BEK: Re-Envisioning In-Browser Privacy

Pieter Hooimeijer	University of Virginia
Benjamin Livshits	Microsoft Research
David Molnar	Microsoft Research
Prateek Saxena	University of California Berkeley
Margus Veanes	Microsoft Research

Microsoft Research Technical Report

MSR-TR-2010-154

Abstract

Web applications must use special string-manipulating *sanitization functions* on untrusted user data, but writing these functions correctly is error prone and time consuming. We present a domain-specific imperative language, BEK, that is expressive enough to capture real web sanitizers used in the Internet Explorer XSS Filter and the Google AutoEscape framework. We exhibit a translation from the BEK language to *symbolic* finite state transducers (SFTs), a novel representation for transducers that annotates transitions with logical formulae. Symbolic finite state transducers give us a new way to marry the classic theory of finite state transducers with the recent progress in *satisfiability modulo theories* (SMT) solvers. We exhibit algorithms for checking equivalence of images of regular languages, idempotency, commutativity, and other properties of SFTs that scale quadratically in the number of states, and we show that BEK's implementation of these algorithms scales near-linearly in practice. We then show how BEK can be applied to checking key security properties of web sanitizers, and how programs written in the BEK language can be compiled to traditional languages such as JavaScript. BEK makes it possible for web developers to write sanitizers supported by deep analysis, yet deploy the analyzed code directly to real applications.

BEK: Modeling Imperative String Operations with Symbolic Transducers

Pieter Hooimeijer
University of Virginia

Benjamin Livshits
Microsoft Research

David Molnar
Microsoft Research

Prateek Saxena
UC Berkeley

Margus Veanes
Microsoft Research

Abstract

Web applications must use special string-manipulating *sanitization functions* on untrusted user data, but writing these functions correctly is error prone and time consuming. We present a domain-specific imperative language, BEK, that is expressive enough to capture real web sanitizers used in the Internet Explorer XSS Filter and the Google AutoEscape framework. We exhibit a translation from the BEK language to *symbolic* finite state transducers (SFTs), a novel representation for transducers that annotates transitions with logical formulae. Symbolic finite state transducers give us a new way to marry the classic theory of finite state transducers with the recent progress in *satisfiability modulo theories* (SMT) solvers. We exhibit algorithms for checking equivalence of images of regular languages, idempotency, commutativity, and other properties of SFTs that scale quadratically in the number of states, and we show that BEK's implementation of these algorithms scales near-linearly in practice. We then show how BEK can be applied to checking key security properties of web sanitizers, and how programs written in the BEK language can be compiled to traditional languages such as JavaScript. BEK makes 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 of the application, 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 is the practice of *sanitization*, where untrusted data is passed through a *sanitizer*, a function that escapes or removes potentially dangerous strings.

Unfortunately, writing sanitizers is difficult to do correctly. Recent static analyses of web applications address cross-site scripting or SQL injection by explicitly modeling sets of values that strings can take at runtime [9, 18, 26, 27]. 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 [8, 14, 15, 17, 20, 24, 25]. String constraint solvers allow any client analysis to express constraints (e.g., path predicates) that include common string manipulation functions. The key problem with existing solutions is that they sacrifice precision to work with sanitizer implementations that are in general purpose languages, such as PHP or Java. No guarantees can be given about the results.

Sanitizers, however, are typically a small amount of code, perhaps tens of lines. Furthermore, security-critical string functions can be modeled precisely using finite state transducers over a symbolic alphabet. Therefore, sanitizers are a prime target for a domain-specific language.

We introduce 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 precise analysis using transducers. BEK can model real-world sanitization functions, such as those in the .NET `System.Web` library, without approximation. We provide a translation from BEK expressions to the theory of algebraic datatypes, allowing BEK expressions to be used directly when specifying constraints for an SMT solver, in combination with other theories.

Key to enabling the analysis of BEK programs is a new theory of *symbolic finite state transducers*, an extension of standard form finite transducers that we introduce formally in this paper. We introduce the notion of and develop a theory of symbolic transducers, showing its integration with other theories in SMT solvers that support E-matching [10]. We give algorithms for *join composition* and *equivalence checking* that show these problems are decidable.

We then show that multiple real world Web sanitization functions, including those used in Internet Explorer 8's cross-site scripting filter and Google's AutoEscape framework, can be converted to BEK programs. We report on which features of the BEK language are needed and which features could be added given our experience. We then use BEK to perform security specific analyses of web 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. Finally, we exhbit a compilation from BEK to JavaScript, which allows developers to use BEK for developing sanitizers that can be transferred rapidly to real applications.

1.1 Contributions

The primary contributions of this paper are:

- We formally describe a domain-specific language, BEK, for string manipulation. We describe a syntax-driven translation from BEK expressions to symbolic finite state transducers.
- We develop a theory of symbolic finite-state transducers (SFTs) and provide algorithms for performing composition computation and equivalence checking. We argue that a symbolic representation is needed to maintain the scalability of the approach since the non-symbolic representation explodes in the case of a large underlying alphabet such as the Unicode or even ASCII. We provide non-trivial extensions of classical decidability results to symbolic case modulo any background theory.
- We show that BEK can encode real-world string manipulating code used to sanitize untrusted inputs in Web applications. We demonstrate several applications that are of direct practical interest. These include checking equivalence of different implementations of the same sanitizer, checking commutativity of sanitizers, and checking whether sanitizers have inputs resulting in known attack vectors.

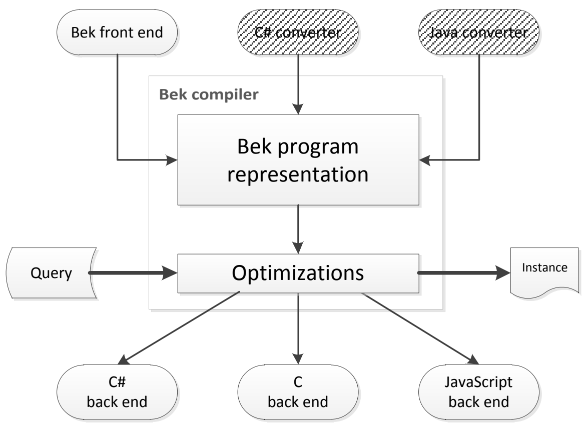

Figure 1: BEK architecture

```
private static string EncodeHtml(string strInput)
{
    if (strInput == null) { return null; }
    if (strInput.Length == 0) { return string.Empty; }
    StringBuilder builder =
         new StringBuilder("", strInput.Length * 2);
    foreach (char ch in strInput)
    {
      if ((((ch > ’‘’) && (ch < ’{’)) ||
      ((ch > ’@’) && (ch < ’[’))) || (((ch == ’ ’) ||
      ((ch > ’/’) && (ch < ’:’))) || (((ch == ’.’) ||
      (ch == ’,’)) || ((ch == ’-’) || (ch == ’_’))))){
        builder.Append(ch);
      } else {
        builder.Append("&#" +
          ((int) ch).ToString() + ";");
      }
    }
    return builder.ToString();
}
```

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

1.2 Paper Organization

The rest of this paper is structured as follows. In Section 2 we provide a motivating example and describe the basics of our approach. Section 3 provides language-theoretic definitions and presents the translation from BEK expressions to transducers. Section 4 describes the core transducer-based algorithms. Section 5 talks about the kinds of analyses we can do on BEK programs and our experimental results. Finally, we discuss closely related work in Section 6 and conclude in Section 7.

2. Overview

This section starts by providing a motivating example for the BEK language (Section 2.1), discusses some of the applications of BEK to analyzing security sanitization functions (Section 2.2), and then describes the overall architecture of the BEK system (Section 2.3).

2.1 Introductory Example

We now show examples of web sanitization functions. Typically this code performs *character escaping*, where a character is compared against a list of prohibited characters and then transformed into an escaped format if it matches.

Example 1. The example 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. ⊠

Example 2. The following BEK program is a basic sanitizer that escapes single and double quotes (but only if they are not escaped already). Note that the structure of the BEK code below closely matches the character-by-character iteration structure of the AntiXSS example above. The **iter** block uses a character variable c and a single boolean state variable b that is initially **f** or false.

```
$\mathbf{iter}(c\ \mathbf{in}\ t)\ \{b := \mathbf{f};\}\ \{$
    $\mathbf{case}(\neg(b) \wedge (c = \text{‘'’} \vee c = \text{‘"’}))\ \{$
        $b := \mathbf{f};\ \mathbf{yield}(\text{‘\textbackslash’});\ \mathbf{yield}(c);\}$
    $\mathbf{case}(c = \text{‘\textbackslash’})\ \{$
        $b := \neg(b);\ \mathbf{yield}(c);\}$
    $\mathbf{case}(\mathbf{t})\ \{$
        $b := \mathbf{f};\ \mathbf{yield}(c);\}$
$\}$
```

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 will 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. ⊠

2.2 Applications of BEK to Security

Web sanitizers are the first line of defense against cross-site scripting attacks for web applications: they are functions 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 [22]. BEK can check two key properties of sanitizer composition: commutativity and idempotence.

Commutativity: Consider `JavaScriptCodec` and `HTMLEntityCodec` [4]. The former performs Unicode encoding (`\u00XX`) for safely embedding untrusted data in JavaScript strings while the latter sanitizer performs HTML entity-encoding (`&lt;`) for embedded untrusted data in HTML content. It turns out that if `JavaScriptCodec` is applied to untrusted data before the application of `HTMLEntityCodec`, certain XSS attacks are not prevented. 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 [7]. 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

		Bool Variables	$b, \ldots$
Bool Constants	$B \in \{\mathbf{t}, \mathbf{f}\}$	Char Variables	c
Char Constants	$d \in \Sigma$	String Variables	t

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

Figure 3: Concrete syntax for BEK. Well-formed BEK expressions are functions of type `string -> 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, or negation.

can then check for inputs that are disallowed by browser filters, but which are allowed by sanitizers. For example, BEK can automatically determine that the sanitizer in Figure 2 does not block attacks such as `javascript&#58;` which are prevented by IE 8 XSS filters. These inputs are likely problematic.

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.

2.3 System Architecture

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.

3. 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), that are symbolic extensions of classical *finite 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 BEK Language

Figure 3 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 extract the prefix (suffix) of t upto (from) and excluding the last occurrence of a character satisfying φ. These built-in operations are not expressible in the core language, because they require the underlying transducer to be *non-deterministic*, while the translation from the core language always yields *deterministic* transducers.

Example 3. $\mathbf{uptoLast}(c =$ `'.'`, `"w.abc.org"`$)$ = `"www.abc"`, $\mathbf{fromLast}(c =$ `'.'`, `"w.abc.org"`$)$ =`"org"`. ⊠

The $\mathbf{iter}$ 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 c in sexpr and evaluaates 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 and the iteration continues. The $\mathit{endcase}$ 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 transducers*. The particular sublass of finite transducers that we are considering here are also called *generalized sequential machines* or GSMs [19], 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*, $T_A : \Sigma_A^* \to 2^{\Gamma_A^*}$:

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

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

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

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

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

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 $\mathbf{t}$ (true) and $\mathbf{f}$ (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.[1] 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 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 $Term^{\text{BOOL}}_\mathcal{T}(\bar{x})$. 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 $\mathbf{u}^M \stackrel{\text{def}}{=} [u_0^M,\ldots,u_{n-1}^M] \in (\mathcal{V}^\gamma)^*$.

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. $Id = (\{q\},q,\{q\},\sigma,\sigma,\{q \xrightarrow{\mathbf{t}/[c]} q\})$. 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. 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 4. ⊠

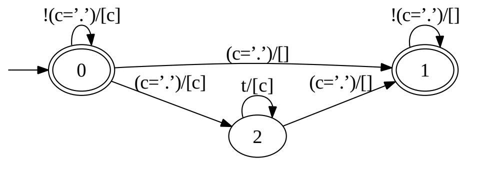

Figure 4: SFT *UptoLastDot* for **uptoLast**(c=`'.'`,*input*).

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

The algorithm for join composition of SFTs, $A \circ B$, is deferred to Section 4.1. At this point we note that the algorithm 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`) {...}, 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.

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).

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*.[2] 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

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

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

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 $c = \perp$, where $\perp$ is a special character used to indicate end-of-string. We assume $\perp$ to be least with respect to $<$. For example, assuming that the BEK programs use concrete ASCII characters, $\perp$: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

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

underlying finite transduction is unchanged. While in most cases checking satisfiability of guards seems "easy", as they are typically small, when considering Unicode, this perception is deceptive. As a simple example, the regex character class `[\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 naive algorithm for checking satisfiability (non-emptiness) of `[\W-[\D]]` may easily time out.

Consider the BEK program in Example 2. The corresponding SFT constructed by the above translation is shown in Figure 5. There are two symbolic transitions from state $\{b \mapsto \mathbf{f}\}$ to itself, corresponding to the cases where the input character c needs (does not need) to be escaped.

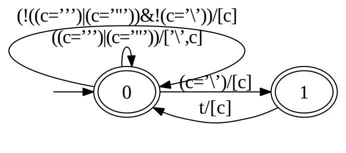

Figure 5: SFT for BEK program in Example 2.

4. Core Algorithms for SFTs

In this section we present two general algorithms for SFTs. First, we describe the symbolic join composition algorithm that is used above in the BEK translation algorithm. Second, we describe an equivalence checking algorithm for single-valued SFTs, where an SFT A is *single-valued* if $[\![A]\!]$ is single-valued (recall Definition 3). The equivalence checking algorithm and the implied decidability result is a nontrivial generalization of the corresponding result known for single-valued finite transducers ,while the general equivalence checking problem for finite transducers and GSMs, and thus for general SFTs (with any background $\mathcal{T}$), is undecidable [16].

In the algorithms we use the definitions $\mathit{Src}(p \xrightarrow{\varphi/\mathbf{u}} q) \stackrel{\text{def}}{=} p$, $\mathit{Tgt}(p \xrightarrow{\varphi/\mathbf{u}} q) \stackrel{\text{def}}{=} q$, $\mathit{Grd}(p \xrightarrow{\varphi/\mathbf{u}} q) \stackrel{\text{def}}{=} \varphi$, and $\mathit{Out}(p \xrightarrow{\varphi/\mathbf{u}} q) \stackrel{\text{def}}{=} \mathbf{u}$. Also, we write $\delta_A(q)$ for the set of all symbolic transitions of A from state q.

4.1 Join Composition

Let A and B be given SFTs and assume that $\rho = \gamma_A = \sigma_B$. The join composition algorithm constructs an SFT $A \circ B$ such that $T_{[\![A \circ B]\!]} = T_{[\![A]\!]} \circ T_{[\![B]\!]}$.

The algorithm is shown in Figure 6, where $Subst(e, u)$ stands for $Subst(e, \{c \mapsto u\})$, i.e., the result of substituting the input variable $c{:}\rho$ in e by the term $u{:}\rho$. The algorithm uses a procedure *GetPaths*$(\varphi, \mathbf{u}, q, B)$ that, given a symbolic label $\varphi/\mathbf{u}$ of A, and a state q of B, returns the collection of all joined paths in B of length $Len(\mathbf{u})$ that start from p and whose guards are feasible for corresponding members of $\mathbf{u}$. *GetPaths* uses satisfiabilty checking to yield only those paths for which the corresponding output from A, when used as input of B, does not cause the resulting guard to become unsatisfiable. If the satisfiability check is removed, the procedure will still be correct in the context of the join algorithm.

The self-join of *UptoLastDot* (from Figure 4) is illustrated in Figure 7. Note that the figure shows for example that *UptoLastDot* is not idempotent. On the other hand, if we consider the self-join of the SFT in Figure 5, then we end up with an identical SFT. Property checking is discussed in more detail below.

Theorem 1. *Let A and B be SFTs such that $\gamma_A = \sigma_B$. Then $T_{A \circ B} = T_A \circ T_B$. This holds also when the satisfiability check is omitted.*

Proof: First note that the satisfiability check removes transitions that are infeasible and therefore do not affect the transduction. We can also ignore states in $A \circ B$ that are not reachable from the initial state, and states that do not reach a final state. Suppose (p_1, p_2) is reachable from the initial state (q_A^0, q_B^0). The definition of *GetPaths* follows exactly the construction of the composed transitions in Definition 2 from (p_1, p_2), which, together with Proposition 2, the main while-loop in the join algorithm that

```
GetPaths(φ, u, q, B):
    if $\mathbf{u} = []$ yield $([], \varphi, q)$;
    else foreach $tr \in \delta_B(q)$
        let $\varphi_1 = \varphi \wedge Subst(\mathit{Grd}(tr), \mathit{First}(\mathbf{u}))$;
        if $IsSat(\varphi_1)$
            foreach $(y, \psi, p)$ in $\mathit{GetPaths}(\varphi_1, \mathit{Rest}(\mathbf{u}), \mathit{Tgt}(tr), B)$
                yield $(Subst(\mathit{Out}(tr), \mathit{First}(\mathbf{u})) \cdot y, \psi, p)$;
Join(A, B):
    let $q^0 = (q_A^0, q_B^0)$; $Q = \{q^0\}$; $\delta = \emptyset$;
    let $S$ be a stack with initial element $q^0$;
    while $S$ is nonempty
        pop $p = (p_1, p_2)$ from $S$;
        foreach $p_1 \xrightarrow{\varphi/\mathbf{u}} q_1$ in $\delta_A(p_1)$
            foreach $(\mathbf{v}, \psi, q_2)$ in $\mathit{GetPaths}(\varphi, \mathbf{u}, p_2, B)$
                let $q = (q_1, q_2)$;
                add $p \xrightarrow{\psi/\mathbf{v}} q$ to $\delta$;
                if $q \notin Q$ then add $q$ to $Q$ and push $q$ to $S$;
    end of while;
    let $F = \{(q_1, q_2) \in Q \mid q_1 \in F_A \wedge q_2 \in F_B\}$;
    eliminate states in $Q \setminus \{q^0\}$ that do not reach a state in $F$;
    return $(Q, q_0, F, \sigma_A, \gamma_B, \delta)$;
```

Figure 6: Join composition algorithm for SFTs.

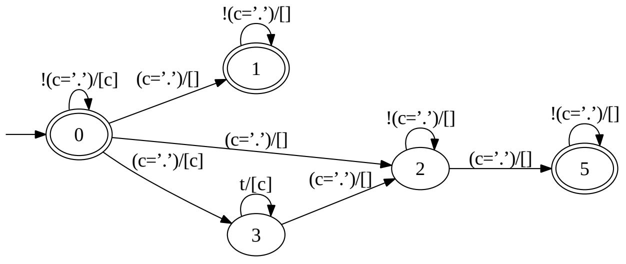

Figure 7: SFT *UptoLastDot* ∘ *UptoLastDot*.

considers all possible (p_1, p_2) that are reachable from the initial state, and Proposition 1, implies the statement of this theorem. ■

4.2 Equivalence

We describe an algorithm for deciding equivalence of single-valued SFTs. Note that all SFTs generated from BEK programs are single-valued and single-valuedness is trivially preserved by join composition. Note also that this does, in general, not outrule nondeterministic SFTs, i.e., SFTs whose underlying finite automaton is nondeterministic. This is important because, several useful single-valued trasductions, such as $T_{\mathit{UptoLastDot}}$, are not expressible as deterministic SFTs.

In the following let A and B be two SFTs such that $\sigma = \sigma_A = \sigma_B$ and $\gamma = \gamma_A = \gamma_B$. A and B are *equivalent* if $T_A = T_B$. Let

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

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

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

Note that 1 and 2 are independent and do not imply each other, while it is easy to see that together they imply equivalence. Checking domain-equivalence is decidable for all SFTs for $\mathcal{T}$, given decidability of $Pred_{\mathcal{T}}(c)$. This follows from results known for *symbolic finite automata* or *SFAs* that generalize finite automata by allowing predicates from $Pred_{\mathcal{T}}(c)$ as labels.

Next, we develop an algorithm for checking partial-equivalence of single-valued SFTs. In the following we assume that *A and B are single-valued.* The algorithm also depends on that *$Pred_{\mathcal{T}}(c_1, c_2)$ is decidable*, i.e., that satisfiabilty can be decided for predicates with up to two uninterpreted constants.

```
*CheckPartialEquivalence*$(A, B)$:
    let $C = Intersect(A, B)$;
    let $Q = \{q_C^0 \mapsto ([], [])\}$;
    let $S$ be a stack with initial element $q_C^0$;
    while $S$ is nonempty
        pop $p$ from $S$;
        let $(\mathbf{a}, \mathbf{b}) = Q(p)$;
        foreach $p \xrightarrow{\varphi/(\mathbf{u},\mathbf{v})} q$ in $\delta_C(p)$
            let $\mathbf{x} = \mathbf{a} \cdot \mathbf{u}$; $\mathbf{y} = \mathbf{b} \cdot \mathbf{v}$;
            if $q \in F_C \wedge Len(\mathbf{x}) \neq Len(\mathbf{y})$ FAIL;
            let $m = min(Len(\mathbf{x}), Len(\mathbf{y}))$;
            let $\psi = \varphi \wedge (\bigvee_{i<m} \mathbf{x}(i) \neq \mathbf{y}(i))$;
            if $IsSat(\psi)$ FAIL;
            let $\mathbf{x}' =$ if $Len(\mathbf{x}) > m$ then $[\mathbf{x}(m), \ldots]$ else $[]$;
            let $\mathbf{y}' =$ if $Len(\mathbf{y}) > m$ then $[\mathbf{y}(m), \ldots]$ else $[]$;
            if $\mathbf{x}' = [] \wedge \mathbf{y}' = []$
                if $q \notin Dom(Q)$ push $q$ to $S$ and set $Q(q) = ([], [])$;
                else if $Q(q) \neq ([], [])$ FAIL;
            else if $\mathbf{y}' = []$
                let $c'$ be a fresh uninterpreted constant of sort $\sigma$;
                let $\varphi_1 = (\bigvee_{i<Len(\mathbf{x}')} \mathbf{x}'(i) \neq Subst(\mathbf{x}'(i), c'))$;
                let $\varphi_2 = \varphi \wedge Subst(\varphi, c') \wedge \varphi_1$;
                if $IsSat(\varphi_2)$ FAIL;
                let $\mathbf{a}' = (\mathbf{x}')^M$ where $M \models \varphi$;
                if $q \notin Dom(Q)$ push $q$ to $S$ and set $Q(q) = (\mathbf{a}', [])$;
                else if $Q(q) \neq (\mathbf{a}', [])$ FAIL;
            else ... (symmetrical case for $\mathbf{x}' = []$)
    end of while;
    SUCCEED;
```

Figure 8: Partial-equivalence algorithm for single-valued SFTs.

The algorithm is given in Figure 8. As the first step, the partial-equivalence checking algorithm uses a procedure called *Intersect* that is a slight variation of the intersection algorithm for SFAs [25]. It constructs the intersection of reachable states for conjoined guards, thus restricting the remainder of the algorithm to the case of inputs from *Dom*$(A) \cap$ *Dom*(B). Symbolic transitions of the intersection *Intersect*(A, B) have the form

$$(p_1, p_2) \xrightarrow{\varphi_1 \wedge \varphi_2/(\mathbf{u}_1, \mathbf{u}_2)} (q_1, q_2)$$

where $(p_1, p_2), (q_1, q_2) \in Q_A \times Q_B$, $p_1 \xrightarrow{\varphi_1/\mathbf{u}_1}_A q_1$, and $p_2 \xrightarrow{\varphi_2/\mathbf{u}_2}_B q_2$. The intersection algorithm eliminates transitions where $\varphi_1 \wedge \varphi_2$ is unsatisfiable, it also eliminates all states that are not reachable from the initial state (q_A^0, q_B^0) (by virtue of DFS), and as a final step, it eliminates all dead states (non-initial states from which no final state is reachable, a final state in the intersection is an element of $F_A \times F_B$).

Next, the algorithm verifies that $T_A(v) = T_B(v)$ for all inputs sequences v in the shared domain *Dom*$(A) \cap$ *Dom*(B). This is a DFS algorithm for symbolic forward analysis of the intersection C computed in the first step. The algorithm requires the solver for $Pred_{\mathcal{T}}(c)$ to be able to provide a model for a satisfiable formula.

In the algorithm, the elements of S are the states still to be verified. Q is the map from reached states to pending outputs associated with those states. The proof of the correctness of the algorithm uses the following lemma.

Lemma 1. *If a state q in Intersect(A, B) is reached twice with different pending outputs then A and B are not partial-equivalent.*

Proof: Suppose we can reach a state q in C from the initial state first time with some input sequence z_1, common output o_1, and pending output (a_1, b_1), and a second time with some input sequence z_2, common output o_2, and pending output (a_2, b_2) such that $(a_2, b_2) \neq (a_1, b_1)$, where, for $i = 1, 2$, either $a_i = []$ or $b_i = []$. Let w be any input sequence from q to a final state of C, w exists because C has no dead states. Since A and B are *single-valued*, we know that

$$\begin{array}{rcl} T_A(z_1 \cdot w) & = & \{o_1 \cdot a_1 \cdot o_3\} \\ T_B(z_1 \cdot w) & = & \{o_1 \cdot b_1 \cdot o_4\} \\ T_A(z_2 \cdot w) & = & \{o_2 \cdot a_2 \cdot o_3\} \\ T_B(z_2 \cdot w) & = & \{o_2 \cdot b_2 \cdot o_4\} \end{array}$$

for some o_3 and o_4 that are the outputs for w starting from q, in A and B respectively.

Suppose now that $T_A(z_1 \cdot w) = T_B(z_1 \cdot w)$ and $T_A(z_2 \cdot w) = T_B(z_2 \cdot w)$. Then $o_1 \cdot a_1 \cdot o_3 = o_1 \cdot b_1 \cdot o_4$ and $o_2 \cdot a_2 \cdot o_3 = o_2 \cdot b_2 \cdot o_4$, so $a_1 \cdot o_3 = b_1 \cdot o_4$ and $a_2 \cdot o_3 = b_2 \cdot o_4$. We reach contradiction by case analysis.

- Case $a_1 = [], b_1 = [], a_2 \neq [], b_2 = []$. Then $o_3 = o_4$ and $a_2 \cdot o_3 = o_4$, but $a_2 \neq []$. $\star$
- Case $a_1 \neq [], b_1 = [], a_2 \neq [], b_2 = []$. Then $a_1 \cdot o_3 = o_4$ and $a_2 \cdot o_3 = o_4$, but $a_1 \neq a_2$. $\star$
- Case $a_1 = [], b_1 \neq [], a_2 \neq [], b_2 = []$. Then $o_3 = b_1 \cdot o_4$ and $a_2 \cdot o_3 = o_4$, and thus $o_3 = b_1 \cdot a_2 \cdot o_3$, but $b_1 \cdot a_2 \neq []$. $\star$

The remaining cases are symmetrical. ■

The following lemma states the correctness of the partial-equivalence algorithm.

Lemma 2. *Let A and B be single-valued SFTs over $\mathcal{T}$ with same input sorts σ and same output sorts γ. If $Pred_{\mathcal{T}}(c{:}\sigma, c'{:}\sigma)$ is decidable, then partial-equivalence of A and B is decidable.*

Proof: For each state p, the while-loop verifies locally, that for any input enabled in p outputs will match up to maximum prefix of outputs from A and B, where p is associated with prior pending outputs $Q(p) = (\mathbf{a}, \mathbf{b})$ from (A, B), where at least one of $\mathbf{a}$ or $\mathbf{b}$ is $[]$. If p is a final state then $Q(p) = ([], [])$. The cases that cause violation of equivalence are the following, in the order of **FAIL**s:

1. There exist outputs of different length when q is final.
2. Some prefix of outputs differ.
3. Pending outputs differ for q, use Lemma 1.
4. If φ_2 is satisfiable there exist two different values for the (symbolic) pending output x' from A, use Lemma 1.
5. When φ_2 is not satisfiable, any model $M \models \varphi$ gives the same interpretation $\mathbf{a}'$ for the (symbolic) pending output $\mathbf{x}'$. If there is already a pending output for q that differs from $(\mathbf{a}', [])$, use Lemma 1.

If all local verifications hold, the equivalence follows, since the scope of the input variable is a single symbolic transition. Termination of the algorithm follows from termination of *Intersect*, termination of satisfiability checks, finiteness of $Q_A \times Q_B$, and that no member of $Q_A \times Q_B$ is pushed to S more than once. ■

The following is the main result regarding decidability of equivalence checking of SFTs.

Theorem 2. *Let A and B be single-valued SFTs over $\mathcal{T}$ with same input sorts σ and same output sorts γ. If $Pred_{\mathcal{T}}(c{:}\sigma, c'{:}\sigma)$ is decidable, then equivalence of A and B is decidable.*

Proof: For partial-equivalence use Lemma 2. For checking domain-equivalence of A and B construct corresponding SFAs D_A and D_B by removing the outputs from the symbolic transitions in A and B. Then check emptiness of $D_A - D_B$ and check emptiness of $D_B - D_A$ using the *difference* algorithm for SFAs [24]. ■

The following classical result is a corollary of Theorem 2, by considering a background $\mathcal{T}$ with a distinct constant for each member of the input and output alphabets and no relation symbols in $Pred_{\mathcal{T}}(c, c')$ besides equality.

Schützenberger [23] *The equivalence problem of single-valued finite transducers is decidable.*

Note however that [23] does not imply Theorem 2 since finite transducers cannot have infinitely many transitions, that are needed for example to represent symbolic transitions where $\mathcal{T}$ is linear arithmetic (the input alphabet is infinite). We believe that Theorem 2 can also be generalized for *finite-valued* SFTs A, i.e. when $T_A(v)$ is finite for all v, where $Pred_{\mathcal{T}}(c_0, \ldots, c_n)$ needs to be decidable for any n.

Regarding algorithmic complexity, there is also a clear advantage in the case of BEK of using SFTs rather than explicit representations with finite transducers (FTs), due to succinctness of SFTs. The background theory $\mathcal{T}$ is in this case k-bit bit vector arithmetic (restricted at the moment to comparison relations only) where k depends on the desired character range (e.g., for basic ASCII $k = 7$, for extended ASCII $k = 8$, and for Unicode $k = 16$). The 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 $O(n^2)$ [12] time. Thus, for an SFT A of size n, using the partial-equivalence algorithm for $[\![A]\!]$ takes $O(2^k n^2)$ time. However, the partial-equivalence algorithm for BEK SFTs is $O(n^2)$.

4.3 Other Algorithms

Other algorithms for SFTs can be realized using the above algorithms. One such algorithm, that is used below, is *inverse image* computation. A *symbolic finite automaton* or *SFA* B is an SFT such that all symbolic transitions in B have an empty output. Let A be an SFT and let B be an SFA such that $\gamma_A = \sigma_B$. The join composition $A \circ B$ is an SFA called the *inverse image of A under B*. The inverse image of *UptoLastDot* (from Figure 4) under the SFA for the regex `"^[a-z]\.$"` is shown in Figure 9. For SFAs the empty output is usually omitted from the transition labels in the figures.

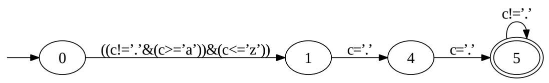

Figure 9: SFA *UptoLastDot* $\circ$ *SFA*(`"^[a-z]\.$"`).

5. Evaluation

Our implementation contains roughly 5,000 lines of C# code implementing the basic transducer algorithms and Z3 [11] integration, and 1,000 lines of F# code for translation from BEK. Our experiments were carried out on a Lenovo ThinkPad W500 laptop with 8GB of RAM and an Intel Core 2 Duo P9600 running at 2.67 GHz, running Windows 7 x64.

5.1 BEK Expressiveness

Compared to previous tools, such as Kaluza [21] or HAMPI [17], BEK supports a wider range of programs. Kaluza presently does not support transducers; special cases such as `replace` are lowered to a series of substrings and concatenations using concrete dynamic information. Furthermore, Kaluza can only answer queries for inputs up to a bounded length. In contrast, BEK can answer questions for inputs of unbounded lengths. Compared to HAMPI, we can handle replace for unbounded strings, and we support concatenation and equality of strings. Length abstractions are also encodable in BEK using symbolic finite transducers. To our knowledge BEK is the first engine capable of such analyses.

The symbolic finite transducer representation we introduce is also much more succinct than traditional transducers. We took 48 BEK programs and counted the number of edges in the resulting symbolic finite transducer. These programs included models of ASP.NET sanitization functions, as well as hand created test programs. We then estimated the number of edges that would be present in a non-symbolic encoding. Figure 10 plots the ratio of the number of edges between encodings and a trend line. The minimum expansion is 1.4x, while the maximum is 256x.

5.1.1 Deployed Web Sanitizers

AutoEscape and OWASP We converted 14 sanitizers from Google AutoEscape and the OWASP HTMLEncode sanitizer to

r4cm

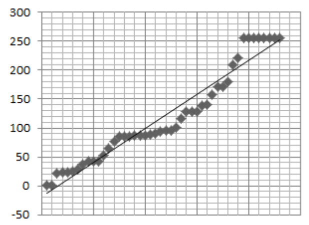

Figure 10: Ratio of number of edges in traditional finite transducer to edges in symbolic finite transducer for 46 BEK programs, including OWASP and Google AutoEscape sanitizers. The ratios are sorted from least to greatest and a trend line added.

BEK programs. For each sanitizer, we checked which features of the original implementation were natively present in the BEK language and which were not present. Figure 11 shows the language features used by these sanitizers. We conclude that few sanitizers use features that we do not natively support. In all these cases, we were able to implement work-arounds in the translation to BEK. Conversion took several days of work by one of the authors.

IE 8 XSS filters. We extracted 21 sanitizers from the binary of Internet Explorer 8 that are used in the IE Cross-Site Scripting Filter feature, denoted `IEFilter1` to `IEFilter18` in Figure 11. 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 express 17 as 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 an infinitely long chain of input characters.

We then focused on whether the IE 8 sanitizers 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.

PHP Builtin Functions. PHP is a widely-used open source server-side scripting language. Minamide's seminal work on the static analysis of dynamic web applications [18] 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. We then focused on how often functions are used that can by modeled as BEK programs.

We used statistics from a study by Hooimeijer [13] 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-

	Native			Not Native		
	boolean	multiple		mult.		
Name	vars	iters	regex	lookahead	arith.	functions
a2bb2a.bek	1	✗	✓	✗	✗	✗
escapeBrackets.bek	1	✓	✗	✗	✗	✗
escapeMetaAndLink.bek	1	✓	✓	✗	✗	✗
escapeString-allinone.bek	1	✗	✗	✗	✗	✗
escapeString.bek	1	✗	✗	✗	✗	✗
escapeStringSimple.bek	1	✗	✗	✗	✗	✗
getFileExtension.bek	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.bek	3	✗	✓	✗	✗	✗
IEFilter2.bek	4	✗	✓	✗	✗	✗
IEFilter3.bek	5	✗	✓	✗	✗	✗
IEFilter4.bek	4	✗	✓	✗	✗	✗
IEFilter5.bek	4	✗	✓	✗	✗	✗
IEFilter6.bek	5	✗	✓	✗	✗	✗
IEFilter7.bek	4	✗	✓	✗	✗	✗
IEFilter8.bek	4	✗	✓	✗	✗	✗
IEFilter9.bek	5	✗	✓	✗	✗	✗
IEFilter10.bek	5	✗	✓	✗	✗	✗
IEFilter11.bek	4	✗	✓	✗	✗	✗
IEFilter12.bek	4	✗	✓	✗	✗	✗
IEFilter13.bek	4	✗	✓	✗	✗	✗
IEFilter14.bek	4	✗	✓	✗	✗	✗
IEFilter15.bek	1	✗	✓	✗	✗	✗
IEFilter16.bek	1	✗	✓	✗	✗	✗
IEFilter17.bek	1	✗	✓	✗	✗	✗

Figure 11: Expressiveness: different language features used by the original 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 number of distinct Boolean variables used by the BEK implementation.

related call sites. By comparison, traditional regular expression functions modeled by tools like Hampi [17] and Rex [25] 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.

Language Features. When constructing the BEK language, we chose to include some language features, such as the ability to have Boolean variables, but excluded others, such as functions. Figure 11 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. Our theorems in section 4 guarantee that these features can be integrated into SFTs, however, by enhancing the language of constraints used for symbolic labels. In addition, 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 conclude that all current "Not Native" features could be added to the BEK language with few or no changes to our SFT algorithms for join composition and equivalence checking.

5.2 Idempotence, Reversibility, and Commutativity

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. Figure 12 reports 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. The number

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

Figure 12: 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 13: Commutativity matrix for seven different implementations of `HTMLEncode`. The `Outsourced` implementations were written by freelancers from a high level English specification.

of states acts as a rough guide to the complexity of the sanitizer. For example we see IE filter 9 out of 17 is quite complicated, with 25 states.

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 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 13 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%.

5.3 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 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

`HTMEncode1`	✓	✓	✓	`0`	—	✓	`0`
`HTMEncode2`	✓	✓	✓	`0`	—	✓	`0`
`HTMEncode3`	✓	✓	✓	`0`	—	✓	′
`HTMEncode4`	`0`	`0`	`0`	✓	`0`	`0`	`0`
`Outsourced1`	—	—	—	`0`	✓	—	`0`
`Outsourced2`	✓	✓	✓	`0`	—	✓	`0`
`Outsourced3`	`0`	`0`	′	`0`	`0`	`0`	✓

Figure 14: 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.

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 automata, 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. This acts as a sanitizer equivalence check.

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 `&amp;` 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 14 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 `HTMLEncodeFreelance0` escapes the − character, while `HTMLEncodeFreelance1` 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.

This case study shows the benefit of automatic analysis of string manipulating functions to check equivalence. 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 14 in less than one minute, including all equivalence and containment checks.

5.4 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 automata, 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.

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 15: Percentage of XSS Cheat Sheet strings, in both HTML and Attribute contexts, that are ruled out by each implementation of `HTMLEncode`

For each of our implementations, for all strings in each set, we then asked BEK whether pre-images of that string exist. Figure 15 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.

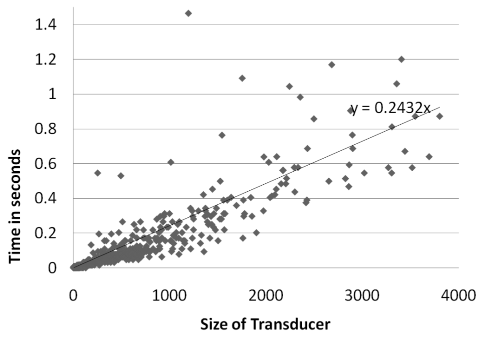

Figure 16: Self-equivalence experiment.

5.5 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 16 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 commutativity 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 17 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.

5.6 From BEK to Other Languages

In addition to analysis highlighted in our previous case studies, we have built compilers from BEK programs to commonly used languages. This allows a developer to write a sanitizer or other string manipulation function in BEK, enjoying the benefits of its analysis and visualization. When the time comes for deployment, the developer can compile to the language of her choice for inclusion into an application.

Figure 18 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

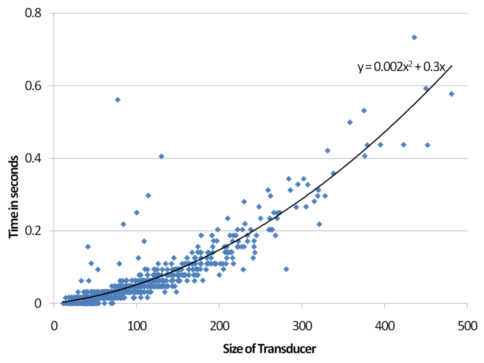

Figure 17: Commutativity experiment.

```
program test0(t);       function   test0(t)
                        {
string s;                 var s  =
                          function ($){
                          var result = new Array();
s := iter(c in t)         for(i=0;i<$.length; i++){
{b := false;} {           var c =$[i];
  case ((c == 'a')):        if ((c == String.fromCharCode(97)))
    b := !(b) && b;         {
    b := b || b;              b := (~(b) && b);
    b := !(b);                b := (b || b);
    yield (c);                b := ~(b);
                              result.push(c);
                            }
  case (true) :             if (tt)
    yield ('$');            {
                              result.push(String.fromCharCode(36));
                            }
                          };
                          return result.join('');
};                        }(str);
                          return s;
                        }
```

Figure 18: A small example BEK program (left) and its compiled version in JavaScript (right). Note the use of `result.push` instead of explicit array assignment.

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.

6. Related Work

Saner combines dynamic and static analysis to validate sanitization functions in web applications [7]. 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[18]. He shows how these grammars can be used to find pages with invalid HTML. 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.

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 [26]. Follow-on work combines this work with dynamic test input generation to find attacks on full PHP web applications [27]. Dynamic analysis of PHP code, using a combination of symbolic and concrete execution techniques, is implemented in the Apollo tool [6]. The work in [28] 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 precision of the analysis over scaling to large amounts of code.

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 [9]. 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 [17] and Kaluza [21] extend the STP solver to handle equations over strings and equations with multiple variables. Rex extends the Z3 solver to handle regular expression constraints [25], while Hooimeijer *et al.*show how to solve subset constraints on regular languages [14]. We in contrast show how to combine any of these solvers with finite automata whose edges can take symbolic values in the theories understood by the solver.

7. Conclusions

Developers can use BEK to write programs that are capable of handling common web sanitization tasks, then translate them automatically to symbolic finite transducers. This new symbolic finite transducer representation makes it possible to quickly answer queries about the sanitization function. Furthermore, our approach allows the class of programs to be extended by extending the theory used to annotate edges of the symbolic transducer. Our algorithms make it possible to check security properties of sanitization functions quickly, aiding developers of web applications. Finally, after using BEK to develop a sanitizer, the developer can compile to an existing language. BEK provides an all around solution for analysis and development of sanitization functions.

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] 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.
[7] 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 *SP*, 2008.
[8] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, 2009.
[9] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise Analysis of String Expressions. In *SAS*, 2003.
[10] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE-21*, 2007.
[11] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS. Springer, 2008.
[12] A. J. Demers, C. Keleman, and B. Reusch. On some decidable properties of finite state translations. *Acta Informatica*, 17:349–364, 1982.
[13] P. Hooimeijer. Decision procedures for string constraints. Ph.D. Dissertation Proposal, University of Virginia, April 2010.
[14] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 188–198, New York, NY, USA, 2009. ACM.
[15] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *ASE*, 2010.
[16] O. Ibarra. The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. *SIAM Journal on Computing*, 4:524–532, 1978.
[17] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, 2009.
[18] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th International Conference on the World Wide Web*, pages 432–441, 2005.

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

[20] 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.

[21] P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for javascript. In *IEEE Security and Privacy*, 2010.

[22] 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, August 2010.

[23] M. P. Schützenberger. Sur les relations rationnelles. In *GI Conference on Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 209–213, 1975.

[24] 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.

[25] M. Veanes, P. de Halleux, and N. Tillmann. Rex: Symbolic Regular Expression Explorer. In *ICST'10*. IEEE, 2010.

[26] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[27] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, 2008.

[28] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, pages 179–192, Berkeley, CA, USA, 2006. USENIX Association.