integrated into Google Chrome. At the time of its publication, the Step 4 approach was a novel contribution, and thus the paper focuses on its advantages over IE8's filter, a Step 3 filter. The filter is called by the HTML parser every time a relevant tag or attribute is consumed, and the corresponding string in the HTML output is searched in the URL. In practice, this means that XSSAuditor expects that for a successful XSS attack entire tags and attributes must be present in the URL, which is not always the case, in particular for partial injections. If an attack is found, the parser modifies the AST and removes the offending node.

Recognizing Execution of Code (Step 4 & 5b): This approach also creates filters that are tightly integrated to a specific browser, but instead of examining data from the HTML parser, it register callbacks in the codebase right before a string is about to be sent to the JavaScript engine for execution. This obviously includes both Step 4 and 5b. Our filter, XSSFilt [104], falls into this category.

Reference [126] presents another filter that uses this approach, published 2 years after XSSFilt. Unlike XSSFilt, this filter assumes the existence of a reflected XSS filter and focuses on stopping DOM-Based attacks. The authors argue that filters based on string matching, while appropriate for reflected XSS attacks, are unnecessary imprecise in the context of DOM-Based XSS: since the flow of untrusted inputs into vulnerable sinks happens entirely in the browser, the filter can collect reliable taint information by modifying the JavaScript engine. In their case, the authors leveraged the work already done in Reference [75] for the Chromium browser and added a straightforward policy to block DOM-Based attacks: the filter's policy prevents tainted strings from becoming a) JavaScript tokens (except terminals such as strings and numbers) and b) URLs in tags and DOM API methods that load new external JavaScript content (specifically, the host part of the URL). With

this policy, the paper reports a false positive ratio of 0.16% on an Alexa Top 10000 crawl.

The paper also discusses the shortcomings of other XSS filters, focusing on XSSAuditor and preseting 13 different issues that can be used to bypass it. Thanks to our improvements over XSSAuditor, only three apply to XSSFilt: *Second order flows*, which applies to attacks that use cookies, `localStorage` and similar flows that outlive the HTTP request/response lifecycle of the filter; *trailing content*, which confuses string matching because neither the input parameter and the injected script content is a substring of the other; *double injection*, previously discovered by Nikiforakis [94]) which splits the payload among different inputs, also confusing the string matching process. These weaknesses can be traced back to the choice of using taint- inference as opposed to full-fledged taint-tracking.

Blocking Unauthorized Scripts (Step 4 & 5b): This approach includes defenses that use directives from the web application to decide which scripts are allowed to run. Although they require explicit cooperation from web applications, these defenses can easily classify scripts as trusted or untrusted and act accordingly. These defenses also protect against stored XSS attacks.

BEEP [62] is a proposal to allow servers to supply a policy for the page through a JavaScript function. This function can interpose on script execution: it receives the script content and its DOM node as arguments, and can allow or deny its execution. The paper provides two sample policies: a whitelisting policy, where the web developers matches every script with a set of known script hashes, and a containment policy, where developers can mark a nodes as untrusted and the function prevents execution of script from any descendant.

DSI [90] and Noncespaces [136] protect against injection attacks by providing an isolation primitive for HTTP. Using this primitive, the server can securely isolate untrusted content and trasmit it to the browser along with the HTML response. The browser can then refuse to execute untrusted content. This combines the advantages of server-side defenses with respect to identification of untrusted content (support for taint-tracking and developer annotation) and of client-side defenses with respect to enforcement (immunity to browser quirks, support for DOM-Based attacks). Both papers use nonces to safely delimit untrusted content; DSI uses random character sequences, while NonceSpaces uses XML namespaces.

Content Security Policies [122] are a W3C standard to restrict the set of

hosts pages can include resources from. For each content type, web developers can specify a list of trusted hosts allowed to embed content in their web page. In the context of XSS, CSPs can prevent attacks by forcing external scripts to be served solely by servers that are trusted or under the control of the web application. Unfortunately, inline scripts cannot be considered same-origin in the presence of XSS vulnerabilities; therefore, to protect against XSS attacks, CSPs must also prevent execution of all inline scripts. Reference [140] noted a low adoption rate for CSPs compared to other security headers and a large percentage of CSP policies opting out of XSS protection by allowing inline scripts and `eval` calls[4]. A more recent version of the spec supports a less restrictive inline script policy: inline scripts are allowed as long as they bear a nonce defined by the policy. Using nonces, developers can deploy CSPs without having to convert all their inline scripts into external scripts, which will hopefully result in more widespread adoption.

Preventing Unintended Information Flows (Step 6): Defenses following this approach do not attempt to prevent execution of malicious code, which is already running at this point. Instead,they try to minimize the extent of the damage upon a successful attack.

Noxes [69] prevents exfiltration in general by monitoring outgoing HTTP data. It is implemented as a client-side proxy that works like a personal firewall: once the user vists a site, the proxy prompts the user if the site causes HTTP requests to URLs that are not a) on the same origin as the site and b) not statically present in the original HTML page. The intuition is that XSS attacks try to exfiltrate data by issuing XMLHttpRequests, creating new images, etc, which are not present in the original page. The evaluation shows that 3.5% of links would generate a warning. (5.7% if protection against implicit flows is enabled, which prevents the attacker from performing too many requests to the same third-party domain). Unfortunately, web pages generate much more dynamic content today than in 2006 (the year of publication of Noxes), and we expect the warning rate to be much higher. For example, Single page applications like Gmail would produce warnings for most of their links.

SessionShield [95] focuses on preventing exfiltration of session identifiers, which are stolen to hijack user sessions. In this case, the proxy inspects responses for `Set-Cookie` headers, filtering out session identifiers by recog-

[4]Even under such policy, CSPs can help with exfiltration in the event of an XSS attack

nizing their name (e.g. `phpsessid`) or the entropy of their value. The proxy adds the session cookie to all responses to the same site, effectively hiding the session identifier from the browser, while supplying it transparently to the server. The effect is the same as defining the session id cookie as `HttpOnly`, which prevents JavaScript code from accessing its value.

Noxes generates many warnings, and SessionShield only protects session IDs. Information Control Flow (IFC) can be used to track the flow of confidential data and only warn the user if outgoing connections contain such data, reducing the number of warnings. Reference [137] presents a modification to Firefox's JavaScript engine that prevents exfiltration using fine grained taint-tracking, refusing to transfer sensitive information (e.g. cookies) to third parties. IFC can also be enforced using Secure Multi- Execution (SME); instead of applying SME to the whole browser, which causes significant performance and memory overhead and usability problems, FlowFox [26] discusses the implementation of SME for the JavaScript engine only, and demonstrates how it can still successfully deal with a multitude of threats (e.g. session exfiltration, malicious advertisements, drive-by- downloads, etc.).

Instead of preventing information leaks, PathCutter [16] focuses on preventing the propagation of XSS worms, by dividing the web application into views and granting limited privileges to each view: an XSS attack compromising a view (e.g. a user comment) might not have the necessary privileges to propagate the worm to other users. Views are separated using `iframe`s pointing to different subdomains, which automatically enforce client-side isolation using the SOP; server-side authentication and access control is done using the `Origin` header.

2.7.2 New Surveys and Attacks

Since the publication of XSSFilt and XSSAuditor, the research community has begun taking XSS filtering into account. Recent work has focused on DOM-Based XSS, which had not been addressed head on by the last generation of defenses, and filter evasion.

In particular, two works [75, 99] focused on detection of DOM-Based XSS attacks in the wild to demonstrate that, as Web Application embed more and more functionality on the client side, DOM-Based XSS vulnerabilities are not a mere variation on reflected XSS vulnerabilities, but a major threat that needs to be addressed. Both works rely on client-side taint-tracking; Reference [75] adds byte-level taint-tracking to Chromium's JavaScript en-

gine and DOM implementation. Once tainted data reaches a sink, it is sent to an exploit generator, which uses context information (e.g. whether the attacker- controlled string appears in a JavaScript string fed to `eval`, in an HTML attribute fed to `document.write`, etc) to generate and verify an exploit. This technique is similar to what we described in our own exploit generator for gDork [105], but has been extended to a) deal with multiple sinks other than the HTML sink implied in reflected XSS attacks and b) use additional information available in the context of taint-tracking (e.g which sanitization functions were applied to the tainted data). Starting a crawl from the Alexa Top 5000 sites, the tool found almost 25 million potentially unsafe data flows. However, because verifying a vulnerability is time consuming, they focused on a subset of flows that is easier to exploit (e.g. no sanitization applied, no second order attacks), which yielded 180 thousand exploits to verify, of which approximately 40% succeeded, for a total of 8.6% sites from the Alexa Top 5000 containing at least one vulnerability.

Instead of working at the C/C++ level, Reference [99] uses a proxy to rewrite incoming JavaScript code and add taint-tracking. This kind of instrumentation does not require learning about the internals of one browser and is not tied to a specific implementation; on the other hand, the overhead is much higher. Beside demonstrating the popularity of DOM- Based XSS vulnerabilities, the paper describes how to automatically produce patches for the vulnerabilities it finds. Starting from the Alexa Top 1000, their tool found 777 thousand flows, which they narrowed down to 80 thousand flows that are easier to exploit, yielding 820 exploits across 89 Alexa Top 1000 domains.

An unsurprising but useful consequence of the increased amount of presentation and application logic that has moved from web servers to browsers is that more of it is now available to researchers for study. Although obfuscation prevents reverse engineering and static analysis, the code is still amenable for runtime analysis. In the context of DOM-Base XSS, Reference [127] defines taint-related complexity metrics (e.g. lines of code between each source access and sink operation) in an attempt to understand the underlying causes of DOM-Based XSS vulnerabilities. Although a third of vulnerabilities are associated to at least one metric having a high complexity, about two thirds are associated to low complexity metrics only, which seems to suggest that the vast majority of XSS attacks are simply caused by total lack of developer awareness about the dangers of handling attacker- controlled data on the browser.

FLAX [114] describes *Client-Side Validation Vulnerabilities* (CSVs); these include, but are not limited to, DOM-Based XSS attacks. This threat includes attacks that use similar sources of untrusted input, but do not inject new JavaScript code; instead, they subvert the application logic in more subtle ways, e.g. they impersonate another origin using `postMessage`, or they inject additional query parameters in otherwise benign `XmlHttpRequest`s. XSS defenses are ineffective against these non-injection threats; FLAX [114] uses taint-aware fuzzing to discover CSV vulnerabilities, while ZigZag [141] uses an Intrusion Detection approach, learning a model of the inputs to each sensitive sink during training.

Researchers also started taking XSS filters into account; Reference [53] discusses how to exploit sites and steal confidential data in a "post-XSS" world, i.e. assuming that XSS filters work and that attacks are able to inject *content* (e.g. HTML, CSS), but not execute JavaScript code. They name these *scriptless attacks*. The paper presents a proof-of-concept attack to steal CSRF tokens (and thus bypass most CSRF protections) and discusses mitigation techniques. In particular, although CSPs are useful to prevent injection of external resources and exfiltration, they cannot prevent attacks relying on inline elements, since the only inline content that can be locked down is script content. Rereference [54] discusses non-standard `innerHTML` behavior that can be used to bypass not only server-side XSS filters, but, unlike normal HTML browser quirks, also client-side filters.

3 jCSRF: A Server- and Browser-Transparent CSRF Defense for Web 2.0 Applications

The second defense presented by this dissertation is jCSRF, another black-box defense that protects legacy web applications from CSRF attacks. CSRF is the name given to a *Confused Deputy Attack* [52] specific to web applications. It is fundamentally different than an injection attack like XSS, and therefore a successful defense requires a different approach than XSSFilt. The term CSRF is as old as 2001 [139]. However, the security community has not paid enough attention to the issue until recent years. CSRF has been infamously named the "sleeping giant" among web based vulnerabilities, the reason being that most websites have been vulnerable to CSRF attacks long before vulnerabilities started to be discovered, but neither malicious individuals nor developers understood the extent of the threat. Given the number of unfixed vulnerabilities still in the wild, it is unsurprising that CSRF, just like XSS, appears both in the CWE Top 25 and the OWASP Top 10.

The stateless nature of HTTP necessitates mechanisms for maintaining authentication credentials across multiple HTTP requests. Most web applications rely on cookies for this purpose: on a successful login, a web application sets a cookie that serves as the authentication credential for future requests from the user's browser. As long as this login session is active, the user is no longer required to authenticate herself; instead, the user's browser automatically sends this cookie (and all other cookies set by the same server) with every request to the same web server.

The same origin policy (SOP), enforced by browsers, ensures confidentiality of cookies: in particular, it prevents one web site (say, `evil.com`) from reading or writing cookies for another site (say, `bank.com`). However, browsers enforce no restrictions on outgoing requests: if a user visits an `evil.com` web page, possibly because of an enticing email (see Figure 9), a script on this page can send a request to `bank.com`. Moreover, the user's browser will automatically include `bank.com`'s cookies with this request, thus enabling Cross-site Request Forgery (CSRF). A CSRF attack thus enables one site to "forge" a user's request to another site. Using this attack, `evil.com` may be able to transfer money from the user's account to the attacker's account [148]. Alternatively, `evil.com` may be able to reconfigure a firewall protecting a home or local area network, allowing it to connect to vulnerable services on this network [21, 22, 25].

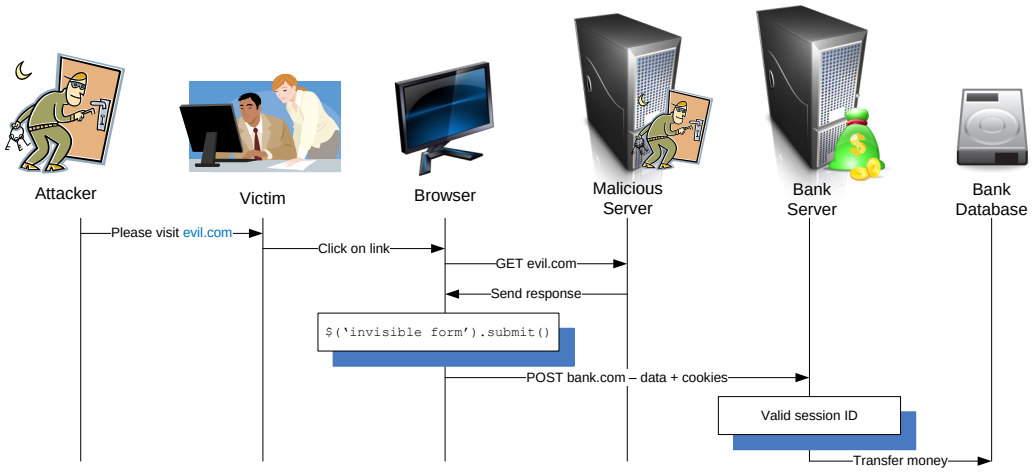

Figure 9: Illustration of a CSRF attack

Since CSRF attacks involve cross-domain requests, a web application can thwart them by ensuring that every sensitive request originates from its own pages. One easy way to do this is to rely on the `Referrer` header of an incoming HTTP request. This information is supplied by a browser and cannot be changed by scripts, and can thus provide a basis for verifying the domain of the page that originated a request. Unfortunately, referrer header is often suppressed by browsers, client-side proxies or network equipment due to privacy concerns [6]. An alternative to the `Referrer` header, called `Origin` header [6], has been proposed to mitigate these privacy concerns, but this header is not supported by most browsers. As a result, it becomes the responsibility of a web application developer to implement mechanisms to verify the originating web page of a request.

A common technique for identifying a same-origin request is to associate a nonce with each web page, and ensuring that all requests from this page will supply this nonce as one of the parameters. Since the SOP prevents attackers from reading the content of pages from other domains, they are unable to obtain the nonce value and include it in a request, thus providing a way to filter them out. Many web application frameworks further simplify the incorporation of this technique [143, 32, 51]. Nevertheless, it is ultimately the responsibility of a web application developer to incorporate these mechanisms. Unfortunately, some web application developers are not

aware of CSRF threats and may not use these CSRF prevention techniques. Even when the developer is aware of CSRF, such a manual process is prone to programmer errors — a programmer may forget to include the checks for one of the pages, or may omit it because of a mistaken belief that a particular request is not vulnerable. As a result, CSRF vulnerabilities are one of the most commonly reported web application vulnerabilities, and is listed as the fourth most important software vulnerability in the CWE Top 25 list [132].

Prompted by the prevalence of CSRF vulnerabilities and their potential impact, researchers have developed techniques to retrofit CSRF protection into existing applications. NoForge [66] implements CSRF protection using the same basic nonce-based approach outlined earlier. On the server-side, it intercepts every page sent to a client, and rewrites URLs found on the page (including hyperlinks and form destinations) so that they supply the nonce when requested. RequestRodeo [64] is conceptually similar but is deployed on the client-side rather than the server side. Unfortunately, since these techniques rely on static rewriting of link names, they don't work well with Web 2.0 applications that construct web pages dynamically on the browser. More generally, existing CSRF defenses suffer from one or more of the following drawbacks:

1. *Need for programmer effort and/or server-side modifications.* Many existing defenses are designed to be used by programmers during the software development phase. In addition to requiring programmer effort, they are often specific to a development language or server environment. More importantly, they cannot be deployed by a site administrator or operator that doesn't have access to application source code, or the resources to undertake code modifications.

2. *Incompatibility with existing browsers.* Some techniques require browser modifications to provide additional information (e.g., the referrer or origin [6] header), while others rely on browsers enforcement of policies on cross-origin requests (e.g., NoScript [43], CsFire [28], SOMA [96], RequestRodeo [64]). These approaches thus leave server administrators at the mercy of browser vendors and users, who may or may not be willing to adopt these browser modifications.

3. *Inability to protect dynamically generated requests.* Existing server-side defenses, including NoForge [66], CSRFMagic [143], and CSRFGuard

[120], do not work with requests that are dynamically created as a result of JavaScript execution on a browser.

4. *Lack of support for legitimate cross-origin requests.* Previous server-side token-based schemes similar to NoForge are aimed at identifying same-origin requests. However, there are many instances where one domain may trust another, and want to permit cross-origin requests from that domain. Such cross-origin requests are not supported by existing server-side solutions, and there does not seem to be any natural way to extend them to achieve this.

Therefore, we developed a new approach for CSRF defense that does not suffer from any of the above drawbacks. jCSRF is implemented in the form of a server-side proxy. Note that on web servers such as Apache that support a plug-in architecture, jCSRF can be implemented as a web server module, thus avoiding the drawbacks associated with proxies such as additional performance overheads and HTTPS compatibility.

jCSRF operates by interposing transparently on the communication between clients and servers, modifying them as needed to protect against CSRF attacks. As a server-side proxy, it avoids any need for server-side changes. jCSRF also avoids client-side changes by implementing client-side processing using a script that it injects into outgoing pages. It can protect requests for resources that are already present in the web page served to a client, as well as requests that are dynamically constructed subsequently within the browser by scripts. Finally, it incorporates a new protocol that enables support of legitimate cross-domain requests.

jCSRF protects all POST requests automatically, without any programmer effort, but as we describe later, it is difficult (for our technique and those of others) to protect against GET-based CSRF without some programmer effort. Moreover, GET-requests are supposed to be free of side-effects as per RFC2616 [38], in which case they won't be vulnerable to CSRF. For these reasons, jCSRF currently does not protect against GET-based CSRF.

3.1 Approach Overview

As described before, the essence of CSRF is a request to a web server that originates from an unauthorized page. We use the terms *target server,* and *protected server* to refer to such a server that is targeted for a CSRF attack.

An authorized page is one that is from the same web server ("same-origin request"), or from a second server that is deemed acceptable by this server ("authorized cross-origin request"). In the former case, no special configuration of jCSRF is needed, but in the latter case, we envision the use of a configurable whitelist of authorized sources for a cross-origin request.

We have implemented jCSRF as a server-side proxy, but it can also be implemented as a server-side module for web-servers that support modules, such as Apache. This proxy is transparent to web applications as well as clients (web-browsers), and implements a server- and browser-independent method to check if the origin of a request is authorized. Conceptually, this authorization check involves three steps:

- In the first step, an authentication token is issued to pages served by the protected server.
- In the second step, a request is submitted to jCSRF, together with the authentication token.
- In the third step, jCSRF uses the authentication token to verify that the page from which the request originated is an authorized page. If so, the request is forwarded to the web server. Otherwise, the request is forwarded to the server *after stripping all the cookies*.

Note that stripping off all cookies will cause an authentication failure within the web application, except for requests requiring no authentication, e.g., access to the login page of the web application, or another informational page that contains no user-specific information. Thus, jCSRF is secure by design and will prevent CSRF attacks. Specifically, its security relies only on three factors: unforgeability of authentication tokens, secure binding between the token and the original page, and the correctness of the authorization policy used in the third step. Other design or implementation errors may lead to false positives (i.e., legitimate requests being denied) but not false negatives.

Note that conceptually, the first two steps are similar to those used in previous defenses such as NoForge [66]. Thus, the key novelty in our approach is the design of protocols and mechanisms that ensure that CSRF protection can be achieved for:

- *dynamically created requests:* requests that are constructed as a result of script execution on the client (web-browser). Such requests are common in Web 2.0 applications using AJAX.

- *cross-origin requests:* requests from a web page served by one web site A to another website B, provided B trusts A for this purpose.

When requests are dynamically created, the strategy used by NoForge of statically rewriting the links (to include an authentication token) is not applicable. We have therefore developed a new approach that uses injected JavaScript to carry out this function. In particular, when a page is served by a web application, jCSRF injects some JavaScript code, called jCSRF-script, into this page. On the browser, jCSRF-script is responsible for obtaining the authentication token, and supplying it together with every request originating from this page. By comparing the domain of the current page and the domain of a request, this script can distinguish between same-origin and cross-origin requests, and use different means to obtain the authentication tokens in each case.

We also point out that a static rewriting strategy does not provide a way to validate cross-origin requests. In particular, if a server A embeds a cross-origin request for server B in its page, then the client would need a token for accessing B, but the server A has no easy way to obtain such a token. Note that it cannot directly request such a token from B since the token would have to be bound in some way to the *user's cookies* for B, and A has no access to these cookies. In contrast, we develop a protocol that can support cross-origin requests naturally.

From a conceptual point of view, jCSRF approach can be applied to both GET and POST requests. However, in practice, the "authorized origin" constraint, which forms the basis of all CSRF defense mechanisms, should not be imposed on many GET-requests. Examples include (a) login pages and other pages that contain no security-sensitive data, and (b) book-marked pages, which may or may not contain sensitive data. Application-specific configuration would be required to list such *landing pages* (case (a)) for each application, and except them from authorization checks. Handling case (b) would require some level of browser cooperation, something we do not assume in our work. Moreover, since it is recommended practice to avoid side-effects in GET-request (as per RFC2616 [38]), they are less likely to be vulnerable as compared to POST-requests. Finally, certain HTML elements such as `img`

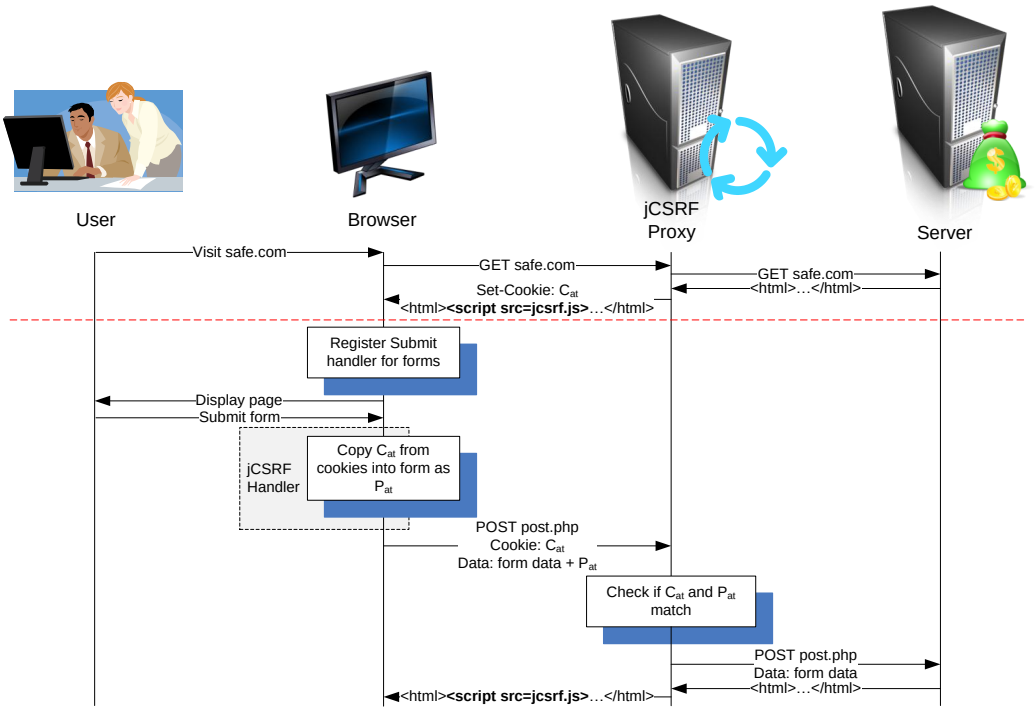

Figure 10: Same-Origin Protocol Workflow

and `frame` cause the browser to issue a GET request before jCSRF-script has a chance to add the authentication token, requiring jCSRF-script to resubmit the requests for these elements and complicate its logic. For these reasons, in our current implementation of jCSRF, GET requests are not subjected to the "authorized origin" constraint.

Below, we provide more details on the key steps in jCSRF.

3.1.1 Injecting jCSRF-script into web pages

When a page is served by a protected server, jCSRF-proxy automatically injects jCSRF-script into the page. This can be done without having to perform the complex task of parsing full HTML. Instead, the new script is added by inserting a line of the form

```
<script type="text/javascript" src=...  ></script>
```

immediately after the `<head>` tag. Also, jCSRF-proxy includes a new cookie in the HTTP response (unless one exists already) that can be used by jCSRF-script to authenticate same-origin requests. The rest of the page is neither examined nor modified by jCSRF-proxy. As a result, the proxy does not know whether the page contains any cross-origin (or same-origin) requests. It is left to the jCSRF-script to determine on the client-side whether a request being submitted is a same-origin or cross-origin request.

If jCSRF-proxy is implemented as a stand-alone proxy, then it may not be easy to handle HTTPS requests as the proxy will now intercept encrypted content. Although this can potentially be rectified by terminating the SSL sessions at the proxy, a simpler and more preferable alternative is to implement the proxy's logic as a module within the web server.

3.1.2 Protocol for Validating Requests

Although there is just a single protocol that uses different mechanisms to validate cross-origin and same-origin requests, it is easier to describe them separately. We first describe the same-origin validation since it is easier to understand, and then proceed to describe the cross-origin case.

Same-Origin Protocol: The same-origin protocol, illustrated in Figure 10, is a simple stateless protocol which authenticates same-origin requests. Red dotted lines in the figure demarcate request-response cycles.

Initially, an authentication token needs to be issued to authorized pages. Since jCSRF permits POST requests only from authorized pages, the very first request from a user has to be a GET request. Such a request is characterized by the fact that a cookie C_{at} used by jCSRF is not set. The server's response to this request is modified by jCSRF-proxy to set this cookie to a cryptographically secure random value. In addition, jCSRF-proxy also injects jCSRF-script in the response as previously described. When this page is received by the browser, jCSRF-script executes, and will ensure (as described further in Section 3.1.3) that the value of C_{at} is copied into a new parameter P_{at} for all requests originating from this page.

Note that all pages returned by a protected server are modified as above, not just the initial GET request. As such, subsequent requests can provide C_{at} as well as P_{at}. This information is then used in the second step of the protocol in Figure 10 to validate POST requests. In particular, jCSRF-proxy checks if $P_{at} = C_{at}$, and if so, the request is forwarded to the server after stripping out P_{at}. A missing P_{at} or C_{at}, or if $P_{at} \neq C_{at}$, it is deemed an

unauthorized request. In this case, jCSRF-proxy strips off all cookies before the request is forwarded to the server. Since web applications typically use cookies to store authentication data, this ensures that the request will be accepted only if it requires no authentication. Note that cross-origin GET requests can be limited in the same way as POST requests, but for reasons described before, the current implementation of jCSRF does not do so.
Correctness — In order to protect against CSRF, this protocol needs to guarantee the following properties:

- scripts running on an attacker-controlled page visited by user's browser cannot obtain the authentication token for the protected domain.
- any token that may be obtained by the attacker, say, using his own browser, cannot be used to authenticate a request from user's browser to the protected domain
- the attacker should not be able to guess an authentication token that is valid for the protected domain

The first property is immediate from the SOP: since the authentication token is stored as a cookie, attacker's code running on the user's browser runs on a different domain and has no access to it.

The second property holds because the attacker, apart from being prevented by the SOP from reading the token, is also prevented from setting the token. Therefore, any token obtained by the attacker and embedded into forms sent by the user would not match the cookie that jCSRF-proxy previously set for the user.

The third property is ensured by the fact that the authentication token is randomly chosen from a reasonably large keyspace. Specifically, jCSRF-proxy for a server S generates C_{at} as follows. First, a 128-bit random value IR is generated from a true random source, such as `/dev/random`. A pseudorandom number generator, seeded with IR, is then used to generate a sequence of pseudorandom numbers $R_1, R_2, \ldots$. From these, nonces $N_1, N_2, \ldots$ are generated using secret-key encryption (specifically, the AES algorithm) as follows:

$$K_s = IR,\ N_i = AES_{K_s}(R_i)$$

Whenever jCSRF-proxy receives a request with a missing (or invalid) C_{at}, it sets C_{at} to N_i and increments i.

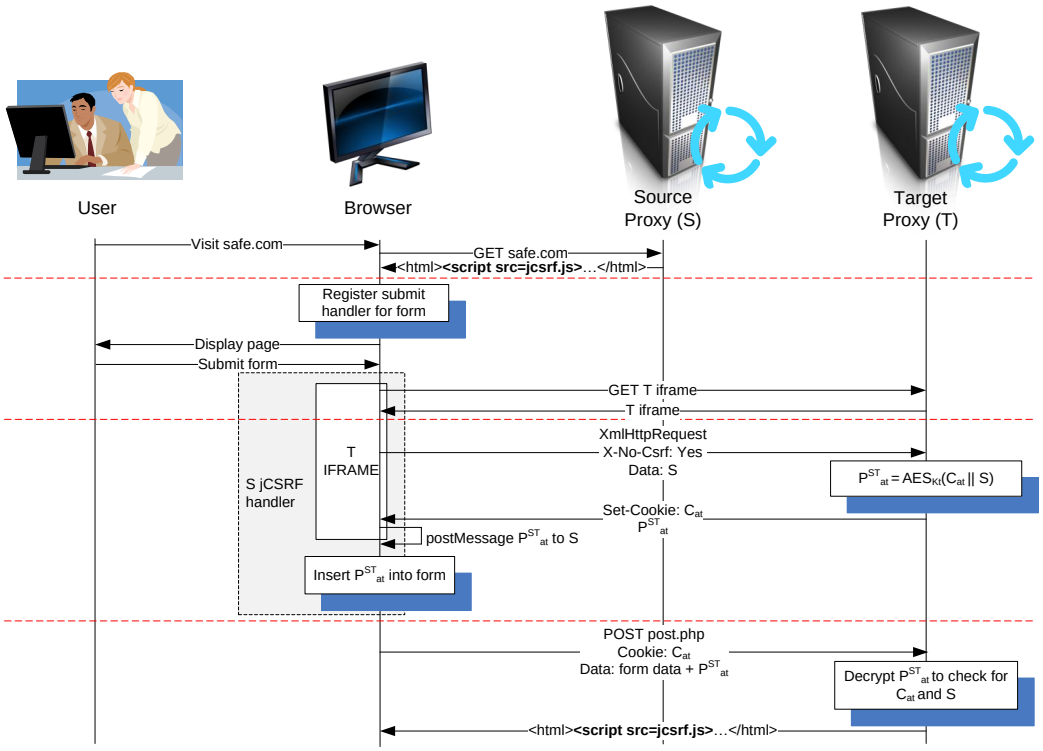

Figure 11: Cross-Origin Protocol Workflow

Note that this protocol design does not require N_i values to be stored persistently, since the validation check is stateless: jCSRF-proxy simply needs to compare C_{at} and P_{at} values in the submitted request. Hence, if jCSRF-proxy crashes, it can simply start all over, generating a new IR and so on. Similarly, K_s can be refreshed on a periodic basis by setting it to a new random value from `/dev/random`.

Cross-Origin Protocol: Figure 11 illustrates our protocol that enables pages from a (source) domain S to submit requests to a (target) domain T. Note that servers have been omitted to reduce the number of actors involved in the picture. Before describing the specifics of the protocol, note that the mechanism used in the same-origin case cannot be used for cross-origin requests: jCSRF-script runs on the source domain and therefore has no access to the target domain's cookies, which should contain the authentication token for requests to that domain. An obvious approach for overcoming this

problem is to have the source domain communicate directly with the target domain to obtain its authentication token, but this is not easy either. In particular, a correct protocol must bind the subset of user's cookies containing security credentials (for domain T) to jCSRF's authentication token (also for domain T). Unfortunately, jCSRF-proxy, being application-independent, is unaware of which cookies contain user credentials, and hence cannot achieve such a binding on its own. We therefore develop a protocol that exploits browser-side functionality to avoid the need for a new protocol between S and T. In this protocol, javascript code executing on the user's browser communicates with T to obtain an authentication token and communicates it to jCSRF-script. This enables jCSRF-script to include the right value of P_{at} when it makes its cross-origin request to T.

Note that there may be many instances where the user loads a page from S containing a form for T, but never actually submits it. To avoid the overhead of additional communication with T in those instances, the steps for passing T's authentication token to jCSRF-script are performed only when the user submits a cross-domain form. In addition to reducing the overhead, this approach has privacy benefits since T does not get to know each time the user visits a page that allows submitting data to T.

The specifics of our cross-domain protocol are as follows. When a POST action is performed on a page from S, jCSRF-script checks if the target domain T is different from S and if T accepts authenticated requests. This information can either be supplied by the web administrator of S as a list of jCSRF-compatible origins or detected by attempting to load a special image `jCSRF-image.jpg` from T: the `error` and `load` events can be used to detect whether the resource was found. If the host does not support jCSRF, then jCSRF-script simply submits the post to T without any authentication tokens. Otherwise, jCSRF-script injects an iframe into the page for the URL `http://T/jCSRF-crossdomain.html?domain=S`. This page will contain javascript code that sets up the authentication token P_{at}^{ST} that a page from S can present to T. The steps involved in this process are as follows:

- First, the script within the iframe makes an `XmlHttpRequest` to the domain T, providing S (given by the parameter `domain` in the above request) as an argument. `XmlHttpRequest`s can only be issued to same-origin resources and, unlike ordinary requests, are allowed to include custom HTTP headers. Therefore, a request bearing the custom header `X-No-CSRF` proves to T that the request came from a page served to

the user's browser by T.

- This `XmlHttpRequest` is served by jCSRF-proxy. If the user's jCSRF cookie (i.e., the cookie C_{at}) is not set, it is set by jCSRF-proxy using a nonce value N_i as described for the same origin case. In addition, jCSRF-proxy sends back the authentication token:

  $$P_{at}^{ST} = AES_{K_T}(C_{at}||S)$$

  Here, K_T is a (random-valued) secret key generated for T using the procedure described for the same-origin protocol.

- In the next step, P_{at}^{ST} needs to be passed on jCSRF-script so that it can complete the request to server T. This is accomplished using the `postMessage` API, which provides a secure mechanism for the framed script from domain T to communicate with a script from domain S. Note that a framing page from a malicious domain A cannot trick the frame from T into sending P_{at}^{ST}: `postMessage` can be instructed to deliver the message to a specific target origin which is chosen by T. Whenever T is instructed to send P_{at}^{ST}, this will be sent to S only, thus preventing A from reading the message.

  Some of the older browsers do not support the `postMessage` API. In that case, a technique called *location hash polling*[5] can be used in its place.

- Once the framing page has received the token, the jCSRF-script from S adds it to the form and submits the POST request to T.

- When jCSRF-proxy for domain T receives a POST request, it decrypts it using K_T, and checks if the cookie C_{at} included with the POST is a prefix of the decrypted data. If so, it checks if the domain S, which represents the remaining part of the decrypted data, is authorized to submit cross-domain POST requests. If so, the request is passed on

[5]In location hash polling [41], a framing page sends its URL to a framed page as a parameter. The framed page can then append a token to the URL of the framed page using an anchor at the end of the URL. The basis for this technique is that this URL change does not cause the framing page to reload; instead the value appended to the URL is available for polling. If a malicious page from domain A lies about its URL (pretending to be a page from S), then the update will cause the outer page to reload from domain S, thus defeating the attempt by A to read data written by T.

the server. In all other cases, jCSRF-proxy treats the request as unauthorized, and strips all cookies before it is forwarded to T.

Correctness — Correctness of the cross-domain protocol relies on the same three properties as the same origin protocol:

- scripts running on an attacker-controlled page visited by user's browser cannot obtain the authentication token for the protected domain.
- any token that may be obtained by the attacker, say, using his own browser, cannot be used to authenticate a request from user's browser to the protected domain
- the attacker should not be able to guess an authentication token that is valid for the protected domain

For the first property, note that because of the semantics of the `postMessage` API, an attacker-controlled page can either receive an authentication token that encodes its true domain A, or it may lie about its origin and not receive the token at all. In the latter case, the first property obviously holds. In the former case, although there is an authentication token, it contains the true origin of the attacker. On receiving this token, jCSRF-proxy will deny the request, as the attacker domain A is not authorized to make cross-origin posts.

The second property holds because the attacker is unable to set (or read) the value of user's cookie C_{at} for the protected domain T. Thus, even if he obtains an authentication token P by interacting with T using his own browser, he cannot use it with user's cookie C_{at} that will have a different value from the cookie value sent to the attacker by T. (Recall that T uses cryptographically random nonces to initialize C_{at}.)

The third property is ensured by the fact that the authentication token is randomly chosen from a reasonably large keyspace.

3.1.3 Design and Operation of jCSRF-script

As noted before, jCSRF-script needs to intercept all POST-requests and add the authentication token as an additional parameter to these requests. There are two ways in which browsers may issue POST requests:

Application	Version	LOC	Type	Compatible
phpMyAdmin	3.3.7	196K	MySQL Administration Tool	Yes
SquirrelMail	1.4.21	35K	WebMail	Yes
punBB	1.3	25K	Bulletin Board	Yes
WordPress	3.0.1	87K	Content-Management System	Yes
Drupal	6.18	20K	Content-Management System	Yes
MediaWiki	1.15.5	548K	Content-Management System	Yes
phpBB	3.0.7	150K	Bulletin Board	Yes

Figure 12: Compatibility Testing

- *Submission of HTML forms,* represented by `form` tags. Note that it is not necessary for the page to contain a `form` tag, because the form can be constructed dynamically using Javascript. Also, it is not necessary for the user to submit the form explicitly, because the form can be submitted automatically using Javascript.

- *`XmlHttpRequest` submissions.* Unlike form submissions, where the response cannot be accessed by the submitting script, the response to `XmlHttpRequest` can be read by the script making the request.

Compatibility requires handling both types of primitives. We now describe how jCSRF-script achieves this.

HTML Form Submission: Modern browsers allow Javascript code to register callbacks for specific events concerning the web page presented to the user. These functions are called *event handlers.* To ensure that every form is submitted to the web application with an authentication token, jCSRF-script registers a *submit* handler for each POST-based form. This handler then checks if the submission is to the same-origin or cross-origin. In the former case, jCSRF-script simply adds the authentication token as an additional parameter to the POST request. In the latter case, it uses the cross-domain protocol to first obtain a token for the target domain, and then adds this token as an additional parameter to the POST request.

Note that the web application might define its own event handlers for the submission event, mostly to validate the form contents. If the web application handler ran after jCSRF's handler, it would have access to the authentication token. In some rare cases, the presence of the token might confuse the web application handler which only expects a predefined set of

fields. Therefore, jCSRF-script detects if the web application defines its own handlers and wraps them with a function which removes the token before calling the web application handler, reinserting the token afterward. There are two types of event handlers: DOM0 handlers (registered with the HTML attribute `onSubmit` or by assigning a function to the JavaScript property `form.submit`) and DOM2 handlers (registered with the `addEventListener` function). The former type of handler is detected by periodically checking all forms for new, unwrapped submit handlers, which can be done through the previously mentioned `submit` property of form elements, since handlers are JavaScript functions and functions are first-class objects in JavaScript. The latter type requires overriding the `addEventListener` method to directly wrap new handlers during their registration, since there is no way to query the set of listeners registered for a specific (event, object) pair.

XmlHttpRequest: For `XmlHttpRequest`s, jCSRF-script modifies the `send` method of the class. For a browser supporting DOM prototypes [47], this can be done simply by substituting the `send` function, while on older browsers it is done by completely wrapping `XmlHttpRequest` functionality in a proxy object that hides the original class, and redirects all requests made by the web application to the proxy class. As explained in 3.1.2, adding a special header `X-No-CSRF` is enough to prove that the request is same-origin and therefore safe.

Compatibility: jCSRF-script uses jQuery's `live` method [131] to reliably interpose on submission of dynamically generated forms: instead of binding an event handler to a specific DOM element at call time, `live` registers a special handler for the root element which is then invoked once the event that fired on one of its descendants bubbles up the DOM tree. The purpose of the special handler is to find the element responsible for the event, check whether it matches the element type specified to `live` and apply the event handler supplied to `live` to it. jCSRF-script can thus bind its handler to the submit event for all future forms by calling

```
$('form').live('submit', handler)
```

Overriding `addEventListener` requires DOM prototypes support, which is not available on old browsers (IE7 and older). On these browsers, only DOM0 events can be wrapped.

Even though we did not encounter this scenario in any of the web application we tested, the techniques used to wrap DOM0 and DOM2 handlers may not work properly if the web application (or another software similar

to jCSRF which transforms the HTML output) is using them as well. For example, if another piece of code other than jCSRF-script polls forms for the presence of submit handlers, a race condition can ensue: either the two wrapper functions are composed in a nondeterministic fashion, or one wrapper is overwritten by a subsequent attempt to wrap the function by the second piece of code.

Note that failure to wrap an event handler does not necessarily imply a compatibility issue with jCSRF: most web applications define their own handlers to predicate on specific form fields, enforcing constraints such as "the field `age` must be a number". Only handlers that predicate on classes of fields might be incompatible with jCSRF on older browsers. For example, the constraint "all fields must be shorter than 10 characters" could create a problem for the token field if the handler is not wrapped.

3.2 Evaluation

3.2.1 Compatibility

To verify that jCSRF is compatible with existing applications, we deployed popular open-source Web applications and accessed them through the proxy, checking for false positives by manually testing their core functionality. We tested jCSRF with two browsers (Firefox and Google Chrome) and the following applications: phpMyAdmin, SquirrelMail, punBB, Wordpress, Drupal, Mediawiki, and phpBB. As Figure 12 shows, these are complex web applications consisting of thousands of lines of code that would require substantial developer effort to audit and fix CSRF vulnerabilities. jCSRF was able to protect all applications without breaking their functionality in any way.

Note that these web applications did not perform cross-origin requests, and therefore our evaluation did not cover the cross-origin protocol. Nevertheless, we believe that the primary source of incompatibility in the cross-origin protocol will remain the same as the same-origin protocol, namely, reliably interposing on submit events. As a result, we expect the compatibility results for the cross-origin protocol to be similar to those shown in Figure 12.

It is worth mentioning that jCSRF requires JavaScript enabled. If it is disabled, say, through the use of a browser extension such as NoScript [43], then requests would be sent out unauthenticated, resulting in false positives.

Application	Version	LOC	Type	CVE	Stopped
RoundCube	0.2.2	54K	Webmail	CVE-2009-4076	Yes
Acc PHP eMail	1.1	3K	List Manager	CVE-2009-4906	Yes

Figure 13: Protection Evaluation

3.2.2 Protection

To test the protection offered by jCSRF, we selected 2 known CVE vulnerabilities and attempted to exploit them. The results are summarized in Figure 13.

First, we exploited the CVE-2009-4076 [23] vulnerability on the open source web mail application RoundCube [109]. Emails are sent using a POST request, but its origin is not authenticated. We built an attack page on an external website that fills out and submits an email message. jCSRF successfully blocked the attack, because the POST request was missing the authentication token. Second, we exploited the CVE-2009-4906 [24] vulnerability on the Acc PHP eMail web application. This vulnerability allows changing the admin password with a POST request from an external website. jCSRF was able to thwart this attack as well.

We limited our evaluation to two because the effectiveness of jCSRF does not need to be established purely through testing. Instead, we have provided systematic arguments as to why the design is secure against CSRF attacks. A secondary factor was that reproducing vulnerabilities is a very time-consuming task, and can be further complicated by difficulties in obtaining specific software versions that are vulnerable, and dependencies on particular configurations of applications, operating systems, etc.

Finally, two attacks are out of scope for a tool such as jCSRF, but should be mentioned for completeness: XSS attacks and same-domain CSRF attacks. XSS attacks can be used to break the assumption that same-origin scripts are under the control of the web developer, to issue token requests and leak results to the attacker, thus defeating the purpose of jCSRF. We point out that a successful XSS attack grants the attacker far more serious capabilities than the ability to craft requests on the victim's browser using his cookies. In fact, the attacker can simply steal the cookies directly and send authenticated requests as the victim from his own machine! To our knowledge, no other server-side CSRF defense can resist in case of an XSS attack. Same-origin CSRF attacks can be carried out by injecting a form in a

server response and tricking the user into submitting it. jCSRF-script would add the correct authentication token, because it has no way to realize that the form present in the DOM tree was indeed supplied by the attacker [149].

3.2.3 Performance

In this section, we estimate the overhead imposed by jCSRF. A page embedding jCSRF-script issues three different type of requests to its target jCSRF-proxy:

1. GET requests. For these, the browser does not perform any special processing, and thus incurs no overhead. On the server-side, jCSRF-proxy only needs to generate a new token if the user does not have one already.

2. Same-Origin POST requests. Before the actual submission, jCSRF-script copies the authentication token C_{at} from the cookies to the form as P_{at}. Therefore, no overhead is introduced on the client, and the proxy only needs to check that $C_{at} = P_{at}$, which is an inexpensive operation.

3. Cross-Origin POST requests. The cross-origin protocol requires three additional GET requests for authentication: one to detect whether the target web application is running jCSRF, one to fetch the iframe from it that requests the token and one for the actual `XmlHttpRequest` that fetches the token. Therefore, this additional network delay dominates any other delay introduced by token generation and verification by the proxy. Although this overhead is non-negligible, we point out that cross-origin POST requests make up only a small fraction of HTTP requests [80], and therefore the delay due to these roundtrips is not likely to affect the overall user browsing experience.

We built a simple web application, deployed it locally and compared the response time of unprotected vs. protected same-origin and cross-origin POST requests. jCSRF protection incurred at most 2ms overhead.

3.3 Related Work

3.3.1 Server-side Defenses

NoForge [66] was the first approach that used tokens to ascertain same-origin requests without requiring modifications to the application's source code. Implemented as a server-side proxy, NoForge parses HTML pages served by a web server, and adds a token to every URL referring to this server. It associates this token with the cookie representing the session id for the application. When a subsequent GET- or POST-request is received, it checks if this request contains the token corresponding to the session id. jCSRF is clearly influenced by NoForge, but makes several significant improvements over it:

- NoForge requires developer help to specify the name of the cookie containing the session id. Not only is this effort unnecessary in our approach, but it is also the case that our technique is compatible with alternative schemes for authentication, such as those that store authentication credentials in multiple cookies, or schemes that support persistent logins that, at different times, may be associated with different session ids.

  Moreover, NoForge needs to maintain server-side state in the form of valid (session id, token) pairs. In contrast, jCSRF does not maintain state, and is less prone to DoS attacks.

- jCSRF supports web sites where URLs are dynamically created by client-side execution of scripts.

- jCSRF supports cross-origin requests whereas NoForge can only protect same-origin requests. NoForge's approach does not easily extend to cross-origin case since it relies on a mapping between cookies and tokens on the server side. In the cross-origin case, the cookies that are visible to the origin and target domains are different, and so it is unclear how the states maintained on the two domains can be correlated.

An important difference between NoForge and jCSRF is that the former protects GET-requests as well. However, as discussed before, there are a number of difficulties in CSRF protection for GET-requests: inability to bookmark pages, need for developer effort to identify "landing pages" that do not need CSRF protection (which are not supported by NoForge), and so

on. In the interest of providing a simple, fully automated solution, jCSRF protects only POST-requests.

Bayawak [59] can be thought of as extending NoForge to enforce a stronger policy: URLs in the web application are augmented with a special token not only to ensure that the request is same-origin, but also to constrain the order in which web pages can be visited. As such, it also protects against *workflow attacks* that aim to disrupt the session integrity by sending out-of-sync requests. This increased protection is obtained at the cost of additional programmer effort needed to specify permissible workflows. Bayawak does not address cross-origin requests or URLs that are dynamically created on the client-side.

X-PROTECT [149] is a server side defense that employs white-box analysis and source code transformation to overcome the shortcomings of other black-box approaches, namely their inability to protect against same-origin CSRF and their need to specify landing pages manually.

Developer Tools: Most web frameworks for rapid application development [32, 51, 106, 35, 148] include functionality to simplify CSRF protection, typically using a NoForge-like approach. For example, Django [32], a Python-based framework, provides CSRF protection for POST requests by requiring a specific template tag to be added to HTML forms, which is translated to a hidden form field containing a token that is also returned through cookies. To check whether the token in the cookies and the form match before executing the application logic, Django provides function wrappers to instrument *views* (python functions associated to URLs). CSRFMagic [143] provides a similar capability for PHP applications. Web developers need to include an import statement in their PHP files to activate this protection. The purpose of the included file is to register output and input filters. The output filter executes before the HTML page is sent to the client, and adds a token to POST forms. The input filter checks for the presence of this token.

CSRFGuard [120] is similar to CSRFMagic, but is designed for Java EE applications. It examines incoming GET and POST requests for the presence of a valid token. CSRFGuard introduced an option for client-side insertion of tokens using a script. Although this appears similar to our technique of injecting jCSRF-script, its operation is different. In particular, their client-side script adds the token to content available when the page fires the `load` event, and hence does not handle requests that may be dynamically constructed by various scripts associated with this page. Moreover, unlike NoForge, it allows web developers to configure a set of landing pages that do not require a valid

token, thus mitigating the usability issues related to GET-request protection at the cost of additional developer effort.

CSRF tools for developers are an invaluable resource for rapid application development. Their benefit is that they provide a finer granularity of control for programmers, as compared to fully automated approaches such as jCSRF. The main drawback of developer tools is the need for programmer effort. Moreover, programmers may overlook to add checks in all places they are required, thus leaving vulnerabilities.

The basic idea of comparing a token and cookie value to verify same origin requests is similar between these approaches and jCSRF. However jCSRF goes beyond these tools by (a) providing support for cross-origin requests, and (b) supporting requests to URLs that are generated dynamically on the client-side.

3.3.2 Browser Defenses

Zeller and Felten [148] present a Firefox plugin which implements a simple policy to prevent POST-based CSRF: cross-site POST requests must be authorized by the user. The drawback of this simple approach is the fatigue stemming from repeated user prompts. NoScript [43] implements a more sophisticated policy that can avoid prompts. In particular, it restricts only those POST requests that go from an untrusted origin to a trusted origin. NoScript primarily targets sophisticated, security-conscious users who are willing to put in the effort needed to populate their list of trusted origins.

De Ryck et al [28] presents a CSRF protection plugin for Firefox, CsFire. It studies cross-domain interactions on the web, and uses the results to design a cross-domain policy that protects against CSRF attacks while optimizing compatibility with existing web applications. This policy relies on the concept of *relaxed SOP*, which allows communication between subdomains of the same registered domain (e.g. `mail.google.com` and `news.google.com`). Their policy also introduces the idea of *direct interaction*: since CsFire is a browser plugin, it has access to UI information such as whether a request was initiated by a user click. Cross-Origin GET requests initiated by user clicks are allowed, while cross-origin POST requests are not allowed in any case. Instead of blocking the request altogether, the plugins strips the cookies from the request, which are necessary to carry out a successful CSRF attack.

RequestRodeo [64] differs from the above techniques in that it is implemented outside of a browser as a client-side proxy. It relies on an approach

similar to NoForge, but rather than blocking a request, it simply strips all cookies from such requests. Another difference is that it has no exceptions for landing pages. This can significantly affect usability.

A key advantage of browser-side defense is that it protects users even if web sites are not prompt in fixing their vulnerabilities. Moreover, they have accurate information about the origin of requests, whether they result from clicking on a bookmark, or a link on a web page trusted by a user. Their primary drawback is that the defense is applied to all web sites and pages, regardless of whether they have any significant security impact. Such indiscriminate application significantly increases the odds of false positives. Moreover, it is easier for a server-side solution residing on a target domain to determine whether it trusts the origin domain of a request. In contrast, browser-based defenses require the user to make this determination, and moreover, do it for all origins and target domains.

3.3.3 Hybrid Defenses

These defenses require both browser and server modifications. The Referrer header is the most known mechanism in this context. Using this HTTP header, a web browser can provide the crucial information that servers lack: namely, the origin domain of the current request. Given this information, a server can implement a simple CSRF protection mechanism that denies requests from domains that it does not trust. Unfortunately, due to privacy concerns, it is common to suppress referrer headers [6]. Barth et al [6] proposed a new header, called the *origin header*, which mitigates these privacy concerns by suppressing most of the sensitive information leaked by the referrer header and providing only the domain name and the protocol.

SOMA [96] is an alternate approach that aims to address a range of threats, including CSRF and XSS. With SOMA, the target domain is able to specify the set of allowable origin domains for incoming requests and an origin domain can specify the set of allowable target domains for outgoing requests. Effectively, SOMA provides support for *mutual authentication* from both the origin and the target of a request. Its implementation relies on a browser plug-in that retrieves these policies from the source and target domain and disallows any cross-origin requests that violate either policy.

Hybrid defenses are the simplest and most effective solutions, because they combine the information and mechanism available on browsers as well as web- servers. However, both browsers and servers have to be modified

in order to achieve CSRF protection. For this reason, it is harder for a hybrid defense to gain real-world traction. For instance, the origin header represents a very modest change from a technical perspective; yet, even after 7 years, not all major browsers support the header as originally proposed by Barth et al. For example, Firefox does not supply the header for same-origin requests, which forces developers to keep implementing more complex token-based authentication as a backup mechanism.

Part II
Principled Security for Web Applications

4 WebSheets

So far, we were focused on mitigating vulnerabilities in legacy applications without requiring any developer effort. However, *black-box* defenses are vulnerability-specific defenses with a narrow scope. In particular, they do not address any of the limitations of traditional application development that we described in Section 1.2. To overcome these limitation and provide broader protection, we focus on the problem of protecting the confidentiality of data in web applications; we start from a clean slate and envision a paradigm where users retain control of their data and can specify a privacy policy that follows the data throughout the web application. The paradigm is built upon the following principles:

- *security is a primary concern, not an afterthought*: our main goal for the new paradigm is to solve the problems of ad-hoc checking described in the introduction. The ability to build secure web applications should be the main driving factor for our design. It should be *simple* for developers to specify and maintain a policy.

- *users maintain ownership and control of their data*: principled web application development is not only about helping developers avoid bugs; users can benefit too: they should not necessarily have to sign off their data to the web application, putting its secrecy at the mercy of the web application developer. The paradigm must *allow users to specify a policy for their own data* and as the data is consumed by untrusted applications, *carry policies along with it* and enforce the policies wherever appropriate, so that third-parties can only tack on additional restrictions on the distribution of the data.

- *ease of use*: if users are to be in charge of their own data and writing their own policies, then it is paramount that policies be simple to specify. Moreover, users and developers alike will have an easier time picking up the new paradigm if we re-use familiar concepts where possible.

We concretized these principles mainly through two design decisions:

- *full separation between application logic and policies*: the mixture of application logic and ad-hoc checks is replaced by a clear separation

between data and permission metadata. Mandatory access control is used to attach policies to data and subject the data to a consistent policy wherever it flows during evaluation.

- *use of the spreadsheet model*: The textual programming model, where data is stored in databases and code is stored in linear text files, is replaced by data tables and permission tables. Both tables can contain not only immediate values, but also formulas, which can refer to other table cells, just like a spreadsheet. While the formulas in data tables implement the application logic, the formulas in permission tables implement the policy. If we want users to write policies for their own data, it makes sense to leverage a user-friendly paradigm

We have named our new paradigm WebSheets, given its tight relationship to spreadsheet programming. The architecture described in the rest of this dissertation overcomes all the problems of traditional web application development: policy checks are now organized in a tabular fashion and evaluated anywhere the data they are attached to is used, with no room for error or forgetfulness. Moreover, instead of simply giving away control of their data to web application developers, users can define their own tables and define their own policies.

The remainder of Part II is structured as follows: first, we provide an overview of the WebSheets model and language (Section 4.1), illustrating it using several example applications. We follow with a in-depth description of the WF language (Section 5.1) and its operational semantics (Section 5.2). Then, we describe our reference implementation (Section 5.3). We conclude the document with background information on spreadsheets and related work (Section 7). Below, we summarize our key contributions:

- We present a new paradigm based on the Spreadsheet model to design secure web applications where the application logic is clearly separated from the security policy.
- We introduce WF, a simple expression language to uniformly define operations on tabular data, as well as privacy and security policies on the data.
- We provide operational semantics for the evaluation of data and permission formulas, comparing the expressivity of our model with other DIFC solutions.

Author	Name	Completed	Shared
author	"Mow Lawn"	false	["Jim"]
author	"Date"	true	[]
author	"Meet Frank"	false	["Frank", "Tom"]
author	"Homework"	false	["Phil"]

Figure 14: `TODO-List`, `Task` Data Table (Expression View)

- To provide a reference implementation for WebSheets, we developed a Node.js application that allows users to create and share websheets through a web interface. Besides adding adding permission tables and mandatory access control as described by the operational semantics, we extended the semantics of traditional spreadsheets with support for temporal and logic triggers, and the ability to upload and safely execute user scripts in two different sandboxed environments for procedural data processing.

4.1 Overview

In this section, we provide an overview of WebSheets. Our goal here is to present the core of the paradigm — our reference implementation, described in Section 5.3, comes with a user interface that provides basic UI elements such as buttons, checkboxes, file upload fields, etc. These features have been added on top of the core semantics in a straightforward way, and we do not discuss them here.

To introduce the paradigm, we use a step-by-step approach, illustrating the WF language and the semantics of formula evaluation using three example applications of increasing complexity. We start with a relatively simple TODO list application, and progress to a moderately complex application for faculty candidate evaluation. An interesting aspect of WebSheets illustrated in these examples is that often, security policies *are* the application logic.

4.1.1 TODO-list

The first, simplest example is `TODO-List`, an application which maintains a private TODO list for each user. Its lone data table, `Task`, is depicted in Figure 14. Each row represents a TODO entry, which has an author,

	Author	Name	Completed	Shared	Row
Read					`user in Shared or user == author`
Write	`false`	`user == author`	`user in Shared or user == author`	`user == author`	
Add Row					
Del Row					`user == author`
Init	`author`	`""`	`false`	`[]`	

Figure 15: `TODO-List`, `Task` Permission Table (Expression View)

an item name, a "Completed" tickbox and a set of users to share the entry with. All users add their entries in the same table. We prepopulated the table with four sample entries, which, under normal operation, would be provided at runtime by different users. In WebSheets, each data table has an accompanying permission table that specifies the privacy and data validation policies of the data contained in the data table. In this case, the permission table for the `Task` data table is shown in Figure 15. The data and permission tables together, both shown as a grid of unevaluated WF formulas, are known as the *expression view* for the `Task` table.

Despite both table containing WF formulas, the formulas in data tables tend to be simpler, because they usually contain input data provided by end users. For example, `Task` contains constants such as `"Mow Lawn"`, which will trivially evaluate to the same value at runtime. The only non-trivial formula in Figure 14 is `author`, which is an environment variable that is bound to the author of the current row (i.e. the user who added the row) upon evaluation: although WebSheets automatically track authorship implicitly, some applications, like this one, benefit from exposing explicit authorship information. Although it is not explicitly shown in the Figure, each row in our sample dataset was entered by a different author, and the `author` formula will evaluate to a different string on each row.

On the other hand, formulas in permission tables are usually more complex, because they are entered by the user who created the table, the WebSheets equivalent of an application developer. They specify the privacy policies and the behavior of the corresponding data table. To grant the specified permission, they must be blank or evaluate to `true`, either using environment variables that depend on the cell being evaluated or by explicitly querying any other cell of the WebSheets repository. For example, `user in Shared`

uses two environment variables: `user` is set to the user who is currently evaluating the permission formula, while `Shared` is bound to the Shared field of the current row.

The permission table shown in Figure 15 implements the following policy: anyone can add new entries, but users can only see, edit and delete their own entries. Optionally, entries can be shared with a set of users specified in the `Shared` column; these users can not only see the entry, but also mark the item as completed. However, they cannot modify the name or assume authorship of the entry. Because the functionality of application stems from the ability of different users to concurrently view and/or modify the list, we could argue that its logic resides almost entirely in the `Task Permissions` table. Note that while data tables can have an arbitrary number of rows and columns, permission tables follow a fixed schema: there are 5 rows and $n+1$ columns, where n is the number of columns in the data table. The schema allows specifying a different policy for all cells along each column (using the first n columns), and one for all cells accross every row of the table, using the `Row` column[6]. When permissions are specified at multiple levels of granularity, the resulting permission is the intersection of all permissions. It is helpful to think about permission tables as a tabular representation for ACLs, where columns represents sets of objects (e.g. all cells along a column) and rows represent operations. There are four permissions: read, write, add row and del row. An additional row contains the Init expressions; these contain default formulas that are used during during add row operations.

In this particular permission table, the read permission is specified once for all columns using the special `Row` column. Intuitively, it will evaluate to true only if the current author of the row or any of the users specified in the `Shared` field of the same row are attempting to evaluate the corresponding data cell. Note how our approach provides a very simple and natural way to express access policies that are a function of data contained in the tables. Write permissions are more strict: only the Completed field is writable by users in the Shared field, while Name and Shared are only writable by the author of the entry. The Author field is read-only and, together with the

[6]This fixed schema does not support specifying formulas at the level of granularity of a single cell. This can be easily supported by an additional shadow table with the same number of columns and rows as the data table, but we leave it out of our description for simplicity. Note that, although multiple cells share the same formula, (e.g. all cells along a column), the formula is evaluated in a different environment for each cell, so the same formula can yield a different result.

Author	Name	Completed	Shared
"Phil"	"Mow Lawn"	false	["Jim"]
~~"Matt"~~	~~"Date"~~	~~true~~	~~[]~~
"Jim"	"Meet Frank"	false	["Frank","Tom"]
"Jim"	"Homework"	false	["Phil"]

Figure 16: `TODO-List`, `Task` (Value View for Jim)

`author` expression from the Init row, implements a common *derived cell* pattern used through these examples. In this case, the field will always evaluate to the implicit author of the row. The add row and del row permissions are only available for the Row column, because associating them to a single field would not make sense. The blank Add Row cell specifies that anyone can create a new TODO entry, while the Del Row formula specifies that only the author of an entry can delete it.

The expression view is only available to the table owner. End users interact with the application using the *value view*, which is computed from the data and permission tables from the expression view, by evaluating the WF expressions of all cells into values and redacting all those values whose read permission evaluated to anything other than `true`. Except for the redaction process, this behavior is similar to that of conventional spreadsheets, where the default is to show values rather than formulas. Note that unlike the *expression view*, the data presented by the *value view* is user dependent, because the permission formulas depend on the user who requested the *value view* for the table, through the `user` environment variable. Figure 16 shows the *value view* of the `Task` table for the user Jim. Note how the second row has been redacted, because Matt has not explicitly shared this entry with Jim. Of course, the contents are striked out in the Figure to show the readers which information has been redacted, but are removed from the response altogether in the actual implementation.

Note that there are no formulas to deal with Denial-Of-Service attacks (e.g. adding 10000 TODO items shared with everyone, to pollute their view). In this dissertation we focus on the problem of data privacy, and leave out both the issue of how to protect against such attacks with the current permission table format (e.g. a count check in the `Add Row` permission) and the issue of how to extend the current permission table format with dedicated fields.

Data Table

Host	Name	Public	Invitees	Attendees
author	"Ph.D. Defense"	false	["Sekar", "Stoller"]	Response[Name == EName and Coming == true].User
author	"Date"	false	["Jennifer"]	Response[Name == EName and Coming == true].User

Permission Table

	Host	Name	Public	Invitees	Attendees	Row
Read						user in Invitees or Public or user == author
Write	false				false	user == author
Add Row						
Del Row						user == author
Init	author	""	false	[]	Response[Name == EName and Coming == true].User	

Figure 17: `RSVP`, `Event` (Expression View)

Extensions: The central benefit of WebSheets is that it enables many simple and varied customizations. This version of `TODO-list` permits any user to add new rows to the table. However, other choices, such a limiting to a specified list of users, is also possible. Note that such a list can also be specified as another table, e.g., we could call it `Task Users`. This list may be defined and modified in the same manner as the `Task` table itself, thus providing another interesting application of data-based security policies.

Another possible extension is to add columns to indicate whether a payment was made for a task (`Paid`), and a column to indicate acknowledgement of payment (`Received`). This extension also needs another column, say, `Selected`, to indicate the specific person from the `Shared` list that selected and completed the job. Permissions would now be modified so that only the person in the `Selected` column can modify the `Received` column. In addition, row deletion would be prohibited until the `Received` column is true.

4.1.2 Event RSVP

The second example is an Event invitation and RSVP web application, `RSVP`, where users can create and RSVP to events. Events can be public or private, and users can only RSVP to private events they have been invited to. This application requires two tables, `Event` and `Response`. The `Event` table, shown in Figure 17, has the `Host`, `Name`, `Public`, `Invitees` and `Attendees` columns. We omit other data such as location, date, reason for refusal, etc

Data Table

User	EName	Coming
author	"Ph.D. Defense"	true
author	"Ph.D. Defense"	false

Permission Table

	User	EName	Coming	Row
Read				user in Event[Name==EName] .0.Invitees or Event[Name==EName].0.Public or user == author
Write	false	newVal == "" or user in Event[Name==newVal] .0.Invitees or Event[Name==newVal].0.Public or user == author		user == author
Add Row				
Del Row				user == author
Init	author	""	false	

Figure 18: `RSVP`, `Response` (Expression View)

to simplify the example. It comes prepopulated with two events, one by the user Riccardo, and one by Jim. The `Attendees` field contains a formula that pulls data from the `Response` table to display the list of users who have RSVP'd to the event.

Figure 17 also shows the permission table for `Event`. Again, read permissions are set using the `Row` column to avoid repetition, and restrict read access to the event information to either the host of the event or to users that have been invited; if the event is public, then every user is implicitly invited. The write row shows how the `Row` permission can be further restricted using column permissions. In this case, the row is editable by the Host, but the `Host` and `Attendees` formula are read-only, because they represent logic that is under the control of the table author.

Note how the formula for `Attendees` effectively republishes data from the `Response` table, and no specific restriction is imposed on such data. In a traditional web application, this would require the read permission policy for the `Attendees` field to take the privacy concerns of the data from `Response` into account. In WebSheets re-exposing data is always a safe operation: because *policies follow data*, the `Attendees` read permission formula can only tack on additional restrictions (in this case, the restriction imposed by the `Row` formula). The runtime will still observe the restrictions imposed by the policy specified in the `Response` table.

Figure 18 shows the *expression view* for the `Response` table, where users

Event

Host	Name	Public	Invitees	Attendees
"Riccardo"	"Ph.D. Defense"	false	["Sekar", "Stoller"]	["Sekar"]
~~"Jim"~~	~~"Date"~~	~~false~~	~~["Jennifer"]~~	~~[]~~

Response

User	EName	Coming
"Sekar"	"Ph.D. Defense"	true
"Stoller"	"Ph.D. Defense"	false

Figure 19: `RSVP` (Value View for Riccardo)

enter their RSVP responses. For our example, it is prepopulated with two responses from different users. The read permission is once again implemented using the `Row` field, using a formula similar to the read permission from Figure 17. However, the formula uses the *table filtering* construct `T[]` to select the relevant row from `Event`, predicating on the value of its fields. The write permission row formula only allows the creator of the response to edit the row. The column fields provide two further restrictions: firstly, the `User` field effectively pins the contents of all cells along the `User` column to `author`, implementing the same pattern we have seen for `Host` in Figure 17. Secondly, the `EName` field performs data validation instead of access control: using the `newVal` symbol, which is bound to the value that the user is trying to insert in the cell, the formula mandates that the user must enter an event name that he has access to.

Once again, developers do not have to worry about the data from `Event` leaking through the permission formula: the policies of the data from `Event` will follow the data even during permission evaluation, so that to gain access to the current `Response` row, the user must not only evaluate the current formula to `true`, but also have permission to read all of its dependencies.

Figure 19 shows the *value view* for the two tables of the application. Note how in the first row the data from `Response` is used to build the value for the `Attendees` field in `Event`, and how the second row is redacted, since the event is not public and Riccardo has not been invited.

4.1.3 Faculty Candidate Review

To further illustrate the applicability of WebSheets using a more involved example, we employ a scenario from our experience in academia: faculty candidate review process. The review process is by nature collaborative,

Faculty

Name
"Bell"
"Murphy"

Faculty Permissions

	Name	Row
Read		
Write		false
Add Row		false
Del Row		false
Init	""	

Figure 20: `Interview`, `Faculty` (Expression View)

since existing faculty contribute to the process: each candidate presents his research to the faculty body, who can later grade the candidate. Finally, the chair picks the best candidate, using the grades provided. WebSheets can be employed not only to automate the grading process, but also to enforce security policies. In particular, we seek to enforce the following properties:

1. Applicants can only enter and view their own application, while Faculty can view any application.

2. Each faculty can enter at most one review per candidate, and cannot enter a review in the name of another faculty.

3. To avoid being influenced by others, faculty should not see each other's grades for a particular candidate until they have graded the applicant themselves.

4. If the candidate and one faculty have a conflict of interest, the latter should be barred from grading and seeing the applicant's grades and average.

We have implemented the `Interview` application using 3 tables: `Faculty`, `Applicant` and `Review`. The first table `Faculty`, shown in Figure 20 is used to indentify which users have faculty privileges. In this relatively simple example, merely having an entry in the table grants the user faculty privileges, and any other user is assumed to be an applicant. A more involved example would use additional fields to assign roles to users, to express role-based access control semantics in the permission formulas. Read permissions are blank, because the faculty roster is public; write, add row and delete row permissions are set to `false`: for simplicity, we assume that the Faculty roster has been pre-populated.

Data Table

Name	Conflicts	AppReviews	Average
author	["Bell", "Murphy"]	(*from init*)	(*from init*)
author	[]	(*from init*)	(*from init*)

Permission Table

	Name	Conflicts	AppReviews	Average	Row
Read					user == author or user in Faculty.Name
Write	false	user == author	false	false	
Add Row					
Del Row					user == author
Init	author	[]	Reviews[AppName=Name].Grade	avg(AppReviews)	

Figure 21: `Interview`, `Faculty` and `Application` (Expression View)

Figure 21 shows the `Applicant` table, which has the columns `Name`, `Conflicts`, `AppReviews` and `Average`, and is used by applicants to fill in their application: each applicant fills in the first two fields, while the latter two fields are dynamically calculated by read-only formulas which use the reviews submitted by faculty in the `Review` table. Since the latter two fields have read-only formulas that are specified by the table creator in the Init row of the permission table, we omit the full formula from the data table, marking it with `(*from init*)`. Once again, we omitted other fields such as CV, Date of Interview, etc for simplicity. The row read formula asserts that applications are only visible by the applicant or by a member of the faculty body. Note how no read permission restrictions are required for `Grades` and `Average`, despite these being sensitive information from another table that should not be shown to the applicant or to faculty with conflicts: because *policies follow data*, it is sufficient to predicate on the grades themselves, which is done in the permission table for `Reviews`.

The `Review` table, shown in Figure 22 has the columns `Author`, `AppName` and `Grade`. This table is filled out by faculty after they have evaluated the applicants. The only sensitive information in each row is the grade, and the read permission for `Grade` restricts access to only the author of the review or to non-conflicting faculty members that have already submitted their own review[7]. The write row permission restricts write access only to the author of the review; however, a validation formula in the `AppName` field mandates

[7]We are explicitly exposing everything but the grade to all users. A more restrictive policy would restrict the other fields to faculty only.

Data Table

Author	AppName	Grade
author	"Smith"	4
author	"Doe"	3.5

Permission Table

	Author	AppName	Grade	Row
Read			user == author or (user in Review[AppName==row.AppName].Author and user not in Applicant[Name==AppName.0.Conflicts])	
Write	false	user not in Applicant[Name==newVal].0.Conflicts and user not in Review[AppName==row.AppName].Author		user == author
Add Row				user in Faculty.Name
Del Row				user == author
Init	author	""	0	

Figure 22: Faculty Admission Application (Expression View)

GoodApplicant

Name	Average
{{Name:a.Name,Average:decl(a.Average) for a in Applicant when decl(a.Average)>3.5}	

Figure 23: Dynamic Table with declassified averages

that the review can be associated only with an applicant that has not listed the current user as a conflict. The validation formula also reject multiple reviews by the same user.

Figure 23 uses a relaxed interpretation of the privacy requirements to showcase two additional features, namely dynamic tables and declassification: while individual reviews are still hidden from faculty in case of conflicts, we want to allow all faculty to view averages. Instead of rewriting another table and entering new data, we leverage dynamic tables to generate an additional `GoodApplicant` table based on the data from `Applicant`, which will list all applicants with a good average. Note that `GoodApplicant`, being a dynamic table, has no permission information of its own; however, because the policies follow the data, the permissions of its dynamic content depend on the permissions of the data that is being used to generate the table. The table builds its contents from a single WF expression, in this case a *list construction* expression. The result of evaluating the dynamic table expression must always be a list of tuples with the same keys, which is used to fill out the rows and columsn of the resulting table. From the user's point of view, a dynamic table is accessed transparently through rows and columns like an ordinary static table, except that its fields are read-only and cannot be evaluated individually, because their value depends on one single formula.

Note that the formula uses the privileged function `decl` to declassify the average grade of each applicant. Declassification is necessary because faculty members need to see the average grade even in case of conflicts. By using declassification, the author of the formula trusts that releasing this information does not constitute a privacy issue. In this case, he assumes that the grades given by each faculty have been anonymized by aggregation, i.e., by the use of `avg`. Although declassification introduces exceptions to policies defined elsewhere, which prevent developers from reasoning about privacy solely by examining permission tables, the state of the art in textual web applications is that every single operation is trusted and has full privileges

by default! Instead, in WebSheets it requires an explicit construct that can be used sparingly.

5 Language and Design

5.1 The WF Language

WebSheets cell and permission formulas are written in WF, a simple expression language that focuses on filtering and processing tabular values. The target audience of the language is the same non-programmers who write Excel formulas. Figure 24 shows the grammar for the language, which starts from the nonterminal ⟨*expr*⟩. NUMBER, STRING and ID are terminals returned by the lexer.

The evaluation of WF formulas is entirely dynamic, i.e., typing errors or unbound identifiers will only cause a runtime error. Evaluation will turn WF expressions into WF values, potentially requiring evaluation of other dependent cells in the spreadsheet. Identifiers are first looked up in the environment, a context-dependent `(name, value)` map containing information such as the current user, the current table or the current column (when applicable). When an identifier is not found in an environment, the name is assumed to be table name.

Besides the usual scalar data types (NUMBER, BOOL, STRING), it supports two composite types:

- *Lists*: A (possibly empty) ordered collection of WF expressions. Lists can be concatenated, sliced and accessed with the operations defined below.
- *Records*: A (possibly empty) unordered `(key, value)` map. Records can also be merged, sliced or accessed.

Lists and Records are not included in the language just to express higher-level application logic: WebSheets semantics define a dualism between tables and WF values. At runtime, a WebSheets table is represented as a list of named tuples: each element of the WF list represents a table row, and each row is represented as a WF tuple, a map of column names to WF values for that particular row.

⟨*expr*⟩ ::= NUMBER (`int`s and `float`s)
| STRING
| ID
| '`true`' | '`false`'
| '`null`'
| '`[`' (⟨*expr*⟩ ('`,`' ⟨*expr*⟩)*)* '`]`' (list)
| '`{`' (ID '`:`' ⟨*expr*⟩ ('`,`' ID '`:`' ⟨*expr*⟩)*)* '`}`' (tuple)
| ⟨*expr*⟩ ⟨*bin_op*⟩ ⟨*expr*⟩ (binary operations)
| ⟨*un_op*⟩ ⟨*expr*⟩ (unary operations)
| '`(`' ⟨*expr*⟩ '`)`' (precedence/associativity override)
| ⟨*expr*⟩ '`.`' ID (tuple/column selection)
| ⟨*expr*⟩ '`.`' NUMBER (list/row selection)
| ⟨*expr*⟩ '`{`' ID ('`,`' ID)* '`}`' (tuple/column projection)
| ⟨*expr*⟩ '`{`' NUMBER ('`,`' NUMBER)* '`}`' (list/row projection)
| ID '`(`' (⟨*expr*⟩ ('`,`' ⟨*expr*⟩)*)* '`)`' (function call)
| '`if`' ⟨*expr*⟩ '`then`' ⟨*expr*⟩ '`else`' ⟨*expr*⟩ (functional if-then-else)
| '`{`' ⟨*expr*⟩ '`for`' ID '`in`' ⟨*expr*⟩ ('`,`' ID '`in`' ⟨*expr*⟩)*
['`when`' ⟨*expr*⟩] '`}`' (list/table construction)
| ⟨*expr*⟩ '`[`' ⟨*expr*⟩ '`]`' (list/table filtering)

⟨*bin_op*⟩ ::= '`+`'
... standard math and logic operators ...
| '`++`' (list/tuple concatenation)
| '`in`' (list/tuple membership)
| '`not in`' (shortand for `not (e in l)`)

⟨*un_op*⟩ ::= '`not`'
| '`-`'

Figure 24: EBNF Grammar for the WF Language

Besides the usual unary and binary operators from math and logic, manipulation of data in WF is supported by four main constructs, which also have a dualism with table operations:

- *Selection*: access a single element from a list or record. The exact semantics depend on the types of the operands involved:
  - *list and integer*: extract the i-th element from the list. For example, `[5,6,7].1` → `6` (we use the arrow as shorthand for "yields").
  - *record and string*: extract a single value from the record. For example, `{a:1,b:2}.a` → `1`.
  - *list of records and string*: perform record selection on all elements of a list. For example, `[{a:1,b:2},{a:3,b:2}].a` → `[1,3]`.

  This is the WF equivalent of selecting one particular row, cell or column of a table respectively. Note how, by not allowing ⟨*expr*⟩ as the right-hand of selection, we somewhat limit the language compared to, say, the C subscript operator. Since we do not rely on statically extracting properties, we could extend the language without complications. However, we did not find a relevant use case, and this limitation yields shorter and cleaner formulas (e.g. `Faculty.Name` vs `Faculty."Name"`).

- *Projection*: accesses a multiple elements from the list or tuple. Again, the exact semantics depend on the types:
  - *list and integers*: return a list containing only the specified indices. For example, `[5,6,7]{0,1}` → `[5,6]`
  - *record and strings*: return a subset of the record containing only the specified keys. For example, `{a:1,b:2,c:3}{a,b}` → `{a:1, b:2}`.
  - *list of records and strings*: perform record projection on all elements of the list. For example, `[{a:1,b:2,c:3}, {a:3,b:2, c:5}].{a,c}` → `[{a:1,c:3}, {a:3,c:5}]`.

  This is the WF equivalent of selecting multiple rows, cells or columns of a table respectively.

- *List Construction*: combines one or more lists to create a new list. For example, {{a:x, b:y} for x in [1,2], y in [3,4] when x+y>4} → [{a:1, b:4}, {a:2,b:3}, {a:2,b:4}].

  This is the most general operator in the language. The expression {a:x, b:y} is evaluated using different bindings for x and y for each iteration. For example, the environment for the first iteration binds x to 1 and y to 3. The set of bindings to iterate on represents the cartesian product of the lists supplied, and the result is a list containing all the evaluations of {a:x, b:y} using all the bindings from the cartesian product. If the expression returns a record, then this is the WF equivalent of building a new table from existing tables. The when clause can be used to filter rows from the resulting table. In the example above, the first binding does not appear in the result because it is filtered out by the clause. Many languages have an equivalent construct called list comprehension. Functional programmers will note how it performs both map and filter.

- *List filtering*: returns the subsets of elements of a list that satisfy the condition in brackets ([{a=1, b=2}][a-b>0] → []). The construct evaluates the condition in the context of each element, and only includes the element in the result if the condition evaluates to true. Note how the keys of the record are automatically added to the environment for each element. List filtering is particularly useful when the WF list is actually a WebSheets table: Table[cond] effectively performs table filtering, returning a subset of Table whose rows all satisfy the condition cond. In terms of expressive power, list filtering is not a new construct, but a shorthand for the list construction operator: e[c] == (v for v in e when c), except that list construction does not automatically add the keys of v to the environment upon evaluation of c.

Note that the language defines function calls, but not function definitions. WF simply provides the ability to call external functions defined by the repository, to perform declassification, execute sandboxed procedural scripts, etc. To focus on the core features of the language, we also purposefully avoided describing how permissions come into play. Section 5.2 and 5.3 provide more details about the interaction between data and permission evaluation and describe the functions that our implementation exposes to WF formulas.

$\mathcal{U}$	(set of WebSheet users)
$\mathcal{P}$	(set of unevaluated read permissions)
$\mathcal{E}$	(evaluation environment)
$\mathcal{E}[]$	(empty evaluation environment)
$\mathcal{E}[a]$	(retrieve symbol a from the environment)
$\mathcal{E}[a := b]$	(return a modified environment where $a := b$)
$\mathcal{E}[\text{type}]$	('perm' when evaluating a permission, 'val' when evaluating a cell formula)
$\mathcal{E}[\text{author}]$	(author of the current cell being evaluated)
$f() :=$	(pattern matching)
$\quad \mid \text{cond}_1 \rightarrow a$	
$\quad \mid \text{cond}_2 \rightarrow b$	

Figure 25: Semantics Notation

5.2 Semantics

In this section we describe the semantics for the evaluation of WF formulas in the context of a WebSheet repository. First, we introduce simplified semantics for functional evaluation with no side effects, showing how our enforcement model offers capabilities equivalent to Decentralized Information Flow Control (DIFC). Then, we highlight the differences between the operational semantics as described in this section and the actual implementation described in Section 5.3. Throughout the section, we will use the notation from Figure 25.

5.2.1 Simplified Semantics

First, we illustrate the semantics for a simplified version of WF, WF_s. The language is the following:

$$e = c \mid e_1 + e_2 \mid @l \mid d(e, u) \mid user$$

The language only contains constants, one binary operation, an abstract reference to a cell l, declassification and a reference to the current user who is evaluating the formula. We define read-only semantics that retrieve information from an abstract store S to return a value to the user, who requests evaluation of a single cell using the getVal operation. The store S maps a cell reference to a triple, containing 1) the author of the cell, 2) the data

formula and 3) the read permission formula. Its signature is thus

$$S :: @l \rightarrow (u, \mathrm{WF}_s, \mathrm{WF}_s)$$

Internally, getVal uses an evaluation function E to convert any WF_s formula to a constant value c. During the evaluation, it also uses a permission checking function P to ensure that the caller of getVal can satisfy the permission formulas extracted from S. The signature for E is

$$E :: (\mathrm{WF}_s, u, \mathcal{U}, \mathcal{E}) \rightarrow c \mid \bot$$

where WF_s is the expression to evaluate, u is the "main" user who is performing the evaluation, $\mathcal{U}$ is the set of users whose credentials can be used in permission checks, and $\mathcal{E}$ is the environment, which is used to store additional information about the evaluation context. In this simplified semantics, the environment contains the the author of the cell and whether the formula is a cell or a permission formula. The possible output value $\bot$ denotes an evaluation error or invalid read permissions. Similarly, the signature for P is

$$P :: (\mathrm{WF}_s, u, \mathcal{U}, \mathcal{E}) \rightarrow c \mid \bot$$

The signature is the same because P is a wrapper over E that evaluates permission formulas instead of data formulas.

Figure 26 shows the simplified semantics for WebSheet evaluation. The most important rule is $E(@l, ...)$. When E evaluates a location reference, before evaluating the cell formula e, it checks the read permissions for the cell using the function P to evaluate the permission formula p. P mandates that at least one of the users in $\mathcal{U}$ must have read permissions for the current cell. getVal is the entry point for the user; it sets up an empty environment and begins evaluation of a single cell and all its dependencies by evaluating its reference. The semantics define two restrictions on WF_s: $d(e, u)$ is only available in cells owned by the principal u, and `user` is only available in permission formulas.

Note that the semantics from Figure 26 perform eager enforcement of the policy specified by the permission formulas: $E(@l, ...)$ evaluates the formula e in $S(@l)$ only *after* P evaluates the read permission formula to verify that at least one user within $\mathcal{U}$ has access to the data. This is the opposite of deferred enforcement, where the evaluation of P can be delayed until necessary. For example, Information-Flow Control implementations attach labels to data

$E(\texttt{user}, u, \mathcal{U}, \mathcal{E}) :=$ (reference to the user
$\quad \mid \mathcal{E}[\text{type}] = \text{'perm'} \rightarrow u$ evaluating the formula)
$\quad \mid \mathcal{E}[\text{type}] = \text{'val'} \rightarrow \bot$
$E(c, u, \mathcal{U}, \mathcal{E}) := c$ (evaluation of constants)
$E(e_1 + e_2, \mathcal{U}, \mathcal{E}) :=$ (binary operators)
$\quad E(e_1, u, \mathcal{U}, \mathcal{E}) + E(e_2, u, \mathcal{U}, \mathcal{E})$
$E(@l, u, \mathcal{U}, \mathcal{E}) :=$ (cell reference)
$\quad \text{let } (a, e, p) := S(@l)$
$\qquad p' = P(p, u, \mathcal{U}, \mathcal{E}[\text{type} := \text{'perm'}, \text{author} := a])$
$\quad \text{in if } (p' = T)$
$\qquad \text{then } E(e, u, \mathcal{U}, \mathcal{E}[\text{type} := \text{'val'}, \text{author} := a])$
$\qquad \text{else } \bot$
$E(d(e, v), u, \mathcal{U}, \mathcal{E}) :=$ (declassification)
$\quad \mid \mathcal{E}[\text{author}] = v \rightarrow E(e, u, \mathcal{U} \cup \{v\}, \mathcal{E})$
$\quad \mid \mathcal{E}[\text{author}] \neq v \rightarrow \bot$
$P(p, u, U, \mathcal{E}) :=$ (permission check)
$\quad \bigvee_{v \in U} E(p, v, U, \mathcal{E}[type := \text{'}perm\text{'}])$
$\text{getVal}(@l, u) := E(@l, u, \{u\}, \mathcal{E}[])$ (entry point for users)

Figure 26: Simplified WebSheet Semantics

representing which users have access to said data; the system does not check whether the current user is allowed by the label to read the data. Instead, the system propagates the label along with the data, deferring the check until the restrictions posed by label cannot be enforced any further, right before the data leaves the trusted runtime.

The main advantage of IFC over other MAC solutions is that it can support *declassification*, a controlled and explicit relaxation of the restrictions imposed by the label: because this is effectively a reversion of an access control decision (since the user might not be allowed to read the data before declassification is applied), eager enforcement cannot traditionally support declassification, forcing MAC systems that enforce a centralized policy to define a weaker policy that takes all corner-cases into account. However, our semantics support declassification! While it is true that in general eager enforcement and declassification do not play along, note that WebSheet semantics impose a very specific path to data flows, because of the expression-driven nature of WF formulas: the runtime always processes the declassification call $d(e, u)$ before evaluating its argument e and performing

the permission checks for e and its dependencies. Therefore, it is sufficient to record the declassification event as evaluation goes deeper into the call stack, so that permission checks can use the authority of the principal who performed the declassification. In the semantics, this is implemented by adding the user to $\mathcal{U}$ in $d(e, u)$ and then using the set of users U in P for permission checks.

In fact, we argue that we obtain expressivity and flexibility similar to Decentralized Information-Flow Control [89]. In his dissertation [88], Myers states that the following two are the essential properties of DIFC:

- *"It allows individual principals to attach flow policies to pieces of data. The flow policies of all principals are reflected in the label of the data, and the system guarantees that all the policies are obeyed simultaneously. Therefore, the model works even when the principals do not trust each other."*

- *"The model allows a principal to declassify data by modifying the flow policies in the attacked label. Arbitrary declassification is not possible because flow policies of other principals are still maintained."*

WebSheets exhibit both properties:

- although modifying the store S is abstracted out of the semantics to focus on evaluation, WebSheets allow users to define their own tables and attach a policy to their cells. Our threat model also supports mutually distrusting users: MAC ensures that the restrictions imposed by read formulas are always obeyed.

- users can declassify their own data by inserting a declassification call in another cell they authored.

5.2.2 Comparison with other DIFC systems

Figure 27 shows alternative DIFC semantics that use deferred enforcement: E does not call P; instead, it returns a set of read permission formulas along with the result. The set of permission formulas is finally evaluated in `getVal`. These semantics use the same store S, but the signatures of E and P are slightly different: since E performs deferred enforcement, it does not require the set of users $\mathcal{U}$. Plus, it returns a value and a set of permissions to evaluate at a later time. Thus, we have

$$
\begin{array}{l}
E(user, u, \mathcal{E}) := \\
\quad \mid \mathcal{E}[\text{type}] = \text{`perm'} \rightarrow u \\
\quad \mid \mathcal{E}[\text{type}] = \text{`val'} \rightarrow \bot \\
E(c, u, \mathcal{E}) := (c, \{\}) \\
E(e_1 + e_2, u, \mathcal{E}) = \quad \text{let} \quad (v_1, \mathcal{P}_1) := E(e_1, u, \mathcal{E}) \\
\qquad\qquad\qquad\qquad\qquad\quad (v_2, \mathcal{P}_2) := E(e_2, u, \mathcal{E}) \\
\qquad\qquad\qquad\qquad\quad \text{in} \quad (v_1 + v_2, \mathcal{P}_1 \cup \mathcal{P}_2) \\
E(@l, u, \mathcal{E}) := \quad \text{let} \quad (a, e, p) := S(@l) \\
\qquad\qquad\qquad\qquad\quad (v, P_v) := E(e, u, \mathcal{E}[\text{type} := \text{`val'}, \text{author} := a]) \\
\qquad\qquad\qquad\quad \text{in} \quad (v, P_v \cup \{p\}) \\
E(d(e, v), u, \mathcal{E}) := \\
\quad \mid \mathcal{E}[\text{author}] = v \rightarrow \text{let } (v, \mathcal{P}_v) := E(e) \\
\qquad\qquad\qquad\qquad\quad \text{in } (v, P_v \setminus \{p \in P_v : P(p, v, \mathcal{E}) = T\}) \\
\quad \mid \mathcal{E}[\text{author}] \neq v \rightarrow \bot \\
P(p, u, \mathcal{E}) := \quad \text{let} \quad (v, \mathcal{P}_v) := E(p, u, \mathcal{E}[\text{type} := \text{`perm'}]) \\
\qquad\qquad\qquad\quad \text{in} \quad (v = T \wedge \bigwedge_{p' \in \mathcal{P}_v} P(p', u, \mathcal{E}[\text{type} := \text{`perm'}])) \\
\text{getVal}(@l, u) := \quad \text{let} \quad (v, \mathcal{P}_v) := E(@l, u, \mathcal{E}[]) \\
\qquad\qquad\qquad\quad \text{in} \quad \text{if} \bigwedge_{p \in P_v} P(p', u, \mathcal{E}[]) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{then } v \\
\qquad\qquad\qquad\qquad\qquad\quad \text{else } \bot
\end{array}
$$

Figure 27: Simplified DIFC WebSheets Semantics

$$E :: (\mathrm{WF}_s, u, \mathcal{E}) \to (c, \mathcal{P})$$

and

$$P :: (\mathrm{WF}_s, u, \mathcal{E}) \to c \mid \bot$$

Although these semantics are functionally equivalent to those from figure 26, they can help readers familiar with DIFC to understand the similarity with WebSheets. There are two main differences between these semantics and Myers's Decentralized Label Model (DLM)[8]. Firstly, WebSheet users write read permission formulas that can not only predicate on the current user (e.g. `user == "admin"`) but also on the contents of the store S; P then verifies whether a specific user is part of the set of allowed users, which are those users for which $P(p)$ evaluates to T. The DLM model instead requires developers to explicitly specify a set of readers. Evaluating WebSheet permission formulas over each principal would yield a set of readers equivalent to DLM labels (e.g. `user in ["admin", "foo"]` is equivalent to DLM's $o \to \{\text{admin}, \text{foo}\}$), but note that in DLM the set of users is fixed at the time the label is created and attached, while in the semantics of Figure 27 the set of users is implicitly calculated when the check is performed, after the label has flowed all the way to `getVal`. Our experience is that in complex web applications, the policies for a specific piece of data continually change (e.g. a private photo is made public) and it is easier to account for such changes using formulas as opposed to explicit set of users, because the formula automatically captures the update to the policy. Plus, this model readily supports revocation, e.g. changing the value of a cell upon which p depends so that $P(p, u)$ is no longer T. The only disadvantage of this model is that calculating the effective set of readers (i.e. all those users for which all permissions from the permission set evaluate to T) is expensive and requires the evaluation of $n * m$ formulas, where n is the number of permission formulas to satisfy and m is the number of users in the system.

Secondly, unlike in the DLM, our semantics do not distinguish between owners and readers in permission formulas. In the DLM model, each set of

[8]Myers and Liskov [89] introduced both the term DIFC and the term DLM. By DIFC, we simply refer to the abstract idea of decentralizing label management compared to traditional IFC systems, so that code from distrustring principals can exchange data. On the other hand, DLM refers to the concrete label format and semantics that were introduced in the same paper and then later used in Jif/JFlow [87].

readers is associated with an owner (e.g. $o \rightarrow \{u_1, u_2\}$), whose authority is required to relax the policy (i.e. add readers to the set $\{u_1, u_2\}$, possibly up to $o \rightarrow \bot$, thus eliminating the restriction altogether) and perform declassification. In our semantics, the formula implicitly represents a set of users who are considered both owners and readers. Because of this, declassification works differently: while in DLM a label $o \rightarrow \{u_1, u_2\}$ requires the authority of the user o for declassification, in our semantics both users u_1 and u_2 can add new readers and owners. This is shown in the definition of $E(d(e, u))$, where $P(u)$ removes all permission formulas p for which $P(p, u) = T$.

Note that our design does not prevent us from defining and tracking ownership, maintaining different sets of readers and owners; we simply opted to keep our design simpler and accept that in a Web Application context, where users display their data on unsecured browsers out of the control of the WebSheet runtime, restrictions on declassification can be easily circumvented out-of-band, thus making the distinctions between readers and owners less valuable. The semantics could be easily extended to include a declassification formula for each cell which defines a set of owners that can declassify its read permission formula: the permission tables shown in Section 4.1 could be extended to include a row "Declassify". Also note that models that unify the concept of readership and ownership have been used successfully in previos work [42]. In particular, our semantics are similar to the DC model [124], which expresses labels as conjuctions of disjunction of principals. It is easy to see that in the DC model $a : (u_1 \vee u_2)$ and $b : u_3$ imply that $a + b : (u_1 \vee u_2) \wedge u_3$ and that the same can be expressed with WF formulas: a : `user in [u1, u2]`, b : `user == u3`, with both semantics from Figure 26 and 27 executing both WF formulas and requiring both to return T during the evaluation of $a + b$. However, like in the DLM, the users in DC Labels are also fixed at the time of creation of the label.

The main reason why the eager enforcement semantics from Figure 26 are preferable to those from Figure 27 (especially after we have shown that, in the context of functional evaluation, eager enforcement does not prevent declassification), is because the latter semantics are vulnerable to covert channels.

5.2.3 Semantics vs. Implementation

The implementation described in Section 5.3 differs from the semantics shown in Figure 26. While some features have been left out of the operational semantics to focus on evaluation, others represent design decision that were

made before the semantics were formalized; because they represent trade-offs which do not affect the core functionality of the paradigm, we did not update the implementation to precisely follow the semantics.

In particular, the following differences are notable:

- The implementation performs deferred enforcement instead of eager enforcement of read permissions, building up permission sets during evaluation. These are evaluated and verified in `getVal` before the data is sent to the user. It is thus more similar to the semantics shown in Figure 27 than those from Figure 26. However, deferred enforcement yields more opportunities to exploit covert channels, which motivated us to define eager enforcement semantics. Section 6.2 describes how to mitigate these covert channels.

- Besides the `getVal` operation, the repository also supports the `setVal` operation. The store S returns an additional write permission formula, thus requiring the following signature for S:

  $$S :: @l \to (u, \mathrm{WF}_s, \mathrm{WF}_s, \mathrm{WF}_s)$$

  However, write permissions do not flow during evaluation (i.e. there is no IFC enforcement of integrity). Rather, the formula is used to check whether a single user can write directly into a specific cell.

- `setVal` is not the only operation that modifies the store S. Besides creating and deleting tables, the implemention defines `addRow` and `delRow` operations, which are subject to different permission checks.

- Although all tables are accessible as rows and columns of values after evaluation, not all tables are directly generated from rows and columns of WF expressions. The implementation supports dynamic tables, which are tables whose source is a single WF expression that must produce a list of uniformly named tuples. The values are then spread over rows and columns and accessed transparently.

- The implementation implements lazy evaluation and caching semantics. The repository maintains a global cache of values and a per-user cache of read permission check results. The read permission formulas that flow during evaluation are actually expressed in the form of cell references, so that they can be used to dynamically build a dependency

graph and update the cache as needed. When a cached value becomes stale or an entirely new cached table is created, the runtime simply stores informations on how to evaluate the original formula, but does not perform evaluation until getVal requires the cell value.

- The implementation introduces the concept of temporal dependencies, which can be used to force re-evaluation of cached values after a certain date, or to trigger evaluation indendently of getVal.

- The implementation extends WF with the ability to perform side effects during evaluation of formulas, by providing the ability to call an external procedural script. This is useful if the application needs to perform I/O (e.g. send mail) or to encapsulate the effects of a transaction (e.g. update the remaining number of seats).

  However, this complicates the semantics: since evaluation is not side-effect free, the semantics cannot re-evaluate data arbitrarily. In the implementation, we tackle this issue using caching and short-circuit semantics (e.g. `guard && script()` does not run `script()` until `guard` evaluates to `true`). The result is kept fresh in the cache and not re-evaluated until any of the dependencies of the formula are changed). The alternative is to require an explicit trigger (e.g. a user initiates the script by clicking on a cell). In this version, the script resides in a separate cell as a read-only formula, and it triggers the execution of a series of assignments to temporary variables and to other cells.

  Procedural scripts should have permission semantics that work organically with the rest of the system. Suppose that there is an assignment $@x := @y + @z$, where all three variables reference cells. Assuming that $E(@y + @z, ...) = v$, we can keep consistent permissions for the new constant value v written in $@x$ by writing $@y; @z; v$ into the cell $@x$, where the semicolon operator evaluates the expressions in order, but only returns the last expression. This way, evaluating $@x$ now returns v, but also requires $P(@y, ...)$ and $P(@z, ...)$ to succeed.

5.3 Implementation

To focus on describing the core semantics of evaluation, permission checking and declassification, Section 5.2 abstracted away important details about how a practical WebSheets implementation would look like. In this section,

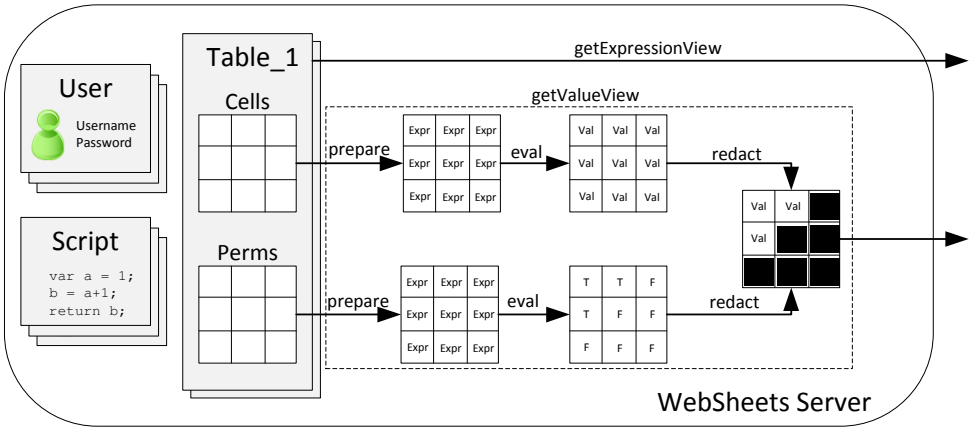

Figure 28: Overview of WebSheets' Evaluation Flow

we describe the actual implementation of WebSheets. We begin our illustration of the implementation by giving a high-level overview of the data model, showing how information flows to end users. Subsequently, we describe the design in more detail by defining the structure of WebSheet tables and explaining their transformation as input formulas are evaluated into values and cached. Finally, we show how permissions are applied and how values are redacted before being sent to end users.

5.3.1 Overview

A WebSheet repository consists of tables of WF formulas, user accounts and user-submitted scripts for procedural data processing that can be safely executed in a sandbox. Figure 28 depicts an overview of the system, showing in particular how these tables are transformed before they are sent to the end user using two different operations, `getExpressionView` and `getValueView`. The former exposes the "Expression View" of a table, a low-level view showing tables as pairs of data tables and permission tables; the operation is only available to table owners and privileged users, and the view is mostly used during development to manage new tables and configure their permissions.

On the other hand, `getValueView` exposes the "Value View" of a table, a high-level view that generates intermediate "output" tables (`prepare`), causes the evaluation of both data and read permission formulas (`eval`) and finally combines the two to produce a redacted result that is sent to

Data Table

	C_1	...	C_n
R_0	Expr	...	Expr
...	...	...	...
R_n	Expr	...	Expr

Permission Table

	C_1	...	C_n	Row
read	Expr	...	Expr	Expr
write	Expr	...	Expr	Expr
add row				Expr
del row				Expr
init	Expr	...	Expr	

Figure 29: WebSheets' Input Table

a specific user (`redact`). This operation is available to all users and, unlike `getExpressionView`, is used during normal operation to interact with existing tables that have already been configured. The purpose of the Value View is to offer an interface similar to spreadsheets, by providing a unified view of input formulas and output values: although the cells displayed to the user contain output values, the user can enter input formulas which are written to the original table, immediately evaluated and displayed to the user as new output values.

Figure 29 shows the format of tables. Besides a name and a description, all tables have an *owner* (i.e. the creator of the table), which can be referenced at runtime. Each table consists of two sub-tables: a data table and a permission table. Internally, these tables are represented as arrays of objects with the same set of properties, which represent the column names of the table. The implementation can thus reference a specific cell using normal object access, i.e. `table[row][col]`; we use the same notation in this section.

Although omitted from the figure, rows and cells also have an author: the row author is the user who added the row, while the cell author is the user who last modified the formula. The format of the permission table depends on the format of the data table: the number of rows is fixed and represents the various permissions like `read`, `write`, etc; the columns are the same columns from the data table plus an additional *row* column. The columns named after the original columns of the data table define permissions that are in scope for all data cells along the column; for example, `perms.write.col1` defines the write permission for all cells along the `col1` column. The `row`

column can be used to specify a permission formula in scope for all cells. If both the column and the row permission are defined for a particular cell, the actual permission is the conjunction of the two.

As shown, in Figure 29, not all the cells of the permission grid are meaningful and can be edited; for example, users cannot define an `init` value for the entire row, because `init` values are specified per-column. The formulas supplied by users as strings are contained in `Expr` objects, which automatically parse the formula according to the WF grammar described in Section 5.1, so that the formula is available both in textual format (for displaying and editing) and as an abstract syntax tree (for evaluation).

5.3.2 Expression View

Despite allowing users to edit formulas directly from the Value View, WebSheets allow users to view and edit tables directly through the Expression View. The client fetches the necessary data using the `getExpressionView` operation. The semantics for `getExpressionView` are straightforward, because unlike `getValueView`, the operation does not cause any evaluation. Instead, the server merely checks that the user requesting the operation is the owner of the table and then returns both the value and the permission table.

5.3.3 Value View

Value Views offer an interface more similar to spreadsheets: the value and permission tables are combined together into a single, redacted table that the users can interact with. The operation `getValueView` supplies the data to the client; we define its semantics using a top-down approach, by describing its three main phases: the operation causes the server to a) generate fresh output data tables and output permission tables if they do not exist yet (`prepare`), b) force evaluation of all the value and permission cells for the current user (`eval`) and c) produce a redacted output table by intersecting the output values and the output permissions (`redact`). The operation is depicted in Figure 28.

We now define the three sub-operations that this pseudocode depends on, namely `prepare`, `eval` and `redact`.

prepare

The `prepare` operation creates fresh output value tables and output permission tables for a specific user. The purpose of these output tables is not only to store the result of the evaluation of the formulas of the original table for the next phase, but also to cache the results for later requests. Note that there is one cache of output data tables for all users, while each user has its own cache of output permission tables. This is because, as shown in Section 5.2, the evaluation result of permission formulas depends on which user is performing the evaluation, while the evaluation result of data formulas can be shared among all users.

The output value table is generated from the input table by simply wrapping all cell formulas of the input value table in a `Cacheable` object. The purpose of `Cacheable` is to encapsulate lazy evaluation and caching semantics: the `prepare` operation simply copies input formulas into the object, delaying actual evaluation to the `eval` operation, which also stores the result in the object; subsequent calls to `eval` can immediately return the cached result.

The output permission table is also generated from the structure of the input data table (i.e. it has the same number of rows and columns as the data table, unlike the permission table shown in the expression view), and it is filled with read permission formulas from the input permission table: each cell contains the `read` column formula from the input permission table ANDed together with the generic `row` read permission. These are also wrapped in a `Cacheable` object.

eval

The `eval` operation produces a `Value` from an `Cacheable`, either evaluating the WF AST in `Expr` or by returning the cached value. The evaluation semantics of most WF AST nodes are straightforward, since they build values bottom-up by first evaluating the children and then combining them into a new `Value`. For example, Section 5.2 shows how to implement addition. However, a few `Expr` nodes have notable semantics:

- *Identifiers*: these are first looked up in the environment, which is a mapping from strings to `Values`. If found, their value is returned; if not found, they are assumed to be references to a table. If the table exists, `eval` does not evaluate any cell of the table, but only returns an unresolved reference to it, a `TableValue`.

- *Selection and Projection*: resolving the aforementioned `TableValue` reference would cause the evaluation of all cells in the table. This is not only wasteful, but can also creates unnecessary dynamic dependencies. For this reason, a few operations can act on a unresolved reference without causing its resolution. The most common cases are selection and projection: on lists and tuples, they return a subset of the elements; on `TableValues` they perform the same function, but do not cause the reference to resolve. Rather, they cause the `TableValue` to "focus" on a subset of the table, which will cause only a subset of table cells to be evaluated and returned upon resolution.

- *Generate and Filter*: like selection and projection, Generate and Filter also avoid resolving `TableValue`s when looping over their rows. For example `T[a==b]` is equivalent to projecting T on all the rows that satisfy the condition in brackets. Thus, only the columns a and b of each row are evaluated.

Any other operation will immediately resolve a `TableValue`. The type of `Value` returned upon resolution depends on how focused the `TableValue` is (i.e. what subset of the original table has been requested using selection, projection and filtering). Generally speaking, `resolve` will expand a single `TableValue` into as many `TableValue`s as the number of cells focused by the original `TableValue`, and then proceed to resolve them one by one.

The basic operation is the resolution of a `TableValue` focused on a single cell, which entails looking up the `Cacheable` wrapper in the cell of the corresponding output table and evaluating it. A `TableValue` focused on a row will generate a record of `TableValue`s focused on each field and evaluate it. A `TableValue` focused on multiple rows (or an entire table) will generate a list of `TableValue`s focused on each row and evaluate the list.

WebSheets rely on dynamic dependencies both for permissions (i.e. a cell can be read if the user is allowed to read the cell itself and all the cells it depends on) and to implement minimal recalculation of cells upon chages (by maintaining a dependency graph). So far, we did not describe how evaluation drives the calculation of dynamic dependencies. All `Value` types store a list of cells they depend on, which are calculated bottom-up and pushed up to the final `Value` during evaluation. The rules for propagating dependencies for most nodes are straightforward (i.e. `a+b` will depend on the dependencies of both `a` and `b`), with only two notable cases:

- *Short-Circuit Evaluation*: IF statements, logical AND and logical OR are implemented with short-circuit semantics. If a value is not evaluated, its dependecies don't flow to the result.
- *TableValue Resolution*: When a `TableValue` is resolved, the result depends on the cell referenced by the `TableValue` (remember that although a `TableValue` can reference sets of cells and sets of rows before resolution, these are expanded into multiple resolutions of single cells). In practice, this is the only source of dependencies in formulas.
- *Lists and Tuples*: Lists and Tuples don't automatically inherit all dependencies from their elements. The runtime extracts all nested dependencies when calculating cell dependencies, but `redact` can use this information to work on a per-element basis. This prevents an entire collection from being redacted if the user does not have access to a single element.

redact

Finally, we describe the last operation that `getValueView` depends on. `redact` reviews an output value table together with its read permission table for the current user. For each cell of the output value table, the runtime checks that:

- the corresponding cell in the permission table evaluates to `true`, and
- for each dependency collected during evaluation of the permission, the corresponding permission cells evaluate to `true` as well.

If either condition is not met, the value is blanked out before it is sent over the network.

5.3.4 Editing Tables

Besides `getExpressionView` and `getValueView`, WebSheets expose several operations to modify existing tables.

Firstly, note that WebSheets use Information-Flow Control only on read operations, to ensure confidentiality; there is no dual mechanism enforced to ensure integrity [9]. Instead, write operations use traditional access control, evaluating the appropriate formula in the context of the user and ensuring

that the user has read access to all the dependencies. Secondly, these permission formulas are not cached — they are ignored by the `prepare` operation, which doesn't copy them into the output table and doesn't wrap them into a `Cacheable` object. Instead, the expression is always fetched from the input table and re-evaluated. Although caching semantics would be simple and similar to those of read permissions, the result of other permission formulas is not required as often and thus caching would not significantly improve performance. This simplifies the semantics of edit operations:

- *WriteCell*: this operation changes the formula of an input cell. To verify that the user is allowed to perform the operation, the runtime takes the conjunction of the write permission for the current column and the generic row write permission and evaluates it. The value must evaluates to `true` and all its dependencies must be readable (i.e. their read permission formula must evaluate to `true`). Unlike read permission formulas, write permission formulas can also refer to the new value of the cell using the identifier `newVal`. This is useful to implement validation logic.

- *WritePerm*: this operation modifies a permission on the input table. Only the table owner can change the table's permissions. In practice, this means that the table owner also "owns" the data of all its cells, since he has read and write access to all of them: if a policy prevents him from accessing the data, he can modify it to remove any restrictions. This is no different than current web application development — users must trust the developers not to misuse the data. We did not investigate or implement the possibility of "locking in" a policy for a piece of data, effectively enforcing a contract between user and table owner that is stipulated at the time the data is entered.

  However, users can still retain control of their data using one extra step: they must enter data into the system using a separate table of their own making, referring to it when supplying data to an untrusted third-party. This way, the third-party can only add new restrictions, but must still satisfy the read restrictions imposed by the user's table, where the data is originally extracted from.

- *AddRow*: this operation adds a new row to the input table, either by specifying an index or by appending it at the end of the table. The

new row is initialized using the cells of the `init` permission row. While the owner of the row is the user who executed `addRow`, the owner of the cells is the owner of the tables (i.e. the owner of the init cells), until the user modifies the default values. The operation is executed only if the `add row` permission evaluates to `true`.

- *DelRow*: this operation removes a row from an input table, but only if the `del row` permission evaluates to `true`.
- *CreateTable*: this operation creates a new empty table, and its caller becomes the table's owner. The user must specify the set of columns at the time of creation; on the other hand, permissions default to "allow all" (i.e. empty formulas) and can be modified at a later time using `WritePerm`. This operation is not privileged and any user can create a new table.
- *DeleteTable* This operation deletes an existing table. Only the owner of the table can perform this operation.

5.3.5 Dependency Recalculation

Beside being used to implement deferred enforcement, dynamic dependencies are also used to decide which cells must be recalculated upon changes to the input tables. Minimal recalculation is not only useful to improve performance and responsiveness upon edits: cell formulas might perform I/O upon evaluation (e.g. send an email) that users do not wish to perform multiple times unless new inputs are available to the function that performs I/O.

Since evaluation is lazy, minimal dependency recalculation involves two different subproblmes:

1. after a modification to the input tables (e.g. WriteCell, AddRow), the runtime must marking a minimal set of evaluated values in the cache as stale. We use the dynamic dependency information collected during evaluation to visit the support graph of the elements that have been modified, marking all the nodes reachable from these items as stale.
2. upon evaluation, decide if a stale cell actually needs reevaluation. This is not always the case: the update that marked the dependencies as stale might still result in the same value, thus causing no changes to propagate. We maintain a generation number to detect this.

Marking a minimal set of cells as stale is implemented through the `trigger` operation, which is called with an event type and a set of arguments every time input tables are modified. The operation supports the following event types:

- *WriteCell*: this is called when an input cell is modified. WriteCell is the most common trigger and thus the one that requires the most efficient semantics. Since we maintain a dependency graph, inverting the edges and visiting the subtree with the current cell as the root yields the support set for the current set, i.e. the set of cells that depend on the current cell that was modified. WriteCell simply marks the current cell and every cell in the support cell as stale.

- *WritePerm*: this is called when a read permission for a cell or a row is modified (other types of permissions are not cached). Because no formula can explicitly refer to a permission formula, the support set for read permissions is always empty; however, the cells of the permission cache for which the current permission is in scope should be marked as stale. When a permission for a column is modified, all the cells on that column of the permission table for all users are marked as stale; when a row permission is modified, all cells in the permission table for all users are marked as stale.

- *WriteOwner*: this is called when the owner of a row is modified. WriteOwner is a privileged operation that does not happen often: normally, the creator of the row its owner as well. In this case, we can still restrict the number of cells that are marked as stale: although dependencies on owner cells are not tracked explicitly, only the cells of the current row can reference its owner (note that the owner can still be accessed indirectly across different rows or tables by setting a cell of the row to `owner`, and then referencing the cell.). Thus, we can approximate the support set by calling `trigger` for all the cells of the row.

- *AddRow*: this operation is called when a new row is added to an input table. Firstly, the server updates output and permission tables by adding a new row at the same index. The initial value for each cell of the new row of the output table is `Cacheable(expr)`, where `expr` is the vaue of the cell in the corresponding input table cell. The initial value for each cell of the new row in the permission table is initialized

as the union of the current column read permission and the row read permission.

Besides updating the data structures, `addRow` can also cause existing cells to be marked as stale in two cases:

- If a `TableValue` references the current row in a looping construct like Filter or Generate, the update will cause the construct to return a different result (e.g. returning an extra row or array element). In this case, `TableValue` returns a special type of dependency that is used by addRow to recognize this exception and mark these cells as stale.
- If a `TableValue` references a specific row and a new row is added before the row referenced, the index of the row will increase by 1. This means that all cells referencing rows beyond the current one must be marked as stale. This can be avoided by distinguishing between relative row references (e.g. current row) and absolute refenrece (e.g. fourth row), so that only cells containing absolute references beyond the inserted index need to be marked as stale. Note that this optimization only saves work if the row is added in any position before the very last; since most `addRow` operations add a row at the end, we did not perform this optimization.

- *DeleteRow*: this operation is called when a row is deleted from a table. The update semantics are similar to `addRow`, including the caveat about deleting any row but the last. In addition, `deleteRow` must mark any cell that refers to the current row as stale.

- *CreateTable*: this operation is called when a new table is added to the server. Because output table are generated as needed by `prepare`, no changes to output tabes are needed at this point. Note that if a cell refers to the current table, we do not need to mark the cell has stale to inform the cell that the table finally exists. Its value would be of type `Error`, because the table did not exist yet, and error cells are always considered stale by recalculation semantics.

- *DeleteTable*: this operation is called when a table is deleted from the repository. Besides deleting its corresponding output table and permission tables for all users, all cells referencing any cell on this table must be marked as stale.

The `trigger` operation only marks cells as stale. Re-evaluation, if necessary, happens during `eval`. All cells have a generation number that updates when the input cell is modified or when the output cell's value is updated; a stale cell can be *revived* if all its dependencies (that at this point have already been evaluated) have a generation number that is less than the current generation number. If a cell is revived, the old value is marked as fresh and the generation number is not updated, thus notifying the cell's support set that no recalculation needs to happen because of this cell.

5.3.6 Built-in Functions

WF function calls can execute builtin functions or user-submitted scripts. When evaluating a function, `eval` first tries to look for the function name in the set of builtin functions provided by the websheet server. Besides the usual functions that one might expect from a spreadsheet (e.g. `sum`, `avg`, etc), WebSheets have five functions with notable semantics, whose names have been capitalized to denote unusual semantics with respect to side-effects, dependencies and permissions:

- *decl*: this function performs declassification on a value. When any user performs `eval` on the cell, all the dependencies that are readable by the author of the cell (i.e. the user that inserted the `decl` call) are marked as non-enforcing and are not used to verify read permissions in `censor`. The dependencies are not removed altogether, because they are still used for recalculation.

- *fix*: this function prevents recalculation on a value, making it constant by turning off recalculation and returning it. All dependencies on the result are marked as non-recalculating and do not affect whether the cell needs recalculation any longer.

- *after*: this function introduces a *temporal dependency*. Instead of depending on changes to another output cell, a temporal dependency registers a callback to mark the cell as stale at a specific time, so that it will be up for recomputation the next time a user needs its value. The function accepts a date and returns `true` when the current data is after the date provided to `after`, and false otherwise. This allows `after` to be used along with short-circuit semantics to prevent evaluation of another formula until a certain time. For example, `after("tomorrow")`

`&& mail("test mail", ...)` prevents users evaluating the cell from sending an email until the next day.

- *trigger*: this function also introduces a temporal dependency and registers a callback, but instead of causing the value to be marked as stale, the callback immediately triggers the evaluation of the cell as the owner of the cell (i.e. the user who wrote the `trigger` call).

- *mail*: this function is used to send mail outside the system. Because data leaving the system cannot be subjected to further permission checks, the function verifies that the user evaluating the function can read all the dependencies of the email's body.

  When a username is supplied in place of the email address, the function checks that the user specified has read permissions instead, and sends the message to the email address associated with his account.

5.3.7 Scripts

If no combination of WF operators and builtin function calls is expressive enough for a particular task, users can register and call scripts that are run securely in a sandboxed environment. If the sandboxed script needs to interact with the websheet server (i.e. the script does not merely process inputs to produce an output, but also reads and writes cells), the script can do so using the same RESTful API that is exposed to users through the web interface. Thus, the script is subject to the same security restrictions. We provide two environment for executing untrusted scripts:

JavaScript: since the current implementation of websheet runs under Node, the simplest and fastest way to sandbox code is to use a JavaScript sandboxing solution. In particular, JaTE [135] enforces strong isolation despite running the untrusted code in the same process. However, JavaScript sandboxing in the context of WebSheets has two shortcomings:

- JaTE (and other similar sandboxing solutions like Ses [84]) enforces isolation but does not address the problem of transparently exposing the Node API, which is require for code reuse and to leverage existing libraries. For example, sandboxed scripts cannot use the `require` function to import Node modules, because many

Node modules interact with the underlying system (e.g. the `fs` module). A `require` implementation for sandboxed scripts must return "clean" modules that do not have these capabilities, which requires manual analysis of a multitude of libraries. To solve a similar problem in Java, Joe-E [83] released *taming libraries*, patches to popular libraries that wrap or delete unsafe methods.

- A JavaScript sandbox forces users to write their own code in JavaScript. If the user is more familiar with a different scripting language or wants to run existing (possibly binary) code, this sandbox is too restrictive.

LXC-Shell: while the first sandboxing method focuses on speed and simplicity, the second provides generality. We use LXC to quickly spin up a throwaway Ubuntu-based container, configure it to access the internet and the WebSheet server itself, and then run the user provided bash script as an entry point. From there, the user can perform arbitrary actions – including writing to disk. Before the container is destroyed, the script can report back a return value to the server, and the control flow returns to the WF formula that evaluated the script. Note that running untrusted program as root in LXC containers is not recommended, and thus we drop root privileges before executing the untrusted script.

Both types of scripts can be uploaded either as normal or setuid. Normal scripts are executed as the user invoking the script (i.e. the cell owner); the user must trust the script he is invoking. Setuid scripts are executed as the author of the script; in this case, the script author must sanitize the inputs and consume them safely. Also, both types of scripts can only be executed if the user running the script (according to the setuid bit) has read permission on all the dependencies of the inputs. This is necessary because the server cannot stop unauthorized data exfiltration beyond this point. It also allows script caller semantics to leave propagation of input dependencies up to the callee — the script can implement `decl-` and `fix`-like semantics, preventing access control and recalculation, or return all dependencies to enforce reasonable read permissions or ensure proper recalculation semantics.

5.3.8 Status

We have implemented WebSheets as a web application which communicates with a backend server written in NodeJS through a JSON API. Through the

web application, users use the backend as a central repository of applications to collaborate with other users.

The implementation of the server and the UI is available on GitHub [102] under the open source GPL3 license. To simplify deployment, a Docker image is available as well [103]. All the features previously discussed have been implemented. In particular, the codebase contains:

- an LR *parser for the WF language*, based on the grammar shown in Figure 24. The Jison grammar is contained in `ws.jison` and the ast definitions are located in `ast.js`.
- support for both *static tables* and *dynamic tables*. Tables are defined in `input.js`, while output tables are defined in `output.js`. Tables are instantiated throughout the codebase on demand by calling into the methods defined in these files.
- *evaluation, caching and recalculation* of data and read permissions. Triggering and permission calculation are defined in `websheets.js`, while evaluation is mostly implemented in `ast.js`.
- support for *redacting* output values, including fine-grained redaction of list and tuple elements. The code is mostly implemented in `ast.js`.
- *access control* on the edit operations, including redacting values when the user does not have the required read permissions. Defined in `websheets.js`.
- two different systems for *sandboxed execution of user-submitted scripts*. The JavaScript sandbox is defined in `sandbox.js`, while the Docker image, based on Ubuntu 14.04, is defined in `docker/`.
- support for *importing WebSheets from XLS files*. Although WebSheets can be created from scratch and managed within the web interface, we also support reading XLS files with a custom format. Importing WebSheets from XLS files is very useful for testing because a) they survive database resets and b) the format also allows pre-populating the table. The import code is in `import.js`

The frontend implements the Single Page Web Application architecture: all HTML, CSS and JS files are static and are served under the `static/`

directory. The `routie` library listens to `location.hash` changes and updates the main content pane using the `jQuery` library. WebSheet operations are requested to server using the JSON api. The only pages that have non-trivial client-side logic are the expression view and the value view. Both of them use the HTML5 `contenteditable` attribute to allow users to edit values in place.

The server is implemented using the `express` framework. The purpose of the server is simply to a) manage the lifecycle of the actual `WS` object, which encapsulates the application logic, b) convert JSON requests into operations on the `WS` object and c) serve the frontend as static content. In particular, the JSON API includes the following endpoints:

- `/user/login`: used to login to the WebSheet application by creating a new session. Most WebSheet operations are invoked with a `user` argument and are not available if the user is not logged in. For debugging, the server also allows the administrator to execute operations as any other user.
- `/admin/[save|load]`: used to serialize and deserialize the state of the web application as a JSON file.
- `/table/create`: executes the `CreateTable` operation on the repository.
- `/table/:name/delete`: executes the `DeleteTable` operation on the .
- `/table/:name/input`: executes the `getExpressionView` operation on the `:name` table.
- `/table/:name/output`: executes the `getValueView` operation on the `:name` table.
- `/table/:name/edit`: executes either the `writeCell` or the `writePerm` operation on the `:name` table, depending on the request body.
- `/table/addrow`: executes the `addRow` operation on table `:name`. If the index is missing, the operation defaults to adding the adding the row to the bottom of the table.
- `/table/:name/:row/deleterow`: executes the `deleteRow` operation on table `:name`, deleting row `:row`.

6 Evaluation

6.1 Case Study

Section 4.1 used three simple examples to give an overview of WebSheets' design and capabilities and to demonstrate its ease of use. Because the nature of that section was mainly didactic, we did not try to demonstrate that Websheets can be used to effectively develop real-world, complex web applications. This section is focused on this task, and presents a rewrite of the popular HotCRP conference management system.

6.1.1 HotCRP

HotCRP [71] is a popular open-source web application developed by Eddie Kohler that allows conference organizers to automate the paper submission and review processes. Over the years, HotCRP has been often been featured in the evaluation section of web security papers; it has been featured both in vulnerability analysis papers that sought to uncover data leaks in existing applications [68] and in policy papers that offered centralized policy specification, to show that they were able to specify its complex policies [145, 100, 144]. Its popularity is due to its substantial privacy requirements: at the very minimum, a modern conference system must securely handle double-blind anonymity and conflicts of interest between PCs and submitters.

Over the years, the author had to fix several information leaks; the project changelog contains 14 mentions of leak-related fixes, although detailed information is not available for all of them. When permission checks are strewn across the codebases and look like the snippet shown in Figure 30, which shows the complexity of the formula to decide whether a user can access a paper's reviews, it is not surprising that the code contains data leak vulnerabilities.

In particular, one vulnerability described in detail by the author caused HotCRP's password recovery feature to reveal the login information of any user when running in "email debug" mode, as shown in Figure 31.

This vulnerability clearly shows the problem with traditional web application development practices: the application logic pushes sensitive data out of the system all throughout the codebase, and the developer must place the appropriate policy checks accross each possible path; if even one path is not

```
return (($this->privChair && $forceShow)
  || ($prow->timeSubmitted > 0
      && (($prow->conflictType >= CONFLICT_AUTHOR
           && $conf->timeAuthorViewReviews() && $rrowSubmitted
           && (!$this->reviewsOutstanding || !$this->isReviewer))
          || ($this->privChair && $prow->conflictType == 0)
          || ($this->isPC
              && $prow->conflictType == 0 && $rrowSubmitted
              && ($conf->timePCViewAllReviews()
                  || defval($prow, "myReviewSubmitted") > 0))
          || (defval($prow, "myReviewType") > 0
              && $prow->conflictType == 0 && $rrowSubmitted
              && defval($prow, "myReviewSubmitted") > 0
              && ($this->isPC
                  || $conf->settings["extrev_view"] >= 1))
          || (defval($prow, "myReviewType") == REVIEW_SECONDARY
              && $prow->conflictType == 0 && $rrowSubmitted
              && $prow->myReviewNeedsSubmit === null)
          || ($rrow && $rrow->paperId == $prow->paperId
              && $rrow->contactId == $this->contactId))));
```

Figure 30: HotCRP Review Read Policy

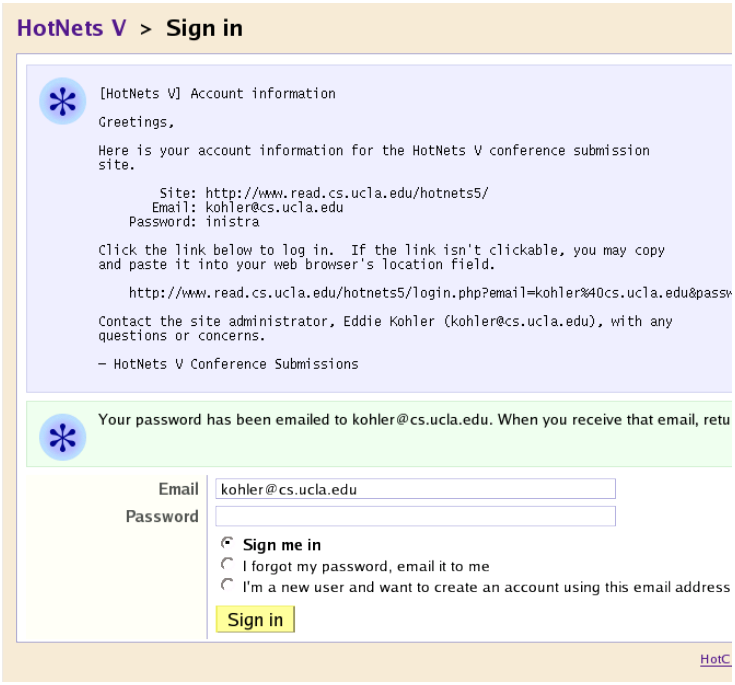

Figure 31: HotCRP Password Leak

properly handled, an attacker will identify it as the weakest link and exploit it. In this particular case, the obscure flow is only present if a debug flag is active.

The author himself, a renowed academic with experience in Information-Flow Control [34], is aware of the problem and proposes two mitigation strategies [70]:

- minimize the number of distinct outgoing flows of sensitive information, to decrease the chances of one flow not being sanitized properly: one of the initial design decisions of HotCRP was to *reduce modes*, i.e. reduce the number of different views on the same set of data. This is a sensible design decision that mitigates the problem, but does not eliminate it.

- use an information-flow control system. However, at the time his thoughts on the matter were published, the author did not believe that IFC systems were expressive enough to handle HotCRP's policies [70], and used the formula in Figure 30 as an example of complex logic.

The researchers behind Resin [145] and Jeeves [144] implemented a significant subset of HotCRP's policies in their IFC systems; since Resin includes a PHP runtime, the language HotCRP is developed in, the authors leveraged the original implementation, simply adding data flow assertions to the system; on the other hand, since Jeeves implements IFC in Scala, the authors implemented a clone of HotCRP named JConf, which implements a subset of the application logic. WebSheets also require us to rewrite the web application from scratch, which we unimaginatively call WSConf. However, in our case we are not only proving that the privacy policies can be expressed in our paradigm, but that the application logic can be expressed as well. For this reason, compared to Resin and Jeeves, we implemented nearly all the features and configuration options of HotCRP, with only a few exceptions noted below. In particular, WSConf implements the following features:

- *double-blind reviews*: PC members cannot view the authors of the papers submitted to the conference, and authors cannot view the name of the reviewers.

- *attachments*: support for uploading and downloading PDF files directly from the repository.

- *conflicts*: prevent PC members from seeing reviews of papers of conflicting authors.
- *deadlines*: support for a submission deadline and a review deadline, preventing new submissions and new reviews respectively.
- *groupthink prevention*: reviewers can see the other reviews for a paper only after submitting their own.
- *email notifications*: reviewers are notified when they are assigned a paper; authors are notified when the chair has decided whether to accept or reject the paper.
- *review preferences*: allow PC members to communicate their willingness to review a particular paper.
- *automatic review assignment*: automatically assign n reviewers to each paper, taking the aforementioned preferences into account.
- *tagging*: support assigning tags to papers. Certain tags are privileged and can only be assigned by the chair.
- *review rating*: supports the rating of reviews by other reviewers, to discourage low-quality reviews.

Because HotCRP is designed to support the use cases of many different conferences, the aforementioned features can be customized through configuration options. We have implemented a significant subset of them. In particular, we ported the following HotCRP configuration options:

- *deadline override*: force acceptance of new papers and new reviews even when their respective deadlines have passed.
- *submission anonymity*: controls whether submission anonymity is always on, always off or opt-in.
- *review anonymity*: controls whether reviewer anonymity is always on, always off or opt-in.
- *non-assigned reviews*: controls whether PC members can spontaneously review papers that have not been assigned to them.

- *groupthink prevention*: controls whether reviewers can always see the other reviews for a paper or only after they have submitted their own.
- *rating reviews*: controls whether reviewers can rate other reviews.

Although we set out to write a realistic HotCRP clone instead of a toy application, we did not quite reach feature parity. The following HotCRP features were not implemented or were simplified:

- *review assignments*: we implemented assignments to showcase the use of JavaScript sandboxes to support complex procedural logic, but we believe replicating all the assignment logic of HotCRP is unnecessary. The problem of optimizing review assignments is NP-hard, so even their algorithm is based on heuristics; in particular, while their algorithm focuses on fairness (most likely, to prevent PC members from bothering the chair about their assignments!), we simply implemented a greedy algorithm that scans through each paper and assigns reviews to the PC members with the highest preference, after filtering for conflicts and review quotas. The original code lives in `src/autoassigner.php` and we believe that porting it to JavaScript would not present significant conceptual challenges.
- *PDF format checking*: HotCRP optionally uses Banal to check whether the format of submitted PDF files conforms to the format specified by the conference. Although LXC sandboxes would be able to interface with this program, we believe this is out of scope.
- *external reviews*: since WebSheet operations are only available to registered users, there is no concept of external reviewer. They can be simulated by manually assigning a review to a WebSheet user that is not on the PC committee, which does not require additional application logic.
- *multiple rounds of reviews*: the privacy policy of reviews is independent from their round. The code to support this feature is therefore related to review assignment, which we only implemented in a simplified way. Note that a second round of reviews can still be assigned manually by the chair.

- *custom submission options and review fields*: while HTML applications need special code to support additional fields, WebSheet developers can easily add a new column to a table to accept further information for each entry.
- *tracks*: tracks use tags to specify different permissions for a subset of submission or a subset of PC members. Although this feature is convenient, the same behavior can be otained by deploying different WSConf instances and configuring them appropriately.

The application requires 7 WebSheet tables: `Committee`, `Config`, `Paper`, `Preference`, `Review`, `Tag` and `RReview`. Unlike the examples in Section 4.1, this example is too involved to be included in this document in tabular format. Our project repository includes the file `hotcrp.xls` which can be used to either view the formulas without a WebSheet repository or to import the application into one. Below, we describe the purpose of each table and a high-level overview of the policies applied to their data.

The first table we describe is `Committee`. This table is readable by all users and writable only by the program chair. The table simply contains a list of usernames that make up the program committee. In practice, the table is used to denote membership to the *PC* role: although WebSheets do not have built-in RBAC support, it can be easily implemented through WF formulas using a table with a list of users and roles. In this case, since the only role is PC member, we omit the role column; simply putting a username in the table confers membership to the PC group. If we did not assume that the chair is the user `admin`, we could add an `isChair` column that specifies whether the username on the same row is the PC chair or a PC member. In other tables, we check if the user belongs to the role using `user in Committee.member`. The chair should fill this table before performing review assignments (although in practice the PC is already known way ahead of time).

Secondly, we describe another table that should be filled ahead of time, `Config`. This table contains flags to configure the behavior and privacy settings of the system. `pDeadline` contains the submission deadline date; `opDeadline` can be set to `true` to override the submission deadline and re-open WSConf for submissions; `rDeadline` and `orDeadline` perform similar functions for the review deadline. `sAnon` decides whether submissions should be anonymous. If set to `false`, then `sAnonOptIn` decides whether paper authors can choose whether their submission is anonymous on their own using

the `anonymous` field in their submission; otherwise, all authors are visible to PC members unless there is a conflict. `rAnon` and `rAnonOptIn` perform a similar function for reviewer anonymity. Note that while paper authors are unblinded after an acceptance decision has been made, anonymous reviews remain anonymous. `rAssigned` controls whether PC members can submit reviews for papers they have not been assigned on. `rGroupThink` controls whether reviews are visible to all PC members (barring conflicts) or if reviewers must first submit their own review for a paper before seeing other reviews. `rRating` controls whether reviewers can anonymously review other reviews. Finally, `tPriv` contains the privileged set of tags that only the chair can write, such as `accept` and `reject`.

The table also contains two cells that work as a trigger: setting `finalized` to true sends acceptance decision notifications to all paper authors, while setting `assigned` to true launches the review assignment procedural function `doAssigned`. This function uses the `maxReviews` and `minReviews` configuration options to set a review quota for PC members and papers respectively. The function is called in the cell `assignedTrigger`. Note that the options are arranged over multiple columns instead of multiple rows; this is merely to produce shorter references; a table with `name`, `value` columns would also be expressible.

The `Paper` table contains one paper submission per row. Authors fill in the following information: a paper title, a paper abstract, a PDF file and a list of conflicts. Each author can only see rows from his own submissions, but the rows are also visible to PC members that do not have conflicts with the author, except for the author field; this field is only visible to PC members if a) the paper has been accepted or b) if the submission is not anonymous. If `sAnonOptIn` config flag is set to `true`, then the column `anonymous` controls the submission's anonymity; otherwise, the value of that column is ignored and the global config value `sAnon` is used.

Unlike the previous fields, the remaining ones are dynamically evaluated using data from other tables, mostly to summarize information about the paper to the chair and the PC members in a convenient format: the `reviewers` field lists the PC members that have reviewed the paper; the `reviews` field shows the `merit, expertise` pairs of the reviews to summarize the value of the paper; the `tags` field pulls information from the `Tag` table and lists all the tags associated with the paper, to allow PC members and the chair to quickly search among them. the `accepted` field looks for specific tags to see whether the paper has been already accepted or rejected. Note that unlike

other tags, the accept and reject tags are also visible to the paper author; a label-based system would perform declassification here, thus splitting the policy for the tags into two different places; WebSheets' WF assertions allow developers to put the normal policy and the exception for acceptance tags together in the `Tag` table, which we describe later.

The `Preference` table holds the review preference for a particular table. This table should be filled in by PC members before review assignments are made: the assignment script will take these preferences into account when assigning reviews, and delete them as they are used. Preferences are only viewable by the chair (who runs the assignment script) and their authors.

`Review` contains the review for the papers. The reviewer fills in the following information: a merit and an expertise score, a review summary, a field for comments to the author and a field for comments to other PC members. The rows can be either created by the automatic assignment script (which runs with admin privileges), or, if the `rAssigned` config flag is set to `false`, manually added by PC members. The rows are only visible to paper authors after an acceptance decision is made, except for the author field (if the review is anonymous) and the PC comments. On the other hand, they are visible to PC members if a) there are no conflicts and b) either the `rGroupThink` flag is set to `false`, or the PC member has already completed a review for the same paper. Normally, the assignment script creates a row for each review assigned, and sets the author and paper fields. Note that reviews assigned by the chair have the `assigned` field set to `true` and cannot be deleted by the assignee.

The `Tag` table contains tag assignments, one per row. The `Paper` table provides a tags field to allow PC members to view sets of tags directly on the paper. PC members are free to assign tags to papers; however, a small subset of tags (specified using the `tPriv` config flag) are privileged and can only be set by the chair. The tag system is also used to make an acceptance decision: once the chair sets the `accept` or `reject` tag for a paper and sets the `finalized` config flag to `true`, reviews become visible to authors and the system sends a notification by mail.

Finally, the `RReview` table contains the grades for the reviews, entered by other PC members. Only the chair can see the reviews of other PC members.

Besides the 7 tables, WSConf defines one procedural function that runs in the JavaScript sandbox, `doAssignment`. The function performs review assignments according to the preferences supplied by PC members in the `Preference` table, avoiding conflicts and observing a maximum number of

```
// ptitle is the current paper that needs a review
// maxR is the quota of reviews per PC member
{ {member: m, pref: Preference[member==m && paper==ptitle]}
    for m in Committee.member
    when Review[author==m&&paper==ptitle]==[] &&
      len(Review[author==m]) < maxR &&
      m not in Paper[title==ptitle].0.conflicts
}
```

Figure 32: Review Assignment WF formula

assignments per PC member and a minimum number of assignments per paper which are supplied as integer arguments. The script is only 40 lines of code, thanks to the high- level API offered by the sandbox. In particular, the sandbox exposes the function `er`, which combines the `eval` and `redact` operations defined in Section 5.3.1 to allow safe evaluation of any WF formula. For the purpose of assigning reviews, we use `er` to extract the set of PC members that a) are not already reviewing the paper, b) have not already been assigned more than their maximum quota of reviews and c) have no conflicts. The WF formula is shown in Figure 32.

The application logic and permissions required a total of 54 WF formulas of varying degrees of complexity. For example, the formula to restrict the addition of new PC members to the chair is simply `user == "admin"`, while the formula to decide whether the row for a paper is readable is show in Figure 33: the row is readable if the user is the chair, the author of the review, a PC member without conflicts with the paper (but if groupthink protection is enabled, the PC member must have already submitted his review for the paper) or the user is the author of the paper after the conference has been finalized and acceptance decision have been sent out. Note that cells can have additional restrictions: for example, the read permission for the author of the review imposes further restrictions if the review is anonymous.

An important metric to assess the usability of IFC systems is not only how readable the policies are, but also how much declassification is used throughout the system. Declassifying a flow represents an exception to the policy under specific conditions, and it can complicate reasoning about the actual policy enforced, since the policy is defined in one place but relaxed in another[9]. We argue that compared to traditional IFC labels, data assertions

[9]Note that this is still preferable to ad-hoc access control, because the exceptions to

```
user == "admin" or
decl(user == author) or
( user in Committee.member and
  user not in Paper[title==paper].0.conflicts and
  ( !Config.0.rGroupThink or
    Review[paper==paper&&author==user&&summary!=""] != [])) or
(Config.0.finalized && user == Paper[title==paper].0.author)
```

Figure 33: Read Row Formula for HotCRP Reviews

allow more expressivity, which lead to more exceptions covered by the formula itself and less declassification. In particular, WSConf only required 2 explicit declassification calls; both of them are related to the fact that read access to any of the `Review` fields is conditional on the current user being the author of the review, but the author of the review itself is not always visible.

6.2 Covert Channels

Generally speaking, a covert channel is a mechanism that can be used by an attacker to read sensitive data without being subjected to the system's policy. Covert channels are particularly important in DIFC systems because the main feature of DIFC over traditional access control is the ability to delay enforcement of the policy, allowing a principal to manipulate data that he might not readily have access to unless it is subsequently declassified. This creates a window of opportunity for an attacker to first access sensitive data and then "launder" said data, shedding the security metadata that DIFC systems typically tie to the data (e.g. a label in traditional lattice-based systems, or a dependency in WebSheets) for deferred enforcement of the policy. There are two conditions that make the threat more realistic:

a) the DIFC system cannot express a policy that require the principal to have any clearance to read sensitive data; instead, the system merely updates the security metadata to account for the flow of information [87, 18, 145]. In this model, principals can always read sensitive data, but cannot leak it through direct, explicit flows. Some systems

the main policy are clearly marked by the usage of declassification primitives and can be inspected carefully for correctness, instead of being implicitly littered around the code where every flow can express a different policy.

support two different secrecy levels for sensitive data — one that allows data to be read but not leaked (deferred enforcement), and one that does not allow data to be read at all (eager enforcement / access control) [34, 74, 110]. These systems can at least limit the amount of information that is vulnerable to covert channels. Otherwise, any piece of information, no matter how sensitive, can be involved in a covert channel attack.

b) malicious code is part of the threat model.

Covert channels exist also in systems that do not have a) and b), but each condition makes the threat less realistic: if a) is false (i.e. the systems allows at least opting-in to eager enforcement), high-confidentiality data can be put out of reach; if b) is false, then only unintentional side- channels from benign code remain. Because WebSheets has both properties, mitigating covert channels is essential. Although we do not attempt to mitigate timing channels, we describe how we mitigate implicit flows and termination channels.

- *Implicit Flow*: an attacker can read a single bit from a secret value without directly reading it by using the secret value in a control flow decision. For example, `secret == true` can be expressed as `if secret then true else false`, which avoids referring to `secret` in either branch. The traditional fix in procedural languages is to push the dependencies of the condition to the program counter, and have those taint all variable writes; in a pure functional language where everything is an expression, it is sufficient to attach the dependencies of the condition to whichever branch is returned.

  However, although the WF language itself does not prescribe side-effects, its implementation in WebSheets has access to a handful of functions that can perform effectful computations, such as sending a mail or invoking a sandboxed script that writes to another cell. For example, `secret == true` can be written as `if secret == true then mail("attacker", "secret is true") else 0`. Attaching the dependencies of `secret` to the return value is not enough, because the data already left the system through the procedural logic once the result is returned.

To prevent this, the dependencies of the condition must be pushed down to its branches before the branches are evaluated, so that functions can check for control flow dependencies. In this example, the mail would be sent only if `secret` is readable by the current user. Note that short-circuiting logical operators must also implement a similar protection, by pushing dependencies of the left node to its sibling if its evaluation is required.

- *Termination*: the attacker can also cause an error that terminates the evaluation of the current formula, thus reading one bit off a termination channel. We distinguish two types of errors: exceptions, which are high-level faults that we catch and display as evaluation errors, and crashes, low-level faults affecting the underlying runtime that cannot be mitigated by definition. The former can be exploited using an expression such as `secret or 1+"hello"`. If `secret` is `true`, the result is redacted, otherwise an error is displayed. To patch this channel, we attach dependencies to exceptions using the same logic for implicit flows as they unwind the stack; then, `censor` can use the dependencies to redact these exceptions, making them indistinguishable from ordinary values that have been redacted.

  Crashes, on the other hand, represent all kinds of software faults that cannot be caught by the JavaScript runtime. For example, an attacker could cause the underlying runtime to allocate too much memory and eventually crash, thus making `secret or crash()` leak one bit by checking if the server ever responds to the operation. Although we could simply claim that the OS, the Node runtime and the WebSheet implementation are part of the TCB and therefore "crash-proof" for the purposes of our covert channel analysis, we acknowledge that it would not be a realistic assumption. Instead, we have devised a simple restriction that mitigates the threat: the user who is creating the new formula must have read access to all its dependencies. In all the scenarios we presented, the table owner fullfills the role of the application developer, which by definition has full access to all the application's data. A more principled alternative, described in Section 5.2, is to avoid deferred enforcement altogether and implement permission checks using eager enforcement.

6.3 Security of WebSheets

In the introduction we identified five major drawbacks of traditional web application development, and we introduced WebSheets as a new paradigm that eliminates or mitigates these problems. In this section, we investigate to what degree our promise has been fulfilled; we tackle each drawback one by one and discuss how they are addressed by WebSheets.

- *Hard to cover all cases*: We claimed that specifying a policy requires placing the same check many times throughout the codebase, and that it is easy to forget a single check and cause a vulnerability. *Because WebSheets policies are specified in permission tables and enforced throughout the repository using Information-Flow Control, WebSheets successfully save developers from the burden of remembering to add checks to every single flow of sensitive data.*

- *Lack of separation of concerns*: we claimed that, because security checks are intermingled with application logic, traditional web application development prevented policy developers and application logic developers from operating on different concerns. *In WebSheets, policy developers maintain permission tables, and application logic developers maintain value tables and scripts, completely separating their concerns. The runtime uses Information-Flow Control to weave the two together into a secure web application.*

- *Difficult to maintain*: we claimed that, when modifying a policy in a traditional web application, developers have to hunt down and update all the checks that informally specified the policy. *Because WebSheets policies are specified in permission tables and enforced throughout the repository, developers only need to update permission tables and can ignore the information contained in the data tables. Moreover, developers immediately know where to look for the formula which specifies the policy for a specific column or cell.*

- *No least-privilege principle*: we claimed that in traditional web applications the application logic has full access to the data, and must place checks to prevent unauthorized access; if a check is missing, the damage can be severe. *In WebSheets, because Information-Flow Control is a form of Mandatory Access Control, the evaluation of WF formulas is subject to a policy that cannot be circumvented. Every formula*

is explicitly evaluated under the authority of a specific user, and the policies contained in the permission tables specify whether each piece of information can be displayed to said user.

- *No formal verification or analysis*: we originally claimed that it is impossible to provide the developer with tools to formally verify the policy or to check for interesting security properties. Admittedly, this thesis does not offer improvements on this point. However, a system where policies are organized and separated from the application logic is a much better starting point for future work that wants to reason about its security properties.

Moreover, we discuss how WebSheet mitigate concrete threats. To focus on relevant, realistic threats, we review the attacks discussed in the latest OWASP Top 10.

A1 - Injection This broad category includes all vulnerabilities where untrusted data is interpreted as code by the web application, such as SQL Injection, XSS and OS Command Injection. *Because WebSheets enforce fine- grained policies using Mandatory Access Control, injection attacks are neutralized. Malicious WF formulas are part of our threat model, and their evaluation cannot circumvent policy checks.*

A2 - Broken Authentication and Session Management Includes all vulnerabilities related to login, password management, sessions, etc. Because developers often write their own implementation, these are commonly flawed, containing flaws such as cleartext passwords, login CSRF, etc. *WebSheets centrally implement authentication and session management, only exposing the current logged-in user to the application logic. Although this doesn't guarantee that WebSheet's implementation is secure, it saves developers from having to re-implement the authentication logic in their tables, thus sharing a single implementation that can be subjected to more scrutiny.*

A3 - Cross Site Scripting We discussed XSS attacks in Section 2.1. Compared to other injection attacks, which are handled using MAC in the runtime, a principled XSS defense also needs to ensure that cells cannot contain JavaScript code, which could then misuse the user's credentials to leak informations to other users against the MAC policy. *Currently,*

we simply prevent XSS attacks by preventing HTML special characters to appear unescaped in cells. If the UI needed to be extended to support untrusted HTML in cells, the provenance information that is maintained by the runtime could be used together with one of the hybrid XSS defenses surveyed in Section 2.7.

A4 - Insecure Direct Object References When the application uses an input parameter to access an object (e.g. referencing a database row by ID using a parameter from the URL querystring), it must check if the current user is authorized to access such object. *Because WebSheets enforce the policies defined in permission tables using Mandatory Access Control, any operation on the repository is subject to the appropriate policy, regardless of whether the application developer expected this particular access to be controlled by untrusted input or not.*

A5 - Security Misconfiguration Developers must configure each level of their application stack correctly before deployment, because default configuration options might not be appropriate for the current deployment or insecure. The most obvious weakness of this kind is forgetting to change a default password. *Because multiple mutually untrusted applications can share a single repository, WebSheets applications can share the same stack, which must be configured only once, except for higher level application specific configuration options. For example, WebSheets users who wish to deploy* `WSConf` *only need to edit the* `Config` *table to fit their needs, but do not need to setup a web server, a DBMS, etc.*

A6 - Sensitive Data Exposure This weakness relates to the ability of attackers to leak private data from the site, such as credit card numbers. The Top 10 authors suggests that encryption should be used on high-confidentiality data to mitigate these leaks. *WebSheets use Information-Flow Control and fine-grained policies to prevent accidental leaks. For example, credit card numbers can easily be prevented from leaking anywhere using a read permission formula such as* `user == owner || user == "bank"`*, which clearly states that only the user who entered the number or the paymeny processor can read the number. Although encryption could be employed to harden the WebSheet runtime against implementation bugs, we leave that for future work since we already have a robust mechanism in place to prevent data leaks.*

A7 - Missing Function Level Access Control When a URL or a single input parameter is used to decide what operation should be executed on the backend (e.g. `user.php?action=delete`), the application should check whether the user is authorized to perform the operation. *Again, because we are enforcing policies through Mandatory Access Control, as long as formulas in permission tables are correct, there is nothing for developers to forget which can cause a user to read or write anything other than what is specified in permission tables.*

A8 - Cross-Site Request Forgery CSRF attacks were described in Section 3. *Because the HTML interface and HTTP-level operations are handled by the underlying runtime, WebSheets do not allow developers to introduce CSRF vulnerabilities. However, the runtime is not guaranteed to be free of CSRF attacks, as it is a traditional web application from the point of view of an attacker. For this reason, we can deploy a CSRF defense like jCSRF to secure all applications deployed in the WebSheet repository.*

A9 - Using Components with Known Vulnerabilities When a web application relies on third party libraries, it can become vulnerable to their vulnerabilities as well. Therefore, developers must ensure that their dependencies are up to date. *The WebSheet runtime is based on Node, which can use the `nsp` tool from the Node Security Project to ensure its components are up to date and free of vulnerabilities.*

A10 - Unvalidated Redirects and Forwards Includes both open redirects (e.g. specify a whole URL as a parameter and redirect the user's browser using a 403 response) and internal redirects within the application's web pages. The former can be used to redirect the victim to an attacker- controlled site, while the latter can be used to direct the user to a page within the same site, to potentially bypass an access control check that is present only if the page is accessed directly. *WebSheets's UI does not have a redirect feature at the moment, so this attack is not applicable. However, if a redirect feature exists, we note that it could not bypass any access check because of the usage of Mandatory Access Control.*

7 Related Work

WebSheets is a new Web application paradigm based on *spreadsheets* which leverages *information-flow control* to support *principled access control* for web applications. Related work comes from all three areas.

7.1 Spreadsheets

VisiCalc [14] introduced spreadsheets in their modern form. Since then, spreadsheets have enjoyed tremendous commercial popularity. Their main contribution to modern computing and the reason for their success is that they enabled non-programmers (accountants, administrators, secretaries, etc) to enter, process and visualize data in a tabular format [91, 119], effectively allowing them to create full-fledged data-driven applications. The major testament to their ease-of-use is that users rarely refer to their spreadsheets as applications.

More generally, the success of spreadsheets can be traced to a handful of key features:

- *User Interface*: while textual programs are developed by editing text files which operate on abstract data structures, spreadsheets programs enable users to organize their data visually into concrete rows, columns and tables.
- *Concrete Programming*: because spreadsheet formulas can only be used to calculate a single cell value, the non-programmer is not bogged down by abstractions: the relationships that make up formulas are always about cells instead of columns, rows and tables.
- *Instant feedback*: while textual programs must be restarted (and possibly recompiled) after their code is changed, spreadsheets have update semantics that only recalculate cells as needed, returning an updated state almost instantly.

As far as commercial spreadsheets are concerned, the spreadsheet paradigm has remained the same since the 1970s, and researchers have eloquently argued their weaknesses[17, 40]:

- *Code Duplication*: the lack of abstraction mentioned above is mitigated through generous use of copy and paste. For example, to sum the

contents of two columns of size n in a third column, because of the lack of a `map` primitive, the formula for `C1 =A1+B1` is copied n times all the way to `Cn`. When the formula is updated (e.g due to a bug) the user must find and update all copies manually.

- *Brittle References*: There are two sources of brittleness. Firstly, most spreadsheet products handle the aforementioned copy and paste by interpreting A1 and B1 as relative references, which become `Ai` and `Bi` respectively when copied into the i-th row. Although convenient, the semantics are confusing to users and are a source of bugs. Secondly, because tables can share worksheets, the addition of rows and columns can often move the cell referenced, causing the original formula to refer to a different cell.

- *Formulas and Macros*: The formula language is very simplistic, and many operations can only be achieved using macros (e.g. define a reusable function). However, macros can only be developed by programmers, which defeats the purpose of using a spreadsheet for most users.

WebSheets preserve most of the benefits of the basic spreadsheet model, while mitigating some of its drawbacks: fixed table schemas with column names reduce the brittleness of formulas, while the WF language reduces the need for a general-purpose imperative language for macros.

Although the programming languages community in general has largely ignored spreadsheets [17], the functional programming community has attempted to improve the spreadsheet paradigm. Functional programmers have a closer relationship with spreadsheets, mainly because spreadsheet computation is a form of pure functional computation [119]. The related work in this area falls roughly into three categories: extension and specialization of the paradigm for a specific use case (e.g. web scraping), backwards-compatible or user-friendly improvements that cater to non-programmers (e.g. define functions in excel formulas), and backwards- incompatible improvements that cater to programmers (e.g. replace Excel's language with a full-fledged functional language).

Domain-Specific Spreadsheets: This research area is motivated by the intuition that spreadsheets are an incredibly productive tool, but cannot be used in specific domains because they lack necessary features. Vegemite [76] and Reference [72] focus on importing and interacting with web data within

the spreadsheet paradigm. The former runs within the browser to closely interact with web pages and offer macro recording capabilities, to avoid repetitive tasks and define web scraping macros, while the latter is an Excel add-on to import data from a multitude of web services. A1 [48] is a spreadsheet tool to simplify system administration. Unlike traditional spreadsheets, cells can contain Java Objects and other cells can invoke these objects to display the result of a computation. The paper also introduces the concept of events, making the spreadsheet reactive in the face of updates, timer intervals, etc. XCelLog [121] introduces *Deductive Spreadsheets*, which are spreadsheet augmented with a DataLog engine for policy development. With its instant updates and tabular representation, the authors argue that spreadsheets are especially suited for explorative policy development.

Spreadsheets for Non-Programmers: This research area focuses on improving traditional spreadsheets while maintaining their ease-of-use. Ideally, the changes should be backwards-compatible and introduced gradually to the user. Reference [65] introduces reusable functions written in tabular form into Excel: currently, users must abandon the spreadsheet paradigm and resort to Visual Basic macros to define even the simplest of functions. Tabular functions use cells for input, output and intermediate computation and allow the user to define functions completely within the spreadsheet paradigm.

Spreadsheets for Programmers: These works attempt to repackage the spreadsheet paradigm with programmer-friendly features. Reference [138] augments Excel with the possibility of calling externally defined Haskell Functions by communicating with a Haskell interpreter. Reference [27] details the implementation of a Spreadsheet Engine and UI using the Clean Language, a lazy functional language. Haxcel [77] presents a Spreadsheet-like interface for Haskell development. Reference [146] describes Mini-SP, a language for spreadsheets focused on supporting non- trivial control flow and message passing among cells. Reference [20] discusses how functional and object-oriented features can come together in the spreadsheet paradigm.

Overall, none of the papers discussed above are similar to WebSheets, because they focus on improving spreadsheet *usability* (for a particular task, for non-programmers and for programmers respectively), while WebSheets expand the *applicability* of the spreadsheet paradigm to an entire new class of applications, namely web applications.

7.2 Information-Flow Control

WebSheets implement a form of *Language-Based*, *Runtime*, *Fine-Grained*, *Decentralized* Information-Flow Control (DIFC) [89] to enforce confidentiality.

In centralized IFC, a single administrator assigns security labels to resources and users and a trusted runtime ensures that data flows are only permitted if they conform to the "no read up, no write down" policy of the Bell-LaPadula model [29, 8]. In the decentralized alternative, users can (a) create and assign new labels to data, (b) declassify data that they own themselves and, in some later implementations, (c) grant declassification capabilities for their data to other users. Although centralized IFC has been successfully applied in the context of military systems [12], recent research work has favored the decentralized approach because of its increased flexiblity.

OS-Level IFC [34, 147, 74] provides broader applicability and stronger isolation guarantees than language-based IFC, especially in the face of malicious code that tries to circumvent enforcement because of the smaller TCB. The main drawback of OS-Level IFC is that labels are typically assigned at the granularity of OS processes or threads, which is fairly coarse-grained for most purposes [110]: in traditional applications OS processes and threads handle data from multiple incompatible sources, and the effect of coarse-grained labelling is that they quickly become tainted with incompatible labels (e.g. an OS process reads sensitive data both for user A and user B, and now cannot communicate with either user), and the application must be rearchitected to work around this limitation (e.g. fork one OS process per user).

Language-Based IFC supports finer-grained assignment of labels, which permits applications to handle different kinds of sensitive data within the same component, as long as the application logic does not *actually* combine them in an unsafe way. In other words, coarse-grained labelling greatly overapproximates the amount of dangerous flows: if a process reads sensitive data, the enforcement mechanism assumes that the process is actively trying to leak the data into files, over the network, to other processes, etc, at any chance it gets; on the other hand, with fine-grained labelling the enforcement mechanism assumes an ongoing leak in a complete yet reasonably sound subset of flows.

While early work focused on enforcing secure data flows using static analysis [30, 89, 87, 19, 112], modern dynamic languages that are popular for

web application development, such as JavaScript, Ruby or Python, are not amenable to precise static flow-to analysis (although simplified subsets can be, e.g. JavaScript$_{\text{SAFE}}$ [46]). Therefore, fine-grained propagation of labels in language-based IFC is done at runtime, i.e. using dynamic taint-tracking [2, 145, 137]. Unfortunately, dynamic taint-tracking has its own shortcomings, such as runtime overhead and the inability to easily deal with covert channels such as implicit flows and termination channels.

Instead of decentralizing the management of labels, WebSheets decentralize their *interpretation*: in traditional lattice-based systems, a set of labels objectively specifies the policy, that is, the effective set of users that can eventually have access to the information; users add and remove their own labels to modify the effective policy. This approach results in scattered management of labels throughout the application's components; Reference [33] describes the issues in defining a decentralized policy in a lattice-based system and leverages an existing lattice-based system by providing a higher-level policy language to define centralized policies that translate to lower-level label assignments. Conversely, in WebSheets the set of labels associated with a piece of data merely specifies its *provenance*; users do not attach or remove labels, bur rather write assertions in permission tables to modify the set of end users that are allowed to view a value with a specific provenance.

In this sense, WebSheets are more similar to Resin [145], which also allows developers to write data-flow assertions in the same language as the application logic. Unlike WebSheets, however, Resin does not implicitly assign a policy to each piece of data; it requires manual configuration and assignment of *policy objects* and *filters*. Policy objects are attached to data; Resin modifies the runtime to propagate policy objects along with data; when data with a policy objects passes through a filter (e.g. HTML output), the filter and the policy object work together to perform a policy check. As a reference implementation, Resin provides modified PHP and Python interpreters; SAFEWEB [56] exploits the dynamic nature of Ruby to add transparent propagation of labels to the unmodified interpreter using a library loaded at runtime. Hails [42] relies on Safe Haskell, a minimally restrictive subset of Haskell, to implement IFC at the granularity of threads using a library. Jeeves [144] presents a policy language and an enforcement library for Scala, leveraging its lazy evaluation semantics to automatically propagate unresolved policy checks to outputs, which are then resolved as needed.

Unlike WebSheets and Resin, Jeeves's policy language supports explicitly specifying a low-confidentiality view for sensitive data to support use cases

where unauthorized users can still benefit from controlled disclosure (e.g. current City instead of the exact GPS coordinates). This feature can be considered as a form of limited declassification, which can conveniently replace full- fledged declassification in some scenarios, thus simplifying reasoning about the policy enforced by the system. Other language-based solutions use traditional DIFC labeling: despite operating in the JVM, Aeolus [18] enforces DIFC at the thread granularity. Similarly, Reference [98] is language-based but implements coarse-grained IFC by leveraging Erlang's lightweight threads: when two threads communicate through message-passing, the runtime verifies that their labels are compatible. Note that these works do not address the problem of storing labels in persistent storage: IFDB [116] discusses how to interface a DIFC lattice-based system like Aeolus with a special DBMS that stores labels at the tuple-level, propagating them back to the client accordingly. There are other DBMS that support IFC; however, they are based on the centralized MLS model (in particular, on the model formalized by the SeaView project [79]). Several commercial DBMSs (e.g. Oracle, DB2, PostgreSQL) support labels for MLS.

One drawback of IFC over traditional access control is that malicious code can exploit covert channels to extract sensitive information: to support declassification at a later point, IFC systems must allow malicious code to read a piece of data in violation of the policy specified by its label. Although IFC prevents explicit leaks, this creates the opportunity for the attacker to leak the piece of data through a covert channel (e.g. a termination channel). Several works minimize the attack surface by leveraging the concept of clearance [34, 147, 74, 110], an eagerly enforced upper bound on the secrecy of the execution context, effectively blending in eager enforcement of a more relaxed policy and deferred enforcement of a more flexible, yet stricter IFC policy that uses declassification to account for exceptions. Other works attempt to control these covert channels; Reference [123] mitigates timing and termination channels in multithreaded systems by making the observation of their temporal behavior a sensitive operation that forces the observer to inherit the secrecy label of the observed thread. Other works simply apply IFC to a threat model where the main concern is developer error, not malicious code.

Figure 34 summarizes related IFC works along the design choices we described. The DLM and DC label models have already been described in Section 5.2. They are both models where the readers for a label are named explicitly. On the other hand, the tag model abstracts away the

Name	Enforcement	Granularity	Labels	Clearance	Implementation
Asbestors	OS	Process	$\star$	Yes	New OS
Histar	OS	Thread	$\star$	Yes	New OS
Flume	OS	Process	Tags	Yes	Linux LSM
Laminar	Both	Security Region	Tags	Yes	Linux LSM + JVM
Jif	Language	Variable	DLM	No	Compiler + JVM
Aeolus	Language	Thread	Tags	No	JVM
Reference [98]	Language	Green Thread	Tags	Yes	Message Passing
Resin	Language	Variable	PHP Code	No	PHP Taint-Tracking
SAFEWEB	Language	Variable	Tags	Yes	Ruby Taint-Tracking
Hails	Language	Thread	DC	Yes	LIO Library
WebSheets	Language	Cell	WF Formula	No	Cell Provenance Tracking

Figure 34: Organization of IFC solutions

concept of readers, making the label itself simply a unique identifier with no implicit meaning: for each tag t, its creator must explicitly grant other users permission to declassify and remove the tag (sometimes noted as t^-) and, if clearance is supported, permission to add the label t (also noted as t^+) and allow the flow of information with tag t. The $\star$ model has similar capabilities, despite having abstractions and semantics more similar to the DLM model. Reference [86] compares the expressive power of tags, $\star$, DLM and DC label models.

7.3 Principled Security in Web Applications

IFC is a powerful, yet heavyweight approach to principled security. It requires extensive modifications to the type system and the compiler (static language-based IFC), the underlying runtime (runtime language-based IFC) or the OS (OS-Based IFC). Web applications currently implement their security policies using scattered checks, so even traditional access control, if applied judiciously with the intent of separating application logic and security policies, can represent an improvement over the status quo. In particular, centralizing the specification of the security policy and applying the least-privilege principle to web application components can reduce the number of vulnerabilities, simplifying reasoning about the policy and limit the damage if a vulnerability is exploited.

Reference [128] presents a Trust Management system that supports decentralized access control policies and protects DBMS data directly within the DBMS by extending SQL with attribute-based GRANT statements. These statements can provide more expressive access control policies compared to

traditional SQL GRANT statements, because instead of applying to an explicit set of users, they can apply to a subset of users based on their attributes (e.g. whether they have been endorsed by a third-party to assume a specific role). Also, unlike traditional GRANT statements, the privileges are extended to new users or revoked from existing users as their user attributes are updated. Thus, the system implements a form of Attribute-Based Access Control (ABAC) that is limited to user attributes; WebSheets implement a more expressive version of ABAC that can predicate on the current user, the current cell, or any other cell in the repository. For example, our HotCRP implementation from Section 6.1.1 mandated that users can read the review for a paper under three conditions: (a) they are members of the PC group, and (b) either they are reviewers for the paper and they have already submitted a review themselves or they are not reviewers and they don't have a conflict with the author. RBAC can only express condition a), while Reference [128]'s form of ABAC can only enforce b) and c) using a hack, by polluting the users' attribute sets with object-related attributes.

GuardRails [15] proposes data-centric policy enforcement for Ruby On Rails (RoR), a popular MVC Web Application Framework: it provides a source- to-source translator to parse policy annotations and add policy checks throughout the web application code whenever data is fetched from the database. Since in RoR the database is always accessed through an ORM, GuardRails replaces the ORM objects at runtime with proxies that enforce the appropriate policy. However, applying these policies at the framework level does not protect against malicious code that has been loaded by a vulnerable or an untrusted application, because the runtime still runs with full privileges. Therefore, researchers have also developed systems that partition the web application into components and restrict the privilege of each component.

Least privilege can be seen as a static version of coarse-grained IFC: in IFC, components start with full privileges and drop privileges to consume labelled data, while in least-privilege isolation components start with a fixed set of limited privileges. CLAMP [100] is a modification to the traditional LAMP stack that enforces strong isolation between users, to limit the damage from not only missing checks, but also malicious code. Least- privilege is enforced for the whole web server using virtualization: each user is served by a dedicated server that is created on demand, and a SQL proxy restricts the amount of information available to each user.

A more lightweight approach compared to virtualization is to leverage

an object-capability language [85], so that the runtime does not need to be modified to enforce least-privilege policies. In an object-capability language, the language semantics prevent untrusted code from referencing anything but a limited set of objects that have been explicitly provided, which can encapsulate access to privileged resources. Capsules [73] uses the Joe-E language [83], a capability-safe subset of Java, to restrict the privilege of J2EE Servlets. Given the recent advancements on the client-side in making JavaScript capability-safe [81, 84, 1, 135], a similar approach could also be applied to an existing server-side JavaScript framework.

Another lightweight approach is to run the components in different processes and use existing OS mechanisms for process isolation. For example, Passe [11] partitions web applications written in the Django framework into components isolated using AppArmor. Passe also features a training mechanism that uses IFC to automatically infer a minimal set of privileges for each component, enforcing them without IFC in enforcement mode. Note that while CLAMP restricts access based on the current user, Passe partitions the web application into modules and uses the current user *and* the current module to make access control decisions, thus performing *data separation* [36] along with traditional user-based access control.

8 Conclusions

While the awareness about web application vulnerabilities has increased in the past decade, these continue to plague applications. Many of these vulnerabilites share the same root cause: the security policy of the web application is implemented using ad-hoc checks scattered throughout their codebase; this is paired with a *reactive* approach to security: vulnerabilities are adressed and fixed after they are found (and possibly exploited with severe consequences) and reported.

In this dissertation, we have shown two approaches to securing web applications: in Part I, we focused on popular vulnerabilities that can be mitigated without developer involvement, and we described two defenses against XSS and CSRF attacks. In Part II, we devised a new web application development paradigm where security policies are clearly separated from the application logic, and where users retain control of their data.

We have shown that both approaches are valid and provide a tangible benefit in terms of security. The first approach is more useful for legacy

applications or to provide security-unaware developers with some degree of protection. On the other hand, the second approach provides broader guarantees and leads to cleaner, more streamlined web applications whose policies can be more easily specified, maintained and audited.

References

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. Jsand: complete client-side sandboxing of third-party javascript without browser modifications. In *ACSAC 2012*.

[2] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. *ACM Sigplan Notices*, 44(8), 2009.

[3] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE S&P 2008*.

[4] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in Web applications. In *IEEE S&P*, 2008.

[5] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. Venkatakrishnan. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24. ACM, 2007.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust Defenses for Cross-Site Request Forgery. In *CCS*, 2008.

[7] D. Bates, A. Barth, and C. Jackson. Regular Expressions Considered Harmful in Client-Side XSS Filters. In *WWW*, 2010.

[8] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, DTIC Document, 1973.

[9] K. J. Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[10] P. Bisht and V. Venkatakrishnan. XSS-Guard: Precise Dynamic Detection of Cross-Site Scripting Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, 2008.

[11] A. Blankstein and M. J. Freedman. Automating isolation and least privilege in web services. In *IEEE S&P 2014*.

[12] S. L. Brand. Dod 5200.28-std department of defense trusted computer system evaluation criteria (orange book). *National Computer Security Center*, 1985.

[13] K. Brandeisky. What to do if your social security number was leaked. http://time.com/money/3620100/sylvester-stallone-social-security-number/.

[14] D. Bricklin. VisiCalc: Information from its creators. http://www.bricklin.com/visicalc.htm, 2014.

[15] J. Burket, P. Mutchler, M. Weaver, M. Zaveri, and D. Evans. Guardrails: a data-centric web application security framework. In *USENIX WebApps*, 2011.

[16] Y. Cao, V. Yegneswaran, P. A. Porras, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS 2012*.

[17] R. J. Casimir. Real programmers don't use spreadsheets. *ACM SIGPLAN '92*.

[18] W. Cheng, D. R. Ports, D. A. Schultz, V. Popic, A. Blankstein, J. A. Cowling, D. Curtis, L. Shrira, and B. Liskov. Abstractions for usable information flow control in aeolus. In *USENIX ATC*, 2012.

[19] S. Chong, K. Vikram, A. C. Myers, et al. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security 2007*.

[20] C. Clack and L. Braine. Object-oriented functional spreadsheets. In *Glasgow Workshop on Functional Programming*, 1997.

[21] CVE Editorial Board. CVE-2007-3574. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-3574, 2007.

[22] CVE Editorial Board. CVE-2009-2073. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2073, 2009.

[23] CVE Editorial Board. CVE-2009-4076. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4076, 2009.

[24] CVE Editorial Board. CVE-2009-4906. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-4906, 2009.

[25] CVE Editorial Board. CVE-2010-1482. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-1482, 2010.

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

[27] W. A. De Hoon, L. M. Rutten, and M. C. D. van Eekelen. Implementing a functional spreadsheet in clean. *Journal of Functional Programming*, 1995.

[28] P. De Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent client-side mitigation of malicious cross-domain requests. In *ESSOS*, 2010.

[29] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5), 1976.

[30] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7), 1977.

[31] Department of Homeland Security. Common Vulnerabilities and Exposures. http://cve.mitre.org/.

[32] Django Software Foundation. Django URL Dispatcher. http://docs.djangoproject.com/en/1.1/topics/http/urls/.

[33] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow.

[34] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *SIGOPS 2005*.

[35] EllisLab Inc. Code Igniter. http://codeigniter.com/, 2002.

[36] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying Privilege Separation to Database Access. In *ACM AsiaCCS 2011*.

[37] K. Fernandez and DP. XSSed. http://xssed.com/.

[38] R. Fielding et al. Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/rfc/rfc2616.txt, 1999.

[39] J. Finkle and D. Volz. Database of 191 million u.s. voters exposed on internet. http://uk.reuters.com/article/us-usa-voters-breach-idUKKBN0UB1E020151229.

[40] C. H. Q. Forster. Programming through spreadsheets and tabular abstractions. *J. UCS*, 2007.

[41] J. Fraser. Backwards compatible window.postMessage(). http://www.onlineaspect.com/2010/01/15/backwards-compatible-postmessage/, 2010.

[42] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazieres, J. C. Mitchell, and A. Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, 2012.

[43] Giorgio Maone. NoScript. http://noscript.net/.

[44] D. Goodin. How a website flaw turned 22,000 visitors into a botnet of ddos zombies. http://arstechnica.com/security/2014/04/how-a-website-flaw-turned-22000-visitors-into-a-botnet-of-ddos-zombies/.

[45] T. P. Group. Php: Prepared statements and stored procedures. http://php.net/manual/it/pdo.prepared-statements.php.

[46] S. Guarnieri and V. B. Livshits. GATEKEEPER: Mostly static enforcement of security and reliability policies for javascript code. In *USENIX Security 2009*.

[47] F. Guisset. JavaScript-DOM Prototypes in Mozilla. https://developer.mozilla.org/en/JavaScript-DOM$_{\text{P}}$rototypes$_{\text{i}}$n$_{\text{M}}$ozilla, 2002.

[48] E. M. Haber, E. Kandogan, A. Cypher, P. P. Maglio, and R. Barrett. A1: Spreadsheet-based scripting for developing web tools. In *LISA*, 2005.

[49] W. G. Halfond and A. Orso. Amnesia: analysis and monitoring for neutralizing sql-injection attacks. In *ACM ASE 2005*.

[50] R. Hansen. XSS Cheat Sheet. http://ha.ckers.org/xss.html.

[51] D. H. Hansson. Ruby on Rails. http://rubyonrails.org, 2011.

[52] N. Hardy. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, Oct. 1988.

[53] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *ACM CCS 2012*.

[54] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *ACM CCS 2013*.

[55] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *USENIX Security 2011*.

[56] P. Hosek, M. Migliavacca, I. Papagiannis, D. M. Eyers, D. Evans, B. Shand, J. Bacon, and P. Pietzuch. Safeweb: A middleware for securing ruby-based web applications. In *Proceedings of the 12th International Middleware Conference*, 2011.

[57] A. Inc. Learn more about the cyber attack against anthem. https://www.anthemfacts.com/faq.

[58] O. Ismail, M. Etoh, Y. Kadobayashi, and S. Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Application (AINA04)*, 2004.

[59] K. Jayaraman, G. Lewandowski, P. Talaga, and S. Chapin. Enforcing Request Integrity in Web Applications. *Data and Applications Security and Privacy*, 2010.

[60] Jeremias Reith. NoXSS. https://addons.mozilla.org/en-US/firefox/addon/noxss/.

[61] Jesse Ruderman. Heuristics to block reflected XSS (like in IE8). https://bugzilla.mozilla.org/show_bug.cgi?id=528661, 2009.

[62] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*. ACM, 2007.

[63] M. Johns, B. Engelmann, and J. Posegga. XSSDS: Server-side Detection of Cross-site Scripting Attacks. In *ACSAC*, 2008.

[64] M. Johns and J. Winter. RequestRodeo : Client Side Protection against Session Riding. In *OWASP Europe*, 2006.

[65] S. P. Jones, A. Blackwell, and M. Burnett. A user-centred approach to functions in excel. In *ACM SIGPLAN*, 2003.

[66] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm*, 2007.

[67] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE S&P 2006*.

[68] T. Kim, R. Chandra, and N. Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *OSDI 2012*.

[69] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing*, page 337. ACM, 2006.

[70] E. Kohler. Hot Crap! In *WOWCS 2008*.

[71] E. Kohler. Hotcrp conference management software. http://read.seas.harvard.edu/~kohler/hotcrp/index.html, 2016.

[72] W. Kongdenfha, B. Benatallah, J. Vayssière, R. Saint-Paul, and F. Casati. Rapid development of spreadsheet-based web mashups. In *WWW '09*.

[73] A. Krishnamurthy, A. Mettler, and D. Wagner. Fine-grained privilege separation for web applications. In *ACM WWW 2010*.

[74] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *SIGOPS 2007*.

[75] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *ACM CCS 2013*.

[76] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *International conference on Intelligent user interfaces*. ACM, 2009.

[77] B. Lisper and J. Malmström. Haxcel: A spreadsheet interface to haskell. In *Workshop on the Implementation of Functional Languages*, 2002.

[78] M. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE S&P*, 2009.

[79] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The seaview security model. *IEEE Transactions on Software Engineering*, 1990.

[80] W. Maes, T. Heyman, L. Desmet, and W. Joosen. Browser protection against cross-site request forgery. In *SecuCode*, 2009.

[81] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *IEEE S&P 2010*.

[82] M.C. Straver. The PaleMoon project homepage. https://www.palemoon.org/, 2016.

[83] A. Mettler, D. Wagner, and T. Close. Joe-e: A security-oriented subset of java. In *NDSS 2010*.

[84] M. S. Miller et al. Secure ecmascript 5. http://code.google.com/p/es-lab/wiki/SecureEcmaScript, 2011.

[85] M. S. Miller and J. S. Shapiro. *Robust composition: towards a unified approach to access control and concurrency control*. 2006.

[86] B. Montagu, B. C. Pierce, and R. Pollack. A theory of information-flow labels. In *CSF 2013*.

[87] A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL 1999*.

[88] A. C. Myers. *Mostly-static decentralized information flow control*. PhD thesis, MIT, 1999.

[89] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP 1997*.

[90] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *NDSS*, 2009.

[91] B. A. Nardi and J. R. Miller. *The spreadsheet interface: A basis for end user programming*. Hewlett-Packard Laboratories, 1990.

[92] E. V. Nava and D. Lindsay. Our favorite xss filters/ids and how to attack them. Black Hat USA 2009.

[93] E. V. Nava and D. Lindsay. Universal xss via ie8's xss filters. Black Hat Europe 2010.

[94] Nick Nikiforakis. Bypassing Chrome's XSS Filter. http://blog.securitee.org/?p=37, 2011.

[95] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. Sessionshield: Lightweight protection against session hijacking. In *ESSOS 2011*.

[96] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *CCS*, 2008.

[97] J. Pagliery. Opm hack's unprecedented haul: 1.1 million fingerprints. http://money.cnn.com/2015/07/10/technology/opm-hack-fingerprints/.

[98] I. Papagiannis, M. Migliavacca, D. M. Eyers, B. Shand, J. Bacon, and P. Pietzuch. Enforcing user privacy in web applications using erlang. *W2SP, Oakland, CA*, 2010.

[99] I. Parameshwaran, E. Budianto, S. Shinde, H. Dang, A. Sadhu, and P. Saxena. Auto-patching dom-based xss at scale. *Foundations of Software Engineering (FSE) 2015*.

[100] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *IEEE S&P*, 2009.

[101] Paul Mutton. Italian Bank's XSS Opportunity Seized by Fraudsters. http://news.netcraft.com/archives/2008/01/08/italian$_{b}$anks$_{x}$ss$_{o}$pportunity$_{s}$eized$_{b}$y$_{f}$raudsters.html.

[102] R. Pelizzi. WebSheets – Source Code. https://github.com/BruceBerry/websheets, 2016.

[103] R. Pelizzi. Websheets docker image. https://hub.docker.com/r/rpelizzi/websheets/, 2016.

[104] R. Pelizzi and R. Sekar. Protection, usability and improvements in reflected xss filters. In *ASIACCS*, 2012.

[105] R. Pelizzi, T. Tran, and A. Saberi. Large-Scale, Automated XSS Detection using Google Dorks. http://www.cs.sunysb.edu/~rpelizzi/gdorktr.pdf, 2011.

[106] Pylons. Pylons Project. http://pylonsproject.org/, 2011.

[107] Riccardo Pelizzi and R. Sekar. XSSFilt: an XSS Filter for Firefox. http://www.seclab.cs.sunysb.edu/seclab/xssfilt/, 2012.

[108] D. Ross. IE 8 XSS Filter Architecture/Implementation. http://blogs.technet.com/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx.

[109] RoundCube.net. RoundCube - Free Webmail for the Masses. http://roundcube.net/, 2010.

[110] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: practical fine-grained decentralized information flow control. In *PLDI 2009*.

[111] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: A study of clickjacking vulnerabilities on popular sites. In *4th Workshop in Web*, volume 2.

[112] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[113] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *SOSP*. ACM, 1974.

[114] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *NDSS 2010*.

[115] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *ACM CCS 2011*.

[116] D. Schultz and B. Liskov. Ifdb: decentralized information flow control for databases. In *EuroSys 2013*.

[117] R. B. Security. Data breach quickview 2015 data breach trends. https://www.riskbasedsecurity.com/2015-data-breach-quickview/.

[118] R. Sekar. An efficient black-box technique for defeating web application attacks. In *NDSS*, 2009.

[119] P. Sestoft. Implementing function spreadsheets. In *ACM Workshop on End-user software engineering*, 2008.

[120] E. Sheridan. OWASP: CSRFGuard Project. https://www.owasp.org/index.php/Category:OWASP_CSRFGuard_Project, 2011.

[121] A. Singh, C. Ramakrishnan, I. Ramakrishnan, S. D. Stoller, and D. S. Warren. Security policy analysis using deductive spreadsheets. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, pages 42–50. ACM, 2007.

[122] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In *WWW*, 2010.

[123] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *ACM ICFP 2012*.

[124] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. Disjunction category labels. In *Information Security Technology for Applications*. 2011.

[125] Stefano Di Paola. DOM XSS Test Cases Wiki Cheatsheet Project. https://code.google.com/p/domxsswiki/, 2013.

[126] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against dom-based cross-site scripting. In *USENIX Security 2014*.

[127] B. Stock, S. Pfistner, B. Kaiser, S. Lekies, and M. Johns. From facepalm to brain bender: Exploring client-side cross-site scripting. In *ACM CCS 2015*.

[128] S. D. Stoller. Trust management and trust negotiation in an extension of SQL. In *TGC 2008*.

[129] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL*, 2006.

[130] S. Technology. Csrf archives - routercheck. http://www.routercheck.com/category/router-vulnerability/csrf/.

[131] The jQuery Project. .live() - jQuery API. http://api.jquery.com/live/, 2011.

[132] The MITRE Corporation. 2011 CWE/SANS Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25/.

[133] The Open Web Application Security Project (OWASP). OWASP Top Ten Project. http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, 2010.

[134] Tim Tomes. DOM-based Cross-Site Scripting, Revisited. http://www.lanmaster53.com/2014/03/dom-based-xss-revisited/, 2014.

[135] T. Tran, R. Pelizzi, and R. Sekar. Jate: Transparent and efficient javascript confinement. In *ACSAC 2015*, 2015.

[136] M. Van Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *NDSS*, 2009.

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

[138] D. Wakeling. Spreadsheet functional programming. *Journal of Functional Programming*, 2007.

[139] P. Waktinks. Cross-Site Request Forgeries (Re: The Dangers of Allowing Users to Post Images). http://www.tux.org/~peterw/csrf.txt, 2001.

[140] M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In *RAID 2014*.

[141] M. Weissbacher, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna. Zigzag: automatically hardening web applications against client-side validation vulnerabilities. In *USENIX Security 2015*.

[142] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *15th USENIX Security Symposium*, pages 121–136, 2006.

[143] E. Z. Yang. CSRFMagic. http://csrf.htmlpurifier.org/, 2008.

[144] J. Yang, K. Yessenov, and A. Solar-Lezama. A language for automatically enforcing privacy policies.

[145] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *ACM SIGOPS*, 2009.

[146] A. G. Yoder and D. L. Cohn. Real spreadsheets for real programmers. In *International Conference on Computer Languages*. IEEE, 1994.

[147] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *OSDI 2006*.

[148] W. Zeller and E. Felten. Cross-site request forgeries: Exploitation and prevention, 2008.

[149] M. Zhou, P. Bisht, and V. Venkatakrishnan. Strengthening XSRF Defenses for Legacy Web Applications Using Whitebox Analysis and Transformation. In *ICISS*, 2011.