Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers

Mike Samuel
Google Inc.
Mountain View, CA, USA
mikesamuel@gmail.com

Prateek Saxena
Computer Science Division
UC Berkeley
Berkeley, CA, USA
prateeks@cs.berkeley.edu

Dawn Song
Computer Science Division
UC Berkeley
Berkeley, CA, USA
dawnsong@cs.berkeley.edu

ABSTRACT

Scripting vulnerabilities, such as cross-site scripting (XSS), plague web applications today. Most research on defense techniques has focused on securing existing legacy applications written in general-purpose languages, such as Java and PHP. However, recent and emerging applications have widely adopted web templating frameworks that have received little attention in research. Web templating frameworks offer an ideal opportunity to ensure safety against scripting attacks by secure construction, but most of today's frameworks fall short of achieving this goal.

We propose a novel and principled type-qualifier based mechanism that can be bolted onto existing web templating frameworks. Our solution permits rich expressiveness in the templating language while achieving backwards compatibility, performance and formal security through a context-sensitive auto-sanitization (CSAS) engine. To demonstrate its practicality, we implement our mechanism in Google Closure Templates, a commercially used open-source templating framework that is used in GMail, Google Docs and other applications. Our approach is fast, precise and retrofits to existing commercially deployed template code without requiring any changes or annotations.

Categories and Subject Descriptors: D.4.6 Operating Systems: Security and Protection; D.1.2 Programming Techniques: Automatic Programming

General Terms: Languages, Security

Keywords: Cross-site Scripting, Type Systems

1. INTRODUCTION

Scripting vulnerabilities, such as cross-site scripting [36] and cross-channel scripting [6], are pervasive in web applications [1, 7, 8, 10, 41], embedded systems [6, 17] and on smartphone platforms [9]. A central reason for this wide-spread prevalence is the ad-hoc nature of output generation from web applications today. Web applications emit code intermixed with data in an unstructured way. *Web application output* is essentially text strings which can be emitted from the server-side code (in Java or PHP) or from client-side code in JavaScript. When a portion of the application output controlled by the attacker is parsed by the browser as a script, a scripting attack results.

The predominant first-line of defense against scripting vulnerabilities is *sanitization*—the process of applying encoding or filtering primitives, called *sanitization primitives* or *sanitizers*, to render dangerous constructs in untrusted inputs inert [1, 38, 49, 50]. However, the practice of manually applying sanitizers is notoriously prone to errors [1, 27, 33, 41–43].

Web Templating Frameworks. To streamline the output generation from application code, numerous web templating frameworks have recently emerged and are gaining wide-spread adoption [11–14, 20, 25, 34, 40, 44, 45, 48, 53, 54]. Web templating frameworks allow developers to specify their application's output generation logic in code units or modules called *templates*. Templates take *untrusted inputs* which may be controlled by the attacker and emit web application outputs, such as HTML or CSS code, as strings. String outputs from templates are composed of static or constant strings written by developers, which are explicitly trusted, and untrusted inputs which must be sanitized. These templates can be compiled into a *target language*, such as JavaScript or Java/PHP, as code functions that take untrusted data as template arguments and emit the application's output as strings. Templates are written in a different language, called a *templating language*, the semantics of which are much simpler as compared to that of the target language. Notably, complex constructs such as JavaScript's `eval` and `document.write` are not included in the templating language. Code external to templates is responsible for invoking compiled templates to obtain the string outputs and evaluating/rendering them in the browser.

Vision. Ideally, we would like to create web applications that are secure by construction. In fact, web templating frameworks offer an ideal opportunity to relieve the developers from the burden of manual sanitization by *auto-sanitizing*—inserting sanitization primitives automatically during the compilation of templates to server-side or client-side code. Despite this ideal opportunity, research so far has not broached the topic of building auto-sanitization defenses in today's commercial templating frameworks.

Challenges. In this work, we first identify the following practical challenges in building reliable and usable auto-sanitization in today's web templating frameworks:

- *Context-sensitivity.* XSS sanitization primitives vary significantly based on the *context* in which the data sanitized is being rendered. For instance, applying the default HTML escaping sanitizer is recommended for untrusted values placed inside HTML tag content context [38]; however, for URL attribute context (such as `src` or `href`) this sanitizer is insufficient because the `javascript` URI protocol (possibly masked) can be used to inject malicious code [21]. We say, therefore, that each sanitization primitive *matches* a context in which it provides safety. Many developers fail to consistently apply the sanitizers matching the context, as highlighted in a recent study performed on a large commercial application well-audited for security [43].

- *Complexity of language constructs.* Templating languages today permit a variety of complex constructs: support for string data-type operations, control flow constructs (`if-else`, loops) and `call`s to splice the output of one template into another. Untrusted input variables may, in such languages, be used in one context along one execution path and a different context along another path. With such rich language features, determining the context for each use of untrusted input variables becomes a path-sensitive, global data-flow analysis task. Automatically applying correct sanitization on all paths in templating code becomes challenging.
- *Backwards compatibility with existing code.* Developers may have already applied sanitizers in existing template code at arbitrary places; an auto-sanitization mechanism should not undo existing sanitization unless it is unsafe. For practical adoption, auto-sanitization techniques should only supplement missing sanitizers or fix incorrectly applied ones, without placing unnecessary restrictions on where to sanitize data.
- *Performance Overhead.* Auto-sanitized templates should have a minimal performance overhead. Previous techniques propose parsing template outputs with a high-fidelity HTML parser at runtime to determine the context [5]. However, the overhead of this mechanism may be high and undesirable for many practical applications.

Context-sensitive Auto-sanitization Problem. We observe that a set of contexts in which applications commonly embed untrusted data is known [50]. And, we assume that for each such context, a matching sanitizer is externally provided. Extensive recent effort has focused on developing a library of safe or correctly-implemented sanitization primitives [22, 23, 29, 30, 38, 41]. We propose to develop an automatic system that, given a template and a library of sanitizers, automatically sanitizes each untrusted input with a sanitizer that matches the context in which it is rendered. By auto-sanitizing templates in this context-sensitive way, in addition to enforcing the security properties we outline in Section 2.2, templating systems can ensure that scripting attacks never result from using template outputs in intended contexts.

Our Approach & Contributions. In this paper, we address the outlined challenges with a principled approach:

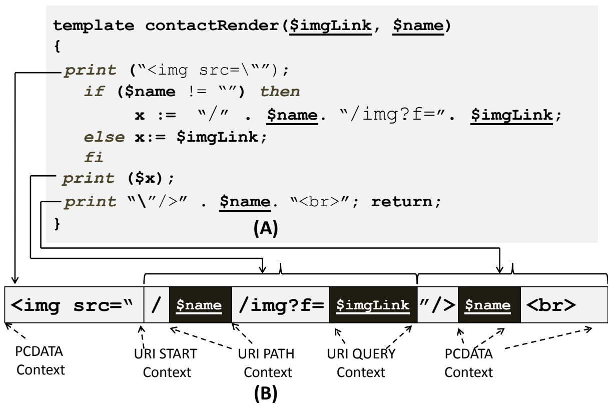
- *Type Qualifier Based Approach.* We propose a type-based approach to automatically ensure context sensitive sanitization in templates. We introduce *context type qualifiers*, a kind of type qualifier that represents the *context* in which untrusted data can be safely embedded. Based on these qualifiers, which refine the base type system of the templating language, we define a new type system. Type safety in our type system guarantees that well-typed templates have all untrusted inputs context-sensitively sanitized.
- *Type Inference during Compilation.* To transform existing developer-written templates into well-typed templates, we develop a Context-Sensitive Auto-Sanitization (CSAS) engine which runs during the compilation stage of a web templating framework. The CSAS engine performs two high-level operations. First, it performs a static type inference to infer context type qualifiers for all variables in templates. Second, based on the inferred context types, the CSAS engine automatically inserts sanitization routines into the generated server-side or client-side code. To the best of our knowledge, our approach is the first principled approach using type qualifiers and type inference for context-sensitive auto-sanitization in templates.
- *Real-world Deployability.* To show that our design is practical, we implement our type system in Google Closure Templates, a commercially used open-source templating framework that is used in large applications such as GMail and Google Docs. Our implementation shows that our approach requires less than 4000 lines of code to be built into an existing commercial web framework. Further, we show that retrofitting our type system to existing templates used in commercial applications requires no changes or annotations to existing code.
- *Improved Security.* Our approach eliminates the critical drawbacks of existing approaches to auto-sanitization in today's templating frameworks. Though all the major web frameworks today support customizable sanitization primitives, a majority of them today do not automatically apply them in templates, leaving this error-prone exercise to developers. Most others automatically sanitize all untrusted variables with the same sanitizer in a *context-insensitive* manner, a fundamentally unsafe design that provides a false sense of security [50]. Google AutoEscape, the only context-sensitive abstraction we are aware of, does not handle the richness of language features we address. We refer readers to Section 7 for a detailed comparison.
- *Fast, Precise and Mostly Static Approach.* We evaluate our type inference system on 1035 existing real-world Closure templates. Our approach offers practical performance overhead of $3 - 9.6\%$ on CPU intensive benchmarks. In contrast, the alternative runtime parsing approach incurs 78% - 510% overhead on the same benchmarks. Our approach performs all parsing and context type inference statically and so achieves significantly better performance. Our approach does not sacrifice any precision in context-determination as compared to the runtime parsing approach— it defers context-sensitive sanitization to runtime for a small fraction of output operations in which pure static typing is too imprecise. Hence, our type system is *mostly static*, yet precise.

2. PROBLEM DEFINITION

The task of auto-sanitization is challenging because state-

$$
\begin{array}{lll}
\text{Base Types} & \alpha ::= & \beta \mid \eta \mid \beta_1 \rightarrow \beta_2 \rightarrow \ldots \beta_k \rightarrow \texttt{unit} \\
 & \beta ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \texttt{unit} \\
\text{Commands} & S ::= & \textbf{print}\ (e : \beta) \\
 & & \mid (v : \beta) := (e : \beta) \\
 & & \mid \textbf{callTemplate}\ f\ (e_1, \ldots, e_k) \\
 & & \mid c_1\ ;\ S_1 \\
 & & \mid \textbf{if}(e : \mathit{bool})\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \\
 & & \mid \textbf{while}(e : \mathit{bool})\ S_1 \\
 & & \mid \textbf{return}; \\
\text{Expressions} & e ::= & (e_1 : \texttt{int}) \oplus (e_2 : \texttt{int}) \\
 & & \mid (e_1 : \texttt{bool}) \odot (e_2 : \texttt{bool}) \\
 & & \mid (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string}) \\
 & & \mid \textbf{const}\ (i : \beta) \\
 & & \mid v : \beta \\
 & & \mid \textbf{San}\ (f, e : \beta) \\
 & v ::= & \mathit{Identifier}
\end{array}
$$

Figure 1: The syntax of a simple templating language. $\oplus$ represents the standard integer and bitvector arithmetic operators, $\odot$ represents the standard boolean operations and $\cdot$ is string concatenation. The *San* expression syntactically refers to applying a sanitizer.



```
template contactRender($imgLink, $name)
{
 print ("<img src=\"");
   if ($name != "") then
       x :=  "/" . $name. "/img?f=". $imgLink;
   else x:= $imgLink;
   fi
 print ($x);
 print "\"/>" . $name. "<br>"; return;
}
```

Figure 2: (A) shows a template used as running example. (B) shows the output buffer after the running example has executed the path including the true branch of the `if` statement.

of-the-art templating frameworks don't restrict templates to be straight-line code. In fact, most templating frameworks today permit control-flow constructs and string data operations to allow application logic to conditionally alter the template output at runtime. To illustrate the issues, we describe a simple templating language that captures the essence of the output-generating logic of web applications. We motivate our approach by showing the various challenges that arise in a running example written in our templating language.

2.1 A Simple Templating Language

Our simple templating language is expressive enough to model Google Closure Templates and several other frameworks. We use this templating language to formalize and describe our type-based approach in later sections. It is worth noting that the simple templating language we present here is only an illustrative example—our type-based approach is more general and can be applied to other templating languages as well.

The syntax for the language is presented in Figure 1. The templating language has two kinds of data types in its base type system: the primitive (`string, bool, int`) types and a special type (denoted as η) for *output buffers*, which are objects to which templates write their outputs. Figure 2(A) shows a running example in our templating language. For simplicity, we assume in our language that there is only a single, global output buffer to which all templates append their output, similar to the default model in PHP.

Command Semantics. The primary command in the language is the `print` command which appends the value of its only operand as a string to the output buffer. The running example has several `print` commands. Note that the syntax ensures that the output buffer (η-typed object) can not be reassigned, or tampered with in the rest of the command syntax.

Templates are akin to functions: they can call or invoke other templates via the `callTemplate` command. This command allows a template to invoke another template during its execution, thereby splicing the callee's outputs into its own. Parameter passing follows standard pass-by-value semantics.

The templating language allows control-flow commands such as `for` and `if-then-else` to allow dynamic construction of template outputs. It supports the usual boolean and integer operations as well as string concatenation. We exclude more complex string manipulation operations like string substitution and interpolation functions from the simple language; with simple extensions, their semantics can be modeled as string concatenations [41].

Restricting Command Semantics. The semantics of the templating language is much simpler than that of a general-purpose language that templates may be compiled to. Notably, for instance, the templating language does not have any dynamic evaluation commands such as JavaScript's `eval` or `document.write`. Therefore, final code evaluation in DOM evaluation constructs or serialization to the HTTP response stream is performed by external application code. For instance, Figure 3 below shows a JavaScript application code written outside the templating language which invokes the function compiled from the running example template. It renders the returned result string dynamically using a `document.write`. Therefore, the template code analysis does not need to model the complex semantics of `document.write` [1] .

```
<script>
var o = new soy.StringBuilder();
contactRender({O: o, imglink: $_GET('extlink'),
               name: [$_GET('name')] }));
document.write(o);
</script>
```

Figure 3: Psuedo-code of how external application code, such as client-side Javascript, can invoke the compiled templates.

2.2 Problem Definition & Security Properties

In this paper, we focus on the following problem: given a templating language such as the one in Section 2.1, and a set of correct sanitization routines for different contexts, the goal is to automatically apply the correct sanitization primitives during compilation to all uses of untrusted inputs in constructing template outputs, while satisfying the following properties.

[1]The semantics of `document.write` varies based on whether the `document` object is open or closed.

Property `NOS`: No Over-Sanitization. The templating language allows string expressions to be emitted at `print` operations. String expressions may be constructed by concatenation of constant/static strings and untrusted input variables; only the latter should be sanitized or else we risk breaking the intended structure of the template output. For instance in our running example, the auto-sanitization engine should *not* place a sanitizer at the statement `print ($x)`, because the expression `x` consists of a constant string as well as untrusted input value. Sanitizing at this print statement may strip out the `/` or `?` characters rendering the link unusable and breaking the intended structure of the page.

Property `CSAN`: Context-Sensitive Sanitization. Each untrusted input variable should be sanitized with a sanitizer matching the context in which it is rendered in. However, this is challenging because untrusted inputs may be used in two different contexts along two different paths. In our running example, the `$imgLink` variable is used both in a URI context as well as a HTTP parameter context, both of which have different sanitization requirements. Similary, untrusted inputs can be rendered in two different contexts even along the same path, as seen for the variable `$name` in Figure 2 (B). We term such use of inputs in multiple contexts as a *static context ambuiguity*, which arise because of path-sensitive nature of the template output construction logic and because of multiple uses of template variables. Section 4 describes further scenarios where context ambiguity may arise.

Property `CR`: Context Restriction. Template developers should be forbidden from mistakenly using untrusted values in contexts other than ones for which matching sanitizers are available. Certain contexts are known to be hard to sanitize, such as in an unqouted JavaScript string literal placed directly in a JavaScript eval [21], and thus should be forbidden.

Determining Final Output Start/End Context. For each template, we infer the contexts in which the template's output can be safely rendered. However, since the final output is used external to the template code, providing a guarantee that external code uses the output in an intended context is beyond the scope of our problem. For example, it is unsafe for external code to render the output of the running example in a JavaScript `eval`, but such properties must be externally checked.

2.3 Motivation for Our Approach

If a templating langauge has no control-flow or `callTemplate` constructs and no constructs to create string expressions, all templates would be straight-line code with `print`s of constant strings or untrusted variables. Auto-sanitizating such templates is a straight-forward 3-step process— (a) parse the template statically using a high-fidelity parser (like HTMLPurify [23]), (b) determine the context at each `print` of untrusted inputs and (c) apply the matching sanitizer to it. Unfortunately, real templating languages are often richer like our templating language and more sophisticated techniques are needed.

One possible extension of the approach for straight-line code is to defer the step of parsing and determing contexts to runtime execution [5]. We call this approach a *context-sensitive runtime parsing* (or CSRP) approach, where a parser parses all output from the compiled template, determines the context of each `print` of untrusted input and sanitizes it at runtime. This approach has additional performance overhead due to cost of parsing all application output at runtime, as previously shown [5] and as we evaluate in Section 6. If string operations are supported in the language, the performance penalty may be exacerbated because of the need for tracking untrusted values during execution.

Instead, we propose a new "mostly static" approach which off-loads expensive parsing steps to a static type analysis phase. Contexts for most uses of untrusted data can be statically determined and their sanitizers can be selected during compile-time; only a small fraction need the more expensive CSRP-like sanitizer selection in our approach—hence our approach is "mostly static".

Assumptions. Our type-based approach relies on a set of assumptions which we summarise below:

1. *Canonical Parser.* To reliably determine the contexts in which untrusted inputs are rendered, constant/static strings in templates must parse according to a canonical grammar which reliably parses in the same way across major browsers. This restriction is necessary to ensure that our context determination is consistent with its actual parsing in the client's browser, which is challenging because browser parsing behaviors vary in idiosyncratic ways. In our approach, templates not complying with our canonical grammar do not typecheck as per our type rules defined in section 4. Google AutoEscape based frameworks such as GWT and CTemplate already tackle the practical issue of developing such a canonical grammar [11, 13, 20]; our engine leverages this existing code base.
2. *Sanitizer Correctness.* As mentioned previously, we assume that a set of contexts in which applications commonly render untrusted inputs is known and their matching sanitizers are externally available. Creating sanitizers that work across major browser versions is an orthogonal challenge being actively researched [22, 23].
3. *End-to-End Security.* As explained earlier, if the external code renders the template outputs in an unintended context or tampers with the template's output before emitting it to the browser, the end-to-end security is not guaranteed. Ensuring correctness of external code that uses template outputs is beyond the scope of the problem we focus here—lint tools, static analysis and code conformance testing can help enforce this discipline externally.

3. OUR APPROACH

In our type-based approach, we enforce the aforementioned security properties by attaching or qualifying variables and expressions in templates with a new kind of qualifier which we call the *context type qualifier*. Type qualifiers are a formal mechanism to extend the basic type safety of langauge to enforce additional properties [16]. Context type qualifiers play different roles for the various expressions they qualify. For an untrusted input variable, the context type qualifier captures the contexts in which the variable can be safely rendered. An untrusted input becomes safe for rendering in a certain context only after it is sanitized by a sanitizer matching that context. Unsanitized inputs have the `UNSAFE` qualifier attached, and are not safe to be a part

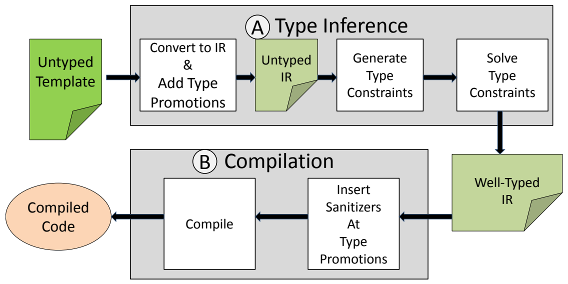


Figure 4: Overview of our CSAS engine.

$$
\begin{array}{lll}
\text{Types} & \tau ::= & \mathcal{Q}_1\ \beta \mid \mathcal{Q}_2\ \eta \\
\text{Base Types} & \alpha ::= & \beta \mid \eta \mid \beta_1 \rightarrow \beta_2 \rightarrow \ldots \texttt{unit} \\
 & \beta ::= & \texttt{bool} \mid \texttt{int} \mid \texttt{string} \mid \texttt{unit} \\
\textbf{Type} & \mathcal{Q} ::= & \mathcal{Q}_1 \mid \mathcal{Q}_2 \mid \vec{\mathcal{Q}_1} \rightarrow [\mathcal{Q}_2 \rightarrow \mathcal{Q}_2] \\
\textbf{Qualifiers} & \mathcal{Q}_1 ::= & \texttt{UNSAFE} \\
 & & \mid \texttt{STATIC}_{c_1 \hookrightarrow c_2} \qquad c_1, c_2 \in \mathcal{C} \\
 & & \mid \texttt{DYN}_{SC} \qquad SC \in 2^{\mathcal{C}\times\mathcal{C}} \\
 & \mathcal{Q}_2 ::= & \texttt{CTXSTAT}_c \qquad c \in \mathcal{C} \\
 & & \mid \texttt{CTXDYN}_S \qquad S \in 2^{\mathcal{C}} \\
\text{Contexts} & \mathcal{C} ::= & \texttt{PCDATA} \mid \texttt{RCDATA} \mid \ldots
\end{array}
$$

Figure 5: The final types τ are obtained by augmenting base types of the language α with type qualifiers $\mathcal{Q}$

of any expression that is used in a `print` statement. For constant/static string expressions, context type qualifiers capture the result of parsing the expression, that is, the start context in which the expression will validly parse and the context that will result after parsing the expression. When the template code constructs an output string expression by concatenating a constant string and an untrusted input, a type rule over context qualifiers of the two strings ensures that the untrusted input is only rendered in contexts for which it is sanitized.

This rule only enforces the `CSAN` property in the concatenation operation. Several additional rules are needed to enforce all the outlined security properties to cover all operations in our templating language. We describe the full type system with formal type rules over context type qualifiers in section 4. The type safety of the type system implies that the security properties outlined in Section 2.2 are enforced.

CSAS Engine. The input to our auto-sanitization engine is an existing template which may be completely devoid of sanitizers. We call these templates *untyped* or vanilla templates. The task of our auto-sanitization engine is two-fold: (a) to convert untyped or vanilla templates into an internal representation (or IR) complying with our type rules (called the well-typed IR), and (b) to compile the well-typed IR to the target language code with sanitization. We develop a CSAS engine in the compiler of a templating framework to handle these tasks. Figure 4 shows the CSAS architecture. It has two high-level steps: (A) Type Qualifier Inference, and (B) Compilation of CSAS templates.

The qualifier inference step transforms the vanilla template into a well-typed IR and automatically infers the type qualifiers for all program expressions in the IR. The inferred type qualifiers exactly determine where and which sanitizers are required for untrusted inputs. The well-typed IR must conform to the type rules that we define in Section 4. The step (B) compiles the well-typed IR and inserts sanitization primitives and additional instrumentation in the final compiled code. The detailed design of the CSAS engine is presented in Section 5.

4. THE CONTEXT TYPE SYSTEM

In this section, we formally describe our type qualifier mechanism that refines the base type system of the language defined in Section 2.1. We define the associated type rules to which well-typed IR code must conform after the type inference step.

4.1 Key Design Points

The CSAS engine must track the parsing context induced by the application's output at each program point. Each string expression, when parsed by the browser, causes the browser to transition from one context to another. We term this transition induced on the browser by parsing strings as a *context transition*, denoted by the notation $c_1 \hookrightarrow c_2$.

Notion of Context Type Qualifiers. Qualifiers play different roles for different kinds of expressions.

For constant/static strings, the context type qualifier captures the context transition it induces when parsed by our canonical grammar. For example, the constant string `<a href="` is qualified with a $\texttt{STATIC}_{\texttt{PCDATA}\hookrightarrow\texttt{URI\_START}}$ context type qualifier indicating that it parses validly as per our canonical grammar (HTML 5), causing the browser to transition from `PCDATA` context (start of tag) to the `URI_START` context. Unsanitized input variables are by default qualified `UNSAFE`. The type system qualifies them with a $\texttt{STATIC}_{c\hookrightarrow c'}$ context qualifier, where c and c' are contexts, only after the variable is sanitized with a sanitizer matching the context c. The sanitizer ensures that the untrusted input safely renders in context c—we define this correctness property of the sanitizer more precisely in Section 4.2. Variables qualified `UNSAFE` are not permitted to be used in string expressions that are emitted to the output buffer.

When data is emitted to the output buffer, the analysis engine must track which context the output buffer is in. The global output buffer (base type η) is also qualified with a different set of context type qualifiers, which indicate the context it is in at any given point in the program. For instance, the output buffer which is in a `URI_START` context, say just after `<a href="` is written to it, is annotated with a $\texttt{CTXSTAT}_{\texttt{URI\_START}}$ context qualifier. The context-sensitivity property is enforced by matching the context type qualifiers of the output buffer and the string expression being written at each `print` command. For example, when an untrusted variable is emitted to a $\texttt{CTXSTAT}_c$ qualified buffer, it must have the $\texttt{STATIC}_{c\hookrightarrow c'}$ qualifier attached, ensuring that it has been sanitized (or made safe) for rendering in context c.

We formally define the qualifiers in Figure 5. As explained, the type system defines two separate sets of type qualifiers: $\mathcal{Q}_1$ and $\mathcal{Q}_2$. Type qualifiers $\mathcal{Q}_2$ annotate the output buffer, which is an object of base type η, whereas the set $\mathcal{Q}_1$ exclusively qualifies other typed expressions. Type qualifiers of the form $\vec{\mathcal{Q}_1} \rightarrow [Q_2 \rightarrow Q_2]$ are inferred for each template function, which capture the expected qualifier types for the arguments and the template's effect on the output buffer, as explained in further detail in Section 4.2.

Handling Context ambiguity with Flow-Sensitivity. Untrusted inputs may be used in different contexts along different or even the same program paths. This leads to

```
template StatAmb($imgLink, $name)
{
   if ($name == “”) then print (“<img src=\“”);
   else print (“<div>”); fi
   print ($imgLink);
   if ($name == “”) then print (“\” />”);
   else print (“</div>”); fi; return;
}
```

Figure 6: An example of a template with static context ambiguity requiring a mixed static-dynamic approach.

context ambiguity, as explained in Section 2.2. A standard flow-insensitive type inference algorithm would infer that such an untrusted input has no single precise context qualifier because of its ambiguous usage in multiple different contexts. To handle such context ambiguity, we design our type system to be *flow-sensitive*— a flow-sensitive type system permits program variables to have varying type qualifiers at different program locations [16].

Mixed Static-Dynamic Typing. Flow-sensitive typing does address static ambiguity to a large extent, but not all of the cases we observe in practice. Consider a template such as the one in Figure 6. In one branch the program writes `<div>` and in the other it writes `<img src="` to the global output buffer. The context that output buffer is in at the join point is statically ambiguous, and consequently, statically selecting sanitizers on subsequent `print` statements in the template is not possible. Similar examples of static ambiguity have been shown to arise in large legacy applications [43].

Our approach avoids throwing type errors for such static ambiguous types by using the following approach: we further divide the type qualifiers into *statically-qualified* and *dynamically-qualified* sets. Qualifiers $\mathcal{Q}_2$ for the output buffer are either static qualifiers ($\texttt{CTXSTAT}_C$) or dynamic ($\texttt{CTXDYN}_S$). At a given program location, if the output buffer is unambiguously determined to be in a single context c, a static qualifier is attached to it. In contrast, when the embedding context of the buffer is statically ambiguous (or imprecise), as in the example of Figure 6, it is over-approximated by a set of contexts S and is qualified with the dynamic qualifer $\texttt{CTXDYN}_S$. $\texttt{CTXDYN}_S$ signifies that the buffer is in one of the contexts determined by the set S. Sanitizers can be statically selected for statically-qualified objects since their contexts are precisely known. For dynamically-qualified buffers, the context-sensitive runtime parsing (or CSRP) approach is employed—data written to such buffers is parsed and sanitized at runtime.

Qualifiers $\mathcal{Q}_1$ for other program expressions are similarly partitioned into static or dynamic sets—for instance, a string expression used in a `print` statement with a dynamically-qualified output buffer is also dynamically-qualified in our type system using the $\texttt{DYN}_S$ qualifier. The set S is a static over-approximation of the set of context transitions that the string expression can induce. Sanitizer selection can be done statically for statically-qualified (such as $\texttt{STATIC}_{c_1 \hookrightarrow c_2}$) expressions and these sanitizers can be placed during compilation. For dynamically-qualified expressions, however, since the context of the output buffer is known only at runtime, the sanitizer selection is performed by the CSRP approach. Specifically, the CSAS engine inserts additional instrumentation for dynamically-qualified string expressions to keep the untrusted substrings in the expression separate from constant substrings. At runtime, when such an expression is being used in a `print`, it is parsed at runtime as per the dynamically-determined start context and the necessary sanitization primitives are applied to the untrusted substrings. In our evaluation, less than 1% of the expressions were dynamically-qualified; a large majority of the cases do not incur the cost of runtime parsing, enabling our type system to be “mostly static”.

Handling Context Ambiguity for Templates. Static context ambiguity may manifest for template start and end contexts as well. A template may be invoked in multiple starting contexts or may be expected to return in multiple ending contexts. In such cases, our CSAS engine resolves the ambiguity purely statically, by cloning templates. For templates that may start or end in more than one context, the CSAS engine generates multiple versions of the template during compilation, each specializing to handle a specific pair of start and end contexts.

Inferring Placement of Sanitizers. Our engine can insert sanitizers into code in which developers have manually applied some sanitizers (chosen from the sanitization library), without undoing existing sanitization if it is correct. Our type rules require additional sanitizers to only be inserted at `print` statements and at *type promotion* operations. Type promotion operations identify points where expressions need to be converted from `UNSAFE`-qualified types to statically- or dynamically-qualified types. These type promotion commands have the form $v := (Q)e$, where Q is a qualified type which are introduced by the CSAS engine when converting templates into the IR. Note that this design separates the type inference task from the type safety rules — type promotions may be added anywhere in the IR by the type qualifier inference algorithm, as long as the resulting IR conforms to the type rules after inference.

4.2 Static Type Rules

In this section, we define a set of type rules which impose static restrictions `S0` - `S4` to acheive the 3 properties (`CR`, `CSAN` and `NOV`) described in Section 2.2.

The type system is subdivided into two main kinds of typing judgements, one for typing language expressions (Figure 7) and one for typing language commands (Figure 8). In our type rules, Γ denotes the type environment that maps program variables, the output buffer (denoted by the symbol ρ) and template name symbols to qualifiers $\mathcal{Q}$.

In a flow-sensitive type system like ours, type qualifier for variables change from one program location to another. Therefore, typing judgements for the language commands (Figure 8) capture the effects of command execution on type environments and have the form $\Gamma \vdash c \Longrightarrow \Gamma'$. This judgement states that the command c is well-typed under the type environment Γ and its execution changes the type environment Γ to Γ'. The expression typing judgement $\Gamma \vdash e : Q$ is standard: it states that at the given program location, the expression e has a type qualifier Q under the environment Γ. All expressions that are neither statically-qualified nor dynamically-qualified, map to `UNSAFE` in Γ. The set of declared variables $\mathcal{V}$ and a map $\mathcal{LF}$ from statement labels to their enclosing functions are assumed to be pre-computed and available externally.

Defining Sanitizer Correctness. The soundness of our type system relies on the correctness of externally provided sanitizers. To define sanitizer correctness more precisely, we

$$\frac{v \in \mathcal{V} \qquad \{v \mapsto Q\} \in \Gamma}{\Gamma \vdash v : Q}\text{T-VAR} \qquad \frac{\alpha_i \neq \texttt{string} \qquad c \in \mathcal{C}}{\Gamma \vdash \mathbf{const}(i : \alpha_i) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-CONST} \qquad \frac{IsParseValid(s, c_1, c_2)}{\Gamma \vdash \mathbf{const}(s : \texttt{string}) : \texttt{STATIC}_{c_1 \hookrightarrow c_2}}\text{T-CONSTSTR}$$

$$\frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c \hookrightarrow c} \qquad \Gamma \vdash e_2 : \texttt{STATIC}_{c \hookrightarrow c} \qquad c \in \mathcal{C}}{\Gamma \vdash (e_1 : \texttt{bool}) \odot (e_2 : \texttt{bool}) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-BOOL} \qquad \frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c \hookrightarrow c} \qquad \Gamma \vdash e_2 : \texttt{STATIC}_{c \hookrightarrow c} \qquad c \in \mathcal{C}}{\Gamma \vdash (e_1 : \texttt{int}) \oplus (e_2 : \texttt{int}) : \texttt{STATIC}_{c \hookrightarrow c}}\text{T-INT}$$

$$\frac{\Gamma \vdash e_1 : \texttt{STATIC}_{c_1 \hookrightarrow c_2} \qquad \Gamma \vdash e_2 : \texttt{STATIC}_{c_2 \hookrightarrow c_3}}{\Gamma \vdash (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string}) : \texttt{STATIC}_{c_1 \hookrightarrow c_3}}\text{T-STRCAT-STAT} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \texttt{UNSAFE} \\ SanMap(c_1 \hookrightarrow c_2, f) \qquad c_1, c_2 \in \mathcal{C}\end{array}}{\Gamma \vdash \mathbf{San}(f, e) : \texttt{STATIC}_{c_1 \hookrightarrow c_2}}\text{T-SAN}$$

$$\frac{IsParseValid(s, c_1, c_2)}{\Gamma \vdash \mathbf{const}(s : \texttt{string}) : \texttt{DYN}_{\{c_1 \hookrightarrow c_2\}}}\text{T-CSTRDYN} \qquad \frac{\Gamma \vdash e_1 : \texttt{DYN}_{S_1} \qquad \Gamma \vdash e_2 : \texttt{DYN}_{S_2}}{\Gamma \vdash (e_1 : \texttt{string}) \cdot (e_2 : \texttt{string}) : \texttt{DYN}_{S_1 \bowtie S_2}}\text{T-STRCAT-DYN}$$

Figure 7: Type Rules for Expressions.

$$\frac{\Gamma \vdash e : Q \qquad v \in \mathcal{V}}{\Gamma \vdash v := e \Longrightarrow \Gamma[v \mapsto Q]}\text{T-ASSIGN} \qquad \frac{\Gamma \vdash e : Q \qquad Q \leq Q'}{\Gamma \vdash v_1 := (Q')e \Longrightarrow \Gamma[v_1 \mapsto Q']}\text{T-PROM} \qquad \frac{\Gamma_0 \vdash c_1 : \Gamma_1 \qquad \Gamma_1 \vdash S : \Gamma_2}{\Gamma_0 \vdash c_1; S \Longrightarrow \Gamma_2}\text{T-SEQ}$$

$$\frac{\Gamma \vdash e : \texttt{STATIC}_{c_1 \hookrightarrow c_2} \qquad \Gamma \vdash \rho : \texttt{CTXSTAT}_{c_1}}{\Gamma \vdash \mathbf{print}(e) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_{c_2}]}\text{T-PRINT-STATIC-1} \qquad \frac{\begin{array}{c}\Gamma \vdash e : \texttt{DYN}_{S_1} \\ \Gamma \vdash \rho : \texttt{CTXDYN}_{S_2} \qquad |CDom(S_1, \mathcal{C}) \cap S_2| \neq 0\end{array}}{\Gamma \vdash \mathbf{print}(e) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXDYN}_{CRange(S_1, S_2)}]}\text{T-PRINT-DYN-2}$$

$$\frac{\begin{array}{c}\Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad \Gamma \vdash \rho : Q_\rho \\ Q_\rho = \texttt{CTXSTAT}_{c_\rho} \qquad Q_{\rho'} = \texttt{CTXSTAT}_{c_{\rho'}} \qquad c_\rho, c_{\rho'} \in \mathcal{C} \qquad \bigwedge_{i \in \{1 \ldots k\}} (\Gamma \vdash e_i : Q_i) \qquad \bigwedge_{i \in \{1 \ldots k\}} ((Q_i \leq \texttt{STATIC}_{c_i \hookrightarrow c_{i'}}) \wedge (c_i \in \mathcal{C}) \wedge (c_{i'} \in \mathcal{C}))\end{array}}{\Gamma \vdash \mathbf{callTemplate} f(e_1, e_2, \ldots, e_k) \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_{c_{\rho'}}]}\text{T-CALL}$$

$$\frac{\begin{array}{c}\Gamma \vdash \rho : \texttt{CTXSTAT}_c \qquad c \in \mathcal{C} \qquad \{\ell \mapsto f\} \in \mathcal{LF} \\ \Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad Q_{\rho'} = \texttt{CTXSTAT}_c\end{array}}{\Gamma \vdash \ell : \ \mathbf{return;} \Longrightarrow \Gamma}\text{T-RET-STAT} \qquad \frac{\begin{array}{c}\Gamma \vdash \rho : Q \\ c \in \mathcal{C} \qquad Q = \texttt{CTXDYN}_S \qquad |S| = 1 \qquad c \in S \qquad \{\ell \mapsto f\} \in \mathcal{LF} \\ \Gamma \vdash f : (Q_1, Q_2 \ldots Q_k) \to [Q_\rho \to Q_{\rho'}] \qquad Q_{\rho'} = \texttt{CTXSTAT}_c\end{array}}{\Gamma \vdash \ell : \mathbf{return;} \Longrightarrow \Gamma[\rho \mapsto \texttt{CTXSTAT}_c]}\text{T-RET-DYN}$$

$$\frac{\Gamma_0 \vdash S_1 : \Gamma \qquad \Gamma_0 \vdash S_2 : \Gamma}{\Gamma_0 \vdash \mathbf{if}(e)\mathbf{then}S_1\mathbf{else}S_2 \Longrightarrow \Gamma}\text{T-IFELSE} \qquad \frac{\Gamma \vdash S \Longrightarrow \Gamma}{\Gamma \vdash \mathbf{while}(e)S \Longrightarrow \Gamma}\text{T-WHILE}$$

Figure 8: Type Rules for Commands. The output buffer (of base type η) is denoted by the symbol ρ.

reuse the notion of *valid syntactic forms*, formalized by Su et. al. [47]. A sanitizer f is *correct* for a context transition $c_s \hookrightarrow c_e$, if all strings sanitized with f are guaranteed to parse validly starting in context c_s yielding an end context c_e according to our canonical grammar, and if the sentential forms generated during such a parse are valid syntactic forms as per the application's intended security policy [47]. In other words, sanitized strings can span different contexts, but all the intermediate contexts induceded during parsing untrusted strings should be syntactically confined to non-terminals allowed by the application's policy. We assume that a relation *SanMap*, mapping each possible context-transition to a matching sanitizer, is available externally.

S0: No Implicit Type Casts. Our type system separates UNSAFE-qualified, statically-qualified and dynamic-qualified types. It does not permit implicit type conversions between them. Type qualifier conversions are *only* permitted through explicit type promotion operations, according to a *promotibility* relation $\leq$ defined in Figure 9.

$$\frac{}{q \leq q} \qquad \frac{c \in S \qquad S \in 2^{\mathcal{C}}}{\texttt{CTXSTAT}_c \leq \texttt{CTXDYN}_S} \qquad \frac{c_1, c_2 \in \mathcal{C}}{\texttt{UNSAFE} \leq \texttt{STATIC}_{c_1 \hookrightarrow c_2}} \qquad \frac{S \in 2^{\mathcal{C} \times \mathcal{C}}}{\texttt{UNSAFE} \leq \texttt{DYN}_S}$$

Figure 9: The promotibility relation $\leq$ between type qualifiers

Our promotibility relation is different from the standard subtyping relation ($\preceq$)—for example, the following subsumption rule applies in standard subtyping, but our promotibility relation does not adhere to it:

$$\frac{\Gamma \vdash e : Q_s \qquad Q_s \preceq Q_t}{\Gamma \vdash e : Q_t}\text{T-SUB}$$

The static type qualifier-based restrictions S1 and S3 defined below together satisfy the no over-sanitization (NOS) property. Similarly, S2 ensures the context restriction (CR) property. The S3 and S4 together satisfy the context-sensitivity (CSAN) property while maintaining strict separation between dynamically-qualified and statically-qualified expressions.

S1: No Sanitization for Constants. The rules T-CONST, T-CONSTSTR and T-CSTRDYN show that constant string values acquire the type qualifier without any sanitization. These values are program constants, so they are implicitly trusted.

S2: Canonical Parsing. The qualifier parameters (denoting the context-transitions) for trusted constant strings are inferred by parsing them according to the canonical grammar. We assume the availability of such a canonical grammar (assumption 1 in Section 2.3), embodied in a predicate *IsParseValid* defined below.

Definition 1. *IsParseValid is a predicate of type* `string` $\times \mathcal{C} \times \mathcal{C} \to$ `bool`, *such that* $IsParseValid(s, c_1, c_2)$ *evaluates to true if and only if the data string* s *parses validly as per the assumed canonical grammar starting in context* c_1 *yielding a final context* c_2.

S3: Safe String Expression Creation. The rules for concatenation do not permit strings qualified as UNSAFE to be used in concatenations, forcing the type inference engine to type promote (and hence sanitize) operands *before* they can be used in concatenation opertions. The T-STRCAT-STAT rule ensures that only statically safe strings can be concatenated whereas the T-STRCAT-DYN rule constructs dynamically qualified strings. The latter rule conservatively over-approximates the result's dynamic set of context-transitions that could occur at runtime. For over-approximating sets, we define an inner-join $S_1 \bowtie S_2$ as the set of all context transitions $c_1 \hookrightarrow c_2$ such that $c_1 \hookrightarrow c_3 \in S_1$ and $c_3 \hookrightarrow c_2 \in S_2$.

`S4`: **Context-Sensitive Output.** The rules for print commands ensure that the emitted string expression can not be `UNSAFE`-qualified. Further, the type rule `T-PRINT-STATIC-1` ensures that the context type qualifier of the emitted string matches the context of the output buffer, when both of them are statically-qualified.

Only dynamically-qualified strings can be emitted to dynamically qualified output buffers—a strict separation between dynamic and static type qualified expressions is maintained. The `T-PRINT-DYN-2` type rule capture this case. This requires a runtime parsing, as described in section 4.3, to determine the precise context. The static type rules compute the resulting context for the output buffer by an over-approximate set, considering the context-transition sets of two dynamically-qualified input operands. To compute the resulting context set, we define 2 operations over a context-transition set S for a dynamically qualified type $\mathtt{DYN}_S$:

$$CDom(S, E) = \{C_i | C_i \hookrightarrow C_e \in S, C_e \in E\}$$

$$CRange(S, B) = \{C_i | C_s \hookrightarrow C_i \in S, C_s \in B\}$$

Control flow Commands. Type rules `T-IFELSE` and `T-WHILE` for control flow operations are standard, ensuring that the type environment Γ resulting at join points is consistent. Whenever static context ambiguity arises at a join point, the types of the incoming values must be promoted to dynamically-qualified type to conform to the type rules. Our type inference step (as Section 5.1 explains) introduces these type promotions at join points in the untyped IR, so that after type inference completes, the well-typed IR adheres to the `T-IFELSE` and `T-WHILE` rules.

Calls and Returns. In our language, templates do not return values but take in parameters passed by value. In addition, templates have side-effects on the global output buffer. For a template f, Γ maps f by name to a type $(Q_1, Q_2, \ldots Q_k) \rightarrow [Q_\rho \rightarrow Q_{\rho'}]$, where $(Q_1, Q_2, \ldots Q_k)$ denotes the expected types of its arguments and $Q_\rho \rightarrow Q_{\rho'}$ denotes the side-effect of f on the global output buffer ρ. The `T-CALL` rule imposes several restrictions.

First, it enforces that each formal parameter either has a statically-qualified type or is promotible to one (by relation $\leq$). Second, it ensures that the types of actual parameters and the corresponding formal parameters match. Finally, it enforces that each (possibly cloned) template starts and ends in statically precise contexts, by ensuring that Q_ρ and $Q_{\rho'}$ are statically-qualified. The output buffer (ρ) can become dynamically qualified within a template's body, as shown in example of Figure 6, but the context of ρ should be precisely known at the return statement. In the example of Figure 6, the context of ρ is ambiguous at the join-point of the first if-else block. However, we point out that at the return statement the dynamically qualified set of contexts becomes a singleton, that is, the end context is precisely known. The `T-RET-DYN` rule applies in such cases and soundly converts the qualifier for ρ back to a statically-qualified type.

For templates that do not start and end in precise contexts, our CSAS engine creates multiple clones of the template, as explained in Section 5.1.3, to force conformance to the type rules.

4.3 Sanitization

Handling manually placed sanitizers. The `T-SAN` rule converts the type of the expression e in the sanitization expression $San(f, e)$ from `UNSAFE` to a statically-qualified type $\mathtt{STATIC}_{c_1 \hookrightarrow c_2}$, only if f is a correct sanitizer for the context transition $c_1 \hookrightarrow c_2$ according to the externally specified $SanMap$ relation.

Auto-sanitization Only at Type Promotions. Other than `T-SAN`, the `T-PROM` type rule is the only way an `UNSAFE`-qualified string can become statically-qualified. The CSAS engine inserts statically selected sanitizers during compilation only at the type promotion command that promote `UNSAFE`-qualified to statically-qualified strings. For such a command $v := (\mathtt{STATIC}_{c_1 \hookrightarrow c_2})e$, the CSAS engine's compilation step automatically inserts the sanitizer which matches the start context c_1 and will ensure that parsing v will safely end in context c_2 .

Type Promotion from `UNSAFE` to Dynamic. For dynamically qualified strings, the CSAS engine needs to perform runtime parsing and sanitization. To enable this for dynamically-qualified strings, our instrumentation uses an auxiliary data structure, which we call the CSRP-expression, which keeps constant substrings separate from the untrusted components. For conceptual simplicity, our CSRP-expression data structure is simply a string in which untrusted substrings are delimited by special characters $\llparenthesis\rrparenthesis$. These special delimiters are not part of the string alphabet of base templating language.

The `T-PROM` rule permits promotions from `UNSAFE`-qualified strings to dynamically-qualified expressions. The CSAS engine inserts instrumentation during compilation to insert the special characters $\llparenthesis\rrparenthesis$ around the untrusted data and to initialize this CSRP-expression with it. The concatenation operation over regular strings naturally extends to CSRP-expressions.

Runtime Parsing and Sanitization. At program points where the output buffer is dynamically-qualified, the CSAS engine adds instrumentation to track its *dynamic context* as a metadata field. The metadata field is updated at each `print`. When a CSRP-expression is written to the output buffer at runtime, the CSRP-expression is parsed starting in the dynamically-tracked context of the output buffer. This parsing procedure internally determines the start and end context of each untrusted substring delimited by $\llparenthesis\rrparenthesis$, and selects sanitizers for them context-sensitively.

We detail the operational semantics for the language and sketch the soundness proof for our type system in the appendix A.

5. CSAS ENGINE

We present the design and implementation of the CSAS engine in this section. The CSAS engine performs two main steps of inferring context type qualifiers and then compiling well-typed IR to JavaScript or server-side Java code with sanitization logic.

5.1 Type Qualifier Inference & Compilation

The goal of the type inference step is to convert untyped or vanilla templates to well-typed IR. In the the qualifier inference step, the CSAS engine first converts template code to an internal SSA representation (untyped IR). The qualifier inference sub-engine is also supposed to add additional type promotions for untrusted inputs, where sanitization primitives will eventually be placed. However, the qualifier inference sub-engine does not *apriori* know where all sanitizations will be needed. To solve this issue, it inserts a set of candidate type promotions, only some of which will be compiled into sanitizers. These candidate type promotions include *type qualifier variables*, i.e., variables whose values are context types and are to be determined by the type inference. They have the form $v' := (Q)e$ where Q is a type qualifier variable, and its exact value is a context type to be determined by the type qualifier inference sub-engine. Next, the type qualifier inference step solves for these qualifier variables by generating type constraints and solving them.

Once constraint solving succeeds, the concrete context type for each qualifier variable is known. These context types can be substituted into the candidate type promotions; the resulting IR is well-typed and is guaranteed to conform to our type rules. In the final compilation step, only some of the candidate type promotions are turned into sanitizer calls. Specifically, type promotions in well-typed IR that essentially cast from a qualified-type to itself, are redundant and don't require any sanitization, whereas those which cast `UNSAFE`-qualified variables into other qualified values are compiled into sanitizers as described in section 4.3.

5.1.1 Inserting Type Promotions with Qualifer Variables

Candidate type promotions are introduced at the following points while converting templates to the untyped IR:

- Each `print` (e) statement is turned into a `print` (v') statement in the IR by creating a fresh internal program variable v'. The CSAS engine also inserts a type promotion (and assignment) statement $v' := (Q)\ e$ preceeding the print statement, creating a qualifier variable Q.
- Each $v = \phi(v_1, v_2)$ statement is turned into equivalent type promotions $v := (Q_1)\ v_1$ and $v := (Q_2)\ v_2$ in the respective branches before the join point, by creating new qualifier variables Q_1 and Q_2 .
- Parameter marshalling from actual parameter "a" to formal parameter "v" is made explicit via a candidate promotion operation $v := (Q)\ a$, by creating new qualifier variable Q.
- A similar type promotion is inserted before the concatenation of a constant string expression with another string expression.

5.1.2 Constraint Solving for Qualifier Variables

The goal of this step is to infer context type qualifiers for qualifier variables. We analyze each template's IR starting with templates that are used by external code— we call these public templates. We generate a version of compiled code for each start and end context in which a template can be invoked, so we try to analyze each public template for each choice of a start and end context. Given a template T, start context c_s and end context c_e, the generic type inference procedure called $TempAnalyze(T, c_s, c_e)$ is described below.

$TempAnalyze(T, c_s, c_e)$ either succeeds having found a satisfying assignment of qualifier variables to context type qualifiers, or it fails if no such assignment is found. It operates over a call-graph of the templates in depth-first fashion starting with T, memoizing the start and end contexts for each template it analyzes in the process. When analyzing the body of a template in IR form, it associates a typemap $\mathcal{L}$ mapping local variables to type qualifiers at each program location. At the start of the inference for T, all local variables are qualified as `UNSAFE` in $\mathcal{L}$. The analysis proceeds from the entry to the exit of the template body statement by statement, updating the context qualifier of each program variable. The context of the output buffer is also updated with the analysis of each statement.

Type rules defined in Figure 8 can be viewed as inference rules as well: for each statement or command in the conclusion of a rule, the premises are type constraints to be satisfied. Similar constraints are implied by type rules for expressions. Our type inference generates and solves these type constraints during the statement by statement analysis using a custom constraint solving procedure.

Several of our type rules are non-deterministic. As an example, the rules `T-CONSTSTR` and `T-CSTRDYN` have identical premises and are non-deterministic because the language syntax alone is insufficient to separate statically and dynamically qualified types. Our constraint solving procedure resolves such non-determinism by backtracking to find a satisfying solution to the constraints. Our inference prefers the most precise (or static) qualifiers over less precise (dynamic) qualifiers as solutions for all qualifier variables during its backtracking-based constraint solving procedure. For instance, consider the non-determinism inherent in the premise involving $IsParseValid$ used in the `T-CONSTSTR` and `T-CSTRDYN` rules. $IsParseValid$ is a one-to-many relation and a constant string may parse validly in many start contexts. Our constraint solving procedure non-deterministically picks one such possible context transition initially, trying to satisfy all instances of the `T-CONSTSTR` rule before that of the `T-CSTRDYN` rule and refines its choice until it finds a context transition under which the static string parses validly. If no instance of the `T-CONSTSTR` rule matches, the engine tries to satisfy the `T-CSTRDYN` rule. Similar, backtracking is also needed when analyzing starting and ending contexts of templates when called via the `callTemplate` operation.

5.1.3 Resolving Context Ambiguity by Cloning

The static typing `T-CALL` rule for `callTemplate` has stringent pre-conditions: it permits a unique start and end context for each template. A templates can be invoked in multiple different start (or end) contexts—our inference handles such cases while keeping the consistency with the type rules by cloning templates. We memoize start and end contexts inferred for each template during the inference analysis. If during constraint generation and solving, we find that a template T is being invoked in start and end contexts different from the ones inferred for T previously during the inference, we create a clone T'. The cloned template has the same body but expects to begin and end in a different start and end context. Cloned templates are also compiled to separate functions and the calls are directed to the appropriate functions based on the start and end contexts.

6. IMPLEMENTATION & EVALUATION

We have implemented our CSAS engine design into a state-of-the-art, commercially used open-source templating framework called Google Closure Templates [45]. Closure Templates are used extensively in large web applications in-

Contexts
HTML PCDATA
HTML RCDATA
HTML TAGNAME
HTML ATTRIBNAME
QUOTED HTMLATTRIB
UNQUOTED HTMLATTRIB
JS STRING
JS REGEX
CSS ID, CLASS, PROPNAME, KEYWDVAL, QUANT
CSS STRING, CSS QUOTED URL, CSS UNQUOTED URL
URL START, URL QUERY, URL GENERAL

Figure 10: A set of contexts $\mathcal{C}$ used throughout the paper.

cluding Gmail, Google Docs and other Google properties. Our auto-sanitized Closure Templates can be compiled both into JavaScript as well as server-side Java code, enabling building reusable output generation elements.

Our implementation is in 3045 lines of Java code, excluding comments and blank lines, and it augments the existing compiler in the Closure Templates with our CSAS engine. All the contexts defined in Figure 10 of the appendix are supported in the implementation with 20 distinct sanitizers.

Subject Benchmarks. For real-world evaluation, we gathered all Closure templates accessible to us. Our benchmarks consist of 1035 distinct Closure templates from Google's commericially deployed applications. The templates were authored by developers prior to our CSAS engine implementation. Therefore, we believe that these examples represent unbiased samples of existing code written in templating languages.

The total amount of code in the templates (excluding file prologues and comments outside the templates) is $21,098$ LOC. Our benchmarks make heavy use of control flow constructs such as `callTemplate` calls. Our benchmark's template call-graph is densely connected. It consists of 1035 nodes, 2997 call edges and 32 connected components of size ranging from 2 - 12 templates and one large component with 633 templates. Overall, these templates have a total of 1224 print statements which write untrusted data expressions. The total number of untrusted input variables in the code base is 600, ranging from $0-13$ for different templates. A small ratio of untrusted inputs to untrusted `print` shows that untrusted inputs are used in multiple output expressions, which are one of the main reasons for context ambiguity that motivate our flow-sensitive design.

Evaluation Goals. The goal of our evaluation is to measure how easily our principled type-based approach retrofits to an existing code base. In addition, we compare the security and performance of our "mostly static", context-sensitive approach to the following alternative approaches:

- *No Auto-Sanitization.* This is the predominant strategy in today's web frameworks.
- *Context-insensitive sanitization.* Most remaining web frameworks supplement each output `print` command with the same sanitizer.
- *Context-sensitive runtime parsing sanitization.* As explained earlier, previous systems have proposed determining the contexts by runtime parsing [5]. We compare the performance of our approach against this approach.

6.1 Compatibility & Precision

Our benchmark code was developed prior to our type system. We aim to evaluate the extent to which our approach can retrofit security to existing code templates. To perform this experiment, we disabled all sanitization checks in the benchmarks that may have been previously applied and enabled our auto-sanitization on all of the 1035 templates. We counted what fraction of the templates that were transformed to well-typed compiled code. Our analysis is implemented in Java and takes 1.3 seconds for all the 1035 benchmarks on a platform with 2 GB of RAM, an Intel 2.6 MHz dual-core processor running Linux 2.6.31.

Our static type inference approach avoids imprecision by cloning templates that are called in more than one context. In our analysis, 11 templates required cloning which resulted in increasing the output **print** statements (or sinks) from 1224 initially to 1348 after cloning.

Our main result is that all 1348 output sinks in the 1035 templates were auto-sanitized. No change or annotations to the vanilla templates were required. We test the outputs of the compiled templates by running them under multiple inputs. The output of the templates under our testing was unimpacted and remained completely compatible with that of the vanilla template code.

Our vanilla templates, being commercially deployed, have existing sanitizers manually applied by developers and are well-audited for security by Google. To confirm our compatibility and correctness, we compared the sanitizers applied by our CSAS engine to the those pre-applied in the vanilla versions of the benchmarked code manually by developers. Out of the 1348 print statements emitting untrusted expressions, the sanitization primitives on untrusted inputs exactly match the pre-applied sanitizers in all but 21 cases. In these 21 cases, our CSAS engine applies a more accurate (`escapeHtmlAttribute`) sanitizer versus the more restrictive sanitizer applied previously (`escapeHTML`) by the developer. Both sanitizers defeat scripting attacks; the pre-existing sanitizer was overly restrictive rendering certain characers inert that weren't dangerous for the context. This evaluation strengthens our confidence that our approach does not impact/alter the compatibility of the HTML output, and that our CSAS engine implementation applies sanitization correctly.

Our type qualifier inference on this benchmark statically-qualified expressions written to all but 9 out of the 1348 sinks. That is, for over 99% of the output sinks, our approach can statically determine a single, precise context.

In these 9 cases, the set of ambiguous contexts is small and a single sanitizer that sanitizes the untrusted input for all contexts in this set can be applied. In our present implementation, we have special-cased for such cases by applying a static sanitizer, which is safe but may be over-restrictive. We have recently implemented the CSRP scheme using an auxiliary data structure, as described in Section 4.3, in jQuery templates for JavaScript [28]; we expect porting this implementation to the Google Closure compiler to be a straightforward task in the future.

6.2 Security

To measure the security offered by our approach as compared to the context-insensitive sanitization approach, we count the number of sinks that would be auto-sanitized correctly in our 1035 templates. We assume that a context-insensitive sanitization would supplement the HTML-entity encoding sanitizer to all sinks, which is the approach adopted in popular frameworks such as Django [14]. Picking another sanitizer would only give worse results for the context-

	No Saniti-zation	Context-Insensitive	Context-Sensitive Runtime Parsing	Our Approach
Chrome 9.0	227	234 (3.0%)	406 (78.8%)	234 (3.0%)
FF 3.6	395	433 (9.6%)	2074 (425%)	433 (9.6%)
Safari 5.0	190	195 (2.5%)	550 (189%)	196 (3.1%)
Server:Java	431	431 (0.0%)	2972 (510%)	431 (0.0%)
# of Sinks Auto-Prot.	0/ 1348 (0%)	982 / 1348 (72%)	1348 / 1348 (100%)	1348 / 1348 (100%)

Figure 11: Comparing the security and runtime overhead (ms.) comparison between our approach and alternative existing approaches for server-side Java and client-side JavaScript code generated from our benchmarks. The last line shows the number of sinks auto-protected by each approach.

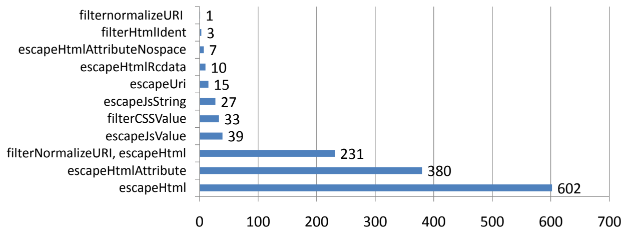


Figure 12: Distribution of inserted sanitizers: inferred contexts and hence the inserted sanitizer counts vary largely, therefore showing that context-insenstive sanitization is insufficient.

insensitive scheme— we show that the most widely inserted sanitizer in auto-sanitization on our benchmarks is also `escapeHtml`, the HTML-entity encoding sanitizer.

The last row in Figure 11 shows the number of sinks auto-protected by existing approaches. Context-insensitive sanitization protects 72% of the total output prints adequately; the auto-sanitization is insufficient for the remaining 28% output `print` opertions. Clearly, context-insensitive sanitization offers better protection than no sanitization strategy. On the other hand, context-sensitive sanitization has full protection whether the context-inference is performed dynamically or as in our static type inference approach. Figure 12 shows that the inferred sanitizers varied significantly based on context across the 1348 output points, showing the inadequacy of context-insensitive sanitization.

6.3 Performance

We measure and compare the runtime overhead incurred by our context-sensitive auto-sanitization to other approaches and present the results in Figure 11. Google Closure Templates can be compiled both to JavaScript as well as Java. We measure the runtime overhead for both cases. For compiled JavaScript functions, we record the time across 10 trial runs in 3 major web browsers. For compiled Java functions, we record the time across 10 trial runs under the same inputs.

The baseline "no auto-sanitization" approach overhead is obtained by compiling vanilla templates with no developer's manual sanitizers applied. For our approach, we enable our CSAS auto-sanitization implementation. To compare the overhead of context-insensitive auto-sanitization, we simply augment all output points with the `escapeHtml` sanitizer during compilation. A direct comparison to Google AutoEscape, the only context-sensitive sanitization approach in templating systems we know of, was not possible because it does not handle rich language features like if-else and loops which create context ambiguities and are pervasive in our benchmarks; a detailed explanation is provided in Section 7. To emulate the purely context-sensitive runtime parsing (CSRP) approach, we implemented this technique for our templating langauge. For Java, we directly used an off-the-shelf parser without modifications from the open-source Google AutoEscape implementation in GWT [15]. For JavaScript, since no similar parser was available, we created a parser implementation mirroring the Java-based parser. We believe our implementation was close to the GWT's public implementation for Java, since the overhead is in the same ballpark range.

Results. For JavaScript as the compilation target, the time taken for parsing and rendering the output of all the compiled template output (total 782.584 KB) in 3 major web browsers, averaged over 10 runs, is shown in Figure 11. The costs lie between 78% and $4.24x$ for the pure CSRP approach and our approach incurs between $3-9.6\%$ overhead over no sanitization. The primary reason for the difference between our approach and CSRP approach is that the latter requires a parsing of all constant string and context determination of untrusted data at runtime— a large saving in our static type inference approach. Our overhead in JavaScript is due to the application of the sanitizer, which is why our sanitization has nearly the same overhead as the context-insensitive sanitization approach.

For Java, the pure CSRP approach has a 510% overhead, whereas our approach and context-insensitive approach incur no statistically discernable overhead. In summary, our approach achieves the benefits of context-sensitive sanitization at the overhead comparable to a large fraction of other widely used frameworks.

We point out that Closure templates capture the HTML output logic with minimal subsidiary application logic — therefore our benchmarks are heavy in string concatenations and writes to output buffers. As a result, our benchmarks are highly CPU intensive and the runtime costs evaluated here may be amortized in full-blown applications by other latencies (computation of other logic, database accesses, network and file-system operations). For an estimate, XSS-GUARD reports an overhead up to 42% for the CSRP approach [5]. We believe our benchmarks are apt for precisely measuring performance costs of the HTML output logic alone. Further performance optimizations can be achieved for our approach as done in GWT by orthogonal optimizations like caching which mask disk load latencies.

7. RELATED WORK

Google AutoEscape, the only other context-sensitive sanitization approach in templating frameworks we are aware of, does not handle the rich language constructs we support— it does not handle conditionals constructs, loops or call operations [3]. It provides safety in straight-line template code for which straight-line parsing and context-determination suffice. To improve performace, it caches templates and the sanitization requirements for untrusted inputs. Templates can then be included in Java code [15] and C code [3]. As we outline in this paper, with rich constructs, path-sensitivity becomes a challenging issue and sanitization requirements for untrusted inputs vary from one execution path to the other. AutoEscape's caching optimization does not directly extend to code where sanitization requirements vary depending on executed paths. Our approach, instead, solves the challenges arising from complex language features representative of richer templating systems like Closure Templates.

Context-inference and subsequent context-sensitive placement for .NET legacy applications is proposed in our recent work [43]. The approach proposed therein, though sound, is a per-path analysis and relies on achieving path coverage by dynamic testing. In contrast, the type-based approach in this work achieves full coverage since it is based on static type inference. The performance improvements in our recent dynamic approach relies heavily on the intuition that on most execution paths, developers have manually applied context-sensitive sanitization correctly. The type-based approach in this work can apply sanitization correctly in code completely lacking previous developer-supplied sanitization. A potential drawback of our static approach is that theoretically it may reject benign templates since it reasons about all paths, even those which may be potentially infeasible. In our present evaluation we have not seen such cases.

Analysis techniques for finding scripting vulnerabilities has been widely researched [1, 2, 6, 18, 24, 27, 31–33, 37, 41, 42, 51, 52]. Defense architectures have targeted three broad categories: server-side techniques [5, 32, 43, 49, 52], purely browser-based techniques [4, 35] and client-server collaborative defenses [19, 26, 36, 46]. Unlike browser-based and client-server defenses, purely server-side approaches are applicable to the server code without requiring modifications to web browsers. Our techniques are an example of this fact.

Among server-side approaches, strong typing has been proposed as a XSS defense mechanism in the work by Robertson et. al [39]. Our approach significantly contrasts theirs in that it does not require any annotations or changes to the existing code, does not rely on strong typing primitives in the base language such as monads and is a mixed static-dynamic type system for existing web templating frameworks and for retrofitting to existing code.

8. CONCLUSIONS

We present a new auto-sanitization defense to secure web application code from scripting attacks (such as XSS) by construction. We introduce context type qualifiers, a key new abstraction, and develop a type system which is directly applicable to today's commercial templating languages. We have implemented the defense in Google Closure Templates, a state-of-the-art templating system that powers GMail and Google Docs. We find that our mostly static system has low performance overheads, is precise and requires no additional annotations or developer effort. We hope that our abstractions and techniques can be extended to other complex languages and frameworks in the future towards the goal of eliminating scripting attacks in emerging web applications.

9. ACKNOWLEDGEMENTS

We thank Avik Chaudhuri, David Wagner, Joel Weinberger, Devdatta Akhawe, Stephen McCamant, Adrian Mettler, Pongsin Poosankam and the anonymous reviewers for their insightful comments to improve this manuscript. This material is based upon work partially supported by the NSF under Grants No. 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0831501, and 0424422, by the AFRL under Grant No. P010071555, by the ONR under MURI Grant No. N000140911081, and by the MURI program under AFOSR Grants No. FA9550-08-1- 0352 and FA9550-09-1-0539. The second author is also supported by the Symantec Research Labs Graduate Fellowship. The third author conducted part of the work as a visiting faculty at Google.

10. REFERENCES

[1] 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.

[2] S. Bandhakavi, S. T. King, P. Madhusudan, and M. Winslett. VEX: Vetting browser extensions for security vulnerabilities, 2010.

[3] Google autoescape implementation for ctemplate (c code). http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html.

[4] 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*, WWW '10, 2010.

[5] 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.

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

[7] Google Analytics XSS vulnerability. http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/.

[8] Google XSS Flaw in Website Optimizer Scripts explained. http://www.acunetix.com/blog/web-security-zone/articles/google-xss-website-optimizer-scripts/.

[9] How I met your girlfriend, DEFCON'10. ohack.us/xss/2010-defcon.ppt.

[10] XSS Attack Identified and Patch-Twitter. http://status.twitter.com/post/1161435117/xss-attackłinebreak-identified-and-patched.

[11] ClearSilver: Template Filters. http://www.clearsilver.net/docs/man_filters.hdf.

[12] CodeIgniter/system/libraries/Security.php. https://bitbucket.org/ellislab/codeigniter/src/8af0fb079f90/system/libraries/Security.php.

[13] Ctemplate: Guide to Using Auto Escape. http://google-ctemplate.googlecode.com/svn/trunk/doc/auto_escape.html.

[14] django: Built-in template tags and filters. http://docs.djangoproject.com/en/dev/ref/templates/builtins.

[15] Google autoescape implementation for gwt (java code). http://code.google.com/p/google-web-toolkit/source/browse/tools/lib/streamhtmlparser/streamhtmlparser-jsilver-r10/streamhtmlparser-jsilver-r10-1.5.jar.

[16] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, 2002.

[17] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Towards secure embedded web interfaces. In *Proceedings of the Usenix Security Symposium*, 2011.

[18] 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*, WWW '09.

[19] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. *16th Annual Network & Distributed System Security Symposium*, 2009.

[20] Google Web Toolkit: Developer's Guide – SafeHtml. http://code.google.com/webtoolkit/doc/latest/DevGuideSecuritySafeHtml.html.

[21] R. Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html.

[22] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the Usenix Security Symposium*, 2011.

[23] HTML Purifier : Standards-Compliant HTML Filtering. http://htmlpurifier.org/.

[24] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04.

[25] JiftyManual. http://jifty.org/view/JiftyManual.

[26] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. *16th International World World Web Conference*, 2007.

[27] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[28] Quasis demo - javascript shell 1.4. http://js-quasis-libraries-and-repl.googlecode.com/svn/trunk/index.html.

[29] 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.

[30] kses - PHP HTML/XHTML filter. http://sourceforge.net/projects/kses/.

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

[32] B. Livshits, M. Martin, and M. S. Lam. SecuriFly: Runtime protection and recovery from Web application vulnerabilities. Technical report, Stanford University, Sept. 2006.

[33] M. Martin and M. S. Lam. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *17th USENIX Security Symposium*, 2008.

[34] The Mason Book: Escaping Substitutions. http://www.masonbook.com/book/chapter-2.mhtml.

[35] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

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

[37] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. *20th IFIP International Information Security Conference*, 2005.

[38] XSS Prevention Cheat Sheet. http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[39] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[40] Ruby on Rails Security Guide. http://guides.rubyonrails.org/security.html.

[41] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, 2010.

[42] P. Saxena, S. Hanna, P. Poosankam, and D. Song. FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *17th Annual Network & Distributed System Security Symposium, (NDSS)*, 2010.

[43] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the ACM Computer and communications security(CCS)*, 2011.

[44] Smarty Template Engine: escape. http://www.smarty.net/manual/en/language.modifier.escape.php.

[45] Google Closure Templates. http://code.google.com/closure/templates/.

[46] S. Stamm. Content security policy, 2009.

[47] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. 2006.

[48] Template::Manual::Filters. http://template-toolkit.org/docs/manual/Filters.html.

[49] 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.

[50] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song. A Systematic Analysis of XSS Sanitization in Web Application Frameworks. In *Proceedings of the European Symposium on Research in Computer Security*, 2011.

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

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

[53] Yii Framework: Security. http://www.yiiframework.com/doc/guide/1.1/en/topics.security.

[54] Zend Framework: Zend_Filter. http://framework.zend.com/manual/en/zend.filter.set.html.

APPENDIX

A. OPERATIONAL SEMANTICS

We have discussed the static type rules in Section 4. In this section, we describe the various runtime parsing checks that our CSAS engine inserts at various operations to achieve type safety. We do this by first presenting a big-step operational semantics for an abstract machine that evaluates our simple templating language. We sketch the proof for the soundness of our type system based on the operational semantics.

$$\frac{c \in \mathcal{C}}{M \vdash \mathbf{const}(n) \Downarrow \langle n, c \hookrightarrow c\rangle}\,\text{E-CINT} \qquad \frac{c \in \mathcal{C}}{M \vdash \mathbf{const}(b) \Downarrow \langle b, c \hookrightarrow c\rangle}\,\text{E-CBOOL}$$

$$\frac{IsParseValid(s, c_1, c_2)}{M \vdash \mathbf{const}(s) \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle}\,\text{E-CONST} \qquad \frac{}{M \vdash \mathbf{const}(s) \Downarrow \lhd s \rhd}\,\text{E-STRING-DYN}$$

$$\frac{\begin{array}{c} M \vdash e_1 \Downarrow \langle b_1, c \hookrightarrow c\rangle \\ M \vdash e_2 \Downarrow \langle b_2, c \hookrightarrow c\rangle \end{array}}{M \vdash e_1 \odot e_2 \Downarrow \langle b_1 \odot b_2, c \hookrightarrow c\rangle}\,\text{E-BL} \qquad \frac{\begin{array}{c} M \vdash e_1 \Downarrow \langle n_1, c \hookrightarrow c\rangle \\ M \vdash e_2 \Downarrow \langle n_2, c \hookrightarrow c\rangle \end{array}}{M \vdash e_1 \oplus e_2 \Downarrow \langle n_1 \oplus n_2, c \hookrightarrow c\rangle}\,\text{E-INT}$$

$$\frac{M \vdash e_1 \Downarrow \langle s_1, c_1 \hookrightarrow c_2\rangle \qquad M \vdash e_2 \Downarrow \langle s_2, c_3 \hookrightarrow c_4\rangle \qquad c_2 = c_3}{M \vdash e_1 \cdot e_2 \Downarrow \langle s_1 \cdot s_2, c_1 \hookrightarrow c_4\rangle}\,\text{E-CAT-STAT}$$

$$\frac{M \vdash e_1 \Downarrow \lhd s_1 \rhd \qquad M \vdash e_2 \Downarrow \lhd s_2 \rhd}{M \vdash e_1 \cdot e_2 \Downarrow \lhd s_1 \cdot s_2 \rhd}\,\text{E-CAT-DYN}$$

$$\frac{M \vdash e \Downarrow s \qquad SanMap(c_1 \hookrightarrow c_2, f) \qquad s' = f(s) \qquad c_1, c_2 \in \mathcal{C}}{M \vdash \mathbf{San}(f, e) \Downarrow \langle s', c_1 \hookrightarrow c_2\rangle}\,\text{E-SAN}$$

$$\frac{M \vdash e \Downarrow s \qquad SanMap(c_1 \hookrightarrow c_2, f) \qquad \underline{s' = f(s)} \qquad c_1, c_2 \in \mathcal{C}}{M \vdash v := (\texttt{STATIC}_{c_1 \hookrightarrow c_2})e \Downarrow M[v \mapsto \langle s', c_1 \hookrightarrow c_2\rangle]}\,\text{E-PROM-ST}$$

$$\frac{M \vdash e \Downarrow s}{M \vdash v := (\texttt{DYN}_{\mathcal{S}})e \Downarrow M[v \mapsto \lhd (\!| s |\!) \rhd\rangle]}\,\text{E-PROM-DYN}$$

$$\frac{M \vdash e \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle}{M \vdash v := (\texttt{STATIC}_{c_1 \hookrightarrow c_2})e \Downarrow M}\,\text{E-CAST-1} \qquad \frac{M \vdash e \Downarrow \lhd s \rhd}{M \vdash v = (\texttt{DYN}_{\mathcal{S}})e \Downarrow M}\,\text{E-CAST-2}$$

$$\frac{M \vdash e \Downarrow \langle s, c_1 \hookrightarrow c_2\rangle \qquad M \vdash \rho \Downarrow \| s', c \| \qquad c = c_1}{M \vdash print(e) \Downarrow M[\rho \mapsto \| s' \cdot s, c_2 \|]}\,\text{E-PRN-STAT}$$

$$\frac{M \vdash e \Downarrow \lhd s \rhd \qquad M \vdash \rho \Downarrow \| s', c \| \qquad \boxed{(s'', c_2) = CSRP(c, \lhd s \rhd)}}{M \vdash print(e) \Downarrow M[\rho \mapsto \| s' \cdot s'', c_2 \|]}\,\text{E-PRN-DYN}$$

$$\frac{M \vdash e \Downarrow true \qquad M \vdash S_1 \Downarrow M'}{M \vdash \mathbf{if}(e)\mathbf{then}S_1\mathbf{else}S_2 \Downarrow M'}\,\text{E-IF} \qquad \frac{M \vdash e \Downarrow false \qquad M \vdash S_2 \Downarrow M'}{M \vdash \mathbf{if}(e)\mathbf{then}S_1\mathbf{else}S_2 \Downarrow M'}\,\text{E-EL}$$

$$\frac{\begin{array}{c} M \vdash e \Downarrow true \\ M \vdash S; \mathbf{while}(e)S \Downarrow M' \end{array}}{M \vdash \mathbf{while}(e)S \Downarrow M'}\,\text{E-WHLTRUE} \qquad \frac{M \vdash e \Downarrow false}{M \vdash \mathbf{while}(e)S \Downarrow M}\,\text{E-WHLFALSE}$$

$$\frac{M \vdash e \Downarrow x}{M \vdash v := e \Downarrow M[v \mapsto x]}\,\text{E-ASSIGN} \qquad \frac{M \vdash c \Downarrow M' \qquad M' \vdash S \Downarrow M''}{M \vdash c; S \Downarrow M''}\,\text{E-SEQ}$$

Figure 13: Operational Semantics for the typed language. Sanitization routines inserted by the CSAS engine after inference are shown Underlined. Runtime parsing and other checks inserted are shown $\boxed{\textit{Boxed}}$, which may produce runtime errors evaluating to `CFail`.

The evaluation rules are shown in Fig 13. Commands operate on a memory M mapping program variables to *values*. Each premise in an evaluation rule has the form $M \vdash e \Downarrow v$ which means that the expression e evaluates to a final value v under the state of the memory M. Command evaluation judgements have the form $M \vdash c \Downarrow M'$ which states that under the memory M evaluation of the command c results in a memory M'. We omit the rules for template calls and returns here which follow standard call-by-value semantics for brevity.

Values. The values produced during the evaluation of the language are mainly of two kinds: (a) V_β values for the data elements of base type β and (b) V_η for objects of base type η. Runtime errors are captured by a third, explicit `CFail` value. The syntax of values is described in Figure 14.

$$\begin{array}{ll}
Value ::= & V_\beta | V_\eta | \texttt{CFail} \\
\\
V_\beta ::= & \langle Val, CTran\rangle | \lhd Expr \rhd | Val \\
Ctran ::= & \mathcal{C} \hookrightarrow \mathcal{C} \\
Val ::= & b|i|s \\
Expr ::= & Val | Expr \cdot Expr | (\!|Val|\!) \\
\\
V_\eta ::= & \|s, EmbCtx\| \\
EmbCtx ::= & \mathcal{C}
\end{array}$$

Figure 14: Syntax of Values

The universe of `string`, `int` or `bool` base typed values is denoted by the letters s,i, and b letters respectively in the syntax above and the standard concatenation operation is denoted by "$\cdot$". The V_β values are of three kinds:

1. Untrusted or unsanitized values are raw untrusted string, integer or boolean values.
2. Other values which are auto-sanitized statically or correspond to program constants are tuples of the form $\langle v, Ctran\rangle$ where $Ctran$ is a metadata field. The $Ctran$ metadata field indicates that the value v safely induces a context transition $Ctran$.
3. The remaining values are a special data-structure $\lhd Expr \rhd$, called the CSRP-expression, which is used for dynamically sanitized values. The data structure stores concatenation expressions, conceptually separating the untrusted peices from trusted peices. In our syntax, we separate untrusted substrings of string expressions by delimiting them with special delimiters, $(\!|$ and $|\!)$, which are assumed to be outside the string alphabet of the base language.

The global output buffer has a value of the form $\|s, EmbCtx\|$, where s is the string buffer consisting of the application's output. The $EmbCtx$ metadata field is the context as a result of parsing s according to our canonical grammar.

Type Safety. Note that the operational semantics ensure the 3 security properties outlined in section 2.2. The `CR` is explicit in the representation of output buffer values—parsing the output buffer at any point results in a permitted context defined in $\mathcal{C}$. Property `NOS` is similarly ensured in the `E-CONST` rule, which evaluate to values of the form $\langle v, Ctran\rangle$. Such values are never sanitized.

Property `CSAN` is immidiate for `E-PRN-STAT` evaluation rule which ensures that the output buffer's context matches the start context in the $Ctran$ field of the written string expression. For the evaluation rule `E-PRN-DYN`, the soundness relies on the procedure $CSRP$ shown boxed which takes a CSRP-expression and a start context to parse the expression in. This procedure parses the string expression embedded in the CSRP-expression, while sanitizing all and only the untrusted substrings delimited by $(\!|\,|\!)$ context-sensitively. If it succeeds it returns a tuple containing the sanitized string expression and the end context.

In order to formalize and sketch the proof of the soundness of our type system, we first define a relation $\mathcal{R}$ mapping types to the set of valid values they correspond to. At any given point in the program, if the type of a variable or object is Q under the typing environment Γ, then its value must correspond to the typing constraints. The relation $\mathcal{R}$ is defined as follows, assuming $\mathcal{U}$ is the universe of strings, integer and boolean values:

Definition 2. ***(Relation $\mathcal{R}$)***

$$\begin{array}{ll}
\mathcal{R}(\mathit{UNSAFE}) & = \{v | v \in \mathcal{U}\} \\
\mathcal{R}(\mathit{STATIC}_{c1\hookrightarrow c2}) & = \{\langle v, c1 \hookrightarrow c2\rangle | v \in \mathcal{U}, IsParseValid(v, c_1, c_2)\} \\
\mathcal{R}(\mathit{DYN}_S) & = \{\lhd v \rhd | v \in \mathcal{U}\} \\
\mathcal{R}(\mathit{CTXSTAT}_c) & = \{\|s, c\| | c \in \mathcal{C}\} \\
\mathcal{R}(\mathit{CTXDYN}_S) & = \{\|s, c\| | c \in \mathcal{C}, c \in S\}
\end{array}$$

At any program location, we define a notion of a well-typed memory M as follows:

Definition 3. ***(Well-Typedness)*** *A memory is well-typed with respect to a typing environment Γ, denoted by $M \models \Gamma$, iff*

$$\forall x \in Dom(M), M[x] \in \mathcal{R}(\Gamma[x])$$

We define the two standard progress and preservation theorems that establish soundness our type system below. Progress states that if the memory if well-typed, then the abstract machine defined in the operational semantics does not get "stuck"— that is, there is at least one evaluation rules that can be applied to the well-typed terms. Preservation states that at any evaluation step if all the subterms (used in the premises) are well-typed then the deduced term is also well-typed.

The machine may get stuck for several reasons. It may be due to the runtime boxed checks failing. This is intended semantics of the language and such behavior is safe. The machine may also get stuck because no evaluaion rule may apply, or in other words, the memory state is such that the semantics do not define how to evaluate further. To distinguish these two cases, we define a value called `CFail`, which the boxed procedure $CSRP$ evaluates to when it fails. Other stuck states may result from reaching an inconsistent memory states for which no evaluation rule applies. We point out the abstract operational semantics we describe here are non-deterministic. Therefore, only when all non-deterministic evaluations of an instance of the procedure $CSRP$ fail, does the boxed check fail and the `print` statement evaluate to the `CFail` runtime error.

Theorem 1. *(Progress and Preservation for Expressions). If $\Gamma \vdash e : T$ and $M \models \Gamma$, then either $M \vdash e \Downarrow$ `CFail` or $M \vdash e \Downarrow v$ and $v \in \mathcal{R}(T)$.*

Proof. By induction on the derivation of $\Gamma \vdash e : T$. □

Theorem 2. *(Progress and Preservation for Commands). If $M \models \Gamma$ and $\Gamma \vdash c \Longrightarrow \Gamma'$, then either $M \vdash c \Downarrow$ `CFail` or $M \vdash c \Downarrow M'$ where $M' \models \Gamma'$.*

Proof. By induction on the derivation of $\Gamma \vdash c \Longrightarrow \Gamma'$. The definition of relation $\mathcal{R}$ serves as the standard inversion lemma. □