Fast and Precise Sanitizer Analysis with BEK

Pieter Hooimeijer, *University of Virginia*
Benjamin Livshits, *Microsoft Research*
David Molnar, *Microsoft Research*
Prateek Saxena, *UC Berkeley*
Margus Veanes [*], *Microsoft Research*

Abstract

Web applications often use special string-manipulating *sanitizers* on untrusted user data, but it is difficult to reason manually about the behavior of these functions, leading to errors. For example, the Internet Explorer cross-site scripting filter turned out to transform some web pages without JavaScript into web pages with valid JavaScript, enabling attacks. In other cases, sanitizers may fail to commute, rendering one order of application safe and the other dangerous.

BEK is a language and system for writing sanitizers that enables precise analysis of sanitizer behavior, including checking idempotence, commutativity, and equivalence. For example, BEK can determine if a target string, such as an entry on the XSS Cheat Sheet, is a valid output of a sanitizer. If so, our analysis synthesizes an input string that yields that target. Our language is expressive enough to capture real web sanitizers used in ASP.NET, the Internet Explorer XSS Filter, and the Google AutoEscape framework, which we demonstrate by porting these sanitizers to BEK.

Our analyses use a novel *symbolic finite automata* representation to leverage fast satisfiability modulo theories (SMT) solvers and are quick in practice, taking fewer than two seconds to check the commutativity of the entire set of Internet Exporer XSS filters, between 36 and 39 seconds to check implementations of `HTMLEncode` against target strings from the XSS Cheat Sheet, and less than ten seconds to check equivalence between all pairs of a set of implementations of `HTMLEncode`. Programs written in BEK can be compiled to traditional languages such as JavaScript and C#, making it possible for web developers to write sanitizers supported by deep analysis, yet deploy the analyzed code directly to real applications.

1 Introduction

Cross site scripting ("XSS") attacks are a plague in today's web applications. These attacks happen because the applications take data from untrusted users, and then echo this data to other users of the application. Because web pages mix markup and JavaScript, this data may be interpreted as code by a browser, leading to arbitrary code execution with the privileges of the victim. The first line of defense against XSS is the practice of *sanitization*, where untrusted data is passed through a *sanitizer*, a function that escapes or removes potentially dangerous strings. Multiple widely used Web frameworks offer sanitizer functions in libraries, and developers often add additional custom sanitizers due to performance or functionality constraints.

Unfortunately, implementing sanitizers *correctly* is surprisingly difficult. Anecdotally, in dozens of code reviews performed across various industries, just about any custom-written sanitizer was flawed with respect to security [38]. The recent SANER work, for example, showed flaws in custom-written sanitizers used by ten web applications [9]. For another example, several groups of researchers have found specially crafted pages that do not initially have cross site scripting attacks, but when passed through anti-cross-site scripting filters yield web pages that cause JavaScript execution [10, 22].

The problem becomes even more complicated when considering that a web application may *compose* multiple sanitizers in the course of creating a web page. In a recent empirical analysis, we found that a large web application often applied the same sanitizers twice, despite these sanitizers not being idempotent. This analysis also found that the order of applying different sanitizers could vary, which is safe only if the sanitizers are commutative [32], providing further evidence suggesting that developers have a difficult time writing correct sanitization functions without assistance.

Despite this, much work in the space of detecting and preventing XSS attacks [19, 23, 25, 27, 39] has optimistically assumed that sanitizers are in fact both known and correct. Some recent work has started exploring the issue of specification completeness [24] as well as sanitizer correctness by explicitly statically modeling sets of values that strings can take at runtime [13, 26, 36, 37]. These approaches use analysis-specific models of strings that are based on finite automata or context-free grammars. More recently, there has been significant interest in constraint solving tools that model strings [11, 17, 18, 20, 31, 34, 35]. String constraint solvers allow any client analysis to express constraints (e.g., path predicates for a

[*] Authors are listed alphabetically. Work done while P. Hooimeijer and P. Saxena were visiting Microsoft Research.

single code path) that include common string manipulation functions.

Sanitizers are typically a small amount of code, perhaps tens of lines. Furthermore, application developers know when they are writing a new, custom sanitizer or set of sanitizers. Our key proposition is that if we are willing to spend a little more time on this sanitizer code, we can obtain fast and precise analyses of sanitizer behavior, along with actual sanitizer code ready to be integrated into both server- and client-side applications. Our approach is BEK, a language for modeling string transformations. The language is designed to be (a) sufficiently expressive to model real-world code, and (b) sufficiently restricted to allow fast, precise analysis, without needing to approximate the behavior of the code.

Key to our analysis is a compilation from BEK programs to *symbolic finite state transducers*, an extension of standard finite transducers. Recall that a finite transducer is a generalization of deterministic finite automata that allows transitions from one state to another to be annotated with *outputs*: if the input character matches the transition, the automaton outputs a specified sequence of characters. In a symbolic finite transducer, transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula. We apply algorithms that determine if two BEK programs are equivalent. We also can check if a BEK program can output a specific string, and if so, synthesize an input yielding that string.

Our symbolic finite state transducer representation enables leveraging *satisfiability modulo theories (SMT) solvers*, tools that take a formula and attempt to find inputs satisfying the formula. These solvers have become robust in the last several years and are used to solve complicated formulas in a variety of contexts. At the same time, our representation allows leveraging automata theoretic methods to reason about strings of unbounded length, which is not possible via direct encoding to SMT formulas. SMT solvers allow working with formulas from any theory supported by the solver, while other previous approaches using binary decision diagrams are specialized to specific types of inputs.

After analysis, programs written in BEK can be compiled back to traditional languages such as JavaScript or C# . This ensures that the code analyzed and tested is functionally equivalent to the code which is actually deployed for sanitization, up to bugs in our compilation.

This paper contains a number of experimental case studies. We conclusively demonstrate that BEK is expressive enough for a wide variety of real-life code by converting multiple real world Web sanitization functions from widely used frameworks, including those used in Internet Explorer 8's cross-site scripting filter, to BEK programs. We report on which features of the BEK language are needed and which features could be added given our experience. We also examine other code, such as sanitizers from Google AutoEscape and functions from WebKit, to determine whether or not they can be expressed as BEK programs. We maintain samples of BEK programs online[1].

We then use BEK to perform security specific analyses of these sanitizers. For example, we use BEK to determine whether there exists an input to a sanitizer that yields any member of a publicly available database of strings known to result in cross site scripting attacks. Our analysis is fast in practice; for example, we take two seconds to check the commutativity of the entire set of Internet Explorer 8 XSS filters, and less than 39 seconds to check an implementations the `HTMLEncode` sanitization function against target strings from the XSS Cheat Sheet [5].

To experimentally demonstrate the difficulty of writing correct sanitizers, we hired several freelance developers to implement `HTMLEncode` functionality. Using BEK, we checked the *equivalence* of the seven different implementations of `HTMLEncode` and used BEK to find counterexamples: inputs on which these sanitizers behave differently. Finally, we performed scalability experiments to show that in practice the time to perform BEK analyses scales near-linearly.

1.1 Contributions

The primary contributions of this paper are:

- **Language.** We propose a domain-specific language, BEK, for string manipulation. We describe a syntax-driven translation from BEK expressions to symbolic finite state transducers.
- **Algorithms.** We provide algorithms for performing composition computation and equivalence checking, which enables checking commutativity, idempotence, and determining if target strings can be output by a sanitizer. We show how JavaScript and C# code can be generated out of BEK programs, streamlining the client- and server-side deployment of BEK sanitizers.
- **Evaluation.** We show that BEK can encode real-world string manipulating code used to sanitize untrusted inputs in web applications. We demonstrate the expressiveness of BEK by encoding OWASP sanitizers, many IE 8 XSS filters, as well as functions written by freelance developers hired through `odesk.com` and `vworker.com` for our experiments presented in this paper. We show how the analyses supported by our tool can find security-critical

[1] http://code.google.com/p/bek/

bugs or check that such bugs do not exist. To improve the end-user experience when a bug is found, BEK produces a counter-example. We discover that only 28.6% of our sanitizers commute, ~79.1% are idempotent, and that only 8% are reversible. We also demonstrate that most hand-written `HTMLEncode` implementations disagree on at least some inputs.

- **A Scalable Implementation.** BEK deals with Unicode strings without creating a state explosion. Furthermore, we show that our algorithms for equivalence checking and composition computation are very fast in practice, scaling near-linearly with the size of the symbolic finite transducer representation. The main reason for this is the symbolic representation of the transition relation.

While the focus of this paper is on XSS attacks[2], our language and analyses are more general and apply to any string manipulating function. For example Chen *et al.* check interactions between firewall rules, finding redundant and order-dependent rules in routers [40]. Cho and Babić [12] check the equivalence between a specification and an implementation for
state machines in SMTP servers.

2 Overview

Figure 1 shows an architectural diagram for the BEK system. At the center of the picture is the transducer-based representation of a BEK program. At the moment, we support a BEK language front end, although other front ends that convert Java or C# programs into BEK are also possible. We provide motivating examples of the BEK language in Section 2.1 and discuss the applications of BEK to analyzing sanitizers in Section 2.2.

2.1 Introductory Examples

Example 1. The following BEK program is a basic sanitizer that backslash-escapes single and double quotes (but only if they are not escaped already). The **iter** construct is a block that uses a character variable c and a single boolean state variable b that is initially $\mathbf{f}$ (false). Each iteration of the block binds the character variable to a single character of the string t; iteration continues until no more characters remain. The block is broken into **case** statements. If a character satisfies the condition of the case statement, the corresponding code is executed.

[2]The dual of the issue of code injection is data privacy; BEK is equally suitable to analyzing the corresponding data cleansing functions.

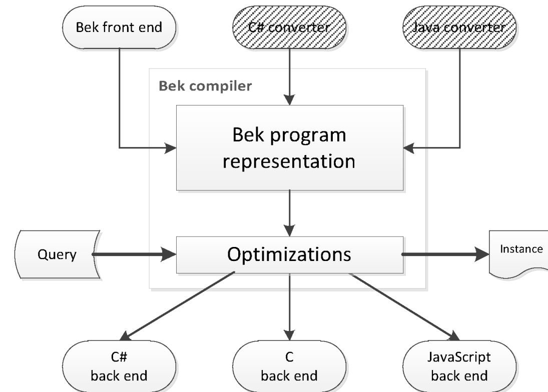

Figure 1: BEK architecture. We use a representation based on *symbolic finite state transducers* (defined in-text) to model string sanitization code without approximation.

```
private static string EncodeHtml(string t)
{
    if (t == null) { return null; }
    if (t.Length == 0) { return string.Empty; }
    StringBuilder builder =
         new StringBuilder("", t.Length * 2);
    foreach (char c in t)
    {
      if ((((c > '`') && (c < '{')) ||
      ((c > '@') && (c < '['))) || (((c == ' ') ||
      ((c > '/') && (c < ':'))) || (((c == '.') ||
      (c == ',')) || ((c == '-') || (c == '_'))))){
        builder.Append(c);
      } else {
        builder.Append("&#" +
          ((int) c).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

Figure 2: Code for `AntiXSS.EncodeHtml` version 2.0.

Here $\mathbf{yield}(c)$ outputs the current character c.

$$
\begin{array}{l}
\mathbf{iter}(c\ \mathbf{in}\ t)\ \{b := \mathbf{f};\}\ \{ \\
\quad \mathbf{case}(\neg(b) \land (c = \text{`'`} \lor c = \text{`"'})) \{ \\
\qquad b := \mathbf{f};\ \mathbf{yield}(\text{`\textbackslash'});\ \mathbf{yield}(c);\} \\
\quad \mathbf{case}(c = \text{`\textbackslash'}) \{ \\
\qquad b := \neg(b);\ \mathbf{yield}(c);\} \\
\quad \mathbf{case}(\mathbf{t}) \{ \\
\qquad b := \mathbf{f};\ \mathbf{yield}(c);\} \\
\quad \}
\end{array}
$$

The boolean variable b is used to track whether the previous character seen was an unescaped slash. For example, in the input `\\"` the double quote is not considered escaped, and the transformed output is `\\\"`. If we apply the BEK program to `\\\"` again, the output is the same. An

interesting question is whether this holds for any output string. In other words, we may be interested in whether a given BEK program is *idempotent*.

If implemented incorrectly, double applications of such sanitization functions can result in duplicate escaping. This in turn has led to command injection of script-injection attacks in the past. Therefore, checking *idempotence* of certain functions is practically useful. We will see in the next section how BEK can perform such checks. ⊠

Example 2. The code in Figure 2 is from the public Microsoft AntiXSS library. The sanitizer iterates over the input character-by-character. Depending on the character encountered, a different action is taken, such as including the character verbatim or encoding it in some manner, such as numeric HTML escaping.

The BEK program corresponding to `EncodeHtml` is

$$
\begin{array}{l}
\textbf{iter}\ (c\ \textbf{in}\ t)\{ \\
\quad \textbf{case}\ (\neg\varphi(c))\{ \\
\quad\quad \textbf{yield}\ [\texttt{'\&'}, \texttt{'\#'}] + \textbf{dec}(c) + [\texttt{';'}];\} \\
\quad \textbf{case}(\textbf{true})\{ \\
\quad\quad \textbf{yield}\ [c];\}\}
\end{array}
$$

where **dec** is a built-in library function that returns the decimal representation of the character and $\varphi(c)$ is the formula

$$
\begin{array}{l}
(\texttt{'a'} \leq c \wedge c \leq \texttt{'z'}) \vee (\texttt{'A'} \leq c \wedge c \leq \texttt{'Z'}) \vee \\
(\texttt{'0'} \leq c \wedge c \leq \texttt{'9'}) \vee c = \texttt{' '} \vee c = \texttt{'.'} \vee \\
c = \texttt{','} \vee c = \texttt{'-'} \vee c = \texttt{'_'}
\end{array}
$$

The BEK program iterates over each character of the input. If the character satisfies the formula $\varphi(c)$, then the program outputs the character. Otherwise the program escapes the character by outputting its decimal encoding, together with the $\&\#$ prefix and semicolon. Note that this sanitizer is not idempotent, because applying the function twice to the string $\&\#$ will result in double escaping. Our tool can detect this in under a second. ⊠

Multiple implementations may exist of the "same" sanitizer. For example, Figure 3 shows the result of running the Red Gate Reflector .NET decompiler on the System.NET implementation of `EncodeHTML`. We have converted this code to BEK as well, noticing that the `goto` structure is the result of a loop after decompilation. Using our analyses, we can check these implementations for equivalence. Our implementation can detect in less than one second that the System.NET implementation does not escape single quote characters, while the AntiXSS implementation does, meaning that the two implementations are not equivalent. Failure to escape single quotes can lead to XSS attacks, so this
difference is significant [33].

```
public static string EncodeHtml(string s)
{
   if (s == null)
     return null;
   int num = IndexOfHtmlEncodingChars(s, 0);
   if (num == -1)
     return s;
   StringBuilder builder=new StringBuilder(s.Length+5);
   int length = s.Length;
   int startIndex = 0;
Label_002A:
   if (num > startIndex) {
     builder.Append(s, startIndex, num-startIndex);
   }
   char ch = s[num];
   if (ch > '>') {
     builder.Append("&#");
     builder.Append(((int) ch).
         ToString(NumberFormatInfo.InvariantInfo));
     builder.Append(';');
   }
   else {
     char ch2 = ch;
     if (ch2 != '"') {
        switch (ch2)
        {
          case '<':
            builder.Append("&lt;");
            goto Label_00D5;

          case '=':
            goto Label_00D5;

          case '>':
            builder.Append("&gt;");
            goto Label_00D5;

          case '&':
            builder.Append("&amp;");
            goto Label_00D5;
        }
     }
     else {
       builder.Append("&quot;");
     }
   }
Label_00D5:
   startIndex = num + 1;
   if (startIndex < length) {
     num = IndexOfHtmlEncodingChars(s, startIndex);
     if (num != -1) {
       goto Label_002A;
     }
     builder.Append(s, startIndex, length-startIndex);
   }
   return builder.ToString();
}
```

Figure 3: Code for `EncodeHtml` from version 2.0 of `System.Net`. This code is not equivalent to the AntiXSS library version.

2.2 Security Applications

Web sanitizers are the first line of defense against cross-site scripting attacks for web applications: they are func-

tions applied to untrusted data provided by a user that attempt to make the data "safe" for rendering in a web browser. Reasoning about the security properties of web sanitizers is crucial to the security of web applications and browsers. Formal verification of sanitizers is therefore crucial in proving the absence of injection attacks such as cross-site and cross-channel scripting as well as information leaks.

2.2.1 Security of Sanitizer Composition

Recent work has demonstrated that developers may accidentally compose sanitizers in ways that are not safe [32]. BEK can check two key properties of sanitizer composition: commutativity and idempotence.

Commutativity: Consider two default sanitizers in the Google CTemplate framework: `JavaScriptEscape` and `HTMLEscape` [4]. The former performs Unicode encoding (`\u00XX`) for safely embedding untrusted data in JavaScript strings while the latter sanitizer performs HTML entity-encoding (`<`) for embedded untrusted data in HTML content. It turns out that if `JavaScriptEscape` is applied to untrusted data before the application of `HTMLEscape`, certain XSS attacks are not prevented [32]. The opposite ordering does prevent these attacks. BEK can check if a pair of sanitizers are commutative, which would mean the programmer does not need to worry about this class of bugs.

Idempotence: BEK can check if applying the sanitizer twice yields different behavior from a single application. For example, an extra JavaScript string encoding may break the intended rendering behavior in the browser.

2.2.2 Sanitizer Implementation Correctness

Hand-coded sanitizers are notoriously difficult to write correctly. Analyses provided by BEK help achieve correctness in three ways.

Comparing multiple sanitizer implementations: Multiple implementations of the same sanitization functionality can differ in subtle ways [9]. BEK can check whether two different programs written in the BEK language are equivalent. If they are not, BEK exhibits inputs that yield different behaviors.

Comparing sanitizers to browser filters: Internet Explorer 8 and 9, Google Chrome, Safari, and Firefox employ built-in XSS filters (or have extensions [3]) that observe HTTP requests and responses [1, 2] for attacks. These filters are most commonly specified as regular expressions, which we can model with BEK. We can then check for inputs that are disallowed by browser filters, but which are allowed by sanitizers. For example, BEK can determine that the AntiXSS implementation of the `EncodeHTML` sanitizer in Figure 2 does not block

$$
\begin{array}{ll}
 & \text{Bool Variables} \quad b, \ldots \\
\text{Bool Constants } B \in \{\mathbf{t}, \mathbf{f}\} & \text{Char Variables} \quad c \\
\text{Char Constants } d \in \Sigma & \text{String Variables} \quad t
\end{array}
$$

$$
\begin{array}{lrl}
\text{Strings} & \mathit{sexpr} ::= & \mathbf{iter}(c\ \mathbf{in}\ \mathit{sexpr})\ \{\mathit{init}\}\ \{\mathit{case}^*\} \\
 & | & \mathbf{fromLast}(\mathit{ccond}, \mathit{sexpr}) \\
 & | & \mathbf{uptoLast}(\mathit{ccond}, \mathit{sexpr}) \mid t \\
 & \mathit{init} ::= & (b\ :=\ B)^* \\
 & \mathit{case} ::= & \mathbf{case}(\mathit{bexpr})\ \{\mathit{cstmt}\} \mid \mathit{endcase} \\
 & \mathit{endcase} ::= & \mathbf{end}(\mathit{ebexpr})\{\mathbf{yield}(d)^*\} \\
 & \mathit{cstmt} ::= & (b\ :=\ \mathit{ebexpr}; \mid \mathbf{yield}(\mathit{cexpr});)^* \\
\text{Booleans} & \mathit{bexpr} ::= & \mathit{Boolcomb}(\mathit{bexpr}) \mid B \mid b \mid \mathit{ccond} \\
 & \mathit{ebexpr} ::= & \mathit{Boolcomb}(\mathit{ebexpr}) \mid B \mid b \\
 & \mathit{ccond} ::= & \mathit{Boolcomb}(\mathit{ccond}) \mid \mathit{cexpr} = \mathit{cexpr} \\
 & | & \mathit{cexpr} < \mathit{cexpr} \mid \mathit{cexpr} > \mathit{cexpr} \\
\text{Char strings} & \mathit{cexpr} ::= & c \mid d \mid \textit{built-in-fnc}(c) \mid \mathit{cexpr} + \mathit{cexpr}
\end{array}
$$

Figure 4: Concrete syntax for BEK. Well-formed BEK expressions are functions of type `string` $\rightarrow$ `string`; the language provides basic constructs to filter and transform the single input string t. $\mathit{Boolcomb}(e)$ stands for Boolean combination of e using conjunction, disjunction, and negation.

strings such as `javascript:` which are prevented by IE 8 XSS filters. These differences indicate potential bugs in the sanitizer or the filter.

Checking against public attack sets: Several public XSS attack sets are available, such as XSS cheat sheet [5]. With BEK, for all sanitizers, for all attack vectors in an attack set, we can check if there exists an input to the sanitizer that yields the attack vector.

3 The BEK Language and Transducers

In this section, we give a high-level description of a small imperative language, BEK, of low-level string operations. Our goal is two-fold. First, it should be possible to model BEK expressions in a way that allows for their analysis using existing constraint solvers. Second, we want BEK to be sufficiently expressive to closely model real-world code (such as Example 2). In this section we first present the BEK language. We then define the semantics of BEK programs in terms of *symbolic finite transducers* (SFTs), an extension of classical *finite state transducers*. Finally, we describe several core decision procedures for SFTs that provide an algorithmic foundation for efficient static analysis and verification of BEK programs.

3.1 The BEK Language

Figure 4 describes the language syntax. We define a single string variable, t, to represent an input string, and a number of expressions that can take either t or another expression as their input. The $\mathbf{uptoLast}(\varphi, t)$ and $\mathbf{fromLast}(\varphi, t)$ are built-in search operations that ex-

tract the prefix (suffix) of t upto (from) and excluding the last occurrence of a character satisfying φ. These constructs are listed separately because they cannot be implemented using other language features. Finally, the **iter** construct allows for character-by-character iteration over a string expression.

Example 3. **uptoLast**(c = `'.'`, `"w.abc.org"`) = `"www.abc"`, **fromLast**(c = `'.'`, `"w.abc.org"`) =`"org"`. ⊠

The **iter** construct is designed to model loops that traverse strings while making imperative updates to boolean variables. Given a string expression ($sexpr$), a character variable c, and an initial boolean state ($init$), the statement iterates over characters in $sexpr$ and evaluates the conditions of the case statements in order. When a condition evaluates to true, the statements in $cstmt$ may yield zero or more characters to the output and update the boolean variables for future iterations. The $endcase$ applies when the end of the input string has been reached. When no case applies, this correspond to yielding zero characters and the iteration continues or the loop terminates if the end of the input has been reached.

3.2 Finite Transducers

We start with the classical definition of *finite state transducers*. The particular sublass of finite transducers that we are considering here are also called *generalized sequential machines* or GSMs [29], however, this definition is not standardized in the literature, and we therefore continue to say finite transducers for this restricted case. The restriction is that, GSMs read one symbol at each transition, while a more general definition allows transitions that skip inputs.

Definition 1. A *Finite Transducer* A is defined as a six-tuple $(Q, q^0, F, \Sigma, \Gamma, \Delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, Σ is the *input alphabet*, Γ is the *output alphabet*, and Δ is the *transition function* from $Q \times \Sigma$ to $2^{Q\times\Gamma^*}$.

We indicate a component of a finite transducer A by using A as a subscript. For $(q, v) \in \Delta_A(p, a)$ we define the notation $p \xrightarrow{a/v}_A q$, where $p, q \in Q_A$, $a \in \Sigma_A$ and $v \in \Gamma^*_A$. We write $p \xrightarrow{a/v} q$ when A is clear from the context. Given words v and w we let $v \cdot w$ denote the concatenation of v and w. Note that $v \cdot \epsilon = \epsilon \cdot v = v$.

Given $q_i \xrightarrow{a_i/v_i}_A q_{i+1}$ for $i < n$ we write $q_0 \xrightarrow{u/v}_A q_n$ where $u = a_0 \cdot a_1 \cdot \ldots \cdot a_{n-1}$ and $v = v_0 \cdot v_1 \cdot \ldots \cdot v_{n-1}$. We write also $q \xrightarrow{\epsilon/\epsilon}_A q$. A induces the *finite transduction*, $T_A : \Sigma^*_A \to 2^{\Gamma^*_A}$:

$$T_A(u) \stackrel{\text{def}}{=} \{v \mid \exists q \in F_A\, (q^0_A \xrightarrow{u/v} q)\}$$

We lift the definition to sets, $T_A(U) \stackrel{\text{def}}{=} \bigcup_{u\in U} T(u)$. Given two finite transductions T_1 and T_2, $T_1 \circ T_2$ denotes the finite transduction that maps an input word u to the set $T_2(T_1(u))$. In the following let A and B be finite transducers. A fundamental composition of A and B is the *join* composition of A and B.

Definition 2. The *join of A and B* is the finite transducer

$$A \circ B \stackrel{\text{def}}{=} (Q_A \times Q_B, (q^0_A, q^0_B), F_A \times F_B, \Sigma_A, \Gamma_B, \Delta_{A\circ B})$$

where, for all $(p, q) \in Q_A \times Q_B$ and $a \in \Sigma_A$:

$$\begin{aligned}\Delta_{A\circ B}((p, q), a) \;\stackrel{\text{def}}{=}\; & \{((p', q), \epsilon) \mid p \xrightarrow{a/\epsilon}_A p'\} \\ & \cup \{((p', q'), v) \mid (\exists u \in \Gamma^+_A) \\ & \qquad p \xrightarrow{a/u}_A p',\; q \xrightarrow{u/v}_B q'\}\end{aligned}$$

The following property is well-known and allows us to drop the distinction between A and T_A without causing ambiguity.

Proposition 1. $T_{A\circ B} = T_A \circ T_B$.

The following classification of finite transducers plays a central role in the sections discussing translation from BEK and decision procedures for symbolic finite transducers.

Definition 3. A is *single-valued* if for all $u \in \Sigma^*_A$, $|A(u)| \leq 1$.

3.3 Symbolic Finite Transducers

Symbolic finite transducers, as defined below, provide a symbolic representation of finite transducers using terms modulo a given background theory $\mathcal{T}$. The background universe $\mathcal{V}$ of values is assumed to be *multi-sorted*, where each sort σ corresponds to a sub-universe $\mathcal{V}^\sigma$. The boolean sort is BOOL and contains the truth values **t** (true) and **f** (false). Definition of terms and formulas (boolean terms) is standard inductive definition, using the function symbols and predicate symbols of $\mathcal{T}$, logical connectives, as well as *uninterpreted constants* with given sorts. All terms are assumed to be well-sorted. A term t of sort σ is indicated by $t : \sigma$. Given a term t and a substitution θ from variables (or uninterpreted constants) to terms or values, $Subst(t, \theta)$ denotes the term resulting from applying the substitution θ to t.

A *model* is a mapping of uninterpreted constants to values.[3] A model *for* a term t is a model that provides an interpretation for all uninterpreted constants that occur in t. (All free variables are treated as uninterpreted constants.) The *interpretation* or *value* of a term t in a

[3] The interpretations of background functions of $\mathcal{T}$ is fixed and is assumed to be an implicit part of all models.

model M for t is given by standard Tarski semantics using induction over the structure of terms, and is denoted by t^M. A formula (predicate) φ is true in a model M for φ, denoted by $M \models \varphi$, if φ^M evaluates to true. A formula φ is satisfiable, denoted by $IsSat(\varphi)$, if there exists a model M such that $M \models \varphi$. Any term $t{:}\sigma$ that includes no uninterpreted constants is called a *value term* and denotes a concrete value $[\![t]\!] \in \mathcal{V}^\sigma$.

Let $Term^{\gamma}_{\mathcal{T}}(\bar{x})$ denote the set of all terms in $\mathcal{T}$ of sort γ, where $\bar{x} = x_0, \ldots, x_{n-1}$ may occur as the only uninterpreted constants (variables). Let $Pred_{\mathcal{T}}(\bar{x})$ denote $Term^{\mathrm{BOOL}}_{\mathcal{T}}(\bar{x})$. In order to avoid ambiguities in notation, given a set E of elements, we write $[e_0, \ldots, e_{n-1}]$ for elements of E^*, i.e., sequences of elements from E. We use both $[]$ and ϵ to denote the empty sequence. As above, if $\mathbf{e_1}, \mathbf{e_2} \in E^*$, then $\mathbf{e_1} \cdot \mathbf{e_2} \in E^*$ denotes the concatenation of $\mathbf{e_1}$ with $\mathbf{e_2}$. We lift the interpretation of terms to apply to sequences: for $\mathbf{u} = [u_0, \ldots, u_{n-1}] \in Term^{\gamma}_{\mathcal{T}}(\bar{x})^*$ let $\mathbf{u}^M \stackrel{\mathrm{def}}{=} [u_0^M, \ldots, u_{n-1}^M] \in (\mathcal{V}^\gamma)^*$.

In the following let $c{:}\sigma$ be a *fixed* uninterpreted constant of sort σ. We refer to $c{:}\sigma$ as the *input variable* (for the given sort σ).

Definition 4. A *Symbolic Finite Transducer (SFT) for* $\mathcal{T}$ is a six-tuple $(Q, q^0, F, \sigma, \gamma, \delta)$, where Q is a finite set of *states*, $q^0 \in Q$ is the *initial state*, $F \subseteq Q$ is the set of *final states*, σ is the *input sort*, γ is the *output sort*, and δ is the *symbolic transition function* from $Q \times Pred_{\mathcal{T}}(c)$ to $2^{Q \times Term^{\gamma}_{\mathcal{T}}(c)^*}$.

We use the notation $p \xrightarrow{\varphi/\mathbf{u}}_A q$ for $(q, \mathbf{u}) \in \delta_A(p, \varphi)$ and call $p \xrightarrow{\varphi/\mathbf{u}}_A q$ a *symbolic transition*, $\varphi/\mathbf{u}$ is called its *label*, φ is called its *input (guard)* and $\mathbf{u}$ its *output*.

An SFT $A = (Q, q^0, F, \sigma, \gamma, \delta)$ denotes the finite transducer $[\![A]\!] = (Q, q^0, F, \mathcal{V}^\sigma, \mathcal{V}^\gamma, \Delta)$ where $p \xrightarrow{a/v}_{[\![A]\!]} q$ if and only if there exists $p \xrightarrow{\varphi/\mathbf{u}}_A q$ and a model M such that $M \models \varphi$, $c^M = a$, $\mathbf{u}^M = v$.

For an STF A let the underlying *transduction* T_A be $T_{[\![A]\!]}$. For a state $q \in Q_A$ let $T^q_A(v)$ ($T^q_{[\![A]\!]}(v)$) denote the set of outputs when starting from q with input v. In particular, if $q = q^0_A$ then $T_C = T^q_A$ and $T_{[\![A]\!]} = T^q_{[\![A]\!]}$. The following proposition follows directly from the definition of $[\![A]\!]$.

Proposition 2. *For* $v \in \Sigma^*_{[\![A]\!]}$ *and* $q \in Q_A$: $T^q_A(v) = T^q_{[\![A]\!]}(v)$.

Example 4. The *identity* SFT Id (for sort σ) is defined follows. $Id = (\{q\}, q, \{q\}, \sigma, \sigma, \{q \xrightarrow{\mathtt{t}/[c]} q\})$. Thus, for all $a \in \mathcal{V}^\sigma$, $q \xrightarrow{a/a}_{[\![Id]\!]} q$, and $[\![Id]\!](v) = \{v\}$ for all $v \in (\mathcal{V}^\sigma)^*$. ⊠

Example 5. Assume σ is the sort for characters. The predicate $c = $ `'.'` says that the input character is a dot.

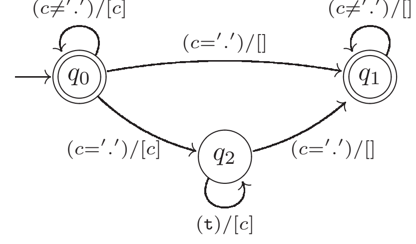

Figure 5: Symbolic finite state transducer for $\mathbf{uptoLast}$(`c='.', input`). This transducer is nondeterministic; there are two transitions that match `'.'` from state q_0.

The SFT $UptoLastDot$ such that for all strings v,

$$UptoLastDot(v) = \mathbf{uptoLast}(c = \texttt{'.'}, v),$$

where **uptoLast** is the BEK function introduced above, is shown in Figure 5. ⊠

Composition works directly with SFTs, and keeps the resulting SFT *clean* in the sense that all symbolic transitions are *feasible*, and eliminates states that are *unreachable from the initial state* as well as non-initial states that are *not backwards reachable from any final state*. In order to preserve feasibility of transitions the algorithm uses a solver for checking satisfiability of formulas in $Pred_{\mathcal{T}}(c)$.

3.4 BEK to SFT translation

The basic sort needed in this section, besides BOOL, is a sort CHAR for characters. We also assume the background relation $<$: CHAR × CHAR → BOOL as a strict total order corresponding to the standard lexicographic order over ASCII (or Unicode) characters and assume $>$, $\leq$ and $\geq$ to be defined accordingly. We also assume that each individual character has a built-in constant such as `'a'`:CHAR. For example,

$$(\texttt{'A'} \leq c \wedge c \leq \texttt{'Z'}) \vee (\texttt{'a'} \leq c \wedge c \leq \texttt{'z'}) \vee$$
$$(\texttt{'0'} \leq c \wedge c \leq \texttt{'9'}) \vee c = \texttt{'_'}$$

descibes the regex character class `\w` of all word characters in ASCII. (Direct use of regex character classes in BEK, such as **case**(`\w`) $\{\ldots\}$, is supported in the enhanced syntax supported in the BEK analyzer tool.)

Each *sexpr* e is translated into an SFT $SFT(e)$. For the string variable t, $SFT(e) = Id$, with Id as in Example 4. The translation of $\mathbf{uptoLast}(\varphi, e)$ is the symbolic composition $STF(e) \circ B$ where B is an SFT similar to the one in Example 5, except that the condition $c = $ `'.'` is replaced by φ. The translation of $\mathbf{fromLast}(\varphi, e)$ is analogous. Finally,

$SFT(\mathbf{iter}(c\ \mathbf{in}\ e)\ \{init\}\ \{case^*\}) = SFT(e) \circ B$ where $B = (Q, q^0, Q, \text{CHAR}, \text{CHAR}, \delta)$ is constructed as follows:

Step 1: Normalize. Transform $case^*$ so that case conditions are mutually exclusive by adding the negations of previous case conditions as conjuncts to all the subsequent case conditions, and ensure that each boolean variable has exactly one assignment in each $cstmt$ (add the trivial assignment $b := b$ if b is not assigned).

Step 2: Compute states. Compute the set of states Q. Let q^0 be an initial state as the truth assignment to boolean variables declared in $init$.[4] Compute the set Q of all reachable states, by using DFS, such that, given a reached state q, if there exists a case $\mathbf{case}(\varphi)\ \{cstmt\}$ such that $Subst(\varphi, q)$ is *satisfiable* then add the state

$$\{b \mapsto [\![Subst(\psi, q)]\!] \mid b := \psi \in cstmt\} \quad (1)$$

to Q. (Note that $Subst(\psi, q)$ is a value term.)

Step 3: Compute transitions. Compute the symbolic transition function δ. For each state $q \in Q$ and for each case $\mathbf{case}(\varphi)\ \{cstmt\}$ such that $\phi = Subst(\varphi, q)$ is satisfiable. Let p be the state computed in (1). Let $\mathbf{yield}(u_0), \ldots, \mathbf{yield}(u_{n-1})$ be the sequence of yields in $cstmt$ and let $\mathbf{u} = [u_0, \ldots, u_{n-1}]$. Add the symbolic transition $q \xrightarrow{\phi/\mathbf{u}} p$ to δ.

The translation of end-cases is similar, resulting in symbolic transitions with guard $c = \bot$, where $\bot$ is a special character used to indicate end-of-string. We assume $\bot$ to be least with respect to $<$. For example, assuming that the BEK programs use concrete ASCII characters, $\bot$:CHAR is either an *additional* character, or the null character `'\0'` if only null-terminated strings are considered as valid input strings. Although practically important, end-cases do not cause algorithmic complications, and for the sake of clarity we avoid them in further discussion.

The algorithm uses a solver to check satisfiability of guard formulas. If checking satisfiability of a formula for example times out, then it is safe to assume satisfiability and to include the corresponding symbolic transition. This will potentially add infeasible guards but retains the *correctness* of the resulting SFT, meaning that the underlying finite transduction is unchanged. While in most cases checking satisfiability of guards seems straightforward, but when considering Unicode, this perception is deceptive. As an example, the regex character class

[4] Note that q^0 is the empty assignment if $init$ is empty, which trivializes this step.

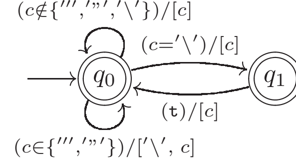

Figure 6: SFT for BEK program in Example 1. This SFT escapes single and double quotes with a backslash, except if the current symbol is already escaped. The application of this SFT is idempotent.

`[\W-[\D]]` denotes an empty set since `\d` is a subset of `\w` and `\W` (`\D`) is the complement of `\w` (`\d`), and thus, `[\W-[\D]]` is the intersection of `\W` and `\d`. Just the character class `\w` alone contains 323 non-overlapping ranges in Unicode, totaling 47,057 characters. A naïve algorithm for checking satisfiability (non-emptiness) of `[\W-[\D]]` may easily time out.

Consider the BEK program in Example 1. The corresponding SFT constructed by the above translation is shown in Figure 6. There are two symbolic transitions from state q_0 to itself. The first corresponds to the cases where the input character c needs to be escaped, and the second to cases where the input does not need to be escaped.

3.5 Join Composition and Equivalence

We now give an informal description of our core algorithms for reasoning about SFTs: *join composition* and *equivalence*. We then show how these algorithms can be used to check properties such as idempotence, existence of an input yielding a target string, and commutativity.

The *join composition* $A \circ B$ corresponds to a program transformation that constructs a single loop over the input string out of two consecutive loops in SFTs A and B. The join composition algorithm constructs an SFT $A \circ B$ such that $T_{[\![A \circ B]\!]} = T_{[\![A]\!]} \circ T_{[\![B]\!]}$. The intuition behind the construction is that the outputs produced by A are substituted *symbolically* in as the inputs consumed by the B. The composition algorithm proceeds by depth-first search, first computing $Q_{A \circ B}$ as constructed as a reachable subset of $Q_A \times Q_B$, starting from (q^0_A, q^0_B). Here we use the SMT solver to determine reachability, calling the solver as a black box to determine if a path from one state to another is feasible or not. This makes our construction *independent* of the particular background theory. In general, this is not true for other recent extensions of finite transducers such as streaming transducers [6], where compositionality depends on properties of the background theory that is being used.

Two SFTs A and B are *equivalent* if $T_A = T_B$. Let

$$Dom(A) \stackrel{\text{def}}{=} \{v \mid T_A(v) \neq \emptyset\}.$$

Checking equivalence of A and B reduces to two separate tasks:

1. Deciding *domain-equivalence*: $Dom(A) = Dom(B)$.
2. Deciding *partial-equivalence*: for all $v \in Dom(A) \cap Dom(B)$, $T_A(v) = T_B(v)$.

Note that 1 and 2 are independent and do not imply each other, but together they imply equivalence. Domain equivalence holds for all SFTs constructed by BEK, because all programs share the same domain, namely that of strings. Checking partial equivalence is more involved. We leverage the fact that all SFTs we construct are single-valued. Our equivalence algorithm first computes the join composition of A and B, then uses the SMT solver to search for inputs that cause A to differ from B. We have a *nonconstructive* proof of termination for this algorithm: it establishes that if A and B are equivalent, then the search must terminate in time quadratic in the number of states of the composed automata. In practice, the SMT solver carries out this search, and our results in Section 4 show scaling is closer to linear in practice.

Equivalence and join composition allow us to carry out a variety of other analyses. Idempotence of an SFT A can be first checked by computing $B = A \circ A$, then checking the equivalence of A and B. If the two SFTs are not equivalent, then A fails to be idempotent. Similarly, commutativity of two SFTs A and B can be determined by computing $C = A \circ B$ and $D = B \circ A$, then checking equivalence. The idea is illustrated in Figure 7. We can also compute the *inverse image* of a SFT with respect to a string s, which lets us find out the set of inputs to the SFT that yield s as an output. We use all of these analyses to check sanitizers for security
properties in the next section.

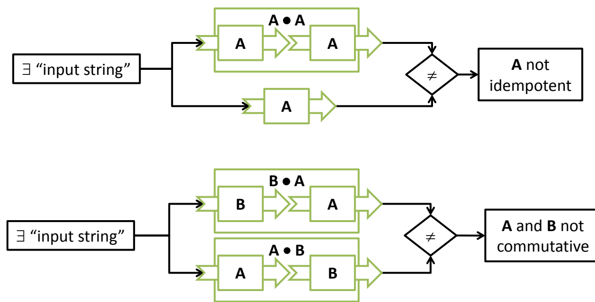

Figure 7: Using composition and equivalence of SFTs to decide idempotence and commutativity.

Our approach has an advantage over traditional finite transducers (FTs), due to succinctness of SFTs. Suppose for example that the background character theory $\mathcal{T}$ is k-bit bit vector arithmetic where k depends on the desired character range (e.g., for Unicode, $k = 16$). An explicit expansion of a BEK SFT A to $[\![A]\!]$ may increase the size (nr of transitions) by a factor of 2^k. Partial-equivalence of single-valued FTs is solvable $O(n^2)$ [15] time. Thus, for an SFT A of size n, using the partial-equivalence algorithm for $[\![A]\!]$ takes $O((2^k n)^2)$ time. In contrast, the partial-equivalence algorithm for BEK SFTs is $O(n^2)$. When the background theory is linear arithmetic, then the alphabet is infinite and a correspoding FT algorithm is therefore not even possible.

4 Evaluation

In the following subsections, we evaluate the real-world applicability of BEK in terms of expressivess,
utility, and performance:

- Section 4.1 evaluates whether BEK can model existing real-world code. We conduct an emperical study of a large body of code to see how widely-used BEK-modelable sanitizer functions are (Section 4.1.1), and we evaluate which BEK features are needed to model sanitizers from AutoEscape, OWASP, and Internet Explorer 8 (Section 4.1.2).
- We put BEK to work to check existing sanitizers for idempotence, commutativity, and reversibility (Section 4.2).
- We perform pair-wise equivalence checks on a number of ported `HTMLEncode` implementations, as well as two outsourced implementations (Section 4.3).
- We evaluate effectiveness of existing `HTMLEncode` implementations against known attack strings taken from the Cross-site Scripting Cheat Sheet (Section 4.4).
- We use a synthetic benchmark to evaluate the scalability of performing equivalence checks on BEK programs (Section 4.5).
- We provide a short example to highlight the fact that BEK programs can be readily translated to other programming languages (Section 4.6).

These experiments are based on an implementation that consists of roughly $5,000$ lines of C# code that implements the basic transducer algorithms and Z3 [14] integration, with another $1,000$ lines of F# code for translation from BEK to transducers. Our experiments were carried out on a Lenovo ThinkPad W500 laptop with 8 GB

of RAM and an Intel Core 2 Duo P9600 processor running at 2.67 GHz, running 64-bit Windows 7.

4.1 Expressive Utility

Thus far, we discussed the expressiveness of BEK primarily in theoretical terms. In this subsection, we turn our attention to real-world applicability instead, through a case study that aims to demonstrate that a wide variety of commonly used sanitizers can be ported to BEK with relative ease.

4.1.1 Frequency of Sanitizer use in PHP code.

PHP is a widely-used open source server-side scripting language. Minamide's seminal work on the static analysis of dynamic web applications [26] includes finite-transducer based models for a subset of PHP's sanitizer functions. These transducers are hand-crafted in several thousand lines of OCaml. We conducted an informal review of the PHP source to confirm that each transducer could be modeled as a BEK program.

Our goal is to perform a high-level quantitative comparison of the applicability of BEK, on the one hand, and existing string constraint solvers (e.g., DPRLE [17], Hampi [20], Kaluza [30], and Rex [35]) on the other. For this comparison, we assume that each Minamide transducer could instead be modeled as a BEK program. We then use statistics from a study by Hooimeijer [16] that measured the relative frequency, by static count, of 111 distinct PHP string library functions. The Hooimeijer study was conducted in December 2009, and covers the top 100 projects on `SourceForge.net`, or about 9.6 million lines of PHP code. The study considered most, but not all, sanitizers provided by Minamide.

Out of the 111 distinct functions considered in the Hooimeijer study, 27 were modeled as transducers by Minamide and thus encodable in BEK. In the sampled PHP code, these 27 functions account for $68,238$ out of $251,317$ uses, or about 27% of all string-related call sites. By comparison, traditional regular expression functions modeled by tools like Hampi [20] and Rex [35] account for just 29,141 call sites, or about 12%. We note that BEK could be readily integrated into an automaton-based tool like Rex, however, and our features are largely complimentary to those of traditional string constraint solvers. These results suggest that BEK provides a significant improvement in the "coverage" of real-world code by string analysis tools.

4.1.2 Language Features

For the remainder of the experiments, we use a small dataset of ported-to-BEK sanitizers. We now discuss that dataset and the manual conversion effort required. The results are summarized in Figure 8, and described in more detail below.

Google AutoEscape and OWASP. We converted sanitizers from the OWASP sanitizer library to BEK programs. We also evaluated sanitizers from the Google AutoEscape framework to determine what language features they would need to be expressed in BEK. These sanitizers are marked with prefixes `GA` and `OWASP`, respectively, in Figure 8. We verified that each of these sanitizers can be implemented in BEK. In several cases, we find additional non–native features that could be added to BEK to support these sanitizers.

Internet Explorer. In addition, we extracted sanitizers from the binary of Internet Explorer 8 that are used in the IE Cross-Site Scripting Filter feature, denoted `IEFilter1` to `IEFilter17` in Figure 8. For this study, we analyze the behavior of the IE 8 sanitizers under the assumption the server performs no sanitization of its own on user data. Of these 21 sanitizers, we could convert 17 directly into BEK programs. The remaining 4 sanitizers track a potentially unbounded list of characters that are either emitted unaltered or escaped, depending on the result of a regular expression match. BEK does not enable storing strings of input characters.

The manual translation took several hours per sanitizer. Figure 8 breaks down our BEK programs based on "Native" features of the BEK language, and "Not Native" features which are not currently in the BEK language. Many of these features can be integrated modeled using transducers, however, by enhancing the language of constraints used for symbolic labels. In addition, with the exception of 4 Internet Explorer sanitizers, we found that a maximum lookahead window of eight characters would suffice for handling all our sanitizers. Finally, we discovered that the arithmetic on characters was limited to right shifts and linear arithmetic, which can be expressed in the Z3 solver we use.

We note that all "Not Native" features could be added to the BEK language with few or no changes to the underlying SFT algorithms for join composition and equivalence checking: only the front end would need to change.

4.1.3 Browser Code

Ideally, we could use BEK to model the parser of an actual web browser. Then, we could use our analyses to check whether there exists a string that passes through a given sanitizer yet causes javascript execution. We performed a preliminary exploration of the WebKit browser to determine how difficult it would be to write such a model with BEK. Unfortunately, we found multiple

Name	Native: boolean vars	Native: multiple iters	Native: regex	Not Native: mult. lookahead	Not Native: arith.	Not Native: functions
`a2bb2a`	1	✗	✓	✗	✗	✗
`escapeBrackets`	1	✓	✗	✗	✗	✗
`escapeMetaAndLink`	1	✓	✓	✗	✗	✗
`escapeString0`	1	✗	✗	✗	✗	✗
`escapeString`	1	✗	✗	✗	✗	✗
`escapeStringSimple`	1	✗	✗	✗	✗	✗
`getFileExtension`	2	✗	✗	✗	✗	✗
GA `HtmlEscape`	0	✗	✗	✗	✗	✗
GA `PreEscape`	0	✗	✗	✗	✗	✗
GA `SnippetEsc`	3	✗	✗	✓	✗	✗
GA `CleanseAttrib`	1	✗	✗	✓	✗	✗
GA `CleanseCSS`	0	✗	✗	✗	✗	✗
GA `CleanseURLEsc`	0	✗	✗	✗	✗	✗
GA `ValidateURL`	2	✓	✗	✓	✓	✗
GA `XMLEsc`	0	✗	✗	✗	✗	✗
GA `JSEsca`	0	✗	✗	✓	✗	✗
GA `JSNumber`	2	✓	✗	✓	✗	✗
GA `URLQueryEsc`	1	✓	✗	✗	✓	✗
GA `JSONESc`	0	✗	✗	✗	✗	✗
GA `PrefixLine`	0	✗	✗	✗	✗	✗
OWASP `HTMLEncode`	0	✗	✗	✓	✗	✗
`IEFilter1`	3	✗	✓	✗	✗	✗
`IEFilter2`	4	✗	✓	✗	✗	✗
`IEFilter3`	5	✗	✓	✗	✗	✗
`IEFilter4`	4	✗	✓	✗	✗	✗
`IEFilter5`	4	✗	✓	✗	✗	✗
`IEFilter6`	5	✗	✓	✗	✗	✗
`IEFilter7`	4	✗	✓	✗	✗	✗
`IEFilter8`	4	✗	✓	✗	✗	✗
`IEFilter9`	5	✗	✓	✗	✗	✗
`IEFilter10`	5	✗	✓	✗	✗	✗
`IEFilter11`	4	✗	✓	✗	✗	✗
`IEFilter12`	4	✗	✓	✗	✗	✗
`IEFilter13`	4	✗	✓	✗	✗	✗
`IEFilter14`	4	✗	✓	✗	✗	✗
`IEFilter15`	1	✗	✓	✗	✗	✗
`IEFilter16`	1	✗	✓	✗	✗	✗
`IEFilter17`	1	✗	✓	✗	✗	✗

Figure 8: Expressiveness: different language features used by the original corpus of different programs. A cross means that the feature was not used by the program in its initial implementation. A checkmark means the feature was used by the program. boolean variables, multiple iterations over a string, and regular expressions are native constructs in BEK. Multiple lookahead, arithmetic, and functions are not native to BEK and must be emulated during the translation. We also show the distinct boolean variables used by the BEK implementation.

functions that require features, such as bounded lookahead and transducer composition, which are not yet supported by the BEK language.

For example, we considered a function in the Safari implementation of WebKit that performs Javascript decoding [7]. This function requires at a minimum the use of functions to connect hexadecimal to ASCII, a lookahead of 5 characters, function composition, and scanning for occurrences of a target character. While as noted above we believe these features could be added to BEK without fundamentally changing the underlying algorithms for symbolic transducers, the BEK language does not yet support them.

4.2 Checking Algebraic Properties

We argued in Section 2 that idempotence and commutativity are key properties for sanitizers. In addition, the property of *reversibility*, that from the output of a sanitizer we can unambiguously recover the input, is important as an aid to debugging.

4.2.1 Order Independence

We now evaluate whether 17 sanitizers used in IE 8 are *order independent*. Order independence means that the sanitizers have the same effect no matter in what order they are applied. If the order does matter, then the choice of order can yield surprising results. As an example, in rule-based firewalls, a set of rules that are not order independent may result in a rule never being applied, even though the administrator of the firewall believes the rule is in use.

Each IE 8 sanitizer defines a specific *input set* on which it will transform strings, which we can compute from the BEK model. We began by checking all 136 pairs of IE 8 sanitizers to determine whether their input sets were disjoint. Only one pair of sanitizers showed a non-trivial intersection in their input sets. A non-trivial intersection signals a potential order dependence, because the two sanitizers will transform the same strings. For this pair, we used BEK to check that the two sanitizers output the same language, when restricted to inputs from their intersection. BEK determined that the transformation of the two sanitizers on thesel inputs was exactly the same — i.e., the two sanitizers were equivalent on the intersection set. We conclude that the IE 8 sanitizers are in fact order independent, up to errors in our extraction of the sanitizers and our assumption that no server-side modification is present.

4.2.2 Idempotence and Reversibility

We now examine the idempotence of several BEK programs, including the IE 8 sanitizers. Figure 9 reports the results. The number of states in the symbolic finite transducer created from each BEK program. For each transducer, we then report whether it is idempotent and whether it is reversible. This shows the number of states acts as a rough guide to the complexity of the sanitizer. For example, we see that IE filter 9 out of 17 is quite complicated, with 25 states.

4.2.3 Commutativity

We investigated commutativity of seven different implementations of `HTMLEncode`, a sanitizer commonly used by web applications. Four implementations were gathered from internal sources. Three were created for our

Name	States	Idempotent?	Reversible?
a2bb2a	1	✗	✓
escapeBrackets	1	✓	✗
escapeMetaAndLink	1	✓	✓
escapeString0	1	✗	✗
escapeString	1	✗	✗
escapeStringSimple	1	✗	✗
getFileExtension	2	✗	✗
IEFilter1	6	✓	✗
IEFilter2	9	✓	✗
IEFilter3	19	✓	✗
IEFilter4	13	✓	✗
IEFilter5	13	✓	✗
IEFilter6	16	✓	✗
IEFilter7	13	✓	✗
IEFilter8	12	✓	✗
IEFilter9	25	✓	✗
IEFilter10	18	✓	✗
IEFilter11	11	✓	✗
IEFilter12	11	✓	✗
IEFilter13	14	✓	✗
IEFilter14	14	✓	✗
IEFilter15	1	✓	✗
IEFilter16	1	✓	✗
IEFilter17	1	✓	✗

Figure 9: For each BEK benchmark programs, we report the number of states in the corresponding symbolic transducer. We then report whether the transducer is idempotent, and whether the transducer is reversible.

HTMLEncode1	✓	✓	✓	✗	✗	✓	✗
HTMLEncode2	✓	✓	✓	✗	✗	✓	✗
HTMLEncode3	✓	✓	✓	✗	✗	✓	✗
HTMLEncode4	✗	✗	✗	✓	✗	✗	✗
Outsourced1	✗	✗	✗	✗	✓	✗	✗
Outsourced2	✓	✓	✓	✗	✗	✓	✗
Outsourced3	✗	✗	✗	✗	✗	✗	✓

Figure 10: Commutativity matrix for seven different implementations of HTMLEncode. The Outsourced implementations were written by freelancers from a high level English specification.

project specifically by hiring freelance programmers to create implementations from popular outsourcing web sites. We provided these programmers with a high level specification in English that emphasized protection against cross-site scripting attacks. Figure 10 shows a *commutativity matrix* for the HTMLEncode implementations. A ✓ indicates the pair of sanitizers commute, while a ✗ indicates they do not. The matrix contains 12 check marks out of 42 total comparisons of distinct sanitizers, or 28.6%. Our implementation took less than one minute to complete all 42 comparisons.

4.3 Differences Between Multiple Implementations

Multiple implementations of the "same" functionality are commonly available from which to choose when writing a web application. For example, newer versions of a library may update the behavior of a piece of code. Different organizations may also write independent implementations of the same functionality, guided by performance

HTMLEncode1	✓	✓	✓	0	–	✓	0
HTMLEncode2	✓	✓	✓	0	–	✓	0
HTMLEncode3	✓	✓	✓	0	–	✓	′
HTMLEncode4	0	0	0	✓	0	0	0
Outsourced1	–	–	–	0	✓	–	0
Outsourced2	✓	✓	✓	0	–	✓	0
Outsourced3	0	0	′	0	0	0	✓

Figure 11: Equivalence matrix for our implementations of HTMLEncode. A ✓ indicates the implementations are equivalent. For implementations that are not equivalent, we show an example character that exhibits different behavior in the two implementations. The symbol 0 refers to the null character.

improvements or by different requirements. Given these different implementations, the first key question is "do all these implementations compute the same function?" Then, if there are differences, the second key question is "how do these implementations differ?"

As described above, because BEK programs correspond to single valued symbolic finite state transducers, computing the image of regular languages under the function defined by a BEK program is decidable. By taking the image of Σ^* under two different BEK programs, we can determine whether they output the
same set of strings.

We checked equivalence of seven different implementations in C# (as explained above) of the HTMLEncode sanitization function. We translated all seven implementations to BEK programs by hand. First, we discovered that all seven implementations had only one state when transformed to a symbolic finite transducer. We then found that all seven are neither reversible nor idempotent. For example, the ampersand character & is expanded to & by all seven implementations. This in turn contains an ampersand that will be re-expanded on future applications of the sanitizer, violating idempotence.

For each BEK program, we checked whether it was equivalent to the other HTMLEncode implementations. Figure 11 shows the results. For cases where the two implementations are not equivalent, BEK derived a counterexample string that is treated differently by the two implementations. For example, we discovered that Outsourced1 escapes the – character, while Outsourced2 does not. We also found that one of the HTMLEncode implementations does not encode the single quote character. Because the single quote character can close HTML contexts, failure to encode it could cause unexpected behavior for a web developer who uses this implementation. For example, a recent attack on the Google Analytics dashboard was enabled by failure to sanitize a single quote [33].

This case study shows the benefit of automatic analysis of string manipulating functions to check equivalence.

Implementation	HTML context	Attribute context
`HTMLEncode1`	100%	93.5%
`HTMLEncode2`	100%	93.5%
`HTMLEncode3`	100%	93.5%
`HTMLEncode4`	100%	100%
`Outsourced1`	100%	93.5%
`Outsourced2`	100%	93.5%
`Outsourced3`	100%	93.5%

Figure 12: Percentage of XSS Cheat Sheet strings, in both HTML tag context and tag attribute contexts, that are ruled out by each implementation of `HTMLEncode`.

Without BEK, obtaining this information using manual inspection would be difficult, error prone, and time consuming. With BEK, we spent roughly 3 days total translating from C# to BEK programs. Then BEK was able to compute the contents of Figure 11 in less than one minute, including all equivalence and containment checks.

4.4 Checking Filters Against The Cheat Sheet

The Cross-Site Scripting Cheat Sheet ("XSS Cheat Sheet") is a regularly updated set of strings that trigger JavaScript execution on commonly used web browsers. These strings are specially crafted to cause popular web browsers to execute JavaScript, while evading common sanitization functions. Once we have translated a sanitizer to a program in BEK, because BEK uses symbolic finite state transducers, we can take a "target" string and determine whether there exists a string that when fed to the sanitizer results in the target. In other words, we can check whether a string on the Cheat Sheet has a *pre-image* under the function defined by a BEK program.

We sampled 28 strings from the Cheat Sheet. The Cheat Sheet shows snippets of HTML, but in practice a sanitizer might be run only on a substring of the snippet. We focused on the case where a sanitizer is run on the HTML Attribute field, extracting sub-strings from the Cheat Sheet examples that correspond to the attribute parsing context. While `HTMLEncode` should not be used for sanitizing data that will become part of a URL attribute, in practice programmers may accidentally use `HTMLEncode` in this "incorrect" context. We also added some strings specifically to check the handling of HTML attribute parsing by our sanitizers. As a result, we obtained two sets of attack strings: HTML and Attribute.

For each of our implementations, for all strings in each set, we then asked BEK whether pre-images of that string exist. Figure 12 shows what percentage of strings have no pre-image under each implementation. All seven implementations correctly escape angle brackets, so no string in the HTML set has a pre-image under any of the sanitizers. In the case of the Attribute strings, however, we found that some of the implementations do not escape the string"&#", potentially yielding an attack. Only one of our implementations of `HTMLEncode` made it impossible for all of the strings in the Attribute set from appearing in its output. Each set of strings took between 36 and 39 seconds for BEK to check the entire set of strings against a sanitizer.

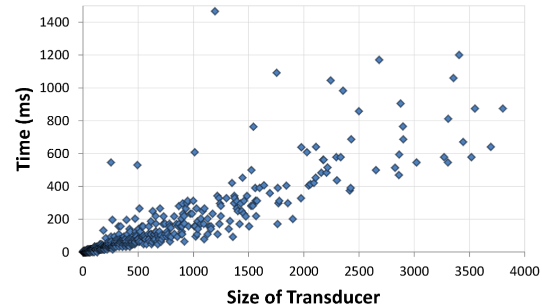

Figure 13: Self-equivalence experiment.

4.5 Scalability of Equivalence Checking

Our theoretical analysis suggests that the speed of queries to BEK should scale quadratically in the number of states of the symbolic finite transducer. All sanitizers we have found in "the wild," however, have a small number of states. While this makes answering queries about the sanitizers fast, it does not shed light on the empirical performance of BEK as the number of states increases. To address this, we performed two experiments with synthetically generated symbolic finite transducers. These transducers were specially created to exhibit some of the structure observed in real sanitizers, yet have many more states than observed in practical sanitizer implementations.

Self-equivalence experiment. We generated symbolic finite transducers A from randomly generated BEK programs having structure similar to typical sanitizers. The time to check equivalence of A with itself is shown in Figure 13 where the size is the number of states plus the number of transitions in A. Although the worst case complexity is quadratic, the actual observed complexity, for a sample size of 1,000, is linear.

Commutativity experiment. We generated symbolic finite transducers from randomly generated BEK programs having structure similar to typical santizers. For each symbolic finite transducer A, we checked commu-

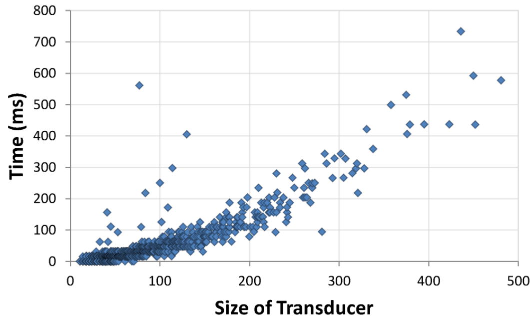

Figure 14: Commutativity experiment.

tativity with a small BEK program *UpToLastDot* that returns a string up to the last dot character. The time to determine that $A \circ$ *UpToLastDot* and *UpToLastDot* $\circ A$ are *equivalent* is shown in Figure 14 where the size is the total number of states plus the number of transitions in A. The time to check non-equivalence was in most cases only a few milliseconds, thus all experiments exclude the data where the result is *not equivalent*, and only include cases where the result is *equivalent*. Although the worst case complexity is quadratic, the actual observed complexity, over a sample size of 1,000
individual cases, was near-linear.

4.6 From BEK to Other Languages

We have built compilers from BEK programs to commonly used languages. When the time comes for deployment, the developer can compile to the language of her choice for inclusion into an application.

Figure 15 shows a small example of a BEK program and the result of its JavaScript compilation. As part of the compilation, we have taken advantage of our knowledge of properties of JavaScript to improve the speed of the compiled code. For example, we push characters into arrays instead of creating new string objects. The result is standard JavaScript code that can be easily included in any web application. By adding additional compilers for common languages, such as C#, we can give a developer multiple implementations of a sanitizer that are guaranteed to be equivalent for use in different contexts.

5 Related Work

SANER combines dynamic and static analysis to validate sanitization functions in web applications [9]. SANER creates finite state transducers for an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. In contrast, our work focuses on a simple language that is expressive enough to capture existing sanitizers or write new ones by hand, but then compile to symbolic finite state transducers that precisely capture the sanitization function. SANER also treats the issue of inputs that may be tainted by an adversary, which is not in scope for our work. Our work also focuses on efficient ways to compose sanitizers and combine the theory of finite state transducers with SMT solvers, which is not treated by SANER.

Minamide constructs a string analyzer for PHP code, then uses this string analyzer to obtain context free grammars that are over-approximations of the HTML output by a server [26]. He shows how these grammars can be used to find pages with invalid HTML. The method proposed in [21] can also be applied to string analysis by modeling regular string analysis problems as *higher-order multi-parameter tree transducers* (HMTTs) where strings are represented as linear trees. While HMTTs al-

```
// orginal Bek program
program test0(t);
string s;
s := iter(c in t)
{b := false;} {
 case ((c == 'a')):  i
    b := !(b) && b;
    b := b || b;
    b := !(b);
    yield (c);
 case (true) :
    yield ('$');
};

//
// JavaScript translation
//
function test0(t) {
  var s  =  function ($){
    var result = new Array();
    for(i=0;i<$.length; i++){
        var c = $[i];
        if ((c == String.fromCharCode(97))) {
            b = (!(b) && b);
            b = (b || b);
            b = !(b);
            result.push(c);
        }
        if (t) {
            result.push(String.fromCharCode(36));
        }
    };
    return result.join('');
  }
  return s(t);
}
```

Figure 15: A small example BEK program (top) and its compiled version in JavaScript (bottom). Note the use of `result.push` instead of explicit array assignment.

low encodings of finite transducers, arbitrary background character theories are not directly expressibly in order to encode SFTs. Our work treats issues of composition and state explosion for finite state transducers by leveraging recent progress in SMT solvers, which aids us in reasoning precisely about the transducers created by transformation of BEK programs and by avoiding state space explosion and bitblasting for large character domains such as Unicode. Moreover, SMT solvers provide a method of extracting concrete counterexamples.

Wasserman and Su also perform static analysis of PHP code to construct a grammar capturing an over-approximation of string values. Their application is to SQL injection attacks, while our framework allows us to ask questions about any sanitizer [36]. Follow-on work combines this work with dynamic test input generation to find attacks on full PHP web applications [37]. Dynamic analysis of PHP code, using a combination of symbolic and concrete execution techniques, is implemented in the Apollo tool [8]. The work in [39] describes a layered static analysis algorithm for detecting security vulnerabilities in PHP code that is also enable to handle some dynamic features. In contrast, our focus is specifically on sanitizers instead of on full applications; we emphasize analysis precision over scaling to large code bases.

Christensen *et al.*'s Java String Analyzer is a static analysis package for deriving finite automata that characterize an over-approximation of possible values for string variables in Java [13]. The focus of their work is on analyzing legacy Java code and on speed of analysis. In contrast, we focus on precision of the analysis and on constructing a specific language to capture sanitizers, as well as on the integration with SMT solvers.

Our work is complementary to previous efforts in extending SMT solvers to understand the theory of strings. HAMPI [20] and Kaluza [31] extend the STP solver to handle equations over strings and equations with multiple variables. Rex extends the Z3 solver to handle regular expression constraints [35], while Hooimeijer *et al.*show how to solve subset constraints on regular languages [17]. We in contrast show how to combine any of these solvers with finite transducers whose edges can take symbolic values in any of the theories supported by the solver.

The work in [28] introduces the first symbolic extension of finite state transducers called a *predicate-augmented finite state transducer* (pfst). A pfst has two kinds of transitions: 1) $p \xrightarrow{\varphi/\psi} q$ where φ and ψ are character predicates or ϵ, or 2) $p \xrightarrow{c/c} q$. In the first case the symbolic transition corresponds to all concrete transitions $p \xrightarrow{a/b} q$ such that $\varphi(a)$ and $\psi(b)$ are true, the second case corresponds to *identity* transitions $p \xrightarrow{a/a} q$ for all characters a. A pfst is not expressive enough for describing an SFT. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode sanitizers such as `EncodeHtml`.

A recent symbolic extension of finite transducers is *streaming transducers* [6]. While the theoretical expressiveness of the language introduced in [6] exceeds that of BEK, streaming transducers are restricted to character theories that are total orders with no other operations. Also, composition of streaming transducers requires an explicit treatment of characters. It is an interesting future research topic to investigate if there is an extension of SFTs or a restriction of streaming transducers that allows efficient symbolic analysis techniques to be applied.

6 Conclusions

Much prior work in XSS prevention assumes the correctness of sanitization functions. However, practical experience shows writing correct sanitizers is far from trivial. This paper presents BEK, a language and a compiler for writing, analyzing string manipulation routines, and converting them to general-purpose languages. Our language is expressive enough to capture real web sanitizers used in ASP.NET, the Internet Explorer XSS Filter, and the Google AutoEscape framework, which we demonstrate by porting these sanitizers to BEK.

We have shown how the analyses supported by our tool can find security-critical bugs or check that such bugs do not exist. To improve the end-user experience when a bug is found, BEK produces a counter-example. We discover that only 28.6% of our sanitizers commute, $\sim$79.1% are idempotent, and only 8% are reversibe. We also demonstrate that most hand-written `HTMLEncode` implementations disagree on at least some inputs. Unlike previously published techniques, BEK deals equally well with Unicode strings without creating a state explosion. Furthermore, we show that our algorithms for equivalence checking and composition computation are extremely fast in practice, scaling near-linearly with the size of the symbolic finite transducer representation.

References

[1] About Safari 4.1 for Tiger. http://support.apple.com/kb/DL1045.

[2] Internet Explorer 8: Features. http://www.microsoft.com/windows/internet-explorer/features/safer.aspx.

[3] NoXSS Mozilla Firefox Extension. http://www.noxss.org/.

[4] OWASP: ESAPI project page. http://code.google.com/p/owasp-esapi-java/.

[5] XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html.

[6] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the Symposium on Princples of Programming Languages*, pages 599–610, 2011.

[7] Apple. Jsdecode implementation, 2011. `http://trac.webkit.org/browser/releases/Apple/Safari%205.0/JavaScriptCore/runtime/JSGlobalObjectFunctions.cpp`.

[8] S. Artzi, A. Kieżun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst. Finding bugs in Web applications using dynamic test generation and explicit-state model checking. *Transactions on Software Engineering*, 99:474–494, 2010.

[9] 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 Symposium on Security and Privacy*, 2008.

[10] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the Conference on the World Wide Web*, pages 91–100, 2010.

[11] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the International Conference on Tools And Algorithms For The Construction And Analysis Of Systems*, 2009.

[12] C. Y. Cho, D. Babić, E. C. R. Shin, and D. Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the Conference on Computer and Communications Security*, pages 426–439, 2010.

[13] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *Proceedings of the Static Analysis Symposium*, 2003.

[14] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on Tools And Algorithms For The Construction And Analysis Of Systems*, 2008.

[15] A. J. Demers, C. Keleman, and B. Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.

[16] P. Hooimeijer. Decision procedures for string constraints. Ph.D. Dissertation Proposal, University of Virginia, April 2010.

[17] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 188–198, 2009.

[18] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the International Conference on Automated Software Engineering*, 2010.

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

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2009.

[21] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 495–508, 2010.

[22] D. Lindsay and E. V. Nava. Universal XSS via IE8's XSS filters. In *Black Hat Europe*, 2010.

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

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

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

[26] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the International Conference on the World Wide Web*, pages 432–441, 2005.

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

[28] G. V. Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:2001, 2001.

[29] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*, volume 1. Springer, 1997.

[30] 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, Mar 2010.

[31] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[32] P. Saxena, D. Molnar, and B. Livshits. ScriptGard: Preventing script injection attacks in legacy Web applications with automatic sanitization. Technical Report MSR-TR-2010-128, Microsoft Research, Sept. 2010.

[33] B. Schmidt. Google analytics XSS vulnerability, 2011. `http://spareclockcycles.org/2011/02/03/google-analytics-xss-vulnerability/`.

[34] M. Veanes, N. Bjørner, and L. de Moura. Symbolic automata constraint solving. In C. Fermüller and A. Voronkov, editors, *LPAR-17*, volume 6397 of *LNCS*, pages 640–654. Springer, 2010.

[35] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2010.

[36] G. Wassermann and Z. Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *Proceedings of the Conference on Programming Language Design and Implementation*, 2007.

[37] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for Web applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, 2008.

[38] J. Williams. Personal communications, 2005.

[39] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the Usenix Security Symposium*, pages 179–192, 2006.

[40] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. Fireman: A toolkit for firewall modeling and analysis. In *Proceedings of the Symposium on Security and Privacy*, pages 199–213, 2006.