Generating Fast String Manipulating Code Through Transducer Exploration and SIMD Integration

Microsoft Research Technical Report MSR-TR-2011-124

Margus Veanes David Molnar Benjamin Livshits Lubomir Litchev

Microsoft Research, Redmond, WA 98052, USA

{margus,dmolnar,livshits,lubol}@microsoft.com

Abstract

Security sanitizers have long been known to be very difficult to implement correctly. Moreover, with the rise of the web, developers need string manipulating functions in both "server" and "client" languages. Hand-writing these functions separately is an open invitation to bugs. At the same time, auto-generated code will not be accepted unless it is significantly faster than previous hand-written code. We address this problem with two complementary approaches centered around Bek, a domain-specific language for writing complex string manipulation routines [8].

First, Bek compiles the input domain-specific program into an intermediate format consisting of symbolic finite state transducers, which extend classical transducers with symbolic predicates. In this paper, we present a novel algorithm that we call *exploration* which performs a *symbolic partial evaluation* of these transducers to obtain simplified, stateless versions of the original program. These simplified versions can be lifted back to Bek, and from there compiled to C#, C, or JavaScript. Second, we explore how SIMD instructions can be combined with Bek compilation to C and C, enabling developers to access parallel features of modern architectures without needing to tweak the C compiler or hand-write assembly.

We have implemented our code generation pipeline for Bek code corresponding to several real string sanitizers. We use an automatic testing approach to compare our generated code to the original C# implementations and found no semantic deviations. Our generated C# code outperforms the previous hand-tuned code by a factor of up to 2.5. For C code with SIMD, we see speedups of 2.5 times compared to native C code for the same sanitizer.

1. Introduction

Much of the motivation for this work comes from doing *security analysis* of large-scale modern web applications, both manually and using a combination of static and runtime analysis techniques. Improperly dealing with untrusted input is at the core of many web application vulnerabilities, as exemplified by cross-site scripting (XSS). 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.

However, sanitizers are surprisingly difficult to get right and are, therefore, often the Achilles heel of many a web application. Anecdotally, in dozens of code reviews performed across various industries, just about any custom-written sanitizer was flawed with respect to security [19]. The recent SANER work, for example, showed flaws in custom-written sanitizers used by ten web applications [2]. Another experiment solicited developers to implement sanitizers based on a verbal description; pretty much all the resulting sanitizers varied in subtle ways [8].

1.1 A Back-End for Bek

The problem of sanitize correctness becomes even more complicated when considering that it is not unusual for a complex web application today to be implemented in a variety of languages, including Java, C#, JavaScript, PHP, and C, spanning both the client and the server. As such, the task of creating *correct* and *consistent* sanitizers that work the same way across all languages is even more complex.

Sanitizers are typically small snippets 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.

This paper shows how programs written in Bek can be compiled down to traditional server- and client-side languages such as C#, C, and JavaScript. This ensures that the code analyzed and tested is functionally equivalent to the code which is actually deployed for sanitization. The focus of this work is on producing an *optimizing back-end* for Bek. Through a series of case studies, we demonstrate how our compilation step generates faster code than manually written sanitization routines. We also give developer a choice of semantically equivalent programs in different languages, with varying performance characteristics to pick from, observing that JavaScript output is usually the slowest, followed by C#, followed by regular C, whereas C with SIMD optimizations is typically the fastest.

1.2 Symbolic Branching Transducers

Key to our analysis is compilation from Bek programs to *symbolic branching transducers*, an extension of standard finite transducers. Recall that a finite transducer is a generalization of deterministic finite automaton 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 symbolic finite transducers [17], transitions are annotated with logical *formulas* instead of specific characters, and the transducer takes the transition on any input character that satisfies the formula.

Our symbolic 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, such as using binary decision diagrams, are specialized to specific types of inputs.

We extend symbolic transducers or STs with *branching rules* and *exception rules*, called ST^{b}s that allow us to be faithful in code generation. At the same time, the extension is a conservative extension of symbolic transducers (without branching rules). For analysis tasks, such as domain equivalence and partial equivalence, we reduce ST^{b}s to STs and apply the analysis techniques introduced in [17]. We introduce a new algorithm for exploration of ST^{b}s that preserves the branching structure of rules.

Figure 1 shows an example of a symbolic branching transducer. This transducer represents the CSSEncode sanitizer from the Microsoft AntiXSS library, version 4. The purpose of this sanitizer is to prevent JavaScript injection in the CSS (cascading style sheet) context. The original version, consisting of 206 lines of C# has been translated by us into 28 lines of Bek code. This sanitize features nested conditionals as well as exception handling, requiring the use of branching transducers. This transducer has two register, one a Boolean, the other a character. This is a fairly typical sanitizer found in libraries such as AntiXSS and others.

1.3 Paper Contributions

This paper has the following contributions.

- We show how to extend symbolic transducers with *branching rules.* These branching rules enable us to carry information about exceptions through transformations of the underlying transducers, allowing for a faithful representation of the original code. In Section 4 we describe how these branching rules allow us to automatically refactor an implementation of the CSSEncode sanitizer while preserving its exception behavior. We also show how the formula simplification feature of our SMT solver can produce more efficient predicates for evaluation of these transducers.
- We show how to compile from Bek into C#, JavaScript, and C. In particular we show how to check the resulting code for semantic differences from the original code. We present a transformation from Bek to C# that outperforms production hand-written versions of the same function by as much as 2.5x.
- Previous work on Bek introduced an extension to symbolic transducers called *registers* [17]. Registers remember small amounts of state and turn out to be essential for modeling real sanitizers. However, the presence of state complicates code parallelization. In this paper, we present a novel algorithm for state space exploration. The result of the exploration is a *stateless* program, which is advantageous for parallelism, as we effectively get rid of the state introduced by registers.
- We develop methods for integrating architectural SIMD instructions with compilation of Bek to C code. Explored transducers lend themselves more naturally to parallelization with SIMD, which results in additional time savings, as illustrated with our case study of a CSSEncode transducer in Section 5.3. Note that existing dependencies in both hand-written and pre-explored Bek programs make it very difficult for a compiler to generate SIMD code.

1.4 Paper Organization

The rest of this paper is organized as follows. Section 2 introduces symbolic branching transducers (SBTs). Section 3 presents algorithms for transducer exploration. Section 4 describes the Bek back-end, focusing on the translation process for C#, C++, and JavaScript. Section 5 provides our experiment evaluation. Finally, Sections 6 and 7 describe related work and conclude.

2. Symbolic Branching Transducers

We now formally introduce symbolic branching transducers or ST^{b}s and give examples of how ST^{b}s capture behavior of programs. We assume a *background structure* that has an effectively enumerable *background universe* $\mathcal{U}$, and is equipped with a language of function and relation symbols with fixed interpretations.

We use τ, σ and γ to denote types, and we write $\mathcal{U}^{\tau}$ for the corresponding sub-universe of elements of type τ. The Boolean type is bool, with $\mathcal{U}^{\mathsf{bool}} = \{\mathsf{t}, \mathsf{f}\}$, the integer type is int, and the type of k-bit bit-vectors is $\mathsf{bv}k$. The Cartesian product type of types σ and γ is $\sigma \times \gamma$. The type σ^* is the type for finite sequences of elements of type σ. The universe $\mathcal{U}^{(\sigma^*)}$ is the Kleene closure $(\mathcal{U}^{\sigma})^*$ of the universe $\mathcal{U}^{\sigma}$. We also write type σ^k as a semantic subtype of σ^* of sequences of elements of length *at most* $k \geq 0$.

Terms and formulas are defined by induction over the background language and are assumed to be well-typed. The

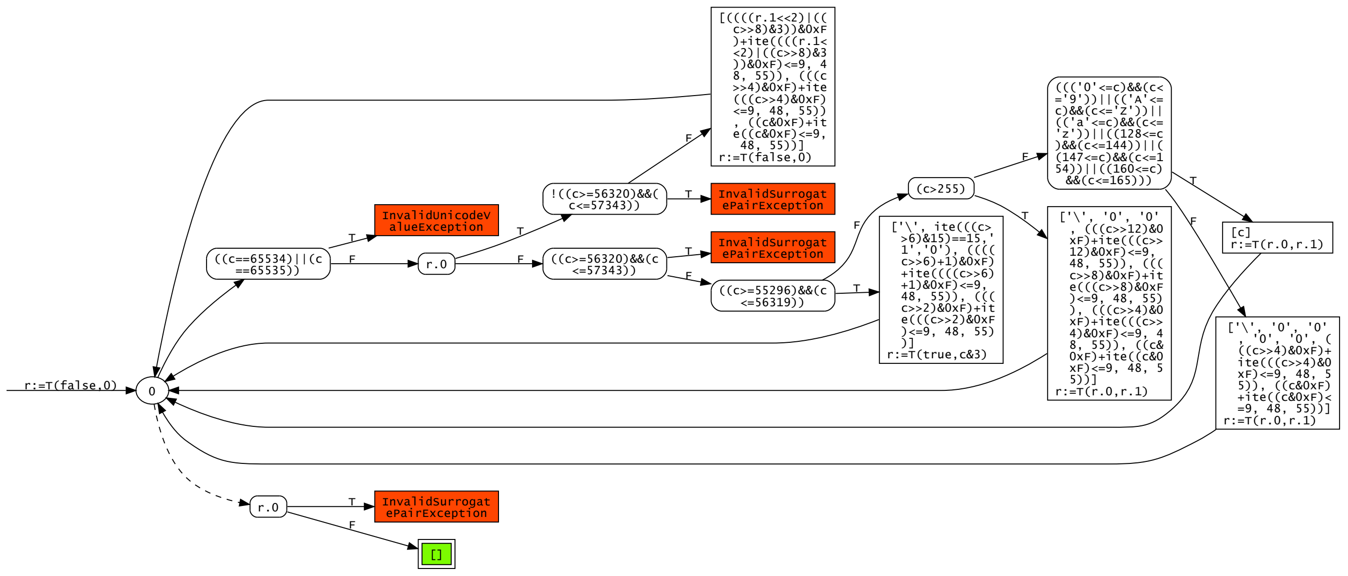
r:=T(false,0)
0
((c==65534)||(c==65535))
InvalidUnicodeValueException
r.0
!((c>=56320)&&(c<=57343))
[((((r.1<<2)|((c>>8)&3))&0xF)+ite((((r.1<<2)|((c>>8)&3))&0xF)<=9, 48, 55)), (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))]
r:=T(false,0)
InvalidSurrogatePairException
((c>=56320)&&(c<=57343))
InvalidSurrogatePairException
((c>=55296)&&(c<=56319))
['\', ite(((c>>6)&15)==15,'1','0'), ((((c>>6)+1)&0xF)+ite((((c>>6)+1)&0xF)<=9, 48, 55)), (((c>>2)&0xF)+ite(((c>>2)&0xF)<=9, 48, 55))]
r:=T(true,c&3)
(c>255)
((('0'<=c)&&(c<='9'))||(('A'<=c)&&(c<='Z'))||(('a'<=c)&&(c<='z'))||((128<=c)&&(c<=144))||((147<=c)&&(c<=154))||((160<=c)&&(c<=165)))
['\', '0', '0', (((c>>12)&0xF)+ite(((c>>12)&0xF)<=9, 48, 55)), (((c>>8)&0xF)+ite(((c>>8)&0xF)<=9, 48, 55)), (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))]
r:=T(r.0,r.1)
[c]
r:=T(r.0,r.1)
['\', '0', '0', '0', '0', (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))]
r:=T(r.0,r.1)
r.0
InvalidSurrogatePairException
[]
T
F

Figure 1. CSSEncode: an example of a symbolic branching transducer.

type τ of a term t is indicated by $t : \tau$. Terms of type bool, or Boolean terms, are treated as formulas, i.e., no distinction is made between formulas and Boolean terms. All elements in $\mathcal{U}$ are also assumed to have corresponding constants in the background language and we use elements in $\mathcal{U}$ also as constants. The set of free variables in a term t is denoted by $FV(t)$, t is *closed* when $FV(t) = \emptyset$, and closed terms t have Tarski semantics $[\![t]\!]$ over the background structure. Substitution of a variable $x : \tau$ in t by a term $u : \tau$ is denoted by $t[x/u]$.

A λ*-term* f is an expression of the form $\lambda x.t$, where $x : \sigma$ is a variable, and $t : \gamma$ is a term such that $FV(t) \subseteq \{x\}$; the type of f is $\sigma \to \gamma$; $[\![f]\!]$ denotes the function that maps $a \in \Sigma$ to $[\![t[x/a]]\!] \in \Gamma$. As a convention, we use f and g to stand for λ-terms. A λ-term of type $\sigma \to$ bool is called a σ-*predicate*. We write φ and ψ for σ-predicates and, for $a \in \Sigma$, we write $a \in [\![\varphi]\!]$ for $[\![\varphi]\!](a) = \mathfrak{t}$. We often treat $[\![\varphi]\!]$ as a *subset* of Σ. Given a λ-term $f = (\lambda x.t) : \sigma \to \gamma$ and a term $u : \sigma$, $f(u)$ stands for $t[x/u]$. A predicate φ is *unsatisfiable* when $[\![\varphi]\!] = \emptyset$; *satisfiable*, otherwise.

The main building block of an ST^{b} is a *rule*. A rule is an expression that denotes a partial function corresponding to a straight-line conditional statement of a program that may *yield outputs*, *produce updates* and *raise exceptions*. We first provide an inductive definition of rules that omits type annotations. We then define additional well-formedness criteria and the semantics for rules.

- $\bot$ is the *exception rule*.
- If f is a λ-term then $br(f)$ is a *basic rule*.
- If φ is a predicate and r_1, r_2 are rules then $ite(\varphi, r_1, r_2)$ is an *if-then-else (ite) rule*.

We say that a rule r is *well-formed* with respect to the type $\sigma \to \gamma$, denoted $r : \sigma \to \gamma$, when one of the following conditions holds:

- r is the rule $\bot$.
- r is a rule $br(f : \sigma \to \gamma)$.
- r is a rule $ite(\varphi : \sigma \to \text{bool}, r_1 : \sigma \to \gamma, r_2 : \sigma \to \gamma)$.

A rule $r : \sigma \to \gamma$ represents a function $[\![r]\!]$ from $\mathcal{U}^\sigma$ to $\mathscr{P}(\mathcal{U}^\gamma)$ For all $a \in \mathcal{U}^\sigma$:

$$\begin{aligned} [\![\bot]\!](a) &\overset{\text{def}}{=} \emptyset \\ [\![br(f)]\!](a) &\overset{\text{def}}{=} \{[\![f]\!](a)\} \\ [\![ite(\varphi, r_1, r_2)]\!](a) &\overset{\text{def}}{=} \begin{cases} [\![r_1]\!](a), & \text{if } a \in [\![\varphi]\!]; \\ [\![r_2]\!](a), & \text{otherwise.} \end{cases} \end{aligned}$$

We now introduce the central definition of a symbolic branching transducer that uses the definition of rules.

Definition 1: A *Symbolic Branching Transducer* or ST^{b} A with *input* type σ, *output* type γ and *state* type τ is a tuple (q^0, R, F), where

- $q^0 \in \mathcal{U}^\tau$ is the *initial state*;
- R is a finite set of rules of type $(\sigma \times \tau) \to (\gamma^k \times \tau)$, for some $k \geq 0$, rules in R are called the *input rules* of A;
- F is a finite set of rules of type $\tau \to \gamma^k$, for some $k \geq 0$, rules in F are called the *final rules* of A.

For a basic subrule $r = br(\lambda(x, y).\langle f(x, y), g(x, y)\rangle)$ of an input rule, f is called the *yield* and g the *update* of r. A basic subrule of a final rule is called a *final yield*. ⊠

We write $p \xrightarrow{a/\boldsymbol{b}}_A q$ for a *concrete transition* of A: there exists $r \in R_A$ such that $[\![r]\!](a, p) = \{(\boldsymbol{b}, q)\}$. Similarly, we write $q \xrightarrow{/\boldsymbol{b}}_A$ for a *final output* of A: there exist $r \in F_A$ such that $[\![r]\!](q) = \{\boldsymbol{b}\}$. Intuitively, a final output is a special case of an input-epsilon move of a classical finite state transducer into a final state, but it is algorithmically useful to keep final rules separate from general input-epsilon moves. Unlike input-epsilon moves in general, final rules do not affect the core algorithms, while providing a very convenient mechanism to yield additional outputs upon reaching the end of the input tape.

We write $A^{\sigma/\gamma;\tau}$ to indicate the input/output types σ/γ and the state type τ of an ST^{b} A. In the following we use the abbreviations $\Sigma = \mathcal{U}^\sigma$, $\Gamma = \mathcal{U}^\gamma$ and $Q = \mathcal{U}^\tau$.

```
program DecodeDigitPairs(input) {
  return iter(x in input) [y := 0;] {
    case (y == 0):  //no previous digit was recorded
      if ((x>='5')&&(x<='9'))
        {y:=x;}     //store the digit in register y
      else
        {yield(x);} //output directly
    case (true):    // y != 0 so y is the previous digit
      if ((x>='5')&&(x<='9'))
        {yield((10*(y-48))+(x-48)); //decode the letter
         y:=0;}
      else
        {yield(x);} //output directly
    } end {
    case (y != 0):
      yield (y);    //there was a digit at the end
  };
}
```

Figure 2. Sample BEK program.

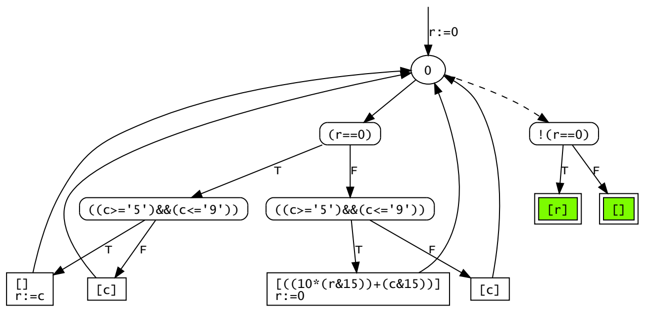

Figure 3. Depiction of the ST$^{\mathrm{b}}$ in Figure 2. Dashed arrows correspond to final rules. Oval nodes correspond to branch conditions and rectangular nodes correspond to basic rules.

The *reachability relation* $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q$ for $\boldsymbol{a} \in \Sigma^*$, $\boldsymbol{b} \in \Gamma^*$, and $p, q \in Q$ is defined through the closure under the following conditions, where '$\cdot$' is concatenation of sequences:

- For all $q \in Q$, $q \xrightarrow{\epsilon/\epsilon}\!\!\!\!\rightarrow_A q$.
- If $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A p'$ $\xrightarrow{a/\boldsymbol{c}}_A q$ then $p \xrightarrow{\boldsymbol{a}\cdot a/\boldsymbol{b}\cdot\boldsymbol{c}}\!\!\!\!\rightarrow_A q$.

Definition 2: The *transduction* of an ST$^{\mathrm{b}}$ A, $\mathscr{T}_A$, is the following function from Σ^* to $\mathscr{P}(\Gamma^*)$.

$$\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b}\cdot\boldsymbol{c} \mid \exists q \in Q\,(q_A^0 \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q \xrightarrow{/\boldsymbol{c}}_A)\}$$

⊠

We say that A is *single-valued* when $|\mathscr{T}_A(\boldsymbol{a})| \leq 1$ for all $\boldsymbol{a} \in \Sigma^*$. A is *deterministic* when $|R_A| = 1$ and $|F_A| = 1$. Note that determinism immediately implies single-valuedness. In this paper we will only consider deterministic ST$^{\mathrm{b}}$s and we will identify R_A (resp. F_A) with the rule it contains.

The following example illustrates the use of ST$^{\mathrm{b}}$s on a typical string transformation scenario and introduces the concrete language BEK that we use for defining ST$^{\mathrm{b}}$s in this paper.

Example 1 Let the input type, output type, and state type be bv7. (Intuitively $\mathcal{U}^{\mathsf{bv7}}$ corresponds to the set of ASCII characters). The BEK program in Figure 2 corresponds to an ST$^{\mathrm{b}}$ that decodes certain occurrences of pairs of digits between 5 and 9 by their corresponding ASCII letters. For example `DecodeDigitPairs("a77")` is `"aM"`.

The initial state is 0 that corresponds to the initial value of y. Let f be the λ-term $\lambda(x, y).((10*(y-48))+(x-48))$ and let φ be the predicate $\lambda(x, y).(\texttt{'5'} \leq x \leq \texttt{'9'})$. The case and if-then-else statements map directly to the following input rule where we lift the λ-prefix to be in the front:

$$\begin{aligned}\lambda(x, y).\mathit{ite}(y = 0, \quad & \mathit{ite}(\varphi(x, y), \mathit{br}([], x), \mathit{br}([x], y)),\\ & \mathit{ite}(\varphi(x, y), \mathit{br}([f(x, y)], 0), \mathit{br}([x], y)))\end{aligned}$$

Similarly, the final rule corresponds to the case statement after the end-construct:

$$\lambda y.\mathit{ite}(y \neq 0, [y], [])$$

Observe that, in this case there are no exception rules. The ST$^{\mathrm{b}}$ is defined for all input strings. Use of exception rules is illustrated in our main case studies below. The graphical illustration of the ST$^{\mathrm{b}}$ for `DecodeDigitPairs` is shown in Figure 3. All graphs in the paper are produced automatically from our analysis framework. ■

3. Exploration of ST$^{\mathrm{b}}$s

In this section we develop an algorithm that allows us to eliminate either all or some of the state registers used in a deterministic ST$^{\mathrm{b}}$ A. In particular, we focus on two, most prominent cases:

- *Full* exploration.
- *Boolean* exploration.

For the purpose of explaining the exploration algorithm, we extend $A = (q^0, R, F)$ with a component P that is a finite set of *control states* and an initial control state $p^0 \in P$. The sets R and F are extended to be maps from P to rules, and each basic subrule on an input rule in R has an additional control state component $p \in P$. With this extension in mind, we write a basic rule as $\mathit{br}(\mathit{yield}, \mathit{update}, p)$. We say that A is *stateless* when the register type τ is the unit type $\mathsf{T0}$ ($\mathcal{U}^{\mathsf{T0}} = \{\langle\rangle\}$), i.e., registers are not used in a stateless ST$^{\mathrm{b}}$, and thus R has the equivalent form

$$\{p_1 \mapsto r_1, p_2 \mapsto r_2, \ldots, p_{|R|} \mapsto r_{|R|}\}$$

where each rule r_i corresponds to a conditional statement that may yield outputs and transition to new control states but does not make use of registers by storing intermediate results in registers. This extension is useful for separation of concerns, it helps to keep the control state separate from the data state.

For example, the ST$^{\mathrm{b}}$ is Example 1 is not stateless because the rules depend on the register y.

By *full exploration* of A, we mean a construction of a stateless ST$^{\mathrm{b}}$ $A^{\mathbf{f}}$ such that $\mathscr{T}_A = \mathscr{T}_{A^{\mathbf{f}}}$, i.e., A and $A^{\mathbf{f}}$ are equivalent. Full exploration is not always possible, because equivalence of stateless ST$^{\mathrm{b}}$s reduces to equivalence of symbolic finite transducers (SFTs), and equivalence of SFTs is decidable [17] modulo a decidable label theory, while equivalence of ST$^{\mathrm{b}}$s is undecidable already for very restricted decidable label theories. Even when full exploration is possible, $A^{\mathbf{f}}$ may still be exponentially larger than A.

By *Boolean exploration* of A, we mean a construction of an ST$^{\mathrm{b}}$ $A^{\mathbf{b}}$ such that $\mathscr{T}_A = \mathscr{T}_{A^{\mathbf{b}}}$ where all Boolean registers of A have been eliminated. For example, if the state type of A is $(\mathsf{bool} \times \mathsf{bool}) \times \mathsf{int}$ then the the state type of $A^{\mathbf{b}}$ is int, i.e., the two Boolean registers have been eliminated by adding new control states.

Note that, in order to completely eliminate the symbolic update of a rule $\mathit{br}([], \lambda(x, y).\varphi(x))$, where φ is a σ-predicate,

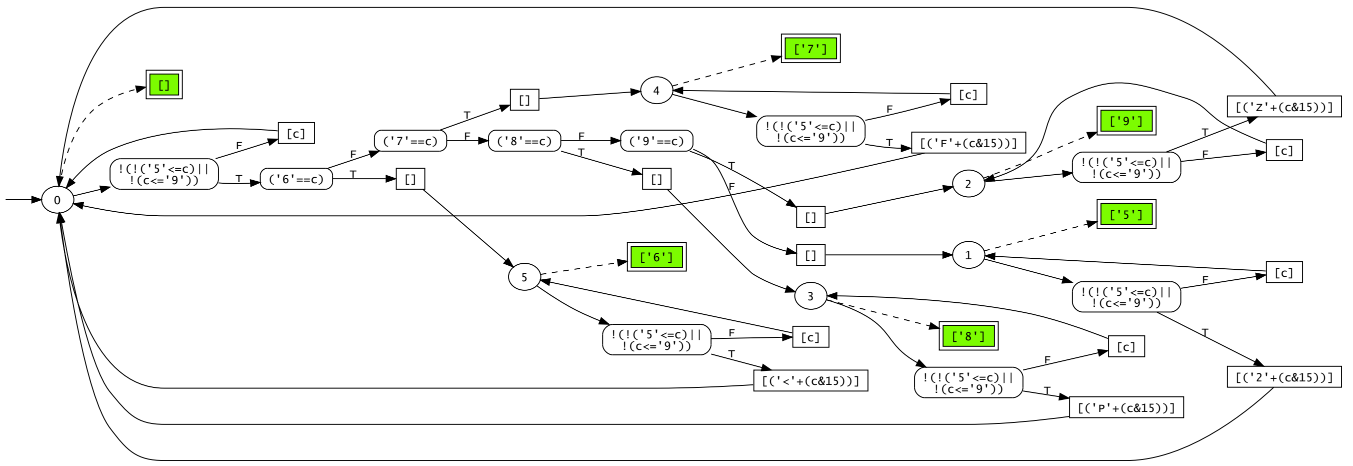

Figure 5. ST^{b} after full exploration of `DecodeDigitPairs` in Figure 2.

i.e., to replace φ by $\lambda x.\mathfrak{t}$ (resp. $\lambda x.\mathfrak{f}$) we would need to decide if $\forall x\, \varphi(x)$ holds, i.e., $\neg\varphi$ is unsatisfiable, (resp. if $\forall x\, \neg\varphi(x)$ holds, i.e., φ is unsatisfiable).

Algorithm. The generic exploration algorithm of ST^{b}s is described in figure 4. The algorithm takes as its input an ST^{b} A, and assumes a projection of the state type τ of A into two parts τ_1 and τ_2. We assume, without loss of generality, that $\tau = \tau_1 \times \tau_2$. The algorithm uses an SMT solver to solve satisfiability and to generate models for formulas.

The algorithm generates a new ST^{b} by exploring the rules with respect to τ_1, effectively eliminating τ_1, i.e. turning it into an explicit state. In order to avoid special cases, we may always assume that either τ_1 or τ_2 can be unit types $\mathtt{T0}$ ($\mathcal{U}^{\mathtt{T0}} = \{\langle\rangle\}$). Now, full exploration of A corresponds to the case when τ_2 is unit type, and Boolean exploration corresponds to the case when τ_1 is a Cartesian combination of Boolean registers and τ_2 is a Cartesian combination of all the non Boolean registers.

Inst(φ,r,p) creates an instance of the rule r with the path condition φ with respect to the fixed register values given by p. For the exception rule this is a noop. For a basic rule this is a partial instantiation of the yield and update with respect to p, where $\lambda y.f(p,y)$ instantiates the first projection of the state register with the value p. An important point for the rules is that unreachable rule instances are incrementally eliminated by deciding satisfiability of corresponding accumulated path conditions.

Expl(φ,r,*add*); is a form of partial exploration of r the with respect to τ_1 or the projection projection function. For the exception rule the operation is a noop. For an ite rule, the step is a direct propagation of the concretizations of the branches. The core of the computation takes place during the concretization of basic rules.

Theorem 1: *Let A be a deterministic* ST^{b} *with state type $\tau_1 \times \tau_2$. If Explore(A) terminates then the result is an* ST^{b} *that is equivalent to A and whose state type is τ_2.*

We omit the formal proof of the theorem but note that termination of the algorithm depends on two factors: decidability of the background theory, and finiteness of the reachable subset of $\mathcal{U}^{\tau_1}$. The first point is already needed in the *Inst* procedure that eliminates unsatisfiable branches. The second point is needed both, for termination of construction of r in *Expl*, as well as for guaranteeing that the search stack is bounded in size. A sufficient condition for the second point is when the functions used for computing the first state projection have the *finite-range* property, i.e., when $\mathcal{U}^{\tau_1}$ can be assumed to be finite.

Example 2 The ST^{b} after full exploration of `DecodeDigitPairs` from Figure 2, is illustrated in Figure 5. The unexplored ST^{b} (in Figure 3) has a single control state 0, while the fully explored ST^{b} has 6 control states. ■

4. Translation

Now that we have described how to refactor ST^{b}s to expose parallelism in BEK programs, we turn to the question of optimizing BEK programs at the symbolic transducer level, then translating BEK to other languages. Our motivation is to "optimize once, analyze once, run everywhere," in contrast to current practice where each language has an independently implemented version of the "same" sanitizer. First, we discuss the question of *exceptions* : we want our generated code to throw exceptions on all the same inputs where the original code threw exceptions. We describe how the *branching rules* in ST^{b}s allow us to carry information through the exploration algorithm. Next, we describe how we employ an SMT solver to simplify formulas that arise inside the ST^{b}s. Finally, we discuss how ST^{b}s can be lifted back to BEK abstract syntax trees, and how we can compile to languages such as C# and JavaScript.

4.1 Exceptions and Branching Rules

Exceptions are common in string manipulating functions, such as web sanitizers. Different error conditions may require different exceptions. For example, the CSSEncode function that ships with the Microsoft AntiXSS library checks for two error conditions: first, input strings that have invalid Unicode values, and second, for invalid "surrogate pairs" of characters. Unfortunately, traditional symbolic transducers do not have an explicit way to express exception behavior. This leaves us with one of two options when modeling the behavior of such functions on inputs that cause exceptions. Either we sacrifice our precise modeling of the behavior of the original program, or we must encode exceptions in an ad hoc way into finite transducers. The first option sacrifices our promise of precision. The second may not be robust to algorithms that manipulate the transducer structure, or

```
$\boldsymbol{Explore}(A^{\sigma/\gamma;\tau_1\times\tau_2}) \stackrel{\text{def}}{=}$
  $p_0 := \mathit{first}(q^0_A)$;
  $q_0 := \mathit{second}(q^0_A)$;
  $S := \mathit{stack}(q^0)$;
  $P := \{p^0\}$;
  $\mathit{Add} \stackrel{\text{def}}{=} \boldsymbol{\lambda} p.$ if $p \notin P$ then $P := P \cup \{p\}$; $\mathit{Push}(S, p)$;
  $R := \{\mapsto\}$;
  $F := \{\mapsto\}$;
  while $S \neq \emptyset$
    $p := \mathit{Pop}(S)$;
    $R(p) := \mathit{Expl}(\lambda y : \tau_2.\mathrm{t}, \mathit{Inst}(\lambda y : \tau_2.\mathrm{t}, R_A, p), \mathit{Add})$;
    $F(p) := \mathit{Expl}(\lambda y : \tau_2.\mathrm{t}, \mathit{Inst}(\lambda y : \tau_2.\mathrm{t}, F_A, p), \mathit{Add})$;
  return $(p^0, q^0, R, F)$;

$\boldsymbol{Inst}(\varphi, \bot, p) \stackrel{\text{def}}{=}$ return $\bot$;
$\boldsymbol{Inst}(\varphi, \mathit{br}(f, g), p) \stackrel{\text{def}}{=}$ return $\mathit{br}(\lambda y.f(p, y), \lambda y.g(p, y))$;
$\boldsymbol{Inst}(\varphi, \mathit{ite}(\psi, t, f), p) \stackrel{\text{def}}{=}$
  $\varphi_t := \lambda y.\varphi(y) \wedge \psi(p, y)$;
  $\varphi_f := \lambda y.\varphi(y) \wedge \neg\psi(p, y)$;
  if $\mathit{IsUnsatisfiable}(\varphi_t)$ return $\mathit{Inst}(\varphi_f, f, p)$;
  else if $\mathit{IsUnsatisfiable}(\varphi_f)$ return $\mathit{Inst}(\varphi_t, t, p)$;
  else return $\mathit{ite}(\lambda y.\psi(p, y), \mathit{Inst}(\varphi_t, t, p), \mathit{Inst}(\varphi_f, f, p))$;

$\boldsymbol{Expl}(\varphi, \bot, \mathit{Add}) \stackrel{\text{def}}{=}$ return $\bot$;
$\boldsymbol{Expl}(\varphi, \mathit{ite}(\psi, t, f), A) \stackrel{\text{def}}{=}$
  return $\mathit{ite}(\varphi, \mathit{Expl}(\varphi \wedge \psi, t, \mathit{Add}), \mathit{Expl}(\varphi \wedge \neg\psi, f, \mathit{Add})$;
$\boldsymbol{Expl}(\varphi, \mathit{br}(f, g), A) \stackrel{\text{def}}{=}$
  $\psi := \lambda y\, z\, .\varphi(y) \wedge (z = \mathit{first}(g(y)))$;
  $r := \bot$;
  while $\exists M \models \psi$
    $r := \mathit{ite}(\lambda y.p = \mathit{first}(g(y)), \mathit{br}(f, \lambda y.\mathit{second}(g(y)), p), r)$;
    $\psi := \lambda y\, z\, .\psi(y, z) \wedge z \neq z^M$;
    $\mathit{Add}(z^M)$;
  return $r$;
```

Figure 4. Exploration algorithm of $\mathrm{ST^b}$s.

```
// Calculate the combined code point
long ccP =
0x10000 + ((ccP - 0xD800) * 0x400)+ (ncP - 0xDC00);
char[] encodedCharacter =
SafeList.SlashThenSixDigitHexValueGenerator(ccP);
for (int j = 0; j < encodedCharacter.Length; j++)
{
    encodedInput[outputLength++] = encodedCharacter[j];
 }
```

Figure 6. Code snippet from hand-written CSSEncode implementation. This code makes use of a computed hash table to map characters to their safe encoded equivalents. The hash table obscures the parallel assignment structure that is possible with CSSEncode.

may have difficulty distinguishing between different types of exceptions.

```
if (...) {}
else if (r0) {
    if (((56320 > c) || (c > 57343)))    {
    throw new Exception("InvalidSurrogatePair");
     }    else  {
      output[pos++] = ((char)(((((c >> 8) & 3) |
      (r1 << 2)) & 0xF) +
     (((((c >> 8) & 3) |
     (r1 << 2)) & 0xF) <= 9 ? 48 : 55)));
     output[pos++] = ((char)(((c >> 4) & 0xF)
     + (((c >> 4) & 0xF) <= 9 ? 48 : 55)));
     output[pos++] = ((char)((c & 0xF)
      + ((c & 0xF) <= 9 ? 48 : 55)));
     r0 = false; r1 = 0;
     }
```

Figure 7. Code snippet from unexplored Bek CSSEncode implementation. While this code makes explicit the parallel assignment structure, it also has a nested if statement that depends on the value of the register r1. Because r1 may change, such a dependency is difficult to encode using SIMD instructions.

```
if (((56320 <= c) && (c <= 57343)))
{
   throw new Exception("InvalidSurrogatePair");
} else {
   if (((55296 <= c) && (c <= 56319)))
   {
    if ((!(1 == ((c >> 0) & 1)) &&
   !(1 == ((c >> 1) & 1))))
    {
     output[pos++] = ((char)92);
     output[pos++] = ((char)(!(1 ==
        (((c >> 6) >> 0) & 1))
      || !(1 == (((c >> 6) >> 1) & 1))
      || !(1 == (((c >> 6) >> 2) & 1))
      || !(1 == (((c >> 6) >> 3) & 1)) ? 48 : 49));
      output[pos++] = ((char)(((1 + (c >> 6)) & 0xF) +
      (((1 + (c >> 6)) & 0xF) <= 9 ? 48 : 55)));
```

Figure 8. Code snippet from explored Bek CSSEncode implementation. The dependence on register r1 has been removed through the exploration algorithm. While there are more nested if statements, each of them depend only on the individual character. This code therefore lends itself to SIMD instructions.

Our solution is to encode exceptions as branching rules in an $\mathrm{ST^b}$. Because $\mathrm{ST^b}$s treat branching rules as an explicit part of the formalism, algorithms on $\mathrm{ST^b}$s will preserve exception semantics. For example, Figure 9 shows an $\mathrm{ST^b}$ for the CSSEncode sanitizer after applying the exploration algorithm of the previous section. We can see that the $\mathrm{ST^b}$ includes rules for both kinds of exceptions. This enables an implementation that is exception-compatible with the original hand-written CSSEncode. In the concrete implementation of the $\mathrm{ST^b}$ exploration algorithm, the exception rules carry an additional label (e.g. `InvalidSurrogatePairException`), although the formal definition indicates exception rules by $\bot$ and does not distinguish them explicitly.

4.2 Formula Simplification

In addition to expressing exception behavior, our use of branching rules makes it easier to apply formula simplification by breaking up formulas into smaller pieces. Again in Figure 9 we see that different predicates evaluated on the input are placed in different branching rules. In a traditional symbolic finite state transducer, this structure would have

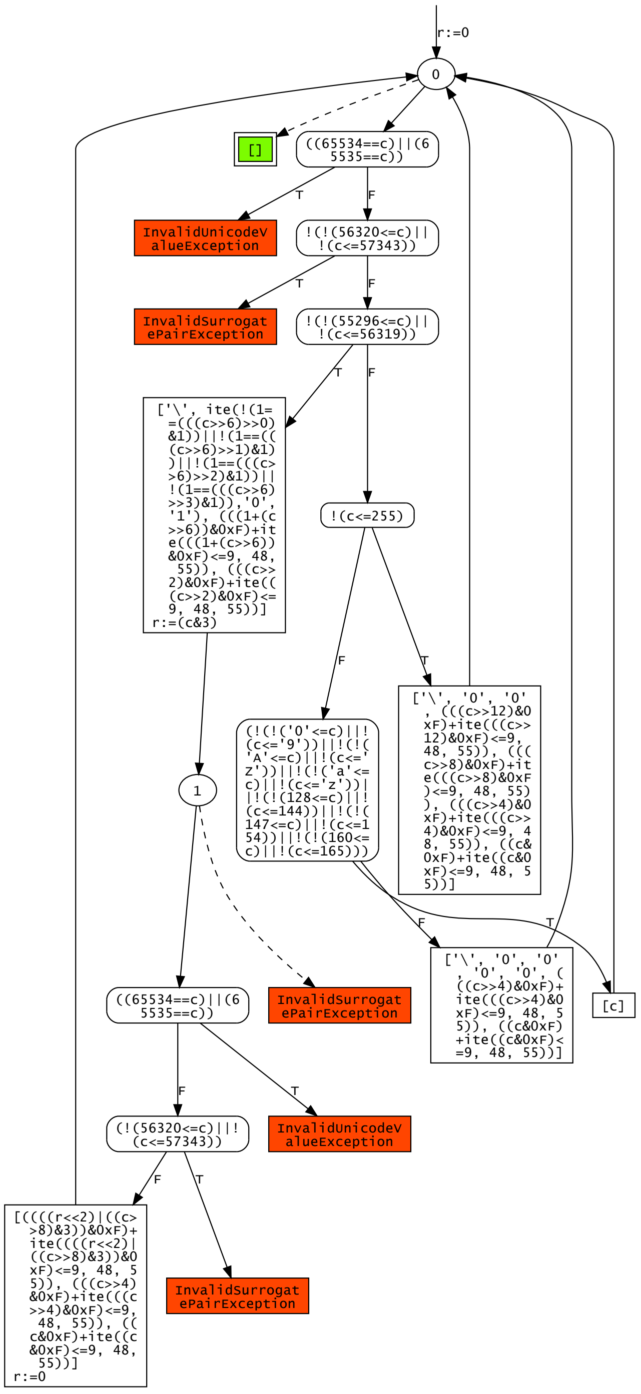
r:=0
0
[]
((65534==c)||(65535==c))
T
F
InvalidUnicodeValueException
!(!(56320<=c)||!(c<=57343))
T
F
InvalidSurrogatePairException
!(!(55296<=c)||!(c<=56319))
T
F
['\', ite(!(1==(((c>>6)>>0)&1))||!(1==(((c>>6)>>1)&1))||!(1==(((c>>6)>>2)&1))||!(1==(((c>>6)>>3)&1)),'0','1'), (((1+(c>>6))&0xF)+ite(((1+(c>>6))&0xF)<=9, 48, 55)), (((c>>2)&0xF)+ite(((c>>2)&0xF)<=9, 48, 55))] r:=(c&3)
!(c<=255)
F
T
['\', '0', '0', (((c>>12)&0xF)+ite(((c>>12)&0xF)<=9, 48, 55)), (((c>>8)&0xF)+ite(((c>>8)&0xF)<=9, 48, 55)), (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))]
(!(!('0'<=c)||!(c<='9'))||!(!('A'<=c)||!(c<='Z'))||!(!('a'<=c)||!(c<='z'))||!(!(128<=c)||!(c<=144))||!(!(147<=c)||!(c<=154))||!(!(160<=c)||!(c<=165)))
1
F
T
['\', '0', '0', '0', '0', (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))]
[c]
((65534==c)||(65535==c))
InvalidSurrogatePairException
F
T
(!(56320<=c)||!(c<=57343))
InvalidUnicodeValueException
F
T
[((((r<<2)|((c>>8)&3))&0xF)+ite((((r<<2)|((c>>8)&3))&0xF)<=9, 48, 55)), (((c>>4)&0xF)+ite(((c>>4)&0xF)<=9, 48, 55)), ((c&0xF)+ite((c&0xF)<=9, 48, 55))] r:=0
InvalidSurrogatePairException

Figure 9. ST^b after Boolean exploration of CssEncode (register `r0` has been eliminated). Information about different types of exceptions has been preserved even though the structure of the ST^b has changed.

been encoded into the formula itself. Our implementation uses recent formula simplification features built into the Z3 solver; we recursively apply this simplification to each formula in a rule.

	States		
Sanitizer	**ST**	**ST_B**	**ST_F**
UTF8Encode	1	2	5
CssEncode	1	2	5
HtmlDecode	1	5	113

Figure 10. Input statistics. Number of control states. ST: unexplored, ST_B: post-Boolean exploration; ST_F: post-Full exploration. HtmlDecode is limited to decimal numeric encodings of ≤ 2 digits.

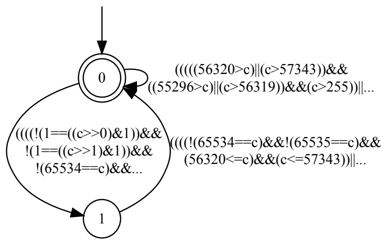
0
(((((56320>c)||(c>57343))&&((55296>c)||(c>56319))&&(c>255))||...
((((!(1==((c>>0)&1))&&!(1==((c>>1)&1))&&!(65534==c)&&...
((((!(65534==c)&&!(65535==c)&&(56320<=c)&&(c<=57343))||...
1

Figure 13. SFA for generating inputs for correctness testing.

Language	Implementation	Runtime
C#	AntiXSS hand-written	18.174
C#	Bek default	6.864
C#	Bek_B	7.020
C#	Bek_F	7.239
C	original	1.497
C	Bek_B	1.872
C	Bek_F	1.919

Figure 14. CSSEncode implementations compared.

4.3 Lifting To AST and Compilation

Finally, we lift from ST^bs to Bek abstract syntax trees. We express the state of the ST^bas a Bek variable inside an iteration block. A series of if statements in Bek checks the current state, then evaluates the appropriate formulas. Once we have a Bek abstract syntax tree, we compile to C# and JavaScript. The key observation is that the structure of a Bek program corresponds to a `foreach` loop carried out over each character in an input string. We then define a visitor over the Bek abstract syntax tree which translates to the appropriate syntax for the target language.

As an example, Figure 6 shows a piece of the original hand-written CSSEncode sanitizer, which uses a table lookup to map potentially dangerous characters to their safe equivalents. Because Bek does not allow table lookups, we represent the actual character ranges in the translated code. Figure 7 shows an snippet of the resulting compilation to C# before any exploration. This code shows the dependence on a register variable r0. Finally in Figure 8 we see an example of the code after exploration is complete compiled to C#; the register variable dependence has been eliminated.

5. Evaluation

Exploration overhead: First we discuss the cost of our exploration algorithm in terms of the speed to perform ex-

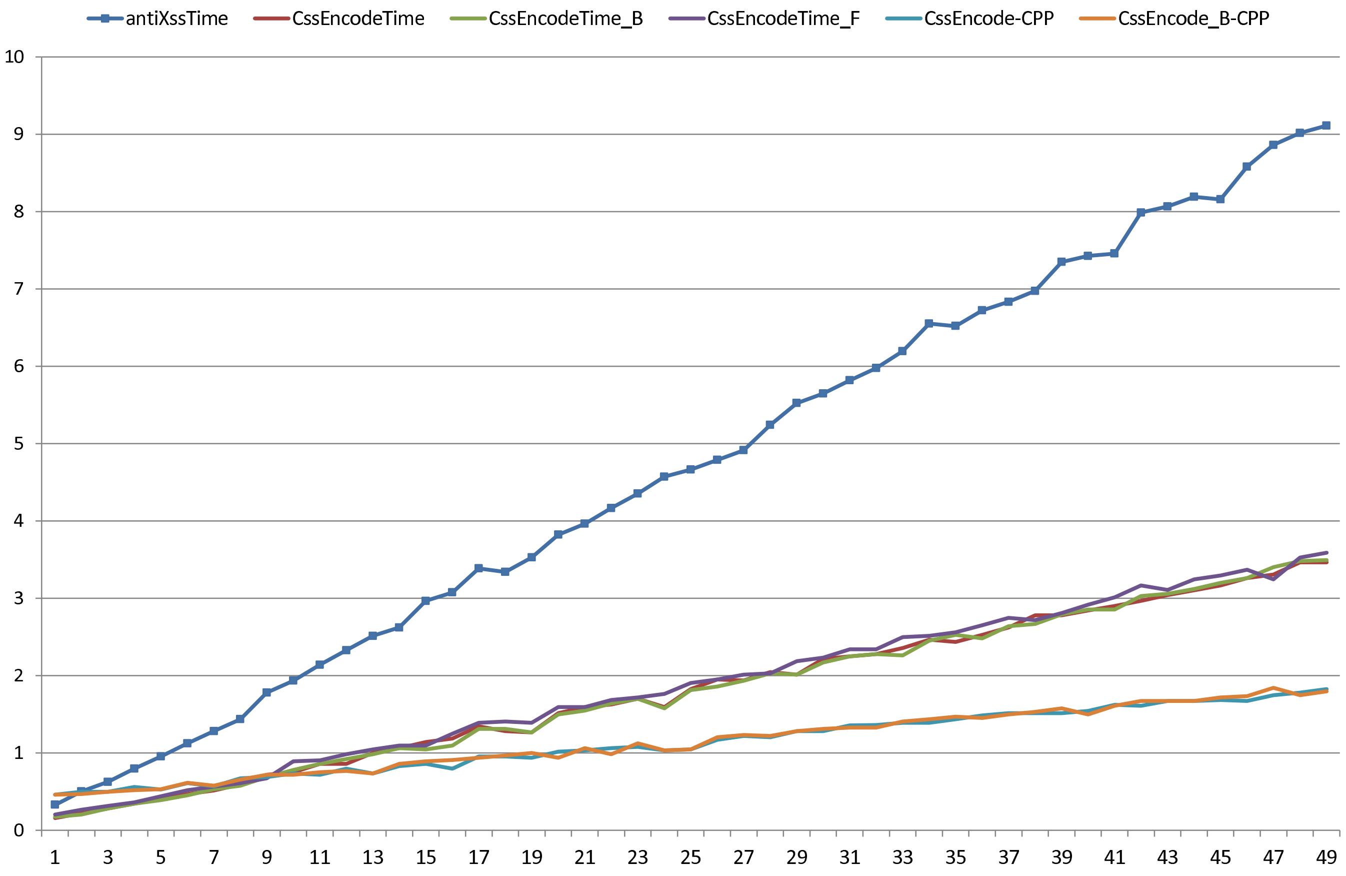

Figure 11. CSS encoder timing comparison over a range of input sizes.

Language	Implementation	Runtime
C#	.NET	3.682
C#	Bek_F	2.496
C	Bek explored	0.343

Figure 15. UTF8Encode implementations compared.

	C	SIMD	Ratio
Full All	1,124	967	1.16
Full ASCII-only	718	671	1.07
Promise ASCII	188	15	12.53
Promise HighChar	889	344	2.58

Figure 16. Table showing speed in ms of C and C/SIMD implementations of fully explored Bek CSSEncode. The first two lines refer to running the "full" implementations on inputs consisting of all valid characters and inputs restricted to ASCII characters. The next two lines reflect optimizations that are possible if we can be sure that the inputs consist only of specific characters. While such a "promise" is not reasonable in general, these numbers demonstrate that in settings that can restrict characters a priori, SIMD offers a major advantage.

	C	SIMD	Ratio
All	202	110	1.84
ASCII-only	156	46	3.39

Figure 17. Table showing speed in ms of C and C/SIMD implementations of fully explored Bek UTF8Encode. The first line shows running on 100-character inputs from a variety of Unicode character sets. The second line shows running time for inputs restricted to ASCII values. Unlike CSSEncode, the UTF8Encode function does not require extensive checks on each character, which removes an obstacle to speedups from SIMD.

Input		UTF8		CSSEncode	
size	PHP	WebTK	Bek	OWASP	Bek
10	2.01	1.66	3.50	13.64	10.71
20	3.43	2.64	6.50	19.48	18.19
30	4.90	3.65	9.70	24.85	25.79
40	6.46	4.69	12.70	30.64	33.27
50	7.96	6.59	15.80	32.61	40.06
100	17.27	10.76	31.60	59.93	72.45

Figure 18. JavaScript running times.

ploration and the number of states added. Figure 10 shows the number of states in three representative sanitizers before exploration, with Boolean exploration, and with full exploration. The speed of the exploration algorithms depends on the size of the reachable state space. For our examples, this

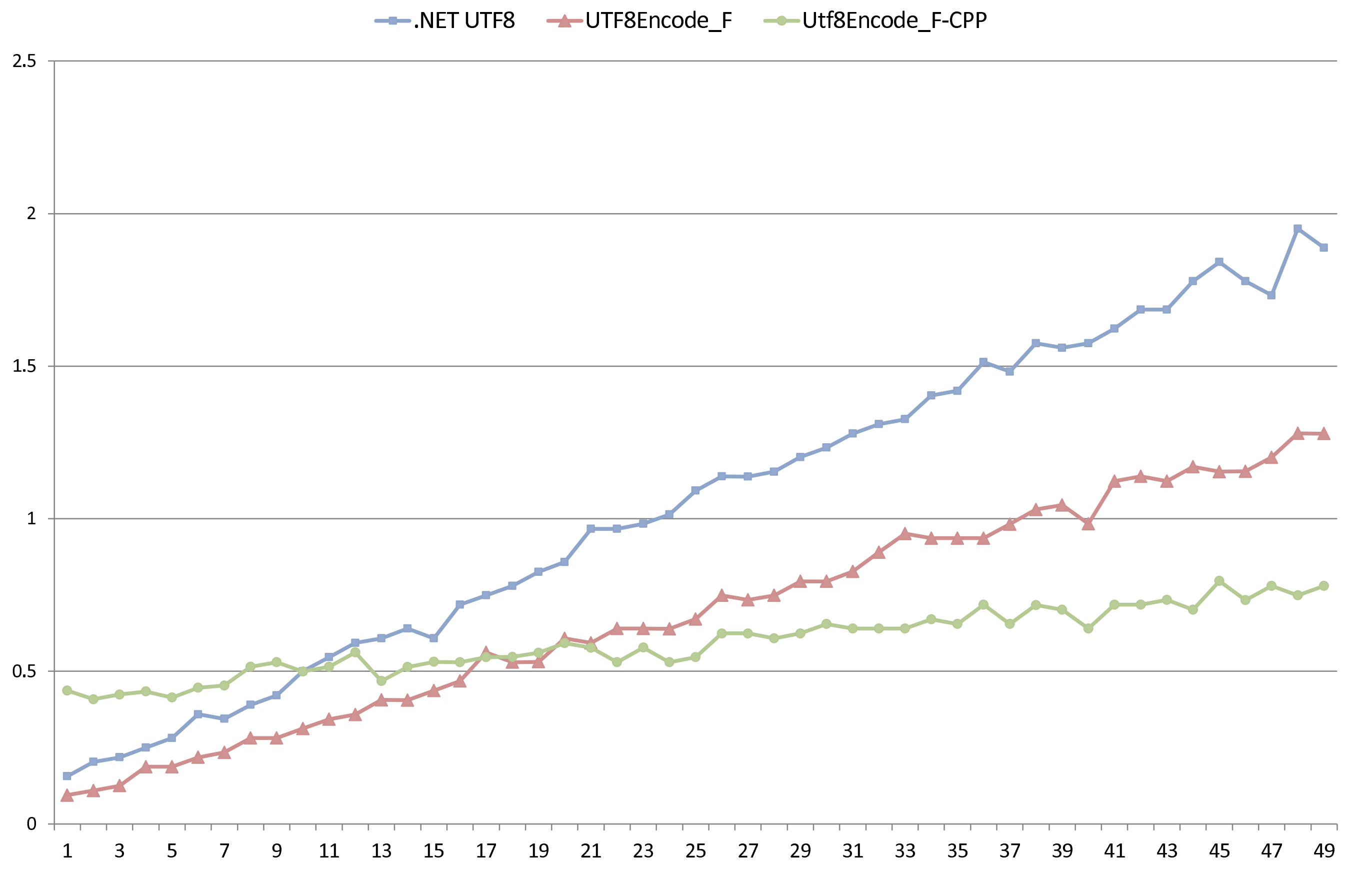

Figure 12. UTF8 encoder timing comparison over a range of input sizes.

Name	Version	Routine	Language	LOC
AntiXSS	2.4	CSSEncode	C#	206
OWASP	0.1.3	CSSEncode	JavaScript	2,953
PHP.JS	1109.2015	UTF8Encode	JavaScript	42
WebTK	NA	UTF8Encode	JavaScript	68

Figure 19. Pre-existing sanitizers used for comparison.

is small due to restricted range of the values stored in the registers. In the given experiments the time to do the full exploration is less than 2 seconds for HtmlDecode and around .1 second for the other cases.

In terms of states, $\mathrm{ST^b}$s provide an exponential reduction, in terms of the size of the alphabet, compared to classical finite state transducers. Classical transducers would need in the order of 2^{16} transitions in all cases. This enables us to perform additional analysis on $\mathrm{ST^b}$s, by reduction to STs, that would not scale on the corresponding classical transducers. In this section, we use the suffix *_B* to refer to *boolean exploration* applied to a basic $\mathrm{ST^b}$, while the suffix *_F* refers to *full exploration*.

5.1 Semantic Checking

Our approach to checking the consistency of the Bek-generated sanitizers with the original versions relies on large-scale testing. We generate a set of 1,000 strings and evaluate both the original sanitizer and the generated code on each input. The strings are chosen randomly and then checked to ensure that they are accepted by the finite state automaton in Figure 13. This ensures the inputs are legal. We found no differences in behavior between the original sanitizer and our generated code.

5.2 Speed Comparison

Bek **compiled C# code is faster than hand-written:** We first focus on server-side implementations of encoders in C# and C. Figure 11 shows a speed comparison of the different ways to compute CSS encoding. To obtain the data in the figure we start with a set of 1,000 strings and then compute successively longer substrings, ranging from 1 to 50 characters in length. To address timer resolution issues, we execute each routine 1,000 times and report the arithmetic mean.

First, we consider CSSEncode implementations. The AntiXSS-provided version of a CSSEncode is considerably slower than the Bek generated implementation. At length 50, the Bek-generated C# version is about 2.5 times faster and the C version is over 5 times faster compared to the AntiXSS routine. Different variants of Bek-generated routines in C# take about the same amount of time and are difficult to distinguish in terms of performance. Our exploration algorithm, despite creating more states, does not appreciably slow down the generated code. We will see in the next subsection the effects of exploration on vectorization.

Next, we consider UTF8Encode. Figure 12 shows a similar comparison of different UTF8 encoder implementations. The base line is the UTF8 encoder found in the standard libraries of the .NET framework. We can see that the .NET version (`.NET UTF8`) is generally the slowest of the three, consistently underperforming the BEK-generated `UTF8Encode_F`. For longer inputs, the C version of the BEK-generated code is as much as twice as fast as the C# version of the same code. These increases in execution time are consistent with the overall speed of a managed runtime such as .NET compared to a C version. We hypothesize that in our case, for small inputs, length 20 or less, the overhead of explicit memory allocation calls to `malloc` and `free` dominates the execution time. At length 50, the built-in .NET version is about 50% slower than the BEK-generated version and is about 2.5 times slower than the C version.

BEK **JavaScript Comparable to Existing Sanitizers:** We then look at client-side Javascript implementations of UTF8Encode and CSSEncode. We evaluated speed on Google Chrome version 15.0.874.106, build 107270, using the JavaScript V8 engine version 3.5.10.22. We compared the BEK generated CSSEncode to the OWASP ESAPI JavaScript library. For UTF8Encode, we compared against two independent implementations, one from the PHP.JS project, the other from the WebToolkit demonstration site. In each case, we run for 10 iterations each on a library of $1,000$ test strings. We then run with varying sizes of strings. The results are shown in Figure 18.

For CSSEncode, the BEK generated implementation is faster than the OWASP implementation until 30 byte strings are reached. At 100 byte strings, the BEK implementation is only 80% as fast as the OWASP implementation. For UTF8Encode, the situation is more interesting. Our BEK program was created directly from the UTF8 specification. While our JavaScript performance falls behind both PHPJS and WebToolKit by a factor of 2, we find during our testing that the sanitizers are *not* equivalent. Specifically our implementation disagrees with PHPJS and WebToolkit's UTF8Encode's implementations on 19 out of $1,000$ inputs. We believe part of our speed difference may arise from edge case checks performed by our BEK program that are not in the other implementations. Overall, our code is close in performance and comes with a guarantee the other implementations do not have: that it has the same semantics as the server side sanitization.

5.3 Exploration and SIMD

Next, we evaluate the effectiveness of our ST^bexploration algorithm from Section 3. We focus on two key questions: first, is code produced by the algorithm easier to vectorize than code that has been either hand-written or has been generated from an unexplored ST^b? Second, does vectorization improve speed, and if so under what conditions?

Exploration Unlocks Vectorization: For the first question, we focus on the case study of CSSEncode. We compared three implementations of CSSEncode. First, we looked at a hand-written implementation that is part of the Microsoft AntiXSS Library. Second, we looked at an C# implementation auto-generated from a BEK program for CSSEncode, without applying any exploration. This BEK program makes use of register variables. Finally, we applied the exploration algorithm from Section 3 to obtain an implementation in C# that removes dependence on registers. For vectorization, we focused on the Intel SSE 4.2 instruction set, which is a set of SIMD instructions found in recent Intel CPUs.

We found that both the hand-written and the unexplored implementations of CSSEncode contain obstacles to applying SSE instructions. Figure 6 is an excerpt from the hand-written CSSEncode implementation, which shows the array lookup technique used to map potentially unsafe characters to their encodings. This table lookup, while fast, is difficult to translate to SSE instructions. Our BEK implementation shows that underneath, what is "really" going on in CSSEncode is a sequence of bit-shifts for each character in the string, conditional on whether the character falls into certain ranges. Figure 7 shows an excerpt with this parallel assignment structure. Unfortunately, this implementation also contains a nested if statement that depends on the register r0. This register may change during processing in an unpredictable way, which makes it difficult to translate into SSE instructions.

In contrast, the post-exploration version shown in Figure 8 has removed the dependence on the register r0 from the nested if statements. While the explored version has more nested if statements, their conditions depend only on the individual character to be processed. Therefore these conditions can be easily encoded using SSE instructions. While the explored version does include an explicit state variable as part of a top level switch statement (not shown), we can optimize each state as a large piece of straight line code.

Vectorization Yields Speedups: We then manually converted the post-exploration C# code to C. Then we rewrote the C code to use SSE instructions. The results are shown in Figure 18. We looked at four different conditions. In the first, we generated test strings that were a mix of ASCII and non-ASCII characters. In the second, we used test strings limited to ASCII values. Each test string had 100 characters. We then ran $1,000$ runs on each test string, over $1,000$ test strings and report the average time in milliseconds. The gains in this case are 16% and 7% respectively.

Upon closer examination, we found that most of the time in the SIMD implementation is spent determining whether each character falls into one of several pre-specified character classes. Therefore we also evaluated speed of both C and SIMD versions under a "promise" that the input would contain only certain characters. We find that if characters are restricted to ASCII alone, then employing SSE yields a speedup of 12.53 times. If the characters are restricted to a specific high character range where CSSEncode still performs checks, we see a 2.5 times speedup. While not realistic in general, this experiment shows that there is significant vectorization inside CSSEncode. This could be unlocked through systems that can make guarantees about preconditions for on the input strings seen by the CSSEncode implementation. Alternatively, future work could focus on fast ways to filter strings so they fit only into a single character bucket.

Our results for UTF8Encode, in Figure 17 also show substantial speed gains from use of SIMD. In the case where inputs are only ASCII characters, we see a 3.3 times speedup. For a mix of ASCII and other Unicode characters, we see an 80% improvement over the C implementation. In part, this is because UTF8Encode performs fewer checks on character ranges than CSSEncode; most of the UTF8Encode function can be easily expressed as vizable bit-shifts and assignments.

6. Related Work

Symbolic finite transducers (SFTs) and BEK were originally introduced in [8] with a focus on security analysis of sanitizers. The key properties that are studied in [8] from a practical point of view are idempotence, commutativity and equivalence checking of sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for deciding equivalence of SFTs, modulo a decidable background theory is studied in [17], including a more general *1-equality* algorithm that factors out the decision problem for single-valuedness, and allows non-determinism without violating single-valuedness. The formalism of SFTs is also extended in [17] to Symbolic Transducers (STs) that allow the use of registers. The focus of the current paper and the motivation is *code generation*. The two extensions of STs that we introduce in the current paper are *conditional branching* and *exception handling*. From the point of view of analysis, such as equivalence checking, the branching rules do not offer more expressiveness, because the branching rules can be flattened, however the branching rules maintain the evaluation order of conditions, and, more importantly, maintain the semantics of *exception handling* that is essential for correct code generation. Exploration algorithms for STs are not studied or analyzed in [8, 17].

In recent years there has been considerable interest in automata over infinite languages [16], starting with the work on *finite memory automata* [10], also called *register automata*. Finite words over an infinite alphabet are often called *data words* in the literature. Other automata models over data words are *pebble automata* [13] and *data automata* [4]. Several characterizations of logics with respect to different models of data word automata are studied in [3]. This line of work focuses on fundamental questions about definability, decidability, complexity, and expressiveness on classes of automata on one hand and fragments of logic on the other hand. A different line of work on automata with infinite alphabets introduces *lattice automata* [6] that are finite state automata whose transitions are labeled by elements of an atomic lattice with motivation coming from verification of symbolic communicating machines. To the best of our knowledge, we do not know of prior work that has investigated the use of extensions of transducers for code generation.

Streaming transducers [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. Streaming transducers are largely orthogonal to SFTs or the extension of STs with branching rules, as presented in the current paper. For example, streaming transducers allow reversing the input, which is not possible with $\mathrm{ST^b}$s, while arithmetic is not allowed in streaming transducers but plays a central role in our applications of $\mathrm{ST^b}$s to string encoders. The work in [14] introduces a different symbolic extension to finite state transducers called *predicate-augmented finite state transducers*. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode transformations such as `UTF8Encode`.

We use the SMT solver Z3 [5] for incrementally solving label constraints that arise during the exploration algorithm. Similar applications of SMT techniques have been introduced in the context of symbolic execution of programs by using path conditions to represent under and over approximations of reachable states [7]. The distinguishing feature of our exploration algorithm is that it computes a precise transformation that is symbolic with respect to input labels, while allowing different levels of concretization with respect to the state variables. The resulting $\mathrm{ST^b}$ is not an under or over approximation, but functionally *equivalent* to the original $\mathrm{ST^b}$. This is important for correct code generation, as opposed to other applications such as test case generation, where under approximations are used, or verification of safety properties, where over approximations are used.

Finite state transducers have been used for dynamic and static analysis to validate sanitization functions in web applications in [2], by an over-approximation of the strings accepted by the sanitizer using static analysis of existing PHP code. Other security analysis of PHP code, e.g., SQL injection attacks, use string analyzers to obtain over-approximations (in form of context free grammars) of the HTML output by a server [12, 18]. Yu et.al. show how multiple automata can be composed to model looping code [20]. Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. HAMPI [11] and Kaluza [15] extend the STP solver to handle equations over strings and equations with multiple variables. The work in [9] shows how to solve subset constraints on regular languages. We are not aware of previous work investigating the use of finite transducers for efficient code generation. One obvious explanation for this is that classical finite transducers are not directly suited for this purpose; as we have demonstrated, finite state $\mathrm{ST^b}$s can be exponentially more succinct than classical finite transducers with respect to alphabet size.

7. Conclusions

In this paper, we have advocated the use of a domain-specific language called Bek to produce consistent and fast sanitizers across a range of languages, some server- and others client-based. At the core of Bek is a novel representation called symbolic branching transducers. SBTs allow for a faithful representation of complex string sanitizers, string manipulation routines that are used to protect applications from untrusted inputs. We have demonstrated how Bek can be used as an optimizing compiler, resulting in significant runtime improvements: our generated C# code outperforms the previous hand-tuned code by a factor of up to 2.5. For C code with SIMD, we see speedups of 2.5 times compared to native C code for the same sanitizer.

References

[1] R. Alur and P. Cerný. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL'11*, pages 599–610. ACM, 2011.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Oakland Security and Privacy*, 2008.

[3] M. Benedikt, C. Ley, and G. Puppis. Automata vs. logics on data words. In *CSL*, volume 6247 of *LNCS*, pages 110–124. Springer, 2010.

[4] M. Bojańczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, pages 7–16. IEEE, 06.

[5] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*, LNCS, 2008.

[6] T. L. Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS 2007*, volume 4634 of *LNCS*, pages 52–68, 2007.

[7] P. Godefroid. Compositional dynamic test generation. In *POPL'07*, pages 47–54, 2007.

[8] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with Bek.

In *Proceedings of the USENIX Security Symposium*, August 2011.

[9] 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, New York, NY, USA, 2009. ACM.

[10] M. Kaminski and N. Francez. Finite-memory automata. In *31st Annual Symposium on Foundations of Computer Science (FOCS 1990)*, volume 2, pages 683–688. IEEE, 1990.

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In *ISSTA*, 2009.

[12] 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.

[13] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. CL*, 5:403–435, 2004.

[14] G. V. Noord and D. Gerdemann. Finite state transducers with predicates and identities. *Grammars*, 4:263–286, 2001.

[15] 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.

[16] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In Z. Ésik, editor, *CSL*, volume 4207 of *LNCS*, pages 41–57, 2006.

[17] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finate state transducers: Algorithms and applications. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'12)*, 2012.

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

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

[20] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *Proceedings of the 15th international conference on Implementation and application of automata*, CIAA'10, pages 290–299, 2011.