# SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications

Prateek Saxena
UC Berkeley
Berkeley, CA
prateeks@cs.berkeley.edu

David Molnar
Microsoft Research
Redmond, WA
dmolnar@microsoft.com

Benjamin Livshits
Microsoft Research
Redmond, WA
livshits@microsoft.com

## ABSTRACT

*We empirically analyzed sanitizer use in a shipping web application with over 400,000 lines of code and over 23,244 methods, the largest empirical analysis of sanitizer use of which we are aware. Our analysis reveals two novel classes of errors:* context-mismatched sanitization *and* inconsistent multiple sanitization*. Both of these arise* not *because sanitizers are incorrectly implemented, but rather because they are not placed in code correctly. Much of the work on cross-site scripting detection to date has focused on finding missing sanitizers in programs of average size. In large legacy applications, other sanitization issues leading to cross-site scripting emerge.*

*To address these errors, we propose* SCRIPTGARD*, a system for ASP.NET applications which can detect and repair the incorrect placement of sanitizers.* SCRIPTGARD *serves both as a testing aid to developers as well as a runtime mitigation technique. While mitigations for cross site scripting attacks have seen intense prior research, we consider both server and browser context, none of them achieve the same degree of precision, and many other mitigation techniques require major changes to server side code or to browsers. Our approach, in contrast, can be incrementally retrofitted to legacy systems with no changes to the source code and no browser changes. With our optimizations, when used for mitigation,* SCRIPTGARD *incurs virtually no statistically significant overhead.*

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*invasive software*;
D.1.2 [**Programming Techniques**]: Automatic Programming—*program transformation, program modification*

## General Terms

Security, Languages, Vulnerabilities


*CCS'11,* October 17–27, 2011, Chicago, Illinois, USA.


## Keywords

Cross-site scripting, Runtime analysis, Web applications

## 1. INTRODUCTION

Web applications are explosively popular, but they suffer from cross-site scripting (XSS) [4, 32] and cross-channel scripting (XCS) [5]. At the core of these attacks is injection of JavaScript code into a context not originally intended. These attacks lead to stolen credentials and actions performed on the user's behalf by an adversary. Ideally, we would create systems that resist attacks by construction. Recent projects such as BLUEPRINT have proposed primitives to encode HTML output in a safe way [20]. Unfortunately, these techniques are difficult to apply to legacy web applications because they make fundamental changes to the way an application creates HTML for consumption by the web browser. Mitigations are needed for XSS attacks against web applications that can be incrementally retrofitted to existing code.

**Prior work:** Much work in this space has focused on missing sanitizers and was performed on relatively small applications. The effort described in this paper goes well beyond that, specifically focusing on applications approaching half a million lines of code and above. We empirically analyze sanitization in a widely used Web application with over 400,000 lines of code, the largest application analysis of which we are aware. In this process, we discovered two novel classes of errors: *context-mismatched sanitization* errors and *inconsistent multiple sanitization* errors. These sanitization errors arise from subtle nesting of HTML contexts and sharing of dataflow paths in the application. Unlike previously reported errors, mismatched sanitization is not due to faulty or missing sanitizers; each individual sanitizer in our test application is *correct* to the best of our knowledge. The new categories of errors only became apparent because of the scale and complexity of the applications we target, however, we believe that they represent a new frontier of complexity, shared by other applications of a similar scale.

As a point of comparison, it is instructive to consider HTML templating systems such as Google GWT or CTemplates, which employs Google AutoEscape's auto-sanitization to remove the need for manual sanitization [10]. Such systems are much easier to sanitize as compared to large-scale *legacy* code that we focus on. For instance, developers are forced to separate untrusted variables from HTML structure explicitly in templates, unlike in legacy code. Their mechanisms does not need to handle the nesting of HTML contexts and sharing of dataflow paths, because

they work with specific frameworks.

**Consistent Sanitizer Placement:** Context-mismatched sanitization errors arise because developers accidentally mismatch the choice of sanitizer with the web *browser context*, the browser's parsing state when processing the sanitized data. This happens when the browser's actual parsing state or context is different from what the developer expects, allowing unintended characters like single quotes in sanitized data. Moreover, we discover inconsistent multiple sanitization errors, in which the *combination* of sanitizers leads to problems. For example, we find that two sanitizers in our test application are not commutative: the order of application matters, only one order is safe, yet both orders appear in our empirical study.

We propose the problem of *consistent sanitizer placement*: apply a sequence of sanitizers, chosen from a pre-specified set, to an untrusted input such that it is safe for all the possible browser contexts in which it is rendered. Sanitization is also needed for cases where data must be encoded to avoid causing functionality errors in a specific context, such as encoding URLs in JavaScript. Prior work does not model the notion of browser contexts precisely, rendering it unable to detect these classes of errors. Depending on the configuration/policy of the application and the authority of the adversary who controls the sanitized data, these inconsistencies may or may not yield cross site scripting attacks, but these remain errors in sanitization practice.

**ScriptGard:** We develop ScriptGard, a system for detecting these sanitization errors, and repairing them by automatically choosing the appropriate sanitizer. Our system requires no changes to web browsers or to server side source code. Instead, we use binary rewriting of server code to embed a *browser model* that determines the appropriate browser parsing context when HTML is output by the web application. In contrast, projects such as Blueprint have proposed primitives to encode HTML output in a safe way [20], but at the cost of requiring large scale server code changes.

Unlike existing template-based HTML writing systems, such as ASP.NET's web and HTML controls, ScriptGard performs *context-sensitive* sanitization. ScriptGard allows developers to create custom nesting of HTML contexts, which we show in Section 6 is common in our test application, without sacrificing the consistency of the sanitization process. During analysis, ScriptGard employs *positive taint-tracking*, which contrasts with traditional taint-tracking because it is conservative (hence does not miss identifying sources of untrusted data) and can provide defense against cross channel-scripting attacks [5, 13]. For example, a recent vulnerability in Google Analytics was due to a non-traditional source of untrusted input, namely event logs [30], which our approach would detect.

We implement our analysis in ScriptGard, which uses binary rewriting techniques to instrument applications running on the ASP.NET platform. We stress, however, that the analyses we perform are general and could be ported to other platforms with sufficient engineering work. ScriptGard can be used either as a testing aid or as a runtime mitigation. As a testing aid, ScriptGard points out sanitizers that are not correct for the runtime parsing context. Our dynamic technique ensures that these reports are for specific, reproducible test cases that exhibit inconsistent sanitization.

As a runtime mitigation, we show how ScriptGard can leverage a *training phase* where it runs with full instrumentation on a target application to learn correct sanitizers for different program paths. Then in deployment, only a lighter path detection instrumentation is necessary. We adapt techniques from preferential path profiling [7]; with this optimization trick, ScriptGard incurs virtually no statistically significant overhead when auto-correcting inconsistently sanitized paths we tested.

## 1.1 Contributions

This paper makes the following contributions.

**Testing for sanitizer placement errors:** In Section 2, we identify two new classes of errors: *context-mismatched sanitization* and *inconsistent multiple sanitization*. We implement a novel analysis that combines automatic server-side instrumentation with a browser model to find inconsistent sanitization. We further refine this analysis with *positive taint tracking* to conservatively over-estimate potentially adversarial inputs, which has been applied to SQL injection in the past, but not to cross site scripting errors [13]. Because we use both server and browser modeling, we find potential sanitization flaws that would be missed with techniques that focus exclusively on the server [2, 17, 21].

**Runtime auto-sanitization:** We show how our analysis can determine the correct sanitization at runtime. While we show the cost to run the full ScriptGard instrumentation is a high multiple of the original run time, we propose a mechanism for using preferential path profiling techniques [7] to shift most of this cost to a pre-deployment *training phase*. With this optimization trick, deployed auto-correction incurs virtually negligible overhead. Our system changes only the server side runtime, requiring no changes to server code or to the browser.

**Evaluation and empirical study:** In Section 6 we evaluate both the testing approach as well as runtime auto-sanitization on an application with over 400,000 lines of code. We performed our security testing on a set of 53 large web pages derived from 7 sub-applications built on top of our test application. Each page contains 350–900 DOM nodes. Out of 25,209 total paths exercised, we found context-mismatched sanitization on 1,207 paths ScriptGard analyzed, 4.7% of the total paths analyzed. We also observed an additional 285 instances of inconsistent multiple sanitization errors. Our runtime mitigation technique safeguards against these inconsistent uses of sanitizers by automatically correcting them at runtime.

## 1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 gives a description of the context-sensitive vulnerabilities ScriptGard is designed to prevent. Section 3 provides an overview of the ScriptGard architecture. Section 4 formalizes the vulnerabilities and provides a notion of correctness. Section 5 provides specific details of ScriptGard implementation. Section 6 gives an experimental evaluation of our techniques. Finally, Sections 7 and 8 describe related work and conclude.

# 2. SANITIZER CONSISTENCY

In this section, we systematically explain the two new class of sanitization errors we commonly observe in our empirical analysis: *context-mismatched sanitization* and *inconsistent*

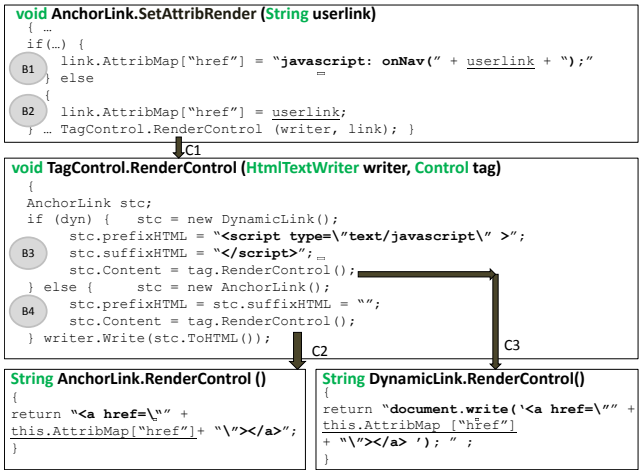



**Figure 1:** Running example: C# code fragment illustrating the problem of automatic sanitizer placement. Underlined values are derived from untrusted data and require sanitization; function calls are shown with thick black arrows C1-C3 and basic blocks B1-B4 are shown in gray circles.

*multiple sanitization*, both of which demonstrate that placement of sanitizers in legacy code is a significant challenge even if the sanitizers themselves are securely constructed.

It is well-known that script-injection attack vectors are highly *context*-dependent — a string such as `expression: alert(a)` is innocuous when placed inside a HTML tag context, but can result in JavaScript execution when embedded in a CSS attribute value context. In fact, the set of contexts in which untrusted data is commonly embedded by today's web applications is well-known. Sanitizers that secure data against XSS in each of these contexts are publicly available [10, 27]. We assume that contexts and their corresponding functionally correct sanitizers, shown in Figure 7, are standard and available.

## 2.1 Inconsistent Multiple Sanitization

Figure 1 shows a fragment of ASP.NET code written in C# which illustrates the difficulty of sanitizer placement. This running example is inspired by code from the large code base we empirically analyzed. Consider the function `DynamicLink.RenderControl` shown in the running example, which places an untrusted string inside a double-quoted `href` attribute which in turn is placed inside a JavaScript string. This code fragment places the untrusted string into two *nested contexts* — when the browser parses the untrusted string, it will first parse it in the JavaScript string literal and then subsequently parse it as a URI attribute.

In Figure 7 we show a sanitizer specification that maps functions to contexts. In particular, two sanitizer functions, `EcmaScriptStringEncode` and `HtmlAttribEncode`, are applied for the JavaScript string context and the HTML attribute context, respectively. However, developers must understand this order of parsing in the browser to apply them in the correct order. They must choose between the two ways of composing the two sanitizers shown in Figure 2.

It turns out that the first sequence of sanitizer composition is *inconsistent* with the nested embedding contexts, while the other order is safe or consistent. We observe that the standard recommended implementation for these sanitizers [27], even in widely deployed public web frameworks and libraries (Django, GWT, OWASP, .NET) do not commute. For instance, `EcmaScriptStringEncode` simply transforms all characters that can break out of JavaScript string literals (like the " character) to Unicode encoding (`\u0022` for "), and, `HtmlAttribEncode` HTML-entity encodes characters (`&quot;` for "). This is the standard recommended behavior these sanitizers [27] with respect to respective contexts they secure.

```
document.write('<a href=" +
HtmlAttribEncode(EcmaScriptStringEncode(this.AttribMap["href"]))
+...
```

(a) Method 1

```
document.write('<a href=" +
EcmaScriptStringEncode(HtmlAttribEncode(this.AttribMap["href"]))
+...
```

(b) Method 2

**Figure 2:** Different sanitization approaches.

**Example 1.** The attack on the wrong composition is very simple. The key observation is that applying `EcmaScriptStringEncode` first encodes the attacker-supplied " character as a Unicode representation `\u0022`. This Unicode representation is not subsequently transformed by the second `HtmlAttribEncode` sanitization, because `\u0022` is a completely innocous string in the URI attribute value context.

However, when the web browser parses the transformed string first (in the JavaScript string literal context), it performs a Unicode decode of the dangerous character `\u0022` back to ". When the browser subsequently interprets this in the `href` URI attribute context the attacker's dangerous " prematurely closes the URI attribute value and can inject JavaScript event handlers like `onclick=...` to execute malicious code. This may also enable parameter pollution attacks in URLs [1]. It is easy to confirm that the other composition of correct sanitizers is definitely safe. ■

## 2.2 Context-Mismatched Sanitization

Even if sanitizers are designed to be commutative, developers may apply a sanitizer that does not match the context altogether; we call such an error as a *context-mismatched sanitization* inconsistency. Context-mismatched sanitization is not uncommon in real applications. To intuitively understand why, consider the sanitization requirements of the running example again.

Notice that the running example has 4 control-flow paths corresponding to execution through the basic-blocks (B1,B3), (B1,B4), (B2,B3) and (B2,B4) respectively. Each execution path places the untrusted `userlink` input string in 4 different contexts (see Figure 3). Determining the browser context is a path-sensitive property, and the developer may have to inspect the global control/data flow to understand in which contexts is a data variable used. This task can be error-prone because the code logic for web output may be spread across several classes, and the control-flow graph may not be explicit (especially in languages with inheritance). We show how two most prevalent sanitization strategies fail to work on this example.

**Failure of output sanitization:** Consider the case in which the application developer decides to delay all sanitization to the *output* sinks, i.e., to the `writer.Write` call in

| HTML output | Nesting of contexts |
|---|---|
| `<script type="text/javascript">`<br>`  document.write(‘`<br>`  <a href="javascript: onNav(`<br>`TOENCODE )";></a>);`<br>`</script>` | JavaScript String Literal, Html URI Attribute, JavaScript Number |
| `<a href="javascript: onNav(`<br>`TOENCODE );"></a>` | Html URI Attribute, JavaScript Number |
| `<a href="TOENCODE"></a>` | Html URI Attribute |
| `<script type="text/javascript">`<br>`  document.write(‘ <a href="`<br>`TOENCODE"> </a>’);`<br>`</script>` | JavaScript String Literal, Html URI Attribute |

**Figure 3:** HTML outputs obtained by executing different paths in the running example. TOENCODE denotes the untrusted string in the output.

`TagControl.RenderControl`. There are two problems with doing so, which the developer is burdened to identify manually to get the sanitization right. First, the execution paths through basic-block B3 embed the untrusted data in a `<SCRIPT>` block context, where paths through basic-block B4 place it in a HTML tag context. As a result, *any* sanitizer picked cannot be consistent for both such paths. Second, even if the first concern did not exist, sanitizing the `stc.Content` variable at the output point is *not* correct. The `stc.Content` is composed of trusted substrings as well as untrusted data — if the entire string is sanitized, the sanitizer could change programmer-supplied constant strings in a way that breaks the intended structure of the output HTML. For example, if the basic-block B1 executes, the untrusted data would be embedded in a JavaScript number context( `javascript: OnNav()` explicitly by the programmer. If we applied `HtmlAttribEncode` to the `stc.Content` the `javascript:` would be eliminated breaking the application's intended behavior.

**Failure of input sanitization:** Moving sanitization checks to earlier points in the code, say at the input interfaces, is not a panacea either. The readers can verify that moving all sanitization to a code locations earlier in the dataflow graph continues to suffer from path-sensitivity issues. Sanitizing in basic-blocks B1 and B2 is not sufficient, because additional contexts are introduced when blocks B3 and B4 are executed. Sanitization locations midway in the dataflow chain, such the concatenation in function `AnchorLink.SetAttribRender`, are also problematic because depending on whether basic-block B1 executes or B2 executes, the `this.AttribMap["href"]` variable may have trusted content or not.

## 2.3 Why is Consistency Hard?

Expressive languages, such as those of the .NET platform, permit the use of string operations to construct HTML output as strings with trusted code intermixed with untrusted data. Plus, these rich programming languages allow developers to write complex dataflow and control flow logic. We summarize the following observations that exist in large legacy applications authored on such rich programming environments:

- **String outputs:** String outputs contain trusted constant code fragments mixed with untrusted data.
- **Nested contexts:** Untrusted data is often embedded in nested contexts.
- **Intersecting data-flow paths:** Data variables are used in conflicting or mismatched contexts along two or more intersecting data-flow paths.
- **Custom Output Controls:** Frameworks such as .NET encourage reusing output rendering code by providing built-in "controls", which are classes that render untrusted inputs in HTML codes. Large applications extensively define custom controls, perhaps because they find the built-in controls insufficient. The running example is typical of such real-world applications — it defines its own custom controls, `DynamicLink`, to render user-specified links via JavaScript.

In this paper, we empirically analyze the extent to which these inconsistency errors arise in practical real-world code. In security-conscious applications, such as the ones we study, security audits are routine and use of lint tools can enforce simple discipline. In particular, notice that running example code is careful to restrict the browser context in which data is allowed to be embedded. For instance, it rigorously appends quotes to each attribute value, as recommended by security guidelines [27].

We point out that state-of-the-art static analysis tools which scale to hundred-thousand LOC applications, presently are fairly limited. Most existing tools detect dataflow paths with sanitizers missing altogether. This class of errors is detected by several static or dynamic analysis tools (such as CAT.NET [22] or Fortify [9]). In this paper, we focus instead on the errors where sanitization is present but is inconsistent.

# 3. SCRIPTGARD APPROACH

ScriptGard employs a dynamic analysis approach to detecting and auto-correcting context-inconsistency errors in sanitization. At a high-level, the overall architecture for ScriptGard is shown in Figure 4. ScriptGard has two main components: (a) a training or analysis phase, and (b) a runtime auto-correction component. In the analysis step, ScriptGard traces the dynamic execution of the application on test inputs. By tracking the flow of trusted values, ScriptGard can identify all untrusted data embedded in the application's output. Given a trace of an application's execution, ScriptGard is capable of mapping the trace to a static program path. ScriptGard is able to determine the correct sequence of sanitizers that should be applied on this program path, by parsing the application's output.

The results of our analysis phase are cached in a *sanitization cache*, which records all execution paths that were seen to have context-inconsistency sanitization errors. This cache serves as a basis for runtime auto-correction of the application during deployed operation. Our intuition is that context-inconsistency arises in a relatively small fraction of the application's code paths. ScriptGard's architecture repairs the placement of sanitizers only on these execution paths by using a light-weight instrumentation. The auto-correction component deploys a low-overhead path-detection technique that detects when these problematic paths are executed at runtime and applies the correct sanitization to untrusted values. The primary motivation for this architecture is to enable separation of expensive analysis to be performed prior to deployment, leaving only low-overhead

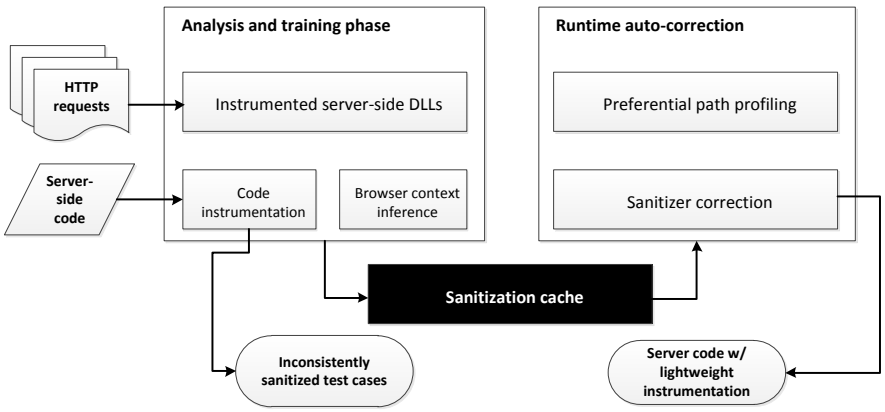



**Figure 4:** ScriptGard architecture.

components enabled at runtime. Another key feature of our auto-correction component is that it requires no updates or changes to the application code, thereby avoiding a long develop-review-patch development cycle.

**Requirements:** ScriptGard requires a map between browser contexts and sanitizer functions appropriate for those contexts. In practice this correspondence is specified by security architects or other experts and can be done once and for all.

Figure 7 shows the example sanitization specification for the running example as well as the applications we study. The sanitizer `SimpleHTMLFormatting` transforms the input such that its output can only contain certain permitted HTML tags such as `<b>`, `<a>` and so on, with the goal of preventing a browser from executing any JavaScript from the resulting string. In contrast, the sanitizer `EcmaScriptStringEncode` takes JavaScript literals and converts them to Unicode. Such conversion is necessary because the JavaScript parser converts Unicode to some other representation for string data. A similar function `UrlPathEncode` performs percent-encoding. This percent encoding is required because the URL parser will decode URLs on entry to the parser.

Reasoning about the localized correctness and completeness properties of the context-sanitizer mapping is an independent problem of interest; techniques for such correctness checking are an area of active research [2, 15, 17]. In this work, we assume the functional completeness and correctness for the specifications.

## 3.1 Training Phase

ScriptGard employs a dynamic analysis which treats each executed path as a sequence of *traces*. Each trace is conceptually a sequence of dataflow computation operations that end in a write to the HTTP stream or an *output sink*. As we highlighted in Section 2, we must consider traces because sanitizer placement is *path-sensitive*. ScriptGard's dynamic analysis checks if the sanitization applied on any untrusted trace is correct. For each untrusted trace observed during program execution, ScriptGard first determines a mapping for each program trace $\vec{t}$ to a sequence of sanitizer functions, $f_1, f_2, \ldots, f_k$, to apply, and second the portion of the output string that should be sanitized.

We call the first step *context inference*. The second step is achieved by a technique called *positive taint-tracking*. If the sequence of sanitizers applied on a trace does not match the inferred sequence, ScriptGard discovers a violating path and it adds the corrected sanitizer sequence for this path to the sanitization cache.

**Positive Tainting:** We have discussed untrusted execution traces in the abstract, but we have not talked about how ScriptGard determines which traces are untrusted and, therefore, need to be sanitized. Exhaustively identifying all the sources of untrusted data can be challenging [19]. Recent work has shown that failure to identify non-web related channels, such as data read from the file system, results in cross-channel scripting attacks [5].

Instead of risking an incomplete specification, which would miss potential vulnerabilities, ScriptGard takes a conservative approach to identifying untrusted data: it employs *positive tainting*, which is a modification of traditional (or negative tainting) used in several previous systems [12, 18, 26, 31, 35, 36]. Instead of tracking untrusted (negative) data as it propagates in the program, we track all safe data. Positive tainting is conservative because if input specifications are incomplete, unknown sources of data are treated as unsafe by default. We describe details of these two steps in Section 5.

## 3.2 Runtime Auto-Correction

From the training phase, ScriptGard builds a *sanitization cache*, which is a map between code paths and the correct sequence of sanitizers to apply for the browser context reached by that code path. Then at runtime, ScriptGard detects which path is actually executed by the program. If the path has been seen in the training phase, then ScriptGard can look up and apply the correct sanitizer sequence from the cache, obviating the need for the full taint flow instrumentation.

If the path has not been seen in the training phase, then the owner of the web site has a policy choice. One choice is to drop the request, then immediately re-execute the application with all ScriptGard checks enabled. This is secure but adds signficant user latency. The other choice is to allow the request to complete, but then log the path taken for later analysis. This is not secure but preserves the per-

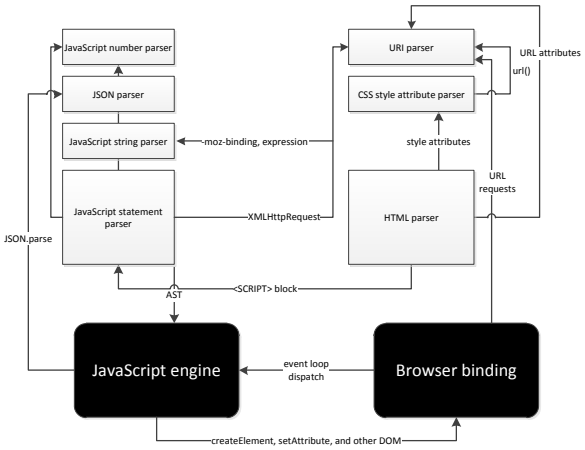



**Figure 5:** An abstract model of an HTML 5-compliant Web browser. Gray boxes represent various parsers for the browser sub-grammars. Black boxes are the major browser execution components.

formance of the application. We leave this choice up to the application administrator.

The common overhead for runtime in both policy choices comes from the cost to detect the path executed by the application. We leverage a technique called *preferential path profiling*, as first used in the Holmes statistical debugging project [7]. Preferential path profiling is a technique for adding lightweight instrumentation to a program to detect if the program executes one of a set of "preferred" paths known in advance of deployment. The technique uses an optimized way of labeling code blocks and updating state during execution to reduce overhead. We report overhead on our application in Section 5.

# 4. FORMALIZATION

In this section, we formalize our approach and the security properties we aim to achieve. We start with an abstract model of the browser. This allows us to define precisely what we mean by a *browser parsing context*. The notion of a context is closely tied to sanitizers that are used, as discussed previously in Section 2. For example, `HtmlAttributeEncode` will properly escape strings in the HTML attribute context, but it is inconsistent to use in other contexts. That in turn allows us to precisely characterize context-sanitizer mismatches. We then define what it means for a server-side program to prevent all such context-sanitizer mismatches. Finally, we show that our strategy in ScriptGard in fact transforms programs into ones that ensure dynamically that no context-sanitizer mismatches are possible.

## 4.1 Browser Model: Definitions

We begin with a browser model as illustrated in Figure 5. For our purposes, we model a web browser as a parser consisting of sub-parsers for several languages. Of course, a real browser has a great deal of implementation issues and side effects to grapple with, but these are out scope of the problems we consider here.

More precisely, we treat the browser as a *collection* of parsers for different HTML standard-supported languages. Figure 5 shows the sub-grammars corresponding to the HTML language, JavaScript language, and the languages for web addresses and inline style attributes.

Because inconsistent application behavior may depend on context that are more fine-grained that regular HTML or JavaScript parsing, we can further divide each sub-grammar into partitions. For instance, the figure shows the JavaScript grammar further subdivided into the string literal sub-grammar, JSON sub-grammar, statement sub-grammar and the number sub-grammar. Formally, we model the browser as a composition of multiple sub-grammars.

**Definition 1.** *Let $G_1, G_2, ...G_n$ be n sub-grammars, where each context-free grammar $G_i = (V_i, \Sigma_i, S_i, P_i)$ is a quadruple consisting of a set of non-terminals $V_i$, terminals $\Sigma_i$ , start symbol $S_i$ and productions $P_i$.*

*Let $\mathcal{T}$ be a set of grammar* cross-grammar transition symbols *and the grammar transition productions $P_\mathcal{T}$, be a set of productions of the form $A \rightarrow T_i$ or $T_i \rightarrow B$, such that $A \in V_i$, $B \in V_j$ $(i \neq j)$ and $T_i \in \mathcal{T}$.*

*We define a web browser as a grammar $\mathcal{G} = \{V, \Sigma, \mathcal{S}, P\}$, with non-terminals $V = V_1 \cup V_2 ... \cup V_n$, terminals $\Sigma = \cup\Sigma_i$, start symbol $\mathcal{S}$ and a set of productions $P = P_\mathcal{T} \cup P_1 \cup P_2 ... P_n$.*

Conceptually, parsers for various languages are invoked in stages. After each sub-parser invocation, if a portion of the input HTML document is recognized to belong to another sub-language, that portion of the input is sent to the appropriate sub-language parser in the next stage. As a result, any portion of the input HTML document may be recognized by one or more sub-grammars. Transitions from one sub-grammar to another are restricted through productions involving special transition symbols defined above as $\mathcal{T}$, which is key for our formalization of context. In a real web browser, each transition from one sub-grammar to another may be accompanied by a one or more transduction steps of the recognized input.

**Example 2.** For instance, data recognized as a JavaScript string is subject to Unicode decoding before being passed to the AST. In addition, HTML 5-compliant browsers subject data recognized as a URI to percent-encoding of certain characters before it is sent to the URI parser [8].

This form of encoding can be modeled using additional rules in either of the sub-grammars. While restricting the browser formalism to a context-free grammar might elide some of the real-world complexities, we find this to be a convenient way for defining the notion of context, which appears to match the reality quite well. ■

**Browser Parsing Contexts:** We formally define the notion of a *browser parsing context* here, with reference to the grammar $\mathcal{G}$. Intuitively, a context reflects the state of the browser at a given point reading a particular piece of input HTML. Each step in the derivation, denoted by $\Rightarrow$ applies a production rule and yields a "sentential form", i.e., a sequence consisting of non-terminals and terminals. We model the parsing context as a sequence of transitions made by the parser between the sub-grammars in $\mathcal{G}$, only allowing the derivations that denote transition from one sub-grammar to another.

**Definition 2.** *Let derivation $D : \mathcal{S} \Rightarrow^* \gamma$ correspond to the sequence $(P_1, P_2, ...P_k)$ of production rule applications to derive a sentential form $\gamma$ from the start symbol $\mathcal{S}$.*

*A browser context $\mathcal{C}_\mathcal{D}$ induced by a derivation $D$ is defined as a projection $(P_1, P_2, \ldots P_k) \rightarrow_\downarrow (P'_1, P'_2 \ldots P'_l)$, that preserves only the productions $P'_i$ in the set of grammar transitions $P_\mathcal{T}$.*

Our grammars are deterministic, so the notion of inducing a parsing context is well-defined. The browser enters a particular context as a result of processing a portion of the input.

**Definition 3.** *We say that an input $I$ induces browser context $\mathcal{C}$, if*

- *$D : S(I) \Rightarrow^* \gamma$ (on input $I$, $\mathcal{S}$ reduces to $\gamma$ following derivation $\mathcal{D}$), and*
- *$\mathcal{D}$ induces context $C$.*

**Sanitizers:** A complex modern web application typically has a variety of both server- and client-side *sanitizers*. We make the simplifying assumption that sanitizers are *pure*, i.e. lacking side-effects. Our review of dozens of real-life sanitizers confirms this assumption. We model sanitizers as abstract functions on strings. Formally,

**Definition 4.** *A* sanitizer *is a function $f : string \mapsto string$.*

**Definition 5.** *A context-sanitizer map is*

$$\psi(\mathcal{C}) = \vec{f}$$

*where $\mathcal{C}$ is a context and $\vec{f}$ is a sequence of sanitizers.*

The goal of sanitization is typically to remove special characters that would lead to a sub-grammar transition. For example, we often do not want a transition from the HTML parsing context to the JavaScript parsing context, which would be enabled by inserting a `<SCRIPT>` block in the middle of otherwise non-offending HTML. Of course, this is but one of many ways that the parser can transition to a different context. Next, we define the *correctness* of a sequence of sanitizers. The intuition is that is after sanitization, the state of parsing is confined to a single context.

**Definition 6.** *Let input $\mathcal{I}$ consist of the* parsed *and* non-parsed *portion: $\mathcal{I} = [\mathcal{I}_P \circ \mathcal{I}_{NP}]$. Let input $\mathcal{I}_P$ induce browser context $\mathcal{C}$ such that $\psi(\mathcal{C}) = \vec{f}$. Then we say that the context-sanitizer map is* correct *if when $\vec{f}(\mathcal{I}_{NP})$ is reduced, the grammar never leaves context $\mathcal{C}$.*

In other words, applying the correct sequence of sanitizers "locks" a string in the current context. In particular, a string locked in the HTML context cannot cause the browser to transition to the JavaScript context, leading to a code injection.

## 4.2 Server-side Program: Definitions

So far, our discussion has focused on parsing HTML strings within the browser regardless of their source. Our goal is to produce HTML on the server that will always have consistent sanitization. Server-side programs take both untrusted and trusted inputs. Untrusted inputs are the well-known sources of injection possibilities such as HTML form fields, HTML headers, query strings, cookies, etc. Trusted inputs are often read from configuration files, trusted databases, etc. Note that the notion of what is trusted and what is not is often not clear-cut. Section 3.1 describes how ScriptGard addresses this problem. Next, we define what it means for a program to properly sanitize its inputs.

**Definition 7.** *A server-side program $\mathcal{P} : \mathcal{I} \rightarrow \mathcal{O}$ defines a relation from untrusted user inputs $I$ to output string $\mathcal{O}$. The program interprocedural data flow graph is a graph $\langle \mathcal{N}, \mathcal{E} \rangle$ with designated sets of nodes*

$$\langle Src, Snk, San \rangle \subseteq \langle \mathcal{N} \times \mathcal{N} \times \mathcal{N} \rangle$$

*where Src are the sources that introduce an untrusted inputs in $\mathcal{P}$, Snk are the sinks that write strings to the output HTTP stream, and San are the sanitizers used by the program.*

Without loss of generality, we assume that sink nodes either write untrusted strings to the output stream or trusted strings, but never strings containing both. Sink operations with mixed content can be translated to an equivalent dataflow graph with only exclusively trusted or untrusted sink nodes using sink node splitting: the output of a mixed string $Snk(q_1 + q_2 + \cdots + r_1 + \ldots q_n)$ can be split into a sequence of exclusive sink writes $Snk(q_1)$, $Snk(q_2) \ldots, Snk(r), \ldots, Snk(q_n)$.

**Definition 8.** *An* untrusted execution trace *$t$ of program $\mathcal{P}$ is a sequence of executed nodes*

$$\vec{t} = n_1, \ldots, n_k \in \mathcal{N}$$

*such that $n_1 \in Src$, $n_k \in Snk$.*

**Definition 9.** *Let $t$ be an untrusted execution trace $\vec{t} = n_1 \ldots n_k$ and let $\vec{f} = f_1, \ldots, f_m$ be a sequence of sanitizers such that $f_1, \ldots, f_m$ is a subsequence of $n_2, \ldots, n_{k-1}$.*

*For all inputs $\mathcal{I}$, let $O$ be the total output string just before the execution of the sink node in $\vec{t}$. We say that trace $\vec{t}$ is* properly sanitized *if $\mathcal{O}$ induces context $\mathcal{C}$ and $\psi(\mathcal{C}) = \vec{f}$.*

In other words, for all possible trace executions, we require that the proper set of sanitizers be applied on trace for the expected parsing context. Note that *trusted* traces are allowed to change the browser context. A common example of that is

```
output.WriteLine("<SCRIPT>");
output.WriteLine("alert('hi');");
output.WriteLine("</SCRIPT>");
```

where each string is a sink and the first and third lines correspond to browser state transitions.

**Theorem 1.** *If untrusted trace $\vec{t}$ is properly sanitized, assume the browser has read string $\mathcal{O}$ which induces context $\mathcal{C}$. Then reading the rest of the string output produced by $\vec{t}$ cannot induce any contexts $\mathcal{C}' \neq \mathcal{C}$.*

**Proof:** *Let input $\mathcal{I} = [\mathcal{I}_P \circ \mathcal{I}_{NP}]$. By Definition 9, for all input-output pairs $\mathcal{I}_P \rightarrow \mathcal{O}$, $\vec{t}$ contains sanitizers $\vec{f}$ correct for any context $\mathcal{C}$ inducible by $\mathcal{O}$. By Definition 6, we know that applying $\vec{f}$ to the remainder of the input $\mathcal{I}_{\mathcal{NP}}$ cannot leave context $\mathcal{C}$.*

For reasons of correctness, we wish to ensure that all untrusted execution traces are properly sanitized.

**Definition 10.** *A server-side program $\mathcal{P}$ is* properly sanitized *if for every untrusted execution trace $\vec{t}$ of $\mathcal{P}$, $\vec{t}$ is properly sanitized.*

As an obvious corollary, if the program is properly sanitized, then no untrusted input to the server program can force the browser to change its context.

# 5. IMPLEMENTATION DETAILS

We now describe our implementation in more detail, with reference to the ASP.NET framework. This section first describes positive taint tracking implemented in ScriptGard in Section 5.1, and then context inference and autocorrecting runtime sanitization in Sections 5.2 and 5.3.

## 5.1 Positive Taint Tracking

We describe our full implementation for positive taint tracking for strings in the .NET platform. The .NET runtime supports two kinds of string objects: *mutable* and *immutable* objects. Immutable objects, instances of the `System.String` class, are called so because their value cannot be modified once it has been created [23]. Methods that appear to modify a `String` actually return a new `String` containing the modification. The .NET language also defines mutable strings with its `System.Text.StringBuilder` class, which allows in-place modification of string values; but all access to the characters in its value are mediated through methods of this class [24]. In essence, all strings in .NET are objects, whose values are accessed through public methods of the class — the language does support a primitive `string` type but the compiler converts `string` type to the `String` object and uses class methods whenever the value of a primitive `string` type is manipulated.

Using the encapsulation features offered by the language, we have implemented the taint status for each string object rather than keeping a bit for each character. The taint status of each string object maintains metadata that identifies if the string is untrusted and if so, the portion of the string that is untrusted. Our implementation maintains a weak hash table for each object, which keys on weak references to objects, so that our instrumentation does not interfere with the garbage collection of the original application objects and scales in size. Entries to freed objects are therefore automatically dropped. Taint propagation, capturing direct data dependencies between string objects, is implemented by using wrapper functions for all operations in string classes. Each wrapper function updates the taint status of string objects at runtime.

We use CCI Metadata [34], a robust static .NET binary rewriting infrastructure to instrument each call to the string object constructors with taint propagation methods. The .NET language is a stack-based language and CCI Metadata provides the ability to interpose on any code block and statically rewrite it. Using this basic facility, we have implemented a library that allows caller instrumentation of specified functions, which allows redirection of original method calls to static wrapper methods of a user-defined class. Redirection of virtual function calls is handled the same way as static calls with the exception that the wrapper function accepts the instance object (sometimes referred to as the `this` parameter) is received as the first argument to the wrapper function.

**Soundness Considerations:** We explain how our positive taint implementation is sound, i.e., does not miss identifying untrusted data, with exception identified in point 5 below. We show that this exception are rare in our test program.

1. The language encapsulation features provide the guarantee that all access to the string values are only permitted through the invocation of methods defined in the string classes.
2. All constant strings belong to the immutable string primitive type. Any modification to the primitive value by the program is compiled to a conversion to an object of the string class, which invokes the string class constructors. Thus, we can safely track all sources of taint by instrumenting these constructors.
3. The string classes `System.String` and `System.Text.StringBuilder` are both *sealed* classes; that is, they cannot be inherited by other classes. This eliminates the possibility that objects that we do not track could invoke methods on the string classes.
4. Conversion between the two string classes is possible. This involves the call to the `Object.ToString` generic method. Statically, we instrument all these calls, and use .NET's built-in reflection at runtime to identify if the dynamic instance of the object being converted to a string and perform the taint metadata update.
5. String class constructors which convert values from non-string types are treated as safe (or positively tainted) by default. This is because we do not currently track taint status for these types. In principle, this is a source of potential unsoundness in our implementation. For example, the following code will lead our tracking to treat untrusted data as trusted:

```
String untrusted = Request.RawUrl;
var x = untrusted.ToCharArray();
....
String outputstr = new String(x);
httpw.Write(outputstr);
```

   Fortunately, these constructors are rare in practice. Figure 11(a) shows the distribution of functions instrumented by ScriptGard. The key finding is that potentially unsound constructions occur only in 42 out of 23,244 functions instrumented for our application. Our implementation ignores this source of false negatives presently; we can imagine verifying that these do not interfere with our needs using additional static analysis or implement more elaborate taint-tracking in the future.

**Output:** The result of ScriptGard's analysis is three pieces of information. First, ScriptGard marks the portion of the server's output which is not positively tainted. The untrusted texts are delimited using special markers consisting of characters that are outside the alphabet used by the application legitimately. Second, for each string written to the HTTP output stream, it records the sequence of propagators (such as string concatenation, format-string based substitution) applied on the output text fragment. In essence, this allows ScriptGard to (a) separate strings that are used for constructing HTML output templates from other strings, and, (b) identify the propagator that places the untrusted data into an output template. Third, it records a path string identifying the control flow path leading to each HTML output sink operation.

In addition to the above information, ScriptGard gathers the sequence of sanitizers applied to a given untrusted

data. To do this, each sanitizer is instrumented similarly to surround the input data with additional special markup identifying that sanitizer's application to the input data. The knowledge of the untrusted data along with the nesting of sanitizers is thus encoded in the HTML output of the server. This output is then subject to the context inference step, which is described next.

## 5.2 Context Inference

For a given analyzed path the server outputs a HTML response encoding the information identifying sub-strings that are untrusted, as well as, the sequence of sanitizers applied. ScriptGard employs a web browser to determine the contexts in which untrusted data is placed, in order to check if the sanitization sequence is consistent with the required sequence.

In our implementation, we use an HTML 5 compliant parser used in the C3 browser, that has been developed from scratch using code contracts to be as close to the current specification as possible. It has a fast JavaScript engine as well. The parser takes an HTML page as input. In the page, untrusted data is identified by special markup. The special markup is introduced at the server's output by our positive taint-tracking.

We augment the HTML parser to track the sequence of browser contexts in which data marked as untrusted appears. In our implementation for HTML, we treat each context to be (a) the state of the lexer (as defined in the HTML 5 draft specification), (b) the stack of open elements (as defined in the HTML 5 draft specification), and (c) specific information about the local state of the parse tree (such as the name of current tag or attribute being processed).

We apply techniques similar to string accenting for tracking untrusted data in other contexts [6]; DOM nodes that correspond to untrusted data are also marked. Similar context tracking is applied for the JavaScript parser. For the policy of our applications and the policies identified in previous work [20], we have found this level of tracking to be adequate.

Using context information, ScriptGard decides the correct sequence of sanitizers to apply for a given untrusted execution trace. To determine the correct sanitizer sequence, ScriptGard applies for each context in the trace, in the nesting order, the appropriate sanitizer from the input sanitization specification. The inferred chain of sanitizers is guaranteed to be correct for the trace, eliminating multiple sanitization errors that could have resulted from manual placement.

## 5.3 Runtime Sanitization Auto-Correction

During analysis or training, for each untrusted trace writing to sink operation $S$, ScriptGard records the static program path by profiling. It also record the first propagator $P$ in the dataflow computation, typically a string concatenation or format-string based string substitution, that places the untrusted data inside the trusted output string emitted at $S$. ScriptGard instruments the deployed application with a low-overhead runtime path detector. During deployed operation, If the currently executing path is is in the sanitization cache, ScriptGard sanitizes the untrusted substring(s) in the output string using the following technique.

**Rewriting untrusted output:** ScriptGard maintains a *shadow copy* of the untrusted data. If the path is not in the sanitization cache, the actual value of the untrusted data is used at output sink. If the path is in the sanitization cache, the shadow copy is sanitized, then the results are output.

To do this, ScriptGard instrumentation-based program transformation for maintaining shadow copies for each untrusted trace computation. Each untrusted trace computation is essentially a sequence of string propagator operations like string concatenation and format-string based substitution writing at a sink $S$. At string propagators using untrusted inputs, the added instrumentation creates a shadow copy of the untrusted data, and delimits it with special markup. At $S$, if the path executed is in the sanitization cache, the added instrumentation strips the actual value and markup out, applies the sanitization on the shadow copy, and writes it to the output stream. Finally, at $S$, if the path is not in the sanitization cache, the instrumentation strips the shadow value out and writes the actual value to the output stream leaving the application behavior unchanged for such paths.

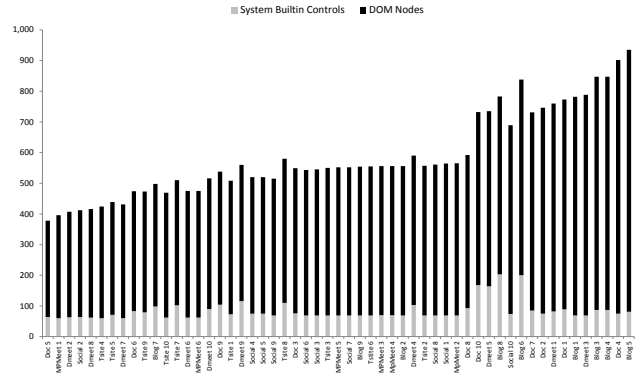



**Figure 6:** Distribution of DOM sizes, in nodes, across our training HTML pages.

| HTML Sink Context | Correct sanitizer that suffices |
|---|---|
| HTML Tag Context | `HTMLEncode`, `SimpleTextFormatting` |
| Double Quoted Attribute | `HTMLAttribEncode` |
| Single Quoted Attribute | `HTMLAttribEncode` |
| URL Path attribute | `URLPathEncode` |
| URL Key-Value Pair | `URLKeyValueEncode` |
| In Script String | `EcmaScriptStringEncode` |
| CDATA | `HTMLEncode` |
| Style | $\texttt{Alpha} - \texttt{numerics}$ |

**Figure 7:** Sanitizer-to-context mapping for our test application.

# 6. EVALUATION

Our evaluation focuses on a large legacy application of over 400,000 lines of server-side C# code. We accessed 53 distinct web pages, which we subjected to ScriptGard analysis. Figure 6 shows the size of the various web pages in terms of the number of DOM nodes they generate from their initial HTML output (ignoring dynamic updates to the DOM via JavaScript, etc.). Page sizes range from 350 to 900 nodes. Our analysis statically instrumented 23,244 functions.

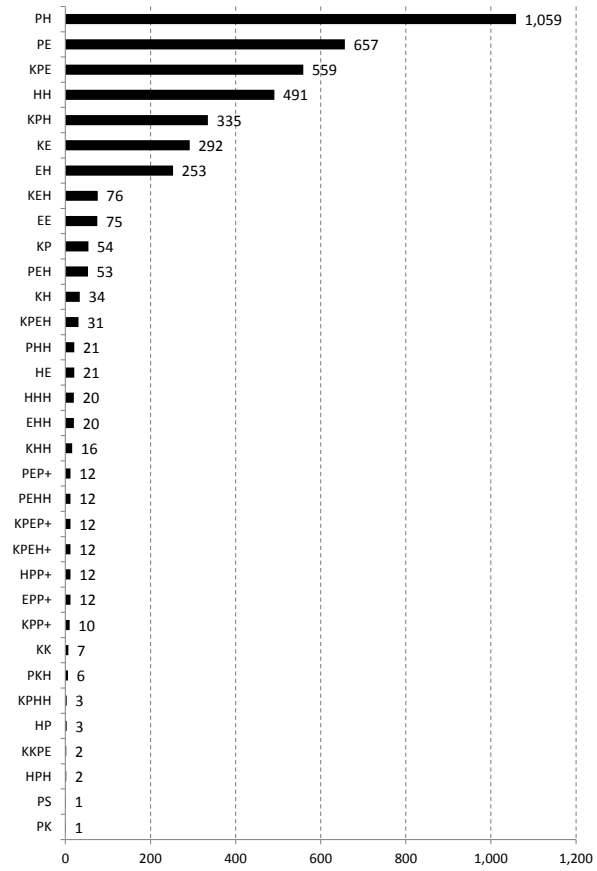



**Figure 8:** Histogram of sanitizer sequences consisting of 2 or more sanitizers empirically observed in analysis, characterizing sanitization practices resulting from manual sanitizer placement. `E`,`H`,`U`, `K`,`P`,`S` denote sanitizers `EcmaScriptStringLiteralEncode`, `HtmlEncode`, `HtmlAttribEncode`, `UrlKeyValueEncode`, `UrlPathEncode`,`SimpleHtmlEncode` respectively.

Our application uses custom objects that handle their own rendering. Figure 6 shows a majority of the DOM nodes in the applications' outputs are derived from custom objects. The language-based solution of ScriptGard, as opposed to .NET runtime library-specific solution, allows it to directly analyze these custom objects.

Figure 7 shows the mapping between contexts and sanitization functions for our application; which is a strict subset of mapping in previous work [20]. In particular, it permits only quoted attributes which have well-defined rules for sanitization [27]. Furthermore, it always sets the page encoding to UTF-8, eliminating the possibility of character-set encoding attacks [14]. We arrived at the result in Figure 7 after several interactions with the application's security engineers.

## 6.1 Analysis Results

**Context-mismatched sanitization:** Figure 9 shows that ScriptGard exercised 25,209 paths on which sanitization was applied. Of these, 1,558 paths (or 6.1%) were improperly sanitized. Of these improperly sanitized paths, 1,207 ( 4.7% of the total analyzed paths) contained data that could not be proven safe by our positive taint tracking infrastructure, so therefore are candidates for runtime automatic choice of sanitization. The remaining 1.4% of paths were sanitizing trusted strings improperly; ScriptGard does not consider these to be safe, therefore they do not need runtime correction.

| Web Page | Sanitized Paths | Inconsistently sanitized | |
|---|---|---|---|
| | | Total | Highlight |
| Home | 396 | 14 | 9 |
| A1 P1 | 565 | 28 | 22 |
| A1 P2 | 336 | 16 | 11 |
| A1 P3 | 992 | 26 | 21 |
| A1 P4 | 297 | 44 | 35 |
| A1 P5 | 482 | 22 | 17 |
| A1 P6 | 436 | 23 | 18 |
| A1 P7 | 403 | 19 | 13 |
| A1 P8 | 255 | 22 | 18 |
| A1 P9 | 214 | 16 | 12 |
| A1 P10 | 1,623 | 18 | 14 |
| A2 P1 | 315 | 16 | 12 |
| A2 P2 | 736 | 53 | 47 |
| A2 P3 | 261 | 21 | 16 |
| A2 P4 | 197 | 16 | 12 |
| A2 P5 | 182 | 22 | 18 |
| A2 P6 | 237 | 22 | 18 |
| A2 P7 | 632 | 20 | 16 |
| A2 P8 | 450 | 23 | 19 |
| A2 P9 | 802 | 26 | 22 |
| A3 P1 | 589 | 25 | 21 |
| A3 P2 | 2,268 | 18 | 14 |
| A3 P3 | 389 | 16 | 12 |
| A3 P4 | 477 | 103 | 15 |
| A3 P5 | 323 | 24 | 20 |
| A3 P6 | 292 | 51 | 45 |
| A3 P7 | 219 | 16 | 12 |
| A3 P8 | 691 | 25 | 21 |
| A3 P9 | 173 | 16 | 12 |
| A4 P1 | 301 | 24 | 20 |
| A4 P2 | 231 | 30 | 25 |
| A4 P3 | 271 | 28 | 22 |
| A4 P4 | 436 | 38 | 32 |
| A4 P5 | 956 | 36 | 24 |
| A4 P6 | 193 | 24 | 18 |
| A4 P7 | 230 | 36 | 32 |
| A4 P8 | 310 | 24 | 20 |
| A4 P9 | 200 | 24 | 18 |
| A4 P10 | 208 | 24 | 20 |
| A4 P11 | 498 | 34 | 29 |
| A4 P12 | 579 | 34 | 29 |
| A4 P13 | 295 | 25 | 20 |
| A4 P14 | 591 | 104 | 91 |
| A5 P1 | 604 | 61 | 55 |
| A5 P2 | 376 | 25 | 21 |
| A5 P3 | 376 | 25 | 21 |
| A5 P4 | 401 | 26 | 21 |
| A5 P5 | 565 | 31 | 26 |
| A5 P6 | 493 | 34 | 29 |
| A5 P7 | 521 | 34 | 29 |
| A5 P8 | 427 | 24 | 20 |
| A5 P9 | 413 | 24 | 20 |
| A5 P10 | 502 | 28 | 23 |
| **Total** | 25,209 | 1,558 | 1,207 |

**Figure 9:** Characterization of the fraction of the paths that were inconsistently sanitized. The right-most column indicates which fraction of those paths could not be proved safe and so were highlighted by our analysis. The difference between last and second last colmn is that some paths sanitize constant strings or provably trusted data.

We used Red Gate's .NET Reflector tool, combined with other decompilation tools, to further investigate the executions which ScriptGard reported as improperly sanitized. Our subsequent investigation reveals that errors result because it is difficult to manually analyze the calling context in which a particular potion of code may be invoked. In par-

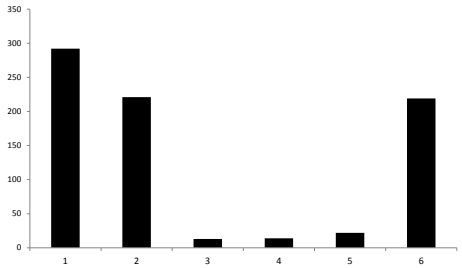
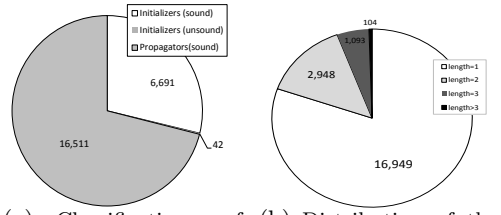



**Figure 10:** Distribution of lengths of paths that could not be proved safe. Each hop in the path is a string propagation function. The longer the chain, the more removed are taint sources from taint sinks.




(a) Classification of functions used for ScriptGard instrumentation.

(b) Distribution of the lengths of applied sanitization chains, showing a sizable fraction of the paths have more than one sanitizer applied.

**Figure 11:** Characterizing ScriptGard instrumentation.

ticular, the source and the sink may be separated by several intervening functions. Since ScriptGard instruments all string operations, we can count how far sources and sinks are removed from each other, as shown in 11(b). In Figure 10, we graph the distribution of these lengths for a randomly selected sample of untrusted paths. This shows that a significant fraction of the chains are long and over 200 of them exceed 5 steps.

Our data on the length of def-use chains is consistent with those reported in previous static analysis based work [18]. As explained in Section 2, the sharing of dataflow paths can result in further ambiguity in distinguishing context at the HTML output point in the server, as well as, in distinguishing trusted data from untrusted data. In our investigation we observed the following cases:

- A single sanitizer was applied in a context that did not match. Typically, the sanitizer applied was in a different function from the one that constructed the HTML template. This suggests that developers may not fully understand how the context — a global property — impacts the choice of sanitizer, which is a local property. This is not surprising, given the complexity of choices in Figure 7.
- A sanitizer was applied to trusted data (on 1.4% of the paths in our experiment) We still report these cases because they point to developer confusion. On further investigation, we determined this was because sinks corresponding to these executions were shared by several dataflow paths. Each such sink node could output potentially untrusted data on some executions, while outputting purely trusted data on others.
- More than one sanitizer was applied, but the applied sanitizers were not correct for the browser parsing context of the data[1].

**Inconsistent Multiple Sanitization:** We found 3,245 paths with more than one sanitizer. Of these, 285 (or 8%) of the paths with multiple sanitization were inconsistent with the context. The inconsistent paths fell into two categories: first, we found 273 instances with the (`EcmaScriptStringLiteralEncode`)(`HtmlEncode`)+ pattern applied. As we saw in Section 2, these sanitizers do not commute, and this specific order is inconsistent. Second, we found 12 instances of the (`EcmaScriptStringLiteralEncode`)(`UrlPathEncode`)+ pattern. This pattern is inconsistent because it does not properly handle sanitization of URL parameters. If an adversary controls the data sanitized, it may be able to inject additional parameters.

We found an additional 498 instances of multiple sanitization that were superfluous. That is, sanitizer $A$ was applied before sanitizer $B$, rendering sanitizer $B$ superfluous. While not a security bug, this multiple sanitization could break the intended functionality of the applications. For example, repeated use of `UrlKeyValueEncode` could lead to multiple percent encoding causing broken URLs. Repeated use of `HtmlEncode` could lead to multiple HTML entity-encoding causing incorrect rendering of output HTML.

We also observed that nesting of parsing contexts is common. For example a URL may be nested within an HTML attribute. Figure 11(b) shows the histogram of sanitizer sequence lengths observed. The inferred context for a majority of these sinks demanded the use of multiple sanitizers. Figure 8 shows the use of multiple sanitizers in the application is widespread, with sanitizer sequences such as `UrlPathEncode HtmlEncode` being most popular. In our application, these sanitizers are not commutative, i.e. they produce different outputs if composed in different orders, which means that paths with different orderings produce different behavior.

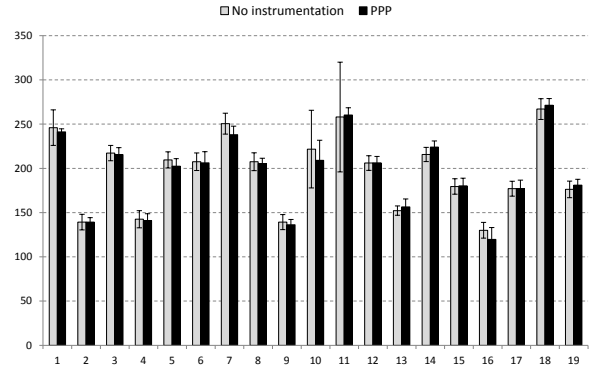
Because ScriptGard is a dynamic technique, all paths found can be reproduced with test cases exhibiting the context-inconsistent sanitization. We investigated a small fraction of these test cases in more depth. We found that while the sanitization is in fact inconsistent, injecting strings in these contexts did not lead to privilege escalation attacks. In part this is because our positive tainting is conservative: if we cannot prove a string is safe, we flag the path. In other cases, adversary's authority and the policy of the test application made it impossible to exploit the inconsistency.

## 6.2 Runtime Overhead

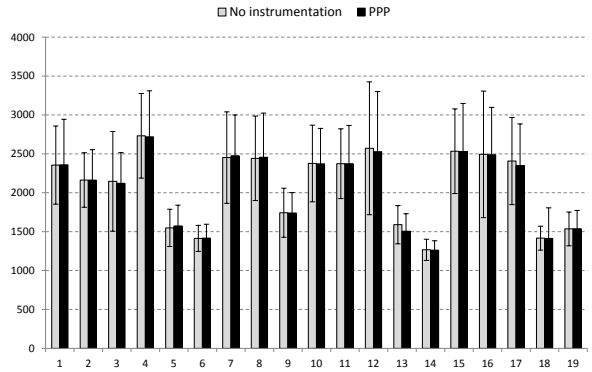
For our experiments, the server was a dual core Intel machine running at 3.16 GHz with 4 GB of RAM, 250 GB of 7200 RPM disk, running Windows Server 2008. Our client was a Core 2 Duo Intel machine running at 2.67 GHz with 8 GB of RAM, 160 GB of 7200 RPM disk, running 64-bit Windows 7. We connected the client and server directly using a network switch.

**Full overhead:** We started by measuring the overhead of the full ScriptGard instrumentation, including positive

[1]Errors where the combination was correct but the ordering was inconsistent with the nested context are reported separately as inconsistent multiple sanitization errors.




(a) Overhead over a switch.




(b) Overhead over an 80211b connection.

**Figure 12:** Overhead of PPP on 19 URLs. Time is in milliseconds. Top of bar is the $75^{th}$ percentile of 20 samples.

| Switch | | | 80211b | | |
|---|---|---|---|---|---|
| **None** | **PPP** | **Overhead** | **None** | **PPP** | **Overhead** |
| 246.0 | 241.3 | -1.93% | 2,355.5 | 2,358.2 | 0.12% |
| 139.3 | 139.3 | 0.00% | 2,163.2 | 2,163.2 | 0.00% |
| 217.3 | 215.5 | -0.81% | 2,146.5 | 2,121.2 | -1.18% |
| 142.5 | 141.0 | -1.05% | 2,731.8 | 2,719.0 | -0.47% |
| 209.5 | 202.5 | -3.34% | 1,548.7 | 1,574.4 | 1.66% |
| 207.5 | 206.3 | -0.60% | 1,413.4 | 1,416.4 | 0.21% |
| 250.5 | 238.0 | -4.99% | 2,451.7 | 2,476.5 | 1.01% |
| 207.5 | 205.5 | -0.96% | 2,442.2 | 2,455.7 | 0.55% |
| 139.3 | 136.3 | -2.15% | 1,742.2 | 1,739.9 | -0.13% |
| 221.8 | 209.0 | -5.75% | 2,376.2 | 2,370.7 | -0.23% |
| 258.0 | 260.3 | 0.87% | 2,373.0 | 2,373.5 | 0.02% |
| 206.0 | 206.3 | 0.12% | 2,571.8 | 2,527.5 | -1.72% |
| 152.3 | 156.5 | 2.79% | 1,588.9 | 1,505.9 | -5.22% |
| 215.8 | 224.0 | 3.82% | 1,266.9 | 1,261.4 | -0.43% |
| 179.5 | 180.3 | 0.42% | 2,533.5 | 2,530.8 | -0.11% |
| 130.0 | 119.8 | -7.88% | 2,493.5 | 2,489.2 | -0.17% |
| 177.0 | 177.3 | 0.14% | 2,408.0 | 2,350.5 | -2.39% |
| 267.0 | 271.3 | 1.59% | 1,416.1 | 1,413.1 | -0.21% |
| 176.3 | 181.0 | 2.70% | 1,534.9 | 1,538.4 | 0.23% |
| **Average** | | *-0.60%* | **Average** | | *-0.13%* |
| **Median** | | *-0.90%* | **Median** | | *-0.45%* |

**Figure 13:** PPP overhead statistics . Time is in milliseconds, $75^{th}$ percentile reported from 20 samples.

taint flow and output parsing. Then we measured the overhead that would be incurred by the deployment phase of path preferential profiling techniques. While the full instrumentation incurs a huge overhead, the preferential profiling incurs negligible overhead.

We took nine URLs, each of which triggered complicated processing on the server to create the resulting web page. For each URL we first warmed the server cache by requesting the URL 13 times. We then measured the *time to first byte* of 13 additional queries to the server for the URL. Time to first byte is appropriate here because we are interested in the overhead added by ScriptGard to server processing. Finally, we took the $75^{th}$ percentile result, following web performance analysis practice used by colleagues at our institution responsible for a large public facing web site. Unfortunately, we found overheads of over 175.6x for the full analysis, for example a URL that took 0.53 seconds to load without instrumentation took 92.73 seconds to load with all checks.

**PPP:** Fortunately, we can shift much of this overhead to a pre-release phase by using preferential path profiling (PPP) techniques. As we described above, the optimizations in PPP include a smart way of numbering code blocks and a way to avoid updating global state when certain "preferred" paths are taken.

We used a wide-area network simulator running on the server to emulate conditions of a typical IEEE 802.11b link between the client and the server, to produce realistic network conditions for evaluating the total overhead. We then instrumented our application using preferential path profiling and performed training on a set of 19 URLs. To test the worst-case performance from PPP, we then chose 19 URLs completely different from those that were used during training. We then visited each URL 20 times and measured the time to first byte.

Figure 12 graphs the time to first byte for each of our test URLs, in milliseconds, after removing the first visit. To account for outliers, the time measurement represents the $75^{th}$ percentile of the measurements over a series of 19 samples. Our graph shows error bars representing one standard deviation in each direction.

Figure 13 shows the median for each URL visited and the overhead for each of the 19 URLs. The average overhead for using preferential path profiling is actually negative, which we attibute to the overhead being too small to be statistically significant. We report overhead both for direct connection to the switch and for when the network simulator is in use configured to simulate an 802.11 connection. Our results show that while the runtime overhead of the full ScriptGard analysis is prohibitive, the deployment overhead can be decreased greatly by shifting analysis to a training phase.

# 7. RELATED WORK

Many defense techniques for scripting attacks have been proposed, which we discuss next. ScriptGard targets a new class of context-inconsistency errors.

**Defense Techniques:** Browser-based mitigations, such as Noncespaces, XSS Auditor, or DSI make changes to the web browsers that make it difficult for an adversary's script to run [3, 11, 25]. These approaches can be fast, but they require all users to upgrade their web browsers.

A server side mitigation, in contrast, focuses on changing the web application logic. Blueprint describes mechanisms to ensure the safe construction of the intended HTML parse

tree on the client using JavaScript in a browser-agnostic way [20]. However, issues in applying BLUEPRINT's mechanisms to various context still exist and developers may place them wrongly. Our work addresses this separate aspect of the problem, in a way that does not require any manual effort from developers.

XSS-GUARD [4] proposes techniques to learn allowed scripts from unintended scripts. The allowed scripts are then white-listed. Like SCRIPTGARD, it employs a web browser for its analysis, but the two approaches are fundamentally different. SCRIPTGARD's defense is based on automatic server-side sanitizer placement, rather than browser-based white-listing of scripting in server output. XSS-GUARD's techniques are intended for applications that allow rich HTML, where designing correct sanitizers becomes challenging. SCRIPTGARD target applications with fairly restrictive policies that have already addressed the sanitizer-correctness issue.

Google AutoEscape is a context-sensitive auto-sanitization mechanism for Google's GWT and CTemplates templating frameworks. As explained earlier, the templating languages it auto-sanitizes are much simpler and AutoEscape does not need to handle if-else or loop constructs which create path-senstivity a major issue. SCRIPTGARD's target is towards large-scale legacy applications, where sanitization placement logic is complex and already deployed, so rewriting all such logic is not practical.

SECURIFLY translates bug specifications written in a special program query to runtime instrumentation that detects these bugs [21]. Our approach reasons both about the browser context as well as the server state, allowing us to tackle sanitizer placement problems not detected by SECURIFLY.

**Software security analysis of web applications:** Software security focuses on using program analysis to find security critical bugs in applications. The WebSSARI project pioneered these approaches for web applications, and several static analysis tools have been propose [16, 35]. Runtime approaches, like ours, has the advantage of demonstrating clear, reproducible test cases over static analysis tools. Multiple runtime analysis systems for information flow tracking have been proposed, including Haldar *et al.* for Java [12] and Pietraszek *et al.* [28] and Nguyen-Tuong *et al.* for PHP [26]. Typically systems use *negative tainting* to specifically identify untrusted data in web applications applications [18, 19, 21, 36]. While negative taint is preferable for finding bugs, it is less desirable for mitigations because it requires specifying all sources of taint. Our design distinguishes itself from most previous work in that it tracks *positive taint*, which is conservative default fail-close approach, and side-steps identifying all sources of taint. The main exception is WASP [13], which does use positive taint, but which concerned SQL injection attacks, which do not exhibit the path sensitivity, use of multiple sanitizers, and need for a browser model to determine if data is a potential cross site scripting attack. WASP was also evaluated on much smaller applications (maximum 20,000 lines of code) than considered in this work.

**Sanitizer correctness:** Balzarotti *et al.* show that custom sanitizer routines are often incorrectly implemented [2].

Livshits *et al.* developed methods for determining which functions in a program play the role of sanitizer. Their MERLIN system is also capable of detecting missing sanitizers [19]. SCRIPTGARD's analysis is complementary to these works. Sanitizers may be present, and they may be functionally correct for contexts they are intended to be used in. Incorrect placement, however, can introduce errors.

The Cross-Site Scripting Cheat Sheet shows over two hundred examples of strings that exercise common corner cases of web sanitizers [29]. The BEK project proposes a systematic domain-specific languages for writing and checking sanitizers [15, 33].

# 8. CONCLUSIONS

We analyzed a set of 53 large web pages in a large-scale web application with over 400,000 lines of code. Each page contained 350–900 DOM nodes. We found 285 multiple-encoding issues, as well as 1,207 instances of inconsistent sanitizers, establishing the prevalence of our two new problem classes and the effectiveness of SCRIPTGARD as a testing aid. With preferential path profiling, when used for mitigation, SCRIPTGARD incurs virtually no statistically significant overhead on cached paths.

## Acknowledgments

We would like to give special thanks to Patrice Godefroid, who has been most instrumental in helping us define the problems. Kapil Vaswani made the ppp tool work with our application. We also thank Kapil Vaswani, David Wagner, Herman Venter, Joel Weinberger, Peli de Halleux, Nikolai Tillmann, Devdatta Akhawe, Adrian Mettler, Dawn Song, our anonymous reviewers, and other Microsoft colleagues. The first author performed this work while interning at Microsoft Research.

# 9. REFERENCES

[1] M. Balduzzi, C. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[2] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[3] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. *International World Wide Web Conference*, 2010.

[4] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[5] H. Bojinov, E. Bursztein, and D. Boneh. XCS: Cross channel scripting and its impact on web applications. In *CCS*, 2009.

[6] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight defense mechanism. In *Proceedings of the Conference on Computer and Communications Security*, October 2007.

[7] T. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, , and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the International Conference on Software Engineering*, May 2009.

[8] D. Connolly and C. M. Sperberg-McQueen. Web addresses in HTML 5. `http://www.w3.org/html/wg/href/draft#ref-RFC3986`, 2009.

[9] Fortify, Inc. Fortify SCA. `http://www.fortifysoftware.com/products/sca/`, 2006.

[10] Google. Google auto escape. http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html, 2011.

[11] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. 2009.

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

[13] W. G. Halfond, A. Orso, and P. Manolios. WASP: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1), 2008.

[14] Y. HASEGAWA. UTF-7 XSS cheat sheet. http://openmya.hacker.jp/hasegawa/security/utf7cs.html, 2005.

[15] P. Hooimeijer, B. Livhsits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the Usenix Security Symposium*, Aug. 2011.

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

[17] A. Kieżun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *International Symposium on Software Testing and Analysis*, 2009.

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

[19] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2009.

[20] M. T. Louw and V. N. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

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

[22] Microsoft Corporation. Microsoft code analysis tool .NET, 2009. http://www.microsoft.com/downloads/en/details.aspx?FamilyId=0178e2ef-9da8-445e-9348-c93f24cc9f9d&displaylang=en,.

[23] Microsoft Corporation. String class (system), 2010. http://msdn.microsoft.com/en-us/library/system.string.aspx.

[24] Microsoft Corporation. StringBuilder class, 2010. http://msdn.microsoft.com/en-us/library/system.text.stringbuilder(v=VS.80).aspx.

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

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

[27] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

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

[29] RSnake. XSS cheat sheet for filter evasion. http://ha.ckers.org/xss.html.

[30] B. Schmidt. google-analytics-xss-vulnerability, 2011. http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/.

[31] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[32] Ter Louw, Mike and V.N. Venkatakrishnan. BluePrint: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[33] M. Veanes, B. Livshits, and D. Molnar. Decision procedures for composition and equivalence of symbolic finite state transducers. Technical Report MSR-TR-2011-32, Microsoft Research, 2011.

[34] H. Venter. Common compiler infrastructure: Metadata API, 2010. http://ccimetadata.codeplex.com/.

[35] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, 2006.

[36] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the Usenix Security Symposium*, 2006.