The SICILIAN Defense: Signature-based Whitelisting of Web JavaScript

Pratik Soni* Enrico Budianto Prateek Saxena
National University of Singapore, Singapore
{pratikso, enricob, prateeks}@comp.nus.edu.sg

ABSTRACT

Whitelisting has become a common practice to ensure execution of trusted application code. However, its effectiveness in protecting client-side web application code has not yet been established. In this paper, we seek to study the efficacy of signature-based whitelisting approach for preventing script injection attacks. This includes a recently-proposed W3C recommendation called Subresource Integrity (SRI), which is based on raw script-text signatures. Our 3-month long measurement study shows that applying such raw signatures is not practical. We then present SICILIAN[1], a novel multi-layered approach for whitelisting scripts that can tolerate changes in them without sacrificing the security. Our solution comes with a deployment model called *progressive lockdown*, which lets browsers to assist the server in composing the whitelist. Such assistance from the browser minimizes the burden of building the signature based whitelist. Our evaluation on Alexa's top 500 sites and 15 popular PHP applications shows that SICILIAN can be fully applied to 84.7% of the sites and all the PHP applications with updates to the whitelist required roughly once in a month. SICILIAN incurs an average performance overhead of 7.02%.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Web Security, whitelisting, script injection attacks

1. INTRODUCTION

Whitelisting has become a common practice to ensure the execution of trusted applications. Mainstream OSes such as Windows and Linux check the signature of application binaries against the public whitelist before installing them [60]. Browsers enable code signature verification on its extensions to make sure that its code is signed by a trusted party (e.g., Mozilla's AMO [38]). Although whitelisting has been successfully implemented in several application platforms, its effectiveness in protecting web applications has not yet been established.

In this paper, we seek to study the efficacy of signature-based whitelisting for defending web applications against script injection attacks. Script injection attacks are the most prevalent threat on the web. They occur through many attack channels, such as cross-site scripting (XSS) attacks or tampering with the scripts' content included from third-party services (e.g., CDNs [45,48] and popular JavaScript libraries [30, 40]). In a web page's context, a *whitelist* can be defined as a set of scripts that are authorized by the website owner to execute at the client side. This makes signature-based whitelisting a strong defense against script injection attacks.

There have been solutions that propose the idea of signature-based whitelisting to secure web applications, such as BEEP [29], DSI [39], and Noncespaces [20]. Some of these ideas have taken shape in mainstream web browsers such as Chrome and Firefox in the form of a W3C recommendation known as Subresource Integrity (SRI) [54]. SRI allows specifying a cryptographic hash (SHA256) of the text of the script's code as a signature that signifies the integrity of a script. This signature can be designated as the integrity attribute of a `<script>` tag — we call such signatures as *raw signatures*. The viability of a signature-based whitelisting approach relies on the premise that script signatures remain static over time. Therefore, we hypothesize that such whitelisting is applicable to websites that remain *mostly static*. To evaluate how many websites fall in this category, we conduct a longitudinal study on the Alexa's top 500 websites (45,066 web pages) and 15 popular PHP applications, crawling with depth limit of 3. In the crawl, we monitor 33,302 scripts for a 3-month period, starting 31st January 2015. The objective of this study is to understand how scripts on these webpages change and determine the fraction of the websites for which JavaScript whitelisting would be sufficient. Our observation suggests that majority of the scripts on these websites are indeed mostly static and have the following characteristics:

- Observation 1: Of the 33,302 scripts that we have crawled, only 2,313 scripts changed over time. The changes in 586 out of 2,313 scripts preserve the code semantics. These changes are merely syntactic changes like minification, variable renaming and differences in the code comments.
- Observation 2: 1,156 of the 2,313 scripts have changes that may impact the code semantics, but the changes introduce no new code. In fact, they affect only the data used in small portion of functionality in the website. For example, random tokens in the query parameter of advertisement URLs.

*This work was done while the author was visiting NUS.

[1] Sicilian is one of the strongest and most successful defense openings in chess where the player does not play for equality but for the advantage.

CCS'15, October 12–16, 2015, Denver, Colorado, USA.

DOI: http://dx.doi.org/10.1145/2810103.2813710.

– Scripts in Observation 1 and 2 together contribute 1,644 of the 2,313 changing scripts (71.07%) – with 98 scripts belonging to both categories.

- Observation 3: We further categorize the changes in the remaining 669 scripts (28.9%) as code-introducing updates. These are due to addition or deletion of code from the script. Based on how frequently these happen, we observe two sub-classes: 461 scripts that change somewhat infrequently and 208 scripts that change very frequently (Section 5).

Our Proposal. In our study, 97.99% of the scripts remain mostly static, that is, no new code is introduced in these scripts. Thus, we believe whitelisting via signatures can be as effective in preventing script injection as it has been in securing desktop application in binary form. The raw signature mechanism, proposed in SRI, can be applied to 93% of the 33,302 scripts. However, only 69 out of 500 websites have *all* the scripts remaining static. Therefore, we expect SRI to have limited practical adoption on real-world websites as-is, and will raise high number of false positives (Section 5). Therefore, we develop SICILIAN, a multi-layered solution for JavaScript whitelisting, with raw signature as the first layer of defense and a more relaxed signature as the second layer of defense. SICILIAN interposes on all functions evaluated by the JavaScript parser module in the browser and allows only scripts in the website's whitelist to be executed. Our relaxed signature mechanism is designed to be secure, that is, without allowing the attacker any significant advantage in constructing illegitimate scripts. We call our signature mechanism *structural signatures* as they are based on the abstract syntax tree representation (structure) of the script code. Structural signatures are efficiently computable and robust against syntactic changes in the scripts (Observation 1). Further, our relaxed signatures are designed to handle certain changes which are beyond syntactic changes — as long as they do not introduce any new code (Observation 2). This extends the applicability of our idea to another 1,156 scripts of the 2,313 scripts which change. Finally, of the 2,313 scripts, 461 scripts (20%) change rarely as described in Observation 3. To handle these, we propose a mechanism based on browser-server collaboration to efficiently update the whitelist with the latest signatures, which we detail in Section 4.4. We consider the remaining 208 scripts of 33,302 (only 0.62%) in Observation 3 as beyond our scope due to their high update frequency.

Progressive Lockdown Deployment. Script injection attacks have been well-recognized in the past decade. Any solution to block them faces a deployment challenge, that is, the difficulty in using the solution without raising false positives and without extensive manual work. To smooth the transition, we propose a new incremental approach called *progressive lockdown*. Our approach consists of three phases: INIT, CRAWL, and LOCKDOWN (Section 4.3). First, our signature protection mechanism is only meant for mostly static websites and therefore it is an opt-in mechanism. When a website registers to use SICILIAN, it can start with an INIT phase. This is a pre-deployment testing phase for compiling the initial whitelist by locally scanning the website. Note that the initial whitelist database may not be complete during this phase. Next, there is an optional CRAWL phase, in which the website can get feedback from the users' browsers about the scripts that it had not covered during the testing — hence the browsers *assist* the website in composing the whitelist. These scripts are added to the whitelist based on a trust-on-first-use (TOFU) assumption, meaning that the first time a browser sees a script out of the whitelist in the CRAWL phase, it locally compiles a signature for the script and sends it to the whitelist. This phase is optional, and only required on websites for which the INIT phase is not enough. The CRAWL phase is assumed to have a similar trust assumption as the trust-on-first-use in SSH [59]. Finally, after the whitelist is sufficiently populated, the site initiates the LOCKDOWN phase. In LOCKDOWN phase, the server can turn on a specific header to initiate a lockdown procedure. During this phase, no scripts outside this whitelist will be executed by the client. Thus, we progressively lockdown the versions of the scripts that can execute under a web page. Similar incremental deployment models have been successful in blocking mixed-content vulnerabilities [27, 37] on the web and we believe this makes a pragmatic alternative for a wider adoption of the solution.

Evaluation on Alexa's Top 500. We evaluate our solution on Alexa's top 500 sites with the objective of measuring reduction in false positives and the performance overhead of our approach. Our evaluation covers changes in scripts with respect to time and multiple user accounts. Our evaluation on Alexa's top websites shows that a SICILIAN-enabled browser introduces 4.68% performance overhead over SRI-enabled browser and 7.02% overhead over vanilla browser. Such overhead is acceptable for all but the most latency-sensitive websites. Further, SICILIAN can be fully applied to 372 domains with updates required only once in a month. Therefore, SICILIAN covers five times more domains than SRI, which is based on raw signatures (69 domains).

Contributions. We make the following contributions in the paper.

- We conduct a 3-month study on how scripts change in the Alexa Top 500 sites and 15 popular PHP applications to evaluate the efficacy of JavaScript whitelisting via script signatures. We monitor changes with respect to time and multiple users.
- We propose SICILIAN, a multi-layered whitelisting approach based on signatures to prevent script injection attacks. SICILIAN employs a novel signature scheme, *structural signatures* which rely on source code structure, and are secure & robust against syntactic changes in the scripts. On average, structural signatures reduce the frequency of signature updates by four times over raw signatures.
- We propose a browser-assisted deployment model called *progressive lockdown*, a pragmatic approach to incrementally build the whitelist for preventing script injection attacks.

2. PROBLEM DEFINITION

We discuss various channels available to carry out script injection attacks (Section 2.1), problems with existing approaches (Section 2.2) and define our problem statement (Section 2.3).

2.1 Channels of Script Injection

Web applications are vulnerable to a variety of script injection attacks. First, such attacks can happen via cross-site scripting vulnerabilities (XSS). XSS proliferates as a result of executing non-legitimate scripts within an origin. Such illegitimate script originates from attacker-controlled data to the server's code which are reflected to the generated HTML content (i.e., reflected XSS), from persistent storage (i.e., persistent XSS), or by utilizing unsafe usage of several JavaScript DOM objects (i.e., DOM-based XSS [31,49]).

Second, an attacker can tamper with the scripts' contents via insecure networks or due to unauthorized modifications at the third-party servers. Such attacks come in many variants, most notably are attacks where malware resides in third-party scripts provider, like those of CDNs or servers of popular JavaScript libraries. An increasing demand for using external JavaScript libraries and CDN-based hosting makes them the primary target to malware attacks, as has been reported recently [30, 45, 48]. Once the malware runs on the server, it has unfettered access to tamper with the scripts'

content which is sent to the embedding websites. Additionally, in typosquatting XSS attacks (TXSS) [40], attacker makes use of the developer's mistake of mistyping the URL address of the script resources. As a result, such mistyping leads to a script resource being loaded from attacker-controlled servers.

2.2 Problems with Existing Approaches

To defend against script injection attacks, a number of countermeasures have been proposed, ranging from sanitization [44, 57], privilege separation [4, 8, 20], confinement [3, 13], filtering [6], to various policy enforcement mechanisms [29, 53]. We make the following observations explaining why previous works are insufficient to handle script injection attacks.

Handling an Injection Channel is Difficult. In script injection attacks, web application is vulnerable to malicious script injection via multiple injection channels in the browser (Section 2.1). Therefore, enforcing a reactive approach by securing each and every script injection channel in the browser is not a pragmatic approach due to: 1) plethora of channels in the browser and web applications to hook; and 2) continuous addition of new features and web specifications to the browser [52]. For example, the client-side web provides a wide array of features, such as browser extensions, cross-origin channels, and local storage that an attacker can exploit to execute malicious scripts.

Weak Notion of Malicious Script. Many existing solutions categorize malicious scripts based on the origin or context in which such scripts are being executed. For example, CSP "over-generalizes" malicious scripts as any scripts injected via inline scripts [18] or `eval()`. As a result, developers who legitimately use inline scripts and would like to retrofit CSP into their web application will suffer from such approach. They must relocate their code from inline HTML tag to make the application comply with CSP – this raises deployability burden as discussed in [15, 16, 58]. Many other solutions also have loose notion of malicious scripts. For example, DSI [39] "under-generalizes" script injection as privilege escalation where untrusted DOM nodes try to execute scripts. While this clearly provides protection against script injection via untrusted inline content, it does not give any security guarantees for attacks like DOM-based XSS, where legitimate-but-vulnerable DOM objects evaluate a potentially untrusted data into code.

Browser Variations. There are many subtleties, incoherencies, and variations in the implementation of browsers that may lead to script injection attacks [47]. For example, filtering bad input for XSS is known to be hard since browsers have their own way of interpreting or blocking certain inputs. Therefore attacker may target certain browsers for injecting payloads. The XSS filter evasion cheat sheet as issued by OWASP is also non exhaustive [41]. Further, attacks such as mXSS work even when a website is properly sanitized because such attacks exploit the subtleties in the underlying browser implementation, particularly the backtick implementation [23].

2.3 Problem Statement

Our goal is to propose a solution to mitigate script injection attacks via signature-based whitelisting of JavaScript. To do so, we first evaluate the efficacy of existing script signature techniques on real-world web applications, consisting of Alexa's top 500 sites and 15 popular PHP applications (Table 1 Section 5). Motivated by the kind of changes observed in these scripts, we propose SICILIAN, a robust multi-layered defense to block script injection attacks. In this work, we answer the following research questions.

- RQ1: Are raw signatures practical in real-world websites?
- RQ2: Do scripts on the web change? If so, what kinds of changes are reflected in them, with respect to time and users?
- RQ3: What are the characteristics of a signature scheme that is secure and robust for practical adoption?
- RQ4: Is whitelisting via script signatures sufficient to prevent script injection attacks?

Threat Model & Scope. We consider the attacker to be a web attacker [2] that tries to actively inject malicious scripts via script injection vulnerabilities in a site or by tampering the content of third-party scripts of the site. Further, we assume that a browser can establish a secure channel with an uncompromised web server of the main site (say via HTTPS), but we make no security assumptions for third-party scripts which are imported by the main site. We also note that other attacks such as tampering of the HTML contents or code reuse attacks of JavaScript are beyond the scope of this paper. One class of attacks that affects all signature-based whitelisting approaches is *mimicry attacks* [55]. In our context, mimicry attacks may allow the attacker to invoke scripts already in the whitelist at unintended code-evaluation points or in an unintended sequence. Indeed, mimicry attacks are a concern, but these are outside the scope of our defenses in this paper. Several orthogonal techniques can be used to detect and block mimicry attacks, such as by using sufficient calling context information [17], creation history [56], or anomalous control-flow patterns [14, 19]. Lastly, we trust all plugins and extensions, and assume an uncompromised web browser.

3. DESIGN

In this section, we introduce SICILIAN, our multi-layered approach for building a script whitelist. The multi-layered signature-based whitelisting scheme is based on our measurement study of changes in JavaScript, which we briefly discuss in Section 3.1. We first start with a basic whitelisting layer and show that it is able to protect only a limited number of websites. Then we show an extension of the basic layer to handle scripts with periodic changes.

3.1 Categories of Changes in JavaScript

To design a robust JavaScript whitelisting scheme, it is important to know the kind of changes reflected in real-world scripts. We analyze the changes in the scripts from two aspects: 1) changes over time; and 2) changes over multiple user accounts differing in privilege and access control. Based on our observation on 33,302 scripts hosted on Alexa's top 500 sites, 93% of the scripts remain static over time and the rest 7% change. The changes in these scripts can be classified into four broad categories to answer our **RQ2**.

- C1: *Syntactic*. Changes that affect only the syntactic structure of code and do not affect the behavior of script execution (syntax-only changes). Belonging to this category are changes in comments, renaming of variables and minification.
- C2: *Data-only*. Scripts contain JavaScript functions which take in data as input. We observe periodic changes in such data whereas the functions themselves remain unchanged. Such data does not affect the execution other than side-effects to the network or the DOM (data-only changes). For example, we observe periodic changes in script's data eventually used as a resource's URL or as HTML content to be rendered on a web page.
- Finally, we observe code-introducing changes in the scripts where the JavaScript functions themselves change. Such changes are meant to add or remove functionality from a script. We label this type of changes as C3. Based on the frequency of updates to the scripts, we further categorize C3 into two classes.

- C3A, a class of changes that happen infrequently. Such infrequent updates are due to manual changes pushed by the scripts developer.
- C3B, a class of changes that are highly frequent. These are typically found in news sites or scripts hosted at optimizer services to show dynamic contents in a web page. This class does not fall within our category of mostly-static scripts, because we see no predictable pattern in such changes. We argue that it is difficult to design a signature scheme that caters with such complexity and hence consider C3B type as beyond our scope.

We give a comprehensive breakdown of the number of scripts in each category in Section 5. The results motivate our multi-layered whitelisting design, whose overview is given in Section 3.2.

3.2 Solution Overview

During the execution of a website under an origin O, the browser maintains its corresponding whitelist W_O. W_O contains the signatures of all legitimate scripts S allowed to be executed under O. Let us denote $\mathcal{S}_{\#}$ as a signature mechanism used to construct the whitelist. The whitelisting logic enforces the following invariants.

1. *Execution of valid scripts:* A script s, included under O, is executed if and only if $\mathcal{S}_{\#}(s) \in W_O$
2. *Collision-hardness:* It should be hard to construct a malicious script $s' \notin S$, such that $\mathcal{S}_{\#}(s') \in W_O$

To guarantee the observation of the aforementioned rules, our approach must capture all the scripts that are about to be executed and execute only those that belong to W_O. To achieve this, we 1) interpose on all channels that lead to new code generation and 2) use signature schemes that are hard to bypass.

Interposing on Script-injection Channels. There are two types of scripts that get executed on a web page: first-order scripts, already included in the page, and higher-order scripts, which are generated or loaded dynamically as the web page's code executes. A whitelisting defense must be turned on for all scripts, including first order and higher-order scripts.

Consider an example of a script in Listing 1 which has two points to generate higher-order scripts, namely `document.write` (point (1) at line 3), and `eval()` (point (2) at line 7). Given that the properties `location.href` and `event.data` are under the attacker's control, she can do the following to inject a malicious higher-order script:

- Influence the value of `location.href` with a string to breakout from <a> tag and inject the malicious script thereafter. This happens during dynamic HTML construction at point (1).
- Send a string containing malicious scripts via `postMessage` to this origin, which will be evaluated to code at point (2).

The code in Listing 1 is vulnerable to script injection attacks as it allows the attacker to inject scripts from different points. To block all such possible injection points, our module interposes on all activities in the underlying browser that generates scripts including script generation due to HTML parsing, network activities, or code execution that invokes generation of a new script (e.g., via `eval()`), thus covering both the first-order and higher-order scripts. Our module is built separately from the browser's JavaScript parser and therefore it is *browser-agnostic* — we explain the implementation detail in Section 4.1. Any new script code generated from such script-inducing point will be captured and flagged as a new script and executed only if its signature is in the whitelist (rule 1). From this point onwards, the focus will be on designing a signature scheme that is hard for the adversary to bypass.

Signature Mechanisms. Inspired from the type of changes described in the Section 3.1, we need to design a relaxed signature scheme $\mathcal{S}_{\#}$ that is robust against them. The idea is to relax the raw signatures by ignoring parts of the script that change. In spite of such relaxation, the attacker's capability of injecting malicious scripts can still be greatly restricted. To understand why, let us consider a mechanism called *naive extension of raw signatures* where part of the code which keeps changing is ignored, i.e., the value of a variable `obj` in Listing 1. The remaining text is then signed using raw signature mechanism (SHA256). Such signature computation for Listing 1 is shown in Listing 2 — notice that the changing value is now removed from the computation.

Such signature scheme is guaranteed to be secure but it cannot tolerate the syntax-only changes (`C1` type). We found cases such as minification and variable renaming on scripts which significantly modify the script's code. For example, Listing 3 describes 2 versions of the code fetched from `img.ifeng.com`. They differ in whitespaces and will result in different signatures under the naive extension. However, they are functionally equivalent. Thus, a more relaxed signature scheme is needed that goes beyond the text based signatures in order to overcome such false positives.

One could consider code similarity techniques for designing relaxed signature schemes. Such techniques map two pieces of code that are similar and find application in plagiarism detection. Attribute-based identification (ABI) is one of such techniques used in detecting document similarity [12]. ABI considers the metric — four tuple $(n1, n2, n3, n4)$ consisting of the following code attributes: 1) number of unique operators, 2) number of unique operands, 3) total number of operators, and 4) total number of operands. If one were to use this metric as a signature, then the two pieces of code in Listing 4 would have the same signatures namely $(2, 4, 2, 5)$, but one is malicious and other is not.

As can be inferred from the naive extension and the code similarity technique, there is a trade-off between the depth of relaxation in signature schemes versus the security they offer. Thus, a "minimally-relaxed" scheme which provides high security is needed. This poses the following question: *how relaxed a signature scheme can be and yet be secure?*

AST-based Signatures. The relaxed signature mechanism must be designed in such a way that it does not give up on security, while at the same time avoids false positives due to syntactic changes in the source code. As we discuss above, computing signature on the text representation of a code is not practical since it is not robust to various source modifications. Therefore, we resort to an abstract syntax tree (AST), which represents the syntactic structure of code.

Our signature mechanisms are based on cryptographically secure hash functions and collision resistant Merkle Hash Trees. The adversary can drive the client to execute a malicious script only if its signature belongs to the whitelist. The whitelist W_O is public knowledge i.e. the adversary knows the scripts in the W_O as well as their signatures. Even then it will be hard for the adversary to construct a malicious script $s' \notin S$ s.t. $\mathcal{S}_{\#}(s') \in W_O$. This is due to the *second pre-image resistance* [42] property of Merkle Hash Trees. We discuss the security of our relaxed signature mechanism in Section 3.4. In conclusion, the adversary has a negligible advantage in driving the client to execute a malicious script due to our combined strategy of interposition and security of our signature mechanism, thereby guaranteeing the observation of rule 1 and 2, thereby answering our **RQ3**.

Scope. Finally, we remind that our relaxed signature will ignore changing data and therefore the only way to bypass this is by making non code-injecting modifications in the data, e.g., replacing

```
1 var x = location.href;
2 var obj = '<div>ADS-CONTENT<div>';
3 document.write('<a href='+x+'>LINK</a>'); (1)
4 document.write(obj);
5 window.addEventListener('message', receiveMessage,
      false);
6 function receiveMessage(event){
7   eval(event.data); (2)
8 }
```

Listing 1: An example of a script in whitelist where value of variable `obj` keeps changing

```
1 // Version 1
2 function Collection(){
3  this.items=[];
4 }
5
6 // Version 2
7 function Collection(){this.items=[];}
```

Listing 3: Different versions of scripts due to minification. The scripts were fetched from `http://img.ifeng.com/tres/pub_res/JCore/TabControl.1.2.U.js`

```
1 SHA256("var x = location.href;
2 var obj = '';
3 document.write('<a href='+x+'>LINK</a>');
4 document.write(obj);
5 window.addEventListener('message', receiveMessage,
      false);
6 function receiveMessage(event){
7   eval(event.data);
8 }");
```

Listing 2: Signature computation of code in Listing 1 using naive extension scheme

```
1 // Code 1 (Benign)
2 var obj = "<div>CONTENT</div>";
3 var b = obj + "<div></div>";
4 document.write(b);
5
6 // Code 2 (Malicious)
7 var obj2 = "function evil(){}";
8 var b = obj + "evil();";
9 eval(b);
```

Listing 4: Benign and Malicious scripts having the same signatures under atttribute-based identification

HTML content of variable `obj` (Listing 1) with non-script contents. Due to rule 1, we guarantee that such attempts will not lead to any new code execution. However, few attacks like defacement, mimicry attacks and data exfiltration may still happen. These are beyond our scope and auxiliary defenses like CSP [53], control-flow analysis [19], and data-confinement [3] thwart such attempts.

In the next subsections, we discuss how we design a multi-layered solution to whitelist changing scripts. Our approach is robust against scripts with types `C1`, `C2`, and `C3A` defined in Section 3.1.

3.3 Layer 1: The Basic Scheme

We observe 30,989 scripts that remain static during our three months of crawling. For these kinds of scripts, a suitable signature scheme would be a raw signature. Raw signature scheme can be implemented using various cryptographic hash algorithms, such as SHA256 or SHA512. This basic scheme provides provable collision-resistant signature of the scripts, which we use to guarantee that the attacker cannot inject malicious scripts that can bypass the whitelist, unless she injects the exact same scripts as the ones in the whitelist which are already authorized to run.

3.4 Layer 2: Relaxed Signature

Key Insights. We have seen 2,105 scripts that change, included in 300+ of Alexa's top sites, which we would like to whitelist using our signature scheme. Such scripts belong to the categories `C1`, `C2`, and `C3A`. We propose the abstract syntax tree (AST) as a representation basis for constructing the signature. Abstract Syntax Tree (AST) is a data structure that captures the structure of the source code via an abstraction of the constructs offered by the language and leaves out redundant details like comments, whitespaces, brace brackets and semicolons.

AST grammar. The AST is built by recursive descent parsing of the JavaScript code. Figure 1 represents a part of the grammar used to form the AST. This tree-grammar conceptually illustrates the grammar for a JavaScript parser such as Esprima [24]. The terminals of the grammar are shown in boldfaced letters (e.g. **Program**) and the non-terminals are placed between angle brackets (e.g. <Identifier>). Each AST's **Program** corresponds to one unit of compilation, e.g., a JS file or independent code blocks in inline script and `eval()` function. An AST node is generated when a rule produces a terminal. The non-terminals do not represent the nodes and are mere placeholders.

As a running example, we give an AST representation of a JavaScript snippet `var x = 10, y; y = x+1;` in Figure 2. In general, nodes in the AST belong to one of the three node-types that we detail below.

1. $\mathbb{L}$:= A set of nodes that contains literals. Literals are any values of type `String`, `Boolean` and `Integer` and elements of the unified types like `Arrays`.
2. $\mathbb{I}$:= A set of nodes that contains identifiers. Identifiers are the unique identities or names of the variables, functions and objects in the source code.
3. $\mathbb{LC}$:= The a priori known set of language constructs which is a finite set of nodes representing the semantic representation of the underlying nodes. For example, **VariableDeclarator** is a node that corresponds to the declaration of a *var*.

Labeling of AST nodes. ASTs are labelled trees where the labels of the nodes come from their node-types. Traditionally, ASTs are supposed to abstract out the details of the source code and hence avoid including the identifiers and literals as nodes. However, it is easy to see that for a security critical venture like ours, one cannot exclude them from the AST. We retain the identifier nodes as well as the literal nodes in our AST. The label of an identifier node is its name and the label of a literal node is its value. Referring to the example in Figure 2.e, the label of the identifier `x` is `x` and the label of the literal `10` is `10`. For the nodes in $\mathbb{LC}$, the labels are defined by mapping the nodes to pre-fixed constants. For example, the label of the node corresponding to **VariableDeclarator** in Figure 2.e can be fixed to `VariableDeclaratorType`. A label is an attribute of the node and is shown by circle-shaped nodes in Figure 2.

AST construction. The AST is constructed with the help of the tree-grammar shown in Figure 1. We describe the construction of the AST for the script snippet `var x = 10,y; y = x+1;` in Figure 2. First, the node **Program** is generated using the production rule (1) shown in Figure 2.a. Then, the left-hand subtree of the AST is generated by application of the production rule (2), that is <SourceElement> to <Statement> and then rule (7) (Figure 2.b) to generate the non-terminal <VariableStatement>. Note that no new nodes have been added to the AST after the **Program** node because no new terminals have been produced. On application of rule (12), a child node of **Program** is created called **VariableDeclaration** which in turn produces the **VariableDeclarator** node (rule (13)) with children <Identifier> and the node **Structure**. The child of

(1) ⟨Program⟩ ⊨ **Program** [⟨SourceElement⟩*]
(2) ⟨SourceElement⟩ ⊨ ⟨FunctionDeclaration⟩ | ⟨Statement⟩
(3) ⟨FunctionDeclaration⟩ ⊨ **FunctionDeclaration** [⟨Identifier⟩ **Structure** [⟨ParameterList⟩ ⟨FunctionBody⟩]]
(4) ⟨Identifier⟩ ⊨ **Identifier** [**IdentifierName**]
(5) ⟨FunctionBody⟩ ⊨ **Body** [⟨SourceElement⟩*]
(6) ⟨ParameterList⟩ ⊨ **ParameterList** [⟨Identifier⟩*]
(7) ⟨Statement⟩ ⊨ ⟨Block⟩ | ⟨VariableStatement⟩ | ⟨ReturnStatement⟩ | ⟨EmptyStatement⟩ | ... | ⟨IfStatement⟩ | ⟨ExpressionStatement⟩
(8) ⟨EmptyStatement⟩ ⊨
(9) ⟨ExpresionStatement⟩ ⊨ ⟨Expression⟩
(10) ⟨ReturnStatement⟩ ⊨ **Return** [⟨Expression⟩]?
(11) ⟨Block⟩ ⊨ **Block** [⟨Statement⟩*]
(12) ⟨VariableStatement⟩ ⊨ **VariableDeclaration** [⟨VariableDeclaration⟩+]
(13) ⟨VariableDeclaration⟩ ⊨ **VariableDeclarator** [⟨Identifier⟩ **Structure** [⟨Initializer⟩]]
(14) ⟨Initializer⟩ ⊨ **Init** [⟨Expression⟩ | **Undefined**]
(15) ⟨Expression⟩ ⊨ ⟨Literal⟩ | ⟨Identifier⟩ | ⟨ObjectExpression⟩ | ⟨ArrayExpression⟩ | ⟨BinaryExpression⟩ | ... | ⟨AssignmentExpression⟩
(16) ⟨IfStatement⟩ ⊨ **IfStatement** [**Test** [⟨Expression⟩] **Consequent** [⟨Statement⟩] (**Alternate** [⟨Statement⟩])?]
(17) ⟨Literal⟩ ⊨ **Literal** [**value**]
(18) ⟨BinaryExpression⟩ ⊨ **BinaryExpression** [⟨BinaryOperator⟩ ⟨LHSExpression⟩ ⟨RHSExpression⟩]
(19) ⟨LHSExpression⟩ ⊨ **left** [⟨Expression⟩]
(20) ⟨AssignmentExpression⟩ ⊨ **AssignmentExpression** [⟨AssignmentOperator⟩ ⟨LHSExpression⟩ ⟨RHSExpression⟩]

Figure 1: A part of the Tree Grammar for AST construction. This grammar has been derived from BNF JavaScript grammar. The nodes of the AST are shown as boldfaced letters (terminals) and the non-terminals are shown between angle brackets <>.

Structure, which is a nonterminal <Initializer>, is also generated (Figure 2.c). Applying (4), (14), (15) and (17) will generate the syntax subtree corresponding to `var x = 10` part of the snippet (Figure 2.d). Similarly, the tree-grammar will generate the entire AST as shown in Figure 2.e by application of rules shown in Figure 1.

A simple signature scheme here is to treat the structure (AST) obtained by parsing as the signature for the given source code. We describe the mechanism to represent this structure with a unique hash value in Section 3.5.1. However, we would also like to relax the signature mechanism a bit more such that it allows two structurally-similar pieces of code to have the same hash value (signature) — thereby allowing syntactic changes in the code while preserving the semantics. We achieve this by allowing two forms of isomorphism on the ASTs and explain the isomorphisms in Sections 3.5.2 and 3.5.3. Note that these isomorphisms are designed in a way to be secure in the sense that whenever we treat two pieces of code to have the same signature they only differ in syntax and will not differ in the semantics. We will revisit these security points when we describe our isomorphisms.

3.5 Structural Signature

Structural signature is a signature mechanism for representing the structure (AST) of the code with a unique hash value. Our mechanism for computing structural signatures, denoted as $\mathcal{S}_{\#}$, is based on the techniques used in set fingerprinting [33] and Merkle Hash Trees [35]. We detail the algorithm for computing structural signatures in Algorithm 1 and describe the isomorphisms in Section 3.5.2 and Section 3.5.3.

3.5.1 Bottom-Up Computation

After obtaining the AST of the script code, we traverse the tree depth-first from left to right and perform a bottom-up signature computation as shown in the Algorithm 1. We compute the structural signatures as follows. Consider a node β labeled l_β and children $b_1, b_2 \dots b_k$ with structural signatures $\mathcal{S}_{\#}(b_1)$, $\mathcal{S}_{\#}(b_2)$, ..., $\mathcal{S}_{\#}(b_k)$. The structural signature for node β is defined as

$$\mathcal{S}_{\#}(\beta) = \mathcal{H}(\mathcal{H}(l_\beta)||\mathcal{S}_{\#}(b_1)||\mathcal{S}_{\#}(b_2)||\dots||\mathcal{S}_{\#}(b_k)) \quad (1)$$

where $\mathcal{H}$ is a cryptographically secure hash function like SHA256. This is represented in the lines 16-21 in the algorithm. For the base case where the node is a non-identifier leaf node, we just compute its hash as its structural signature represented in the lines 5-6. For when the node is an identifier leaf node, we handle it Section 3.5.3.

Our mechanism computes the structural signature for the entire tree in a bottom-up fashion. The structural signature of the root node is the signature for the entire AST and hence of the corresponding code.

3.5.2 Isomorphism 1: Node Permutations

This is the first isomorphism that we define on the AST which allows ASTs of two programs differing in the order of the nodes to have the same signature. The mechanism described in Section 3.5.1 can be relaxed to achieve this isomorphism. While computing the structural signature of a node, we sort the structural signatures of its children and then concatenate them in the sorted order. For the example node β mentioned in Section 3.5.1, the structural signature will be redefined as follows:

$$\mathcal{S}_{\#}(\beta) = \mathcal{H}(\mathcal{H}(l_\beta)||\mathcal{S}_{\#}(b_{\pi(1)})||\dots||\mathcal{S}_{\#}(b_{\pi(k)})) \quad (2)$$

where $\pi(i)$ is the index of i^{th} node in the sorted order.

However, enforcing such isomorphism for every node-type might lead to semantic differences. For example, the ordering of children nodes corresponding to arguments of a **CallExpression** or parameters of a **FunctionDeclaration** is important as allowing permutation here may lead to semantically different code. Thus, we impose the sorting mechanism for the unordered node-types like properties of an object. We do not perform such sorting for function arguments and parameters. The lines 15-21 in Algorithm 1 perform this computation. This isomorphism enables the snippets in Listing 8 to have the same signature. This isomorphism can be easily extended to take into account permutation of independent code statements at

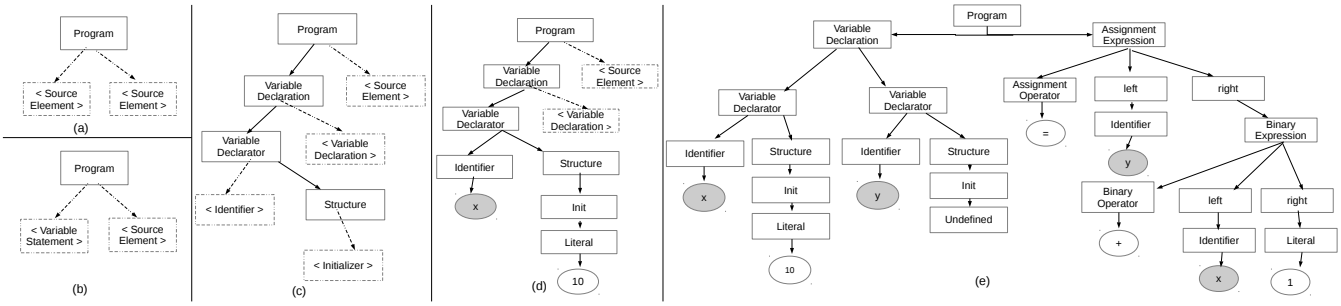

Figure 2: An AST representation of the code snippet "`var x = 10, y; y = x+1;`" derived from the tree-grammar production rules specified in Figure 1. The solid rectangular nodes represent the terminals of the tree grammar (belong to the AST nodes) and the dotted rectangular are the non-terminals of the grammar.

Algorithm 1 Structural Signature Algorithm

```
1:  StructId ← {} //contains structural identities
2:  IdentityPos ← {} //contains positions of an identifier
3:  procedure STRUCT_SIGN(n, pos)
4:      if is_leaf(n) then
5:          if is_non_identifier(n) then
6:              s ← H(n.type||l_n)
7:          else if n ∈ StructId then
8:              s ← StructId[n]
9:              IdentityPos[n].append(pos||"Identifier")
10:         else
11:             n_struct ← STRUCTURE_NODE(n)
12:             s ← H(H(n.type)|| STRUCT_SIGN(n_struct, pos))
13:             StructId[n] = s
14:     else
15:         sign ← {}
16:         for each ∈ n.children do
17:             sign.append(STRUCT_SIGN(each, pos || n.type))
18:         if n is of UnorderedType then
19:             s ← H(H(l_n) || join(sorted(sign)))
20:         else
21:             s ← H(H(l_n) || join(sign))
22:         if n is an child of Program then
23:             UPDATE_STRUCTIDS(s)
24:             FLUSH_IDENTITYPOS()
25:     return s
```

the same level in the AST, the order of variable declarators and the order of function declarations in the global scope.

Safety. The isomorphism under node permutations restricted to unordered node types is secure because ASTs that only differ in the order of these unordered node types will have the same structural signature. Such ASTs correspond to code that are syntactically different but have the same code semantics.

3.5.3 *Isomorphism 2: Label Renaming*

The name of the identifier, which is its label, is unique and can be used for computing its structural signature, i.e., $S_{\#}(x) = \mathcal{H}(x)$ for an identifier `x`. However, in our experiments, we find that identifiers get consistently renamed with no change in the logic of the source code, as shown by code hosted at `sstatic.net` in Listing 5. So, we would like two pieces of code differing only in variable names to have the same structural signature. To achieve this, we define the isomorphism under renaming of the labels. The isomorphism proceeds by trying to define a unique identity for every

```
1 // Script during first visit
2 var f = parseInt(o.css("margin-top")),
3 h = f + d;
4 t.is("textarea") || (h =
      parseInt(o.prevAll(":visible").eq(0)
      .css("margin-bottom")) + f), o.css("margin-top",
      h)
5
6 // Script during second visit
7 var h = parseInt(o.css("margin-top")),
8 f = h + d;
9 t.is("textarea") || (f =
      parseInt(o.prevAll(":visible").eq(0)
      .css("margin-bottom")) + h), o.css("margin-top",
      f)
```

Listing 5: An example of variable renaming in a script monitored over two subsequent visits. Underlined variables are renamed during the second visit.

identifier which can further be used to replace the identifier's name in the signature computation. If it is not possible to represent an identifier with a unique identity, then we do not apply this kind of relaxation — we fallback to using the name of the identifier and resort to the signature computation as explained in Section 3.5.1 and Section 3.5.2 in order to avoid semantically different code to have the same signature.

Structural Identity of an Identifier. In order to define an identity for an identifier, we first take into account the identifier during its declaration. Consider the following variable declaration `var x = 10`. From this statement, we can infer that `x` can be identified as a variable which has a number `10` assigned to it. In general, the type and the initialization value can be used to identify an identifier. We refer to this as the initial identity of the identifier. The type of an identifier can either be `VariableType` or `ObjectType` and the initialization value is either the value it is assigned to during declaration or undefined, if nothing is assigned to it.

Note that type and initialization alone do not always uniquely identify the identifier as there may be more than one identifiers initialized to the same value. These identifiers then may be used in different statements and at different positions in a statement. Simply using type and initialization value as the identity might lead to semantically different code having the same structural signature. Therefore, we propose to use the *position set* of the identifier in the statement to further refine its identity. The position set of an identifier describes the structure of the statements it is used in and the position(s) of the identifier in these statements. For example, the variables `x` and `y` in the statement "`y = x + 1;`" are in different positions — `y` is positioned as the left hand side of the assignment expression while `x` positioned as the left hand side of a binary expression which is also the right hand side of the assignment expression (Figure 2.e). Therefore, the set of positions at which an

identifier is used in the statement along with the structure of the statement captures the usage pattern of the identifier and thereby uniquely identifies it.

We refer to the identity of the identifier as "structural identity" as it is based on its structure that consists of the type, initialization value and the position set in the AST. An identifier initially has a structural identity based on its initial structure (type, initialization value) which gets refined during the tree traversal according to the statements it is used in and its positions in the statements. This refinement takes place at the end of each statement or, in terms of the tree grammar, at the end of the signature computation of the source elements[2]. Henceforth, we will use statement and source elements interchangeably whereas both terms refer to source elements.

The above idea of structural identity readily extends for function identifiers whose initial structural identity is based on the structure of the function body and the parameter list and further gets refined as per its usage in the whole **Program**'s scope. We explain how to compute structural identity next.

Structural Identity Initialization. As previously mentioned, any identifier when declared has an initial structure and hence a structural identity. To capture this initial structure, we add a new node to the language constructs called **Structure** as shown in Figures 2[c-e]. The generation of the **Structure** node is described in rules (3) and (13) of the grammar in Figure 1. This structure node, along with the type defines the structural identity of the identifier, referred to as `StructId`. For example, the `StructId` of the identifier node `x` in Figure 2 is computed as follows

$$StructId(x) = \mathcal{H}(\mathcal{H}("VariableType")||\mathcal{S}_{\#}(Structure)) \quad (3)$$

Here, $\mathcal{S}_{\#}$(**Structure**) node is computed in the bottom up fashion as described in Algorithm 1. The lines 9-11 of the algorithm return a signature for the identifier node that is independent of its name but completely dependent on its initial structure. `StructId` of the identifier is stored in the datastructure `StructId` and updated according to the line 13 of Algorithm 1. The structural signature of `x` is the same as its structural identity. After computing the structural identity of `x`, the signature computation proceeds in the bottom-up fashion to compute signatures of the parent nodes.

Refining Structural Identity. The initial structural identity needs to be refined as per the usage of the identifier in the code. We propose to use the position set of the identifier in the AST to refine its identity. The position set of an identifier consists of two things: i) structure of the statement it is used in and ii) the positions of the identifier in the statement.

The structural signature of a node defines the structure of the entire subtree rooted at itself. This is because of the bottom up computation nature of structural signatures. Referring to the AST of our example statement in Figure 2, the signature of **AssignmentExpression** compactly describes the structure of the statement `y = x + 1`. Additionally, to define the position(s) of an identifier in the statement we use a notion similar to XPath in HTML [36]. For example, sequence of nodes from **AssignmentExpression** to the identifier node corresponding to `y` in Figure 2 uniquely describes the position of `y` in the statement `y = x + 1`. Thus, both $S_{\#}$(**AssignmentExpression**) and the position of `y` represented by (**AssignmentExpression** $\rightarrow$ **left** $\rightarrow$ **Identifier**) collectively describe the usage of `y`. These are used to refine the structural identity of `y` as shown in the Equation 4. Similarly the structural identity of `x` is refined as shown in Equation 5.

[2] A program is a collection of source elements which are either statements or function declarations according to the rules (1) and (2) of Figure 1

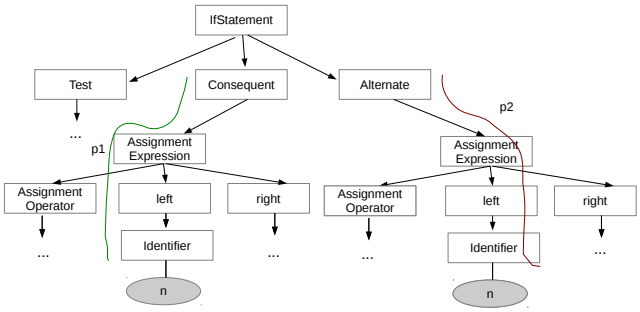

Figure 3: Partial AST for the IfStatement present in the code of Listing 6. The positions of the identifier n are marked with green (p1) and red (p2).

$$StructId(y) = \mathcal{H}(StructId(y)||s||\mathcal{H}(pos_y)) \quad (4)$$

$$StructId(x) = \mathcal{H}(StructIdentity(x)||s||\mathcal{H}(pos_x)) \quad (5)$$

where $s = \mathcal{S}_{\#}(AssignmentExpression)$, pos_x and pos_y are the positions of `x` and `y` respectively in the statement.

The positions of an identifier are described as a sequence of nodes starting at the source element or the child of the Program node which is the node **AssignmentExpression** for the statement `y = x + 1;`. This position can be recursively computed by passing the position to the children while computing their structural signatures as shown in lines (12) and (17) of Algorithm 1. The positions of the identifiers are stored in the data structure `IdentityPos`. A single identifier can have more than one position in a source element. For example, `x = x + 5`, there are two positions of `x` and both are to be recorded as they both describe the usage of `x`. This is done by making a list of positions and hence `IdentityPos[x]` is a list of positions of `x` in the current statement. `IdentityPos[x]` is updated according to line 9 in Algorithm 1.

Algorithm 2 Identity Refinement Algorithm

```
1: procedure REFINE_STRUCTIDS(s)
2:     for each ∈ IdentityPos.keys do
3:         PathHash ← ""
4:         for p ∈ IdentityPos[each] do
5:             PosHash = PosHash||H(p)
6:         t ← s||PosHash
7:         StructId[each] = H(StructId[each]||t)
```

The `StructId` of the identifiers are updated at the end of the signature computation of the source element node (a child of the Program node). This is described in the lines 22-24 of Algorithm 1. The algorithm for refining the structural identities is shown in Algorithm 2. At the end of the signature computation of a source element, the data structure `IdentityPos` is flushed as the current positions have already refined the structural identity of the identifiers.

Similarly, if the source element were an **IfStatement** with an identifier involved in both the "if-block" and the "else-block" as shown in Listing 6, the signature computation proceeds as before. The `IdentityPos[n]` will contain two positions of `n` namely `p1` and `p2` as shown in the Figure 3. We therefore merge both the positions at the join point, i.e., after the end of the **ifStatement**. The structural identity of `n` is updated by concatenating the hash of the positions to the structural identity as shown in Equation 6.

```
1 var b = generate_random_number(0,1), n;
2
3 if(b==0)
4   n = "even"
5 else
6   n = "odd"
```

Listing 6: An example of a script consisting of IfStatement

$$StructId(n) = \mathcal{H}(StructId(n)||s||H(p1)||H(p2)) \quad (6)$$

where $s = \mathcal{S}_{\#}(\textbf{IfStatement})$ and `p1` and `p2` are the positions of n in the **IfStatement**.

Safety. Using the type, initialization value and the position set in the AST, we refine the structural identities of the identifiers in order to generate unique identity for each of them. But if two identifiers have the same structural identity then they conflict with each other. Replacing such identifiers by their structural identities could open avenues for the attacker to replace one by the other and achieve script injections. Therefore, if such a case happens, we retain the names of the identifiers and do not replace them with their structural identities. With this restriction, we compute the structural signature for the script again using Algorithm 1. Such cases of conflicting identities did not come up in our large scale evaluation as structural identities of every identifier in the script were unique. Such restriction is, however, required to be enforced in order to guarantee that two scripts with different semantics are not treated as one signature.

3.5.4 *Collision-Resistance*

The structural signatures generation that follows the mechanism explained in the previous steps will lead to the same structural signatures for two scripts that are equivalent. However, no two structurally non-equivalent scripts must lead to the same structural signatures. This means our mechanism must not lead to false negatives. Since the underlying idea of structural signatures is inspired from Merkle Hash Trees, we borrow the collision resistance property of our signatures from the collision resistance of the top level root signature of Merkle Hash Trees.

Thus, structural signatures establish the definition of structural equivalence formulated in terms of AST isomorphism. This implies that it is very difficult for the attacker to introduce a new script that is structurally different from a script in the whitelist, while at the same time has the same structural signature with the script in the whitelist — this answers our **RQ4**.

4. ARCHITECTURE

In this section, we discuss SICILIAN, a solution to implement JavaScript whitelisting based on multi-layered signature schemes. We detail how our whitelisting mechanism works, specify the whitelisting policies, as well as techniques for constructing such whitelist. Finally, we outline the deployment scenario of our approach.

4.1 Browser Modification

We explain the implementation detail of SICILIAN with respect to a browser flow model discussed in [57] (illustrated in Figure 4) and Chromium design, although our approach is also applicable to other mainstream browsers. Our whitelisting logic must guarantee that only the authorized scripts (scripts with legitimate signature) are allowed to enter the browser's JavaScript parser module, as mentioned in rule 1 in Section 3.2. Therefore, our whitelisting module must be placed in a location where it is exposed to all the interfaces JavaScript parser has with client-side web components like HTML parser and CSS parser. In Chromium, the right point to

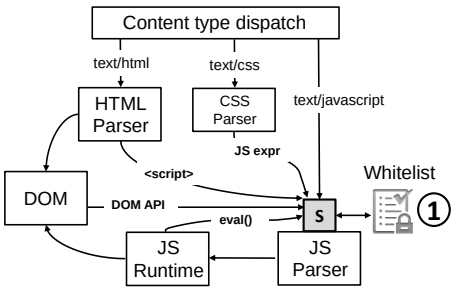

Figure 4: Deployment point of our whitelist logic. Our whitelisting module (illustrated as a greyed box **S**) is a standalone program which validates any scripts entering browser's JavaScript Parser.

place our whitelisting logic is in a function `executeScript()` of class `ScriptLoader`.

Next, our solution employs multi-layered approach when validating the scripts against the whitelist. First, all the scripts are checked against the whitelist (Figure 4 point ①) using raw signatures. Scripts that pass the first layer of validation will then be allowed by the browser to execute. Otherwise, the browser carries out the second layer of validation, which computes the signature of the remaining scripts with structural signatures. Scripts that do not pass this validation are blocked by the browser.

Finally, our whitelisting module is not built by modifying the implementation of the browser's JS parser. Rather, we build our solution as a standalone module hooked into the browser's JavaScript parser, shown as greyed box in Figure 4. To compute the structural signature of a script, SICILIAN needs to parse it and perform a tree traversal. Therefore, we equip the whitelisting module with our own parser written in JavaScript, modified from Esprima [24]. Since our parser is written in JavaScript, the script to be validated and our parser logic are processed by Chromium's JavaScript Engine via a function call `Script:Compile()`. The reason for not using browser's JavaScript parser is that the parsing logic varies across browsers [1, 9]. This may lead to inconsistent signatures across browsers. So, we champion a *browser-agnostic* approach[3].

Although our approach requires modification to the browser — as with many other defenses like HTTP Strict Transport Security [37], CSP [53], or W3C's Subresource Integrity [7] — it requires no proactive action at the end-user since minor patches for SICILIAN can be delivered transparently via browser auto-updates.

4.2 Whitelist Policy Specification

Our structural signature comes with a *fail-safe* policy that is composed of directives that manage how SICILIAN is imposed on a script. After applying such directives, the browser will block a script from running if its signature is not in the whitelist. All policies are defined at a web page granularity such that they remain applicable throughout the execution of the webpage. Here, we define a web page with respect to a URL of a page including the origin and its subpath (i.e., anything before the '?' character in the URL). We discuss the construction of such whitelist in Section 4.3.

Signature Directive. In our policy definition, the script's signature is specified by a directive `signature`. This directive is specified by a JSON object containing four properties, namely `type` which specifies the signature type to be imposed on the script, `id` which is the identifier of a script (e.g., script's URL), `value` which emphasizes the expected hash value of the script, and `policy` which specifies certain JavaScript literals which will be ignored during

[3]Browser-agnostic here means that the module is agnostic to a particular browser's parsing logic and JavaScript engine

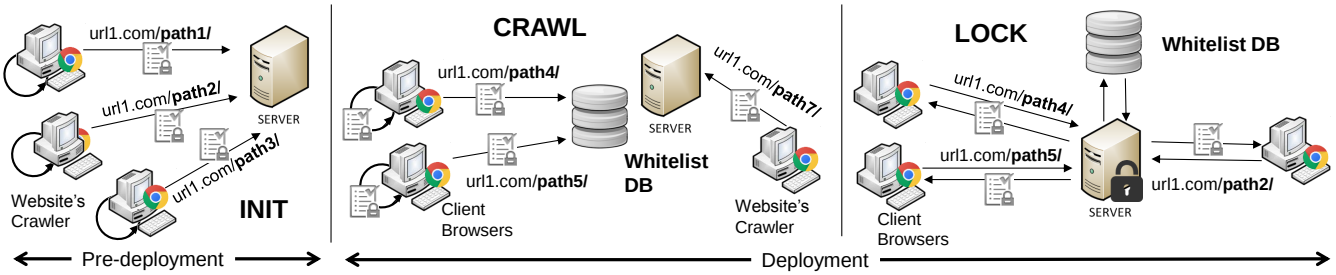

Figure 5: High-level overview of whitelist construction. Our technique comprises of three phases namely INIT, CRAWL, and LOCK.

structural signature computation — this is specified in the data directive below. The browser validates the script by computing either raw signature if the `type` is `raw-sign` or structural signature if the type is `struct-sign`.

Data Directive. This directive specifies the data that keeps changing in the script. Such data will be ignored by our structural signature computation. The changing data is specified by a directive `dynamic-data` which is a JSON object containing two properties, namely `name` which denotes the identifier of the data (e.g., variable name) and `data_loc` which uniquely identifies the relative scope chain of the identifier. This directive can either be served as annotations by the website admin or inferred automatically by the browser (during the crawl phase in Section 4.3)

Example Policies. We provide examples of policies according to the real scripts we encountered in Alexa's top sites.

Example 1. Site want to employ raw signature on a script hosted at `somewhere.com/site.js`

```
1 --signature:{type:'raw-sign',
      id:'something.com/site.js',
      value:'aLp+5608ed+rwmLOHboV=='}
```

Example 2. Site has a script hosted at third-party site `somewhere.com/site.js` which has a variable which value keeps changing. The variable's name is `'bar'` inside a function name `'foo'`. The website owner wants to sign this script with structural signature.

```
1 --signature:{type:'struct-sign',
      id:'something.com/site.js',
      value:'aLp+5608ed+rwmLOHboV==',
      policy:{dynamic-data:{name:'bar',
      data_loc:root-foo}}}
```

Our approach requires the web page to whitelist all scripts that are going to be loaded and executed. However, this raises deployability issues since web application can generate a massive amount of web pages. Crawling such pages during pre-deployment analysis may not always lead to complete code coverage.

4.3 Deployment: Progressive Lockdown

We propose a browser-assisted whitelist construction method that helps the website owner to specify a correct whitelist for web pages which do not yet have it. Our entire whitelist construction is based on *progressive lockdown*, which slowly introduces whitelists as new web pages are being discovered by the browser. We introduce three important whitelist construction phases, namely 1) INIT where the developer carries out a pre-deployment phase to compile initial whitelists for a limited set of web pages, say all web pages at depth of two from the landing page, 2) CRAWL where the browser assists the website in building a whitelist for web pages that have not been visited during the pre-deployment, and 3) LOCK to switch the website into a full-fledged whitelisting-based mode. Note that this is an *opt-in* approach where the website owner can build such whitelist without any significant changes to the server-side code. We illustrate our whitelist construction technique in Figure 5.

INIT Phase. During the pre-deployment phase, the developer carries out INIT to sign its web pages. The goal is to create an initial whitelist and manually add policies in the whitelist. Such manual policy configuration includes adding `dynamic-data` directives to specify which data keeps changing in a script. This step is necessary to whitelist scripts in web pages that are likely to be visited by the user. As a result, such pages are "guarded" from potential malicious script injection.

CRAWL Phase. After the INIT phase, the website can opt for a second step, that is the CRAWL phase. CRAWL is carried out at the deployment phase with the help of client's web browser to compile a whitelist for web pages that have not been visited during INIT. Our crawling mechanism is based on trust-on-first-use (TOFU), meaning that the first time a browser sees a script out of a whitelist, it locally compiles a whitelist for the script and sends it to the whitelist database. Although TOFU is applicable for the entire CRAWL phase, the browser is not going to trust any new scripts on the second visit to the script. Additionally, the server can himself continue the INIT phase to populate the whitelists.

We apply our multi-layered solution described in Section 3 to construct such whitelist. First, if the browser does not encounter any changes during multiple visits on the same scripts, the browser will set such scripts as static and use the raw signature (Layer 1 in Section 3). Otherwise, the browser will resort to structural signatures (Layer 2). While compiling the whitelist, the browser records the data that changes in the scripts to suggest additional policies regarding the data-only changes to the website admin. Note that this can be done as a background job without delaying page load.

A typical CRAWL phase may vary, but may last up a month in order to get a precise and fine-grained policy. This is based on our observation that the majority of non-static scripts that we study had already been changed within 1 month interval. Once a complete whitelist is constructed for all scripts on a web page, the browser then sends a tuple of page's URL and its whitelist to a database owned by the website via an out-of-band channel.

LOCK Phase. Once the whitelist database is sufficiently populated, the website owner can initiate the LOCK phase. The whitelists returned by the browsers in the CRAWL phase may conflict with each other. The server can resolve such conflicts and decide the final whitelist database through manual intervention. However, we expect the majority of the scripts to be whitelisted automatically without conflict since they are mostly static scripts (see Section 5). This final whitelist is thus stored on the server who is trusted for its integrity. Once the final whitelist is ready, brow-

sers must strictly follow the whitelist provided by the server — all the partial whitelists compiled on the browsers during the CRAWL phase must be flushed. For each browser's request to a web page, the web server will query its database and attach the corresponding whitelist for the web page during the HTTP response. In the LOCK phase, the whitelist is communicated to the browsers using a custom HTTP header `X-Whitelist:[directives]`. If there are any changes to the server code post-lockdown, the server can update the whitelist database itself and serve the updated whitelists for future visits.

4.4 Signature Updates

We recall that 461 scripts in our crawl belong to `C3A`, where website developers add or remove functionality from the scripts and this happens in an infrequent manner. Scripts within this category undergo a non-syntactic change and no longer preserve the code semantics. Once this happens, signature mismatch errors will be raised on all browsers loading these scripts. Since the changes are legitimate and infrequent, the website admin can do the following to handle such updates:

1. If the script is hosted within a website, website admin can recompute the signature of the script and update the value in the whitelist. This way, browser that accesses a web page will get the updated whitelist. Signature mismatch can only occur when the browser still retains the old signature of the script. In such cases, the browser can simply communicate back to the server to get an updated signature.
2. If the script is imported from some third-party services, the browser will report such incidents back to the main website's server. The admin of such site can either analyze the script and decide whether such changes are legitimate, or it can block the inject attempt and contact the third-party script provider about the script changes. If the changes are indeed legitimate, the admin can update their value in the whitelist database.

5. EVALUATION

In this section, we give an empirical analysis of our 3-month long measurement study (from 31st January 2015 to 30th April 2015) on Alexa's top 500 sites and 15 popular PHP applications (see Table 1) with the following goals. First, we would like to determine whether raw signatures are practical for scripts in Alexa's Top 500 websites and popular PHP apps. Second, we show that our signature scheme and policies defined in Section 3 are easy to implement and expressive enough to whitelist benign scripts on Alexa's top websites. Further, structural signatures are able to reduce false positives as compared to the raw signatures. Lastly, we show that our approach incurs only a small performance overhead on the browser.

efront	X2CRM	SquirrelMail	AstroSpaces
elgg	Magento	PhpScheduleIt	Cubecart
ownCloud	osCommerce	OpenCart	Dokeos
PrestaShop	ZenCart	Gallery	

Table 1: Popular applications that we investigate for user-related changes.

Implementation. We implement SICILIAN by modifying Chromium version 43.0.2315, an open source version of Google Chrome. We implemented a module which can be patched to the Chromium's JavaScript engine by adding approximately 200 lines of C code and 500 lines of JavaScript code spreading over 2 files in the Chromium's codebase. We have released our patch to the Chromium's JavaScript engine on a public repository [46].

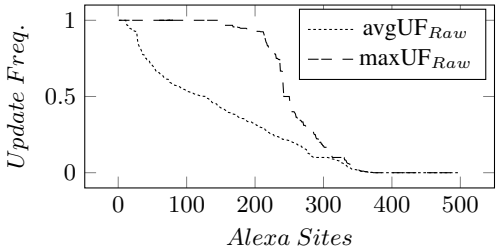

Figure 6: The figure represents the variation of average and maximum of update frequencies of all scripts of a domain across Alexa's top 500 sites, sorted in descending order of their update frequencies. The graph is based on a UF_{raw} metric, which indicates how often raw signatures are updated for a number of visits.

Platform. All experiments were conducted in Chromium v43 set up on a Dell Latitude 6430u host, configured with Intel(R) Core(TM) i7-3687U 2.10GHz CPU and 8GB RAM running 64-bit Ubuntu Linux 14. We use Ninja v.1.5.1 for compiling our modified Chromium browser.

5.1 Insufficiency of Raw Signatures

Our study on the Alexa's top 500 shows that none of these popular websites employs signature checking on their script resources, which we suspect is due to the deployability burden in retrofitting such mechanisms. Such burden is caused by the highly dynamic scripts which require developers to frequently update the signatures. To justify this claim, we crawl for 45066 webpages logging 33,302 scripts from Alexa's top 500 and record their changes periodically over 3-months starting 31st January, 2015. Each script was visited 52 times on average in our experiments, giving an average of 17 visits per month. Therefore, every website has an average of 17 visits per script per month. We introduce a metric called *update frequency (UF)*, which measures how often a script's signature changes per visit. Note that change in script's signature implies that the content of the script itself has changed. Update frequency for raw signature is calculated as follows:

$$UF_{Raw} = \frac{\text{\# times the raw signature changes}}{\text{\# visits to the script}} \tag{7}$$

Our result of Update Frequency (UF) in Alexa's top 500 website is summarized in Figure 6, which shows the sorted version of maximum and average UF of scripts in Alexa's 500 domains. In the figure, 59 error domains were assumed to have both maximum and average UF = 0. As seen from the figure, 300+ domains have an average UF > 0 which suggests that such domains have at least one script that changes. We further observe from the `maxUF` graph that there are almost 200 domains which have at least one script with UF = 1, i.e., this script keeps changing for every visit. If raw signatures via SRI are to be imposed on websites with such scripts, the developer needs to undertake the burden of updating the signature of the script for each request — which is not practical in real world settings.

Deployability of Raw Signatures. We investigate the number of scripts on which raw signature based solutions (e.g., SRI, BEEP [29]) can be fully applied. As mentioned earlier, we find 30,989 static scripts which accounts for approximately 93% of all the scripts that we have crawled. However, only 69 websites of the Alexa's top 500 (13.8%) and 7 out of 15 PHP apps can be fully retrofitted to raw signature-based solution because only these sites do all the scripts remain static. Thus, raw signatures have limited practical adoption thereby answers our **RQ1**.

5.2 Quantitative Analysis of SICILIAN

Deployability of SICILIAN. We find that 33,094 scripts can be whitelisted using SICILIAN, which are approximately 99.4% of all the scripts that we have crawled. Among the Alexa's top 500 sites and 15 PHP applications we analyze, SICILIAN can be fully applied to 372 websites and 15 PHP apps. This means that all the scripts imported by those websites and apps fall either in the `C1`, `C2`, or `C3A` category of script changes. By Equation 8 for these 372 domains, we get an average UF_{Struct} of 0.057. This is equivalent to 1 whitelist update[4] per month. Given that 59 of them were not visited due to errors on the websites, this accounts for 84.7%[5] of all websites we crawled. SICILIAN covers five times more domains than SRI (69 websites), which is a raw signature-based whitelisting solution. Our approach also works on popular and highly-dynamic websites such as Google, Ebay, and Amazon.

$$UF_{Struct} = \frac{\text{\# times the structural signature changes}}{\text{\# visits to the script}} \quad (8)$$

Rate of Signature Updates. We compare the rate of signature updates for scripts with raw and structural signatures by comparing their update frequencies (Equation 7&8). Figure 7 shows the variation of update frequency of raw and structural signatures among Alexa's 500 domains, sorted according to UF values. Of the 500 domains, 59 domains are excluded from the figure as these sites either returned no HTML or had at least one DNS error (unreachable) in our crawl. Of the 441 websites, there are 153 and 334 domains with an UF $\leq$ 0.1 for raw signatures and structural signatures, respectively. This implies that our signature mechanism works for 334 web sites, assuming that they are update frequency is lesser than 0.1 (5 times in our 3 month measurement[6]). In contrast, SRI is only applicable to 153 of the sites with such a rate of updates. We point out that each signature update in SRI requires changing all integrity attributes on all parent web-pages, whereas our approach is one update in the server-side database. Further, the number of domains increases to 433 (98% of all non-error domains) for UF $\leq$ 0.5 as compared to 314 domains of raw signatures. Thus, structural signatures significantly reduce the update frequency of Alexa's top 500 sites. On average, the UF for structural signatures (μS = 0.075) is four times smaller than the average UF for raw signatures (μR = 0.28)(Figure 7).

Time-related Changes. We classify our crawled scripts into categories defined in Section 3.2 and summarize the result in Figure 8. As seen from the figure, 71.07% of the changes in the scripts belong to `C1` union `C2` and 4% of them fall in both `C1` and `C2`. Scripts belonging to `C2` are the majority of the changes (46%+4% = 50%) and thus `C2` is a common practice in Alexa's websites (271 domains). If site admin wishes to impose raw signatures on scripts in `C1` and `C2`, she needs to extensively rewrite server code to retrofit to raw signatures. Therefore, such script changes are not easily fixable. As opposed to raw signatures, structural signatures can robustly handle them without enforcing any burden on the admin.

Further, `C3A` contributes significantly (20%) to the changes and affects 247 domains, which suggests that developers regularly maintain their site's codebase. Scripts in `C3A`, however, have very small UFs (0.066 on average). This is equivalent to approximately three

[4] 0.057*17, given an average of 17 script visits per month per domain.

[5] This is obtained from (372/(500-59))*100%

[6] Out of the 372 websites on which Sicilian can be fully applied, 334 require an at most 5 whitelist updates in 3 months while the remaining 38 require an at most 5 whitelist updates in 2 months. Of the 334 sites, majority of the sites (243) require an at most of 1 whitelist update per month. Together all 372 sites give an average of 1 whitelist update per month due to an average UF = 0.057.

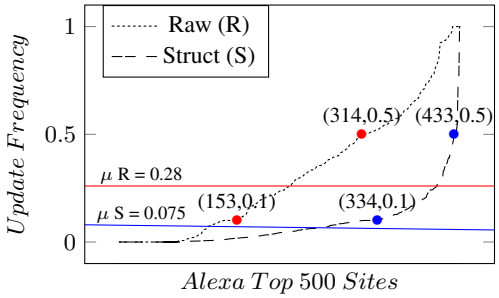

Figure 7: Variation of update frequency of Alexa's 500 domains for raw and structural signatures . The figure represents domains sorted in ascending order of their update frequencies. We remove 59 error domains from the graph. μR and μS represent the average UF of 441 domains for raw signatures and structural signatures, respectively.

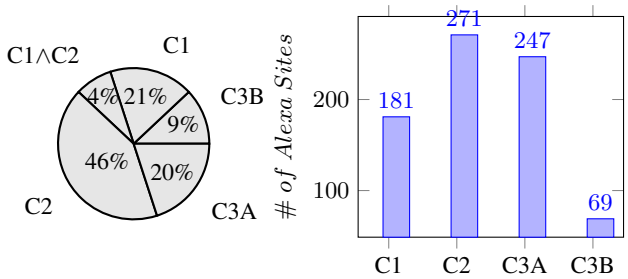

Figure 8: Pie chart on the left represents the distribution of scripts that have changed into four categories and Bar graph on right represents the number of Alexa sites affected by these four categories of changes.

script updates during our 3-month crawling period, given an average of 52 script visits. Therefore, although such scripts are significant and affect many domains, the effort of updating their signatures is relatively low (once a month).

Finally, we find that scripts in `C3B` are highly dynamic, and cannot be whitelisted with any kind of signature schemes due to their high UFs (0.652 on average). Such highly dynamic scripts include advertisements with customized scripts [28] or scripts from news websites. However, such "non-static" scripts are scarce (0.62%) with respect to all the crawled scripts and used only in 69 domains. These domains can be retrofitted to SICILIAN by placing the `C3B` scripts into separate `iframe` container, as done in [4]. Such modification requires additional development effort, but it may be feasible given the small number of `C3B` scripts.

App.	# P	# C	# D	Value of `name` in SICILIAN Policy
efront	100	1	1	var `chat_listmenu`
Elgg	100	100	1	var `lastcache`
ownCloud	100	100	1	var `oc_current_user`
PrestaShop	100	100	1	var `static_token`
x2CRM	100	1	3	var `yii`
Magento	100	1	3	var `region_id`
osCommerce	100	1	2	var `theme_token`
ZenCart	100	1	1	var `s`
Rest of seven applications (Table 1) exhibit no user-related changes.				

Table 2: Changes in scripts due to multiple users. SRI handles none of these changes and SICILIAN can handle all of them with the corresponding policy mentioned in the last column. # P represents the pages crawled, # C represents the pages where inline scripts change, # D represents the number of data that change.

Per-user Changes. Beside time-related changes, we analyse 15 PHP applications (listed in Table 1) to determine changes related to multiple users. Such standalone web applications enable us to

create multiple user accounts with different privilege levels which is difficult in popular live websites. Note that we do not apply any patches on the installed PHP applications thus excluding C3 category from our observation set. We crawl for 100 webpages on each application and summarize our observations in Table 2. Seven out of the 15 applications exhibited no change in external, internal and inline scripts and hence can be retrofitted to raw signatures as stated in Section 5.1. In the rest eight applications, only inline scripts were observed to change. These applications maintain user specific information like var oc_current_user mentioned in Table 2. All the crawled webpages changed in three of the applications (eg. Elgg, owncloud and PrestaShop) but only in small number of data (see column "#D"). The changes observed across users fall in the C1 and C2 category. These can be handled by structural signatures by specifying the dynamic-data policy for variables mentioned in column SICILIAN's policy. Thus, we infer that such standalone applications remain mostly static across different users and structural signatures can be used to whitelist them.

Performance. To measure performance, we sample one page for every domain in the Alexa's top 500 websites and compare the page load time between vanilla browser, SRI-enabled browser (implements raw signature), and SICILIAN-enabled browser — averaged over five attempts. We outline the performance overhead of five websites and the overall 500 websites in Table 3. As listed in the table, our structural signature mechanism gives an average performance overhead of 7.02% for pages in 500 domains. Such overhead is acceptable for all but the most latency-sensitive websites. Our multi-layered approach guarantees structural signatures are only applied on changing scripts. The remaining (7.02% - 2.34%) = 4.68% overhead is due to procedures required to compute structural signatures like building the AST and computing hashes while walking the tree. However, the overhead for computing structural signatures is overshadowed by the time for loading non-script resources such as videos. Therefore, structural signature tends to give high overhead on sites that contain only texts, such as wikipedia.org (20.94%) and wordpress.com (9.5%). Despite such overhead, the total page load time for these sites is small (below 3s) and hence not impacting user experience significantly.

Domain	Vanilla	Raw-Sign		Struct-Sign	
	LD	LD	OVD(%)	LD	OVD(%)
blogspot.sg	8.25	8.7	5.35	8.708	5.44
wikipedia.org	2.158	2.168	0.46	2.61	20.94
mashable.com	23.74	24.75	4.26	25.14	5.89
google.cn	4.55	4.65	2.19	4.86	6.94
twitter.com	2.79	2.86	2.57	3.03	8.51
Average Overhead of Alexa's 500 Sites			2.34%		7.02%

Table 3: Performance evaluation of structural signature on Alexa's Top 500 sites. LD represents load time of the page, measured in *seconds* whereas OVD represents overhead percentage (%), as compared to the load time in vanilla browser.

Policy Development Effort. A fine-grained policy for a script can be derived with modest development effort. For our dataset, it took us less than 5 minutes of manual effort per script to compile a fine grained policy. The average policy size is 83 bytes for static scripts and 263 bytes for changing scripts. Since there are 30,989 static scripts and 2,105 changing scripts, we expect the entire whitelist database size to be around 2.9 MB for the 33,302 scripts we have crawled. The most time consuming steps were code-beautification and the use of a text differencing tool to find out the dynamic parts of the code. The developer may need to compare several versions of the script to derive a fine grained policy, however in our experience three script versions proved to be sufficient. This manual task of determining the policy and which variables change can be automated by comparing the AST of the different script versions. In our semi automatic analysis of the crawled data set, we determined the types of changes in all the scripts along with the variables whose values change therefore building the policy for all 33,302 scripts.

5.3 Examples of Script Changes

In this section, we give examples and comprehensive breakdown of scripts in C1 and C2 category of changes. These changes were observed in our 3-month crawling period.

5.3.1 *Scripts in* C1

We first categorize changing scripts that preserve the code semantics as C1 which can be handled by using structural signatures. Table 4 gives details about the number of scripts falling in C1. Scripts in C1 can be further classified into four sub-categories (see Table 4), with changes in code comments section as the majority. We point out that few scripts in C1 exhibit more than one type of change and hence belong to more than one sub-categories.

Changes	# ES	# IS	Total	UF	# AD
Comments	274	192	466	0.401	94
Permutations of Object	51	10	61	0.749	92
Variable Renaming	8	1	9	0.22	5
Others	7	85	92	0.269	30

Table 4: Changes in scripts belonging to C1. #ES and #IS represent the number of external and internal scripts respectively, Total denotes the total number of scripts (sum of external and internal), UF represents the update frequency and #AD the number of affected domains.

Code Comments. In our experiments, out of the 586 scripts falling in C1, 466 scripts had changes in the comments section of the code. These form the majority of the script changes observed in C1. As shown in Table 4, these 466 scripts are included in 94 domains and have an UF value of 0.401 on the average, which indicates that scripts with changes in comments tend to be updated frequently. We present an example where the comments section of a script changes periodically in Listing 7.

```
/* requested Sat, 14 Feb 2015 8:10:57 GMT */
/* generated February 14, 2015 12:10:58 AM PST*/

(function(){ geolocation = {};
  ...}();
```

Listing 7: An example of script where the comments contain time related information which changes on every visit.

```
// First version of the script, requested at 2 April
"wgAvailableSkins":{"cologneblue":"CologneBlue",
"myskin":"MySkin","simple":"Simple","modern":"Modern",
"nostalgia":"Nostalgia","monobook":"MonoBook",
"standard":"Standard","chick":"Chick"}

// Second version of the script, requested at 3 April
"wgAvailableSkins":{"cologneblue":"CologneBlue",
"monobook":"MonoBook","myskin":"MySkin",
"simple":"Simple","chick":"Chick","modern":"Modern",
"nostalgia":"Nostalgia","standard":"Standard"}
```

Listing 8: An example of permutation of properties during object initialization. The underlined strings are properties which get permuted.

Permutation of properties in JavaScript Objects. A JavaScript Object is a collection of properties of the form key:value. In our experiments, a total of 53 scripts were found to differ in the ordering of object properties during initialization. This was especially observed in scripts like www.google.com/jsapi where

	# ES	# IS	UF	# AD
Unused Data	64	12	0.632	43
Side-effects of Changing JS Literals				
1) Data-JS	36	36	0.533	47
2) Data-URL	423	242	0.722	203
3) Data-HTML	113	229	0.76	65
4) Data-Cookie	50	8	0.563	21
Multi-Version Scripts	51	17	0.303	62

Table 5: Changes in dynamically-generated scripts which affect execution of the scripts (type C2).(#ES,#IS) represents the number of external and internal scripts, UF represents the update frequency and # AD the number of affected domains.

a loader function takes in an object as an argument that is a collection of modules to be loaded. An example of a script with different ordering of properties is shown in Listing 8 hosted at weibo.com. We encounter different ordering of several properties owned by an object wgAvailableSkins.

Other Syntactical Changes. In addition to previously mentioned categories, we find a few other interesting changes. In our study, we have versions of a script hosted at bol.com.br where all the double-quote characters (") around a string were replaced by single quotes (’). JavaScript offers the flexibility of optional semi-colons (;) at the end of each statement and optional brace brackets around single line conditional statements. We find 1 script where two different versions differ in semi-colons and braces for single *if* statements and 2 scripts where equality operators (==,!=) were replaced by their strict versions (===,!==) for comparing strings and boolean. All such changes bring no semantic change to the code. The number of scripts falling prey to such syntactical changes are handful, however, we aim to show that such cases are prevalent.

5.3.2 *Scripts in* C2

Next, we discuss types of changes in scripts belonging to C2, which is summarized in Table 5.

Side-effects of Changing JavaScript Literals. Since changes in C2 are data-only, we observe the side-effects of such data in the script’s execution, i.e., in what form do such data eventually get used in the scripts (sink). We classify the sinks into: HTML (Data-HTML), part of resource URL (Data-URL), stored in a DOM cookie object (Data-Cookie), or used as part of the script’s logic (Data-JS). Table 5 shows the summary of these side-effects. In our study, we observe 342 scripts where dynamic data finds sink in functions like document.write as shown in Listing 9. The periodically-changing JavaScript data also flows into sensitive JavaScript objects like URL and Cookies. As seen in Table 5, 203 websites include scripts with changing data going into the URL whereas 21 websites include scripts where changing data affect cookies. The scripts in Data-URL are a majority (665 scripts) while scripts in Data-HTML have the highest UF of 0.76 followed by Data-URL at 0.722. All the categories under side-effects have high UF (≥ 0.5) which characterizes their highly dynamic behavior.

```
1 document.write('<link rel=\"stylesheet\"
      href=\"http:\/\/staticd.cdn.industrybrains.
2     com\/css\/zonesuu\/zone796.css\"uu
          type=\"text\/css\" \/>\n')
```

Listing 9: Sample usage of periodically-changing JavaScript data used to construct HTML content.

Unused Variables. We observe a number of unused variables of the scripts which have periodically-changing values. To decide whether such changing variables are unused, we apply a semi-automatic analysis by removing them from the code and check whether such removal affects the script execution. Such changes in data happen in 76 scripts, which were embedded in a total of 43 sites.

Multi-Version Scripts. We find cases where the server returns completely different versions of the script across visits. At first, we thought that it might have been a complete API Update since the changes were so drastic. However, after subsequent visits, we observe that the changes converge into a finite set of script versions. As an example, such case happens in a script hosted at the following URL http://v6.top.rbk.ru/rbc_static/version-1.2.1/scripts/build/~_layout-main-live-tv.js. In total, we observe 68 scripts that indicate similar characteristics.

6. RELATED WORK

Structural Integrity on the Web. Several solutions have proposed different notion of structural integrity on the web. In Document Structure Integrity (DSI) [39] and Blueprint’s [34] notion of structural integrity, there is a distinction between nodes in the DOM’s abstract syntax tree which are trusted and those which are not trusted. As a result, both the solutions try to impose certain policy to confine the untrusted nodes in order to prevent code injection, either by rendering such nodes using special DOM construction techniques (implemented in Blueprint) or using taint-tracking (in DSI). In contrast to that notion, we guarantee that every node in the script’s AST is trusted and therefore technique to confine such untrusted nodes in the AST is no longer needed.

JavaScript Whitelisting. Whitelisting is a promising direction for preventing malicious script injection into a website. Jim et.al. introduce the idea of whitelisting script by embedding a policy in its pages that specifies which scripts are allowed to run in their system called BEEP [29]. Braun et.al., proposes a validation scheme to extend several HTML elements with an integrity attribute that contains a cryptographic hash of the representation of the resource the author expects to load. This scheme is named Subresource Integrity (SRI) in their latest W3C recommendation [54]. These works mainly use raw signature scheme, which validates the integrity of a resource based on the cryptographic hash computation on the resource’s source code. In SICILIAN, we introduce a new signature scheme that is robust against scripts’ changes, which are pervasive in real world websites. On the other hand, web primitive such as Content Security Proposal [53] performs *domain whitelisting*, which specifies the origin of resources but does not give any particular restriction on the integrity of the resources. The policy effort for CSP seems to be higher than SICILIAN [16].

Script Injection Prevention. A different approach for preventing script injection has been proposed by a variety of XSS defenses. The main line of research has focused on sanitizing untrusted input [5, 25, 44, 57]. However, performing only sanitization does not fully protect web applications from script injection attacks due to other client-side attack variants such as DOM-based XSS or second-order vulnerabilities [11]. Other XSS defenses include privilege separation [4, 8, 10, 15, 39], DOM isolation [3, 13, 21], and taint tracking [22, 31, 43, 50, 51]. Most of the above solutions assume the content integrity of site’s scripts. However, this may not always the case in the presence of attacks on third-party library or CDNs [32]. SICILIAN is not designed solely for preventing XSS attacks. Rather we build our solution as a *channel-agnostic* mechanism which checks any injected scripts wherever it came from.

JavaScript Measurement Study. To the best of our knowledge, there has been no study on how scripts change in popular websites that is of comparable breadth and depth to our work. We are aware of the *archive.org* project [26] which records content changes in

the internet over period of years, including JavaScript. However, it does not completely record all the web contents in Alexa's top websites nor are we aware of any study in JavaScript changes that makes use of such data. Nikiforakis et.al. study the evolution of JavaScript inclusions over time and identify the trust relationships of these sites with their library providers [40]. This study shows types of vulnerabilities that are related to unsafe third-party inclusion practices although there is no particular study on the changes reflected as well as any defenses proposed in the paper to prevent those.

7. CONCLUSION

In this paper, we conduct a longitudinal study on changes in scripts to evaluate the efficacy of signature-based JavaScript whitelisting for preventing script injection attacks. We then propose a system called SICILIAN, which 1) employs a multi-layered whitelisting approach using a novel signature scheme, structural signature, that is robust against mostly static scripts; and 2) comes with an incremental deployment model called progressive lockdown to ensure its practicality in real-world settings. Our large-scale evaluation shows that SICILIAN can whitelist scripts with reasonable performance.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers of this paper for their helpful feedback, and our shepherd Deepak Garg for his insightful comments and suggestions for preparing the final version of the paper. We thank Zhenkai Liang, Xinshu Dong, and Yaoqi Jia for their constructive feedback on the paper. This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This work is also supported in part by a university research grant from Intel.

9. REFERENCES

[1] E. Abgrall, Y. L. Traon, M. Monperrus, S. Gombault, M. Heiderich, and A. Ribault. Xss-fp: Browser fingerprinting using html parser quirks. *arXiv preprint arXiv:1211.4812*, 2012.

[2] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song. Towards a formal foundation of web security. In *Computer Security Foundations Symposium (CSF), 2010 23rd IEEE*, pages 290–304. IEEE, 2010.

[3] D. Akhawe, F. Li, W. He, P. Saxena, and D. Song. Data-confined html5 applications. In *Computer Security–ESORICS 2013*, pages 736–754. Springer, 2013.

[4] D. Akhawe, P. Saxena, and D. Song. Privilege separation in html5 applications. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 23–23. USENIX Association, 2012.

[5] 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 *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 387–401. IEEE, 2008.

[6] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side xss filters. In *Proceedings of the 19th international conference on World wide web*, pages 91–100. ACM, 2010.

[7] F. Braun, D. Akhawe, J. Weinberger, and M. West. Subresource integrity. `https://rawgithub.com/w3c/webappsec/master/specs/subresourceintegrity/index.html`.

[8] E. Budianto, Y. Jia, X. Dong, P. Saxena, and Z. Liang. You can't be me: Enabling trusted paths and user sub-origins in web browsers. In *Research in Attacks, Intrusions and Defenses*, pages 150–171. Springer, 2014.

[9] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 8–9. ACM, 2012.

[10] Y. Cao, V. Yegneswaran, P. Porras, and Y. Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *Proceedings of the 19th NDSS Symposium*, 2012.

[11] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *USENIX Security Symposium*, 2014.

[12] J. L. Donaldson, A.-M. Lancaster, and P. H. Sposato. A plagiarism detection system. In *ACM SIGCSE Bulletin*, volume 13.

[13] X. Dong, Z. Chen, H. Siadati, S. Tople, P. Saxena, and Z. Liang. Protecting sensitive web content from client-side vulnerabilities with cryptons. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1311–1324. ACM, 2013.

[14] X. Dong, K. Patil, J. Mao, and Z. Liang. A comprehensive client-side behavior model for diagnosing attacks in ajax applications. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 177–187. IEEE, 2013.

[15] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1205–1216. ACM, 2013.

[16] M. Fazzini, P. Saxena, and A. Orso. Autocsp: Automatically retrofitting csp to web applications. In *the Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015.

[17] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *ACM SIGPLAN Notices*, volume 37, pages 307–318. ACM, 2002.

[18] Google. Content security policy (csp). `https://goo.gl/Y7u2ee`.

[19] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for ajax intrusion detection. In *Proceedings of the 18th international conference on World wide web*, pages 561–570. ACM, 2009.

[20] M. V. Gundy and H. Chen. Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In *Proceedings of the 16th Network and Distributed System Security Symposium Symposium*, 2009.

[21] W. He, D. Akhawe, S. Jain, E. Shi, and D. Song. Shadowcrypt: Encrypted web applications for everyone. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1028–1039. ACM, 2014.

[22] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671. ACM, 2014.

[23] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 777–788. ACM, 2013.

[24] A. Hidayat. `http://esprima.org`.

[25] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX conference on Security*. USENIX Association, 2011.

[26] Internet Archive. `https://archive.org/index.php`.

[27] C. Jackson and A. Barth. Forcehttps: protecting high-security web sites from network attacks. In *Proceedings of the 17th international conference on World Wide Web*, pages 525–534. ACM, 2008.

[28] Jeremiah Grossman and Matt Johansen. `https://goo.gl/kwgWPm`.

[29] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 601–610. ACM, 2007.

[30] jQuery. Update on jquery.com compromises. `http://goo.gl/uFcPKM/`.

[31] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1193–1204. ACM, 2013.

[32] A. Levy, H. Corrigan-Gibbs, and D. Boneh. Stickler: Defending against malicious cdns in an unmodified browser. 2015.

[33] R. Lipton. Fingerprinting sets. `http://goo.gl/tx7pWq`.

[34] M. T. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 331–346. IEEE, 2009.

[35] R. C. Merkle. A certified digital signature. In *CRYPTO*, 1989.

[36] Mozilla. `https://developer.mozilla.org/en-US/docs/Web/XPath`.

[37] Mozilla. Http strict transport security. https://developer.mozilla.org/en-US/docs/Web/Security/HTTP_strict_transport_security/.

[38] Mozilla. Signing a xpi. https://goo.gl/Ffls5r.

[39] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[40] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: large-scale evaluation of remote javascript inclusions. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[41] OWASP. Xss filter evasion cheat sheet. `https://goo.gl/Iq60U0`.

[42] P. Rogaway and T. Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *Fast Software Encryption*, pages 371–388. Springer, 2004.

[43] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[44] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[45] Security Affairs. Afghanistan cdn network compromised by chinese hackers. `http://goo.gl/Kh8zqN`.

[46] SicilianCCS15. Chromium patches for sicilian. https://github.com/sicilianccs15/sicilian.

[47] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 463–478. IEEE, 2010.

[48] Softpedia. Exploit kit dropped through akamai content delivery network. `http://goo.gl/1UgGgT`.

[49] S. Son and V. Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *Proceedings of the 20th Network and Distributed System Security Symposium*, 2013.

[50] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against dom-based cross-site scripting. In *Proceedings of the 23rd USENIX security symposium*.

[51] 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 *Proceedings of the 14th Network and Distributed System Security Symposium*, 2007.

[52] W3C. All standards and drafts. http://www.w3.org/TR/.

[53] W3C. Content security policy 2.0. `http://www.w3.org/TR/CSP2/`.

[54] W3C. Subresource integrity. `http://www.w3.org/TR/SRI/`.

[55] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[56] J. Wang, Y. Takata, and H. Seki. Hbac: A model for history-based access control and its model checking. In *Computer Security–ESORICS 2006*, pages 263–278. Springer, 2006.

[57] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A systematic analysis of xss sanitization in web application frameworks. In *ESORICS*, 2011.

[58] M. Weissbacher, T. Lauinger, and W. Robertson. Why is csp failing? trends and challenges in csp adoption. In *Proceedings of the 17th Research in Attacks, Intrusions and Defenses*, 2014.

[59] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving ssh-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.

[60] Z. Yan and S. Holtmanns. Trust modeling and management: from social trust to digital trust. *IGI Global*, 2008.