Symbolic Finite State Transducers: Algorithms and Applications

Margus Veanes
Microsoft Research
margus@microsoft.com

Pieter Hooimeijer *
University of Virginia
pieter@cs.virginia.edu

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

David Molnar
Microsoft Research
dmolnar@microsoft.com

Nikolaj Bjørner
Microsoft Research
nbjorner@microsoft.com

Abstract

Finite automata and finite transducers are used in a wide range of applications in software engineering, from regular expressions to specification languages. We extend these classic objects with symbolic alphabets represented as parametric theories. Admitting potentially infinite alphabets makes this representation strictly more general and succinct than classical finite transducers and automata over strings. Despite this, the main operations, including composition, checking that a transducer is single-valued, and equivalence checking for single-valued symbolic finite transducers are effective given a decision procedure for the background theory. We provide novel algorithms for these operations and extend composition to symbolic transducers augmented with registers. Our base algorithms are unusual in that they are nonconstructive, therefore, we also supply a separate model generation algorithm that can quickly find counterexamples in the case two symbolic finite transducers are not equivalent. The algorithms give rise to a complete decidable algebra of symbolic transducers. Unlike previous work, we do not need any syntactic restriction of the formulas on the transitions, only a decision procedure. In practice we leverage recent advances in satisfiability modulo theory (SMT) solvers. We demonstrate our techniques on four case studies, covering a wide range of applications. Our techniques can synthesize string pre-images in excess of 8,000 bytes in roughly a minute, and we find that our new encodings significantly outperform previous techniques in succinctness and speed of analysis.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic

General Terms Algorithms, Theory, Verification

Keywords Automata, Composition, Equivalence, SMT

1. Introduction

Finite automata are used in a wide range of applications in software engineering, from regular expressions to specification languages. Nearly every programmer has used a regular expression at one point or another to parse logs or manipulate text. Finite transducers are an extension of finite automata to model functions on lists of elements, which in turn have uses in fields as diverse as computational linguistics and model-based testing. While this formalism is of immense practical use, it suffers from certain drawbacks: in the presence of large alphabets, they can "blow up" in the number of transitions, as each transition can encode only one choice of element from the alphabet. Furthermore, the most common forms cannot handle infinite alphabets.

Symbolic finite transducers (SFTs) are an extension of traditional transducers that attempt to solve these problems by allowing transitions to be labeled with arbitrary formulas in a specified theory. While the concept is straightforward, traditional algorithms for deciding composition, equivalence, and other properties of finite transducers *do not* immediately generalize to the symbolic case. In particular, previous work on symbolic finite transducers have needed to impose restrictions on character theories to achieve decidable analysis [1, 38]. Our work breaks this barrier and allows for arbitrary formulas from *any decidable background theory.* In practice, we leverage the recent progress in satisfiability modulo theory (SMT) solvers to provide this decision procedure. We find that our algorithms are fast when used with Z3, a state of the art SMT solver.

The restriction we do make on SFTs is a semantic one: that the SFT is *single-valued*. This restriction is needed because equivalence is undecidable even for standard finite transducers. The single-valuedness property is decidable for symbolic finite transducers. This gives us a way to check transducers arising from practical applications before applying our algorithms.

While it was previously known that equivalence was decidable for single-valued finite transducers, again it does not immediately follow that equivalence should be decidable for single-valued *symbolic* finite transducers because typically

* Work done while the author visited Microsoft Research.

POPL'12, January 25–27, 2012, Philadelphia, PA, USA.

	Effective closure under composition	Equivalence	Alphabet
Finite State Transducers	closed	undecidable in general [23], decidable for single-valued case [43] and finite-valued case [12, 50]	finite set of elements, comparable with equality
Streaming Transducers [1]	closed (for finite alphabets)	decidable	total orders (infinite)
Symbolic Finite Transducers	closed (Proposition 1)	decidable for single-valued case (Theorem 2)	**any decidable theory**
Symbolic Transducers	closed (extension of Proposition 1)	undecidable (already for the single-valued case, through direct encoding of 2-counter machines)	**any decidable theory**

Figure 1. Summary of decidability results. The bottom two rows summarize our contributions.

Case study	Section	Feature
HTMLDecode	4.1	ST representation compactness, integer-linear arithmetic
Malware fingerprinting	4.2	SFT composition for program analysis
Image blurring	4.3	non-string module theory
Location privacy	4.4	stream manipulating programs

Figure 2. Summary of case studies.

even very restricted extensions of finite automata and finite transducers lead to undecidability of the core decision problems. In fact, our proof requires a delicate separation between the "automata theoretic" parts of our algorithms and the use of the decision procedure. Unusually, our algorithm for deciding equivalence is *nonconstructive*: while we can determine that two symbolic finite automata are not equivalent, our proof does not provide a way to find a counterexample. Fortunately, we provide a separate *model generation* semi-decision procedure that can find counterexamples once it is known that two automata are not equivalent.

Figure 1 summarizes known results about finite state transducers (over sequences) and extensions thereof, focusing on the key properties studied in this paper, namely functional *compositionality*, decidability of *equivalence*, and the role of the *alphabet*, in order to place our contributions in a clear context. In Section 3.2 we compare our main techniques to closely related techniques used for *streaming transducers* [1]. Section 5 describes further related work, including work on extending automata to trees.

1.1 Applications

Section 4 presents four comprehensive case studies in different areas. Our first case study extends previous work in using symbolic finite transducers to model web "sanitization functions" [4]. Our techniques allow the addition of *register variables*, which enable encoding a sanitizer that could not be handled efficiently by previous symbolic approaches. Our second case study shows an application to analysis of Javascript malware found on the Web. Our third and fourth case studies showcase additional theories beyond strings. Figure 2 summarizes how each case study reflects a novel feature of our work. In addition to these case studies, finite transducers have been employed in other areas such as analysis of web sanitization frameworks, host-based intrusion detection, and natural language processing. Our work immediately applies to these application domains.

1.2 Contributions

Our contributions are the following:

- We present novel algorithms for *composition* and *equivalence checking* of symbolic finite transducers. Our algorithms, unlike previous work, make no restrictions on the formulae used in the transducers: we require only a decision procedure for the background theory.
- We show that the single-valuedness property of symbolic transducers is decidable. This gives rise to a decidable complete algebra of symbolic transducers. The impact is that single-valued symbolic transducers can now be "first class" objects for constructing program analyses.
- We present four case studies that demonstrate how our new algorithms enable new applications. We demonstrate experimentally that our algorithms not only terminate, but that they run quickly in practice for problem instances of interest.

1.3 Paper Organization

The rest of this paper is structured as follows. In Section 2 we provide an introduction to symbolic finite state transducers. Section 3 describes the core transducer-based algorithms. Section 4 provides four detailed case studies of transducer use. Finally, we discuss closely related work in Section 5 and conclude in Section 6.

2. Symbolic Finite Transducers

We now formally define symbolic finite transducers, we give examples of how these objects model program behavior, and we define analyses that may be conducted on such transducers. We assume a *background structure* that has an effectively enumerable (countable) multi-typed carrier set or *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. As a convention, we abbreviate $\mathcal{U}^\sigma$ by Σ and $\mathcal{U}^\gamma$ by Γ, because these symbols are used frequently. The Boolean type is $\mathbb{B}$, with $\mathcal{U}^\mathbb{B} = \{\mathrm{t}, \mathrm{f}\}$ and the integer type is $\mathbb{Z}$. Terms and formulas are defined by induction over the background language and are assumed to be well-typed. The type τ of a term t is indicated by $t\!:\!\tau$. Terms of type $\mathbb{B}$, 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 $\sigma \to \mathbb{B}$ is called a σ-*predicate*. We write φ and ψ for σ-predicates and, for $a \in \Sigma$, we write $a \in [\![\varphi]\!]$ for $[\![\varphi]\!](a) = \mathtt{t}$. 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 $t[x/u]$ (i.e., we assume implicit β-reduction). A predicate φ is *unsatisfiable* when $[\![\varphi]\!] = \emptyset$; *satisfiable*, otherwise.

The following is a key notion in the paper. Two λ-terms $f, g:\sigma \to \gamma$ are *equivalent relative to a* σ-*predicate* φ, denoted $f \equiv_\varphi g$, when $f \not\equiv_\varphi g$ is unsatisfiable, where

$$f \not\equiv_\varphi g \quad \stackrel{\text{def}}{=} \quad \lambda x.\varphi(x) \wedge f(x) \neq g(x)$$

is called the *difference predicate of* f *and* g *relative to* φ.

Definition 1: A *label theory for* $\sigma \to \gamma$ is associated with an effectively enumerable set of λ-terms of type $\sigma \to \gamma$ and an effectively enumerable set of σ-predicates that is effectively closed under Boolean operations and relative difference. ⊠

Effective closure under Boolean operations in Definition 1 means that $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$ and $[\![\neg\varphi]\!] = \Sigma \setminus [\![\varphi]\!]$. An important and direct consequence is the following:

$$[\![\neg(f \not\equiv_\varphi g)]\!] \quad = \quad [\![\lambda x.\neg(\varphi(x) \wedge f(x) \neq g(x))]\!].$$

In particular, it is not possible to express relative equivalence of λ-terms directly, other than through unsatisfiability of a difference predicate. This is important in order to maintain our decidability results while making minimal assumptions about the label theory. In our use of label theories below, difference predicates play a central role and are always used in a positive context (i.e., not negated as above).

A label theory Ψ is *decidable* when satisfiability for $\varphi \in \Psi$, $IsSat(\varphi)$, is decidable. We assume an effective *witness* function for a σ-predicate φ such that, if $IsSat(\varphi)$ then $witness(\varphi) \in [\![\varphi]\!]$.

Example 1: As an example of a decidable label theory for $(\mathbb{Z} \times \mathbb{Z}) \to \mathbb{Z}$, consider the combined theory of pairs and quantifier-free integer linear arithmetic. Suppose π_1 and π_2 are the projection functions from pairs. Let f be $\lambda x.(2 * \pi_1(x))$, let g be $\lambda x.(\pi_1(x) + \pi_2(x))$, and let φ be $\lambda x.(\pi_1(x) = \pi_2(x))$. Then f and g are equivalent relative to φ, because $f \not\equiv_\varphi g$ is unsatisfiable, i.e.,

$$\lambda x.(\pi_1(x) = \pi_2(x) \wedge 2 * \pi_1(x) \neq \pi_1(x) + \pi_2(x))$$

is unsatisfiable. ⊠

Given a set X, we write X^* for the *Kleene closure* of X. Similarly, τ^* is the type of sequences over τ. A sequence of length $k \geq 0$ is denoted either by $[x_0, \ldots, x_{k-1}]$ or $\boldsymbol{x}$ with $\boldsymbol{x}_i$ as the i'th element of $\boldsymbol{x}$ for $0 \leq i < |\boldsymbol{x}|$.

Next, we describe an extension of finite state transducers through a symbolic representation of labels. The advantage of the extension is succinctness and modularity with respect to any given label theory. It naturally separates the finite state transition graph from the label theory.

Definition 2: A *Symbolic Finite Transducer (SFT)* over $\sigma \to \gamma$ is a tuple (Q, q^0, F, R), 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*, and R is a set of *rules* $(p, \varphi, \boldsymbol{f}, q)$, where $p, q \in Q$, φ is a σ-predicate and, $\boldsymbol{f}$, is a a sequence of λ-terms, over a given label theory for $\sigma \to \gamma$. ⊠

We use the more intuitive notation $p \xrightarrow{\varphi/\boldsymbol{f}}_A q$ for a rule $(p, \varphi, \boldsymbol{f}, q) \in R_A$ and call φ its *guard*. We omit the index A when A is clear from the context. We treat $\boldsymbol{f} : (\sigma \to \gamma)^*$ as a function $\lambda x\ .\ [\boldsymbol{f}_0(x), \ldots, \boldsymbol{f}_k(x)]$ where $k = |\boldsymbol{f}| - 1$. We lift the definition of relative equivalence of λ-terms to sequences $\boldsymbol{f}$ and $\boldsymbol{g}$ of λ-terms: $\boldsymbol{f} \equiv_\varphi \boldsymbol{g}$ iff $|\boldsymbol{f}| = |\boldsymbol{g}|$ and for all i, $0 \leq i < |\boldsymbol{f}|$, $\boldsymbol{f}_i \equiv_\varphi \boldsymbol{g}_i$. We use the notation of rules to also denote concrete transitions when the intension is clear. For $p, q \in Q_A$, $a \in \Sigma$ and $\boldsymbol{b} \in \Gamma^*$:

$$p \xrightarrow{a/\boldsymbol{b}}_A q \ \text{ if for some } p \xrightarrow{\varphi/\boldsymbol{f}}_A q \in R\text{: } a \in [\![\varphi]\!],\ \boldsymbol{b} = [\![\boldsymbol{f}]\!](a)$$

i.e., the transition $p \xrightarrow{a/\boldsymbol{b}} q$ is an *instance* of a rule. Concatenation of two sequences seq_1 and seq_2 is denoted $seq_1 \cdot seq_2$.

Definition 3: For $\boldsymbol{a} \in \Sigma^*$ and $\boldsymbol{b} \in \Gamma^*$, $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q$ denotes the *reachability relation*: there exists a path of transitions from p to q in A with input sequence $\boldsymbol{a}$ and output sequence $\boldsymbol{b}$: let $n = |\boldsymbol{a}| - 1$, when there exist subsequences $\boldsymbol{b}^i$, such that $\boldsymbol{b} = \boldsymbol{b}^0 \cdot \boldsymbol{b}^1 \cdots \boldsymbol{b}^n$ and

$$p = p_0 \xrightarrow{\boldsymbol{a}_0/\boldsymbol{b}^0} p_1 \xrightarrow{\boldsymbol{a}_1/\boldsymbol{b}^1} p_2 \cdots p_n \xrightarrow{\boldsymbol{a}_n/\boldsymbol{b}^n} p_{n+1} = q$$

We let $p \xrightarrow{\epsilon/\epsilon}\!\!\!\!\rightarrow_A p$ for all $p \in Q_A$. ⊠

Definition 4: The *transduction* of A, denoted $\mathscr{T}_A$, is the following function from Σ^* to 2^{Γ^*}:

$$\mathscr{T}_A(\boldsymbol{a}) \stackrel{\text{def}}{=} \{\boldsymbol{b} \in \Gamma^* \mid \exists q \in F_A\ (q^0_A \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow q)\}.$$

Equivalently, $\mathscr{T}_A$ is viewed as the binary relation, or subset of $\Sigma^* \times \Gamma^*$, such that $\mathscr{T}_A(\boldsymbol{a}, \boldsymbol{b})$ iff $\boldsymbol{b} \in \mathscr{T}_A(\boldsymbol{a})$. The *domain* of A is $\mathscr{D}(A) \stackrel{\text{def}}{=} \{\boldsymbol{a} \in \Sigma^* \mid \mathscr{T}_A(\boldsymbol{a}) \neq \emptyset\}$. ⊠

The following subclass of SFTs captures transductions that behave as partial functions from Σ^* to Γ^*.

Definition 5: A is *single-valued* when $|\mathscr{T}_A(\boldsymbol{a})| \leq 1$ for all $\boldsymbol{a} \in \Sigma^*$. ⊠

A sufficient condition for single-valuedness is determinism.

Definition 6: A is *deterministic* when, for all $p \xrightarrow{\varphi/\boldsymbol{f}}_A q$ and $p \xrightarrow{\psi/\boldsymbol{g}}_A r$, if $IsSat(\varphi \wedge \psi)$ then $q = r$ and $\boldsymbol{f} \equiv_{\varphi\wedge\psi} \boldsymbol{g}$. ⊠

In terms of concrete transitions, determinism of A means that if $p \xrightarrow{a/\boldsymbol{b}}_A q$ and $p \xrightarrow{a/\boldsymbol{b}'}_A q'$ then $(\boldsymbol{b}, q) = (\boldsymbol{b}', q')$. It follows by induction over $|\boldsymbol{a}|$ for $\boldsymbol{a} \in \Sigma^*$ that, if $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q$ then $(\boldsymbol{b}, q)$ is unique for the given p and $\boldsymbol{a}$, in particular when $p = q^0_A$ and $q \in F_A$, and thus A is single-valued. Determinism is, however, not a necessary condition for single-valuedness, as is illustrated below. In the following examples, all SFTs are single-valued. The first example illustrates a few simple functional list transformations, expressed as deterministic SFTs that illustrate how global properties of SFTs depend on the theory of labels.

Example 2: Let the input type and the output type be $\mathbb{Z}$. All SFTs have a single state here. Predicates and terms are terms in integer linear arithmetic. *Negate* multiplies all elements by -1. *Increment* adds 1 to each element. *DeleteZeros* deletes all zeros from the input.

$R_{Negate} = \{p \xrightarrow{\lambda x.\mathtt{t}/[\lambda x.(-x)]} p\}$

$R_{Increment} = \{q \xrightarrow{\lambda x.\mathtt{t}/[\lambda x.(1+x)]} q\}$

$R_{DeleteZeros} = \{r \xrightarrow{\lambda x.(x=0)/[]} r, \quad r \xrightarrow{\lambda x.(x\neq 0)/[\lambda x.x]} r\}$

Properties such as commutativity and idempotence of SFTs depend on the theory of labels. For example, whether *Negate* and *DeleteZeros* commute or whether *DeleteZeros* is idempotent depend on properties of integer addition and multiplication. None of the examples can be expressed as traditional finite state transducers over a finite alphabet. Our results about composition and equivalence checking, discussed below, allow us to effectively establish such properties modulo decidability of a given label theory. ⊠

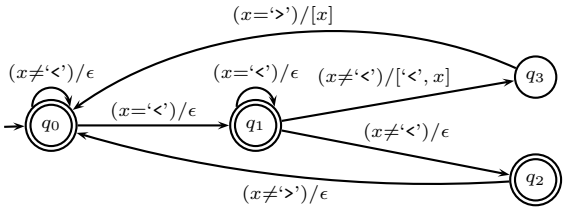
The following example illustrates a common string transformation where the use of nondeterministic SFTs is essential.

Example 3: Suppose that the following C# code is intended to implement a function `GetTags` that extracts from a given input stream of characters all substreams of the form $[\text{`<'}, x, \text{`>'}]$, where $x \neq$ `'<'`. For example

`GetTags("<<s><<>><f><t") = "<s><>><f>"`

```
1:  int q = 0; char c = (char)0;
2:  foreach (char x in input) {
3:    if (q == 0) { if (x == '<') q = 1; else q = 0;}
4:    else if (q == 1) { if (x == '<') q = 1; else q = 2;}
5:    else if (q == 2) { if (x == '>') { yield return '<';
                                         yield return c;
7:                                       yield return '>';}
8:                       q = 0; }
9:    c = x; }
```

Note that the variable `q` keeps track of the relative position in the pattern $[\text{`<'}, x, \text{`>'}]$ and `c` records the previous character. The corresponding single-valued SFT is:[1]



GetTags is nondeterministic because there are two rules from q_1 to q_2 and q_3, respectively, yielding different outputs for the same input. If the characters are represented as integers, then a deterministic version of *GetTags* does not exists, and if the characters are represented as 16-bit bitvectors (that corresponds precisely to the standard UTF-16 encoding of characters in C#) then the size of the equivalent deterministic SFT is 2^{16} times larger. (Example 7 below explains how *GetTags* is constructed from the C# code.) ⊠

3. SFT Algorithms

In this section we study algorithms for *composition* and *equivalence* of SFTs. First, we show that SFTs are effectively closed under composition. Next, we provide an efficient algorithm for *single-valued equality* of SFTs modulo a decidable theory of labels. Finally we introduce an algebra of SFTs that enables a variety of practically useful decision problems, such as deciding single-valuedness, and deciding commutativity and idempotence of single-valued SFTs.

[1] We omit λ's in figures for a more compact view.

3.1 Composition of SFTs

Given two transductions $\mathscr{T}_1$ and $\mathscr{T}_2$, $\mathscr{T}_1 \circ \mathscr{T}_2$ denotes the following function:

$$\mathscr{T}_1 \circ \mathscr{T}_2 \stackrel{\text{def}}{=} \lambda \boldsymbol{b}.\ \bigcup_{\boldsymbol{a} \in \mathscr{T}_1(\boldsymbol{b})} \mathscr{T}_2(\boldsymbol{a}).$$

This definition follows the convention in [21]. Notice that $\circ$ applies first $\mathscr{T}_1$, then $\mathscr{T}_2$, contrary to how $\circ$ is used for standard function composition. Note also that single-valuedness is trivially preserved by composition.

We say that label theories for $\sigma \to \tau$ and $\tau \to \gamma$ are *composable* if there is a label theory Ψ for $\sigma \to \gamma$ such that

- if $f : \sigma \to \tau$ and $g : \tau \to \gamma$ are a λ-terms then $\lambda x.g(f(x))$ is a valid λ-term in Ψ.
- if φ is a τ-predicate and $f : \sigma \to \tau$ is a λ-term, then $\lambda x.\varphi(f(x))$ is a valid σ-predicate in Ψ.

Proposition 1: Let A and B be SFTs over composable label theories. Then there exists an SFT $A \circ B$ that is obtained effectively from A and B such that $\mathscr{T}_{A \circ B} = \mathscr{T}_A \circ \mathscr{T}_B$.

The algorithm for $A \circ B$ can be implemented with a DFS procedure that, by assuming decidability of the label theory, eliminates incrementally all composed rules that have unsatisfiable guards and finally eliminates all *deadends* (*deadlock states*: states from which no final state is reachable).

3.2 Equivalence of SFTs

We introduce an algorithm for deciding equivalence of *single-valued* SFTs. While general equivalence of finite state transducers is undecidable [23] the undecidability is caused by allowing unboundedly many different outputs for a given input. Single-valued transducers, furthermore, correspond closely to functional transformations over lists computed by concrete programs. As illustrated above, this does not (in general) rule out nondeterministic SFTs.

SFTs A and B are *equivalent*, $A \equiv B$, when $\mathscr{T}_A = \mathscr{T}_B$. Deciding $A \equiv B$ reduces to two independent tasks:

Domain equivalence : $\mathscr{D}(A) = \mathscr{D}(B)$.

Partial equivalence : $(\forall \boldsymbol{a} \in \mathscr{D}(A) \cap \mathscr{D}(B))\ \mathscr{T}_A(\boldsymbol{a}) = \mathscr{T}_B(\boldsymbol{a})$

A *symbolic finite automaton* or *SFA* is an SFT all of whose outputs are empty. Let $\mathbf{d}(A)$ denote the SFA obtained from the SFT A by replacing all outputs by ϵ. Then $\mathscr{D}(A) = \mathscr{D}(B)$ iff $\mathbf{d}(A) \equiv \mathbf{d}(B)$. Equivalence of SFAs is decidable over decidable label theories [48]. The decidability of SFA equivalence depends on the assumption that the label theory is closed under *complementation*, this assumption is not needed for partial equivalence.

For developing a decision procedure for partial equivalence of single-valued SFTs we use the following weak form of partial equivalence.

Single-valued equality (or 1-equality $A \stackrel{1}{=} B$) :
$\forall \boldsymbol{a}\, \boldsymbol{b}\, \boldsymbol{c}\, ((\boldsymbol{b} \in \mathscr{T}_A(\boldsymbol{a}) \wedge \boldsymbol{c} \in \mathscr{T}_B(\boldsymbol{a})) \Rightarrow \boldsymbol{b} = \boldsymbol{c})$

Proposition 2: A is single-valued iff $A \stackrel{1}{=} A$. If A and B are single-valued then $A \stackrel{1}{=} B$ iff A and B are partially equivalent.

Single-valued equality of two SFTs A and B may fail for two reasons. There is an input $\boldsymbol{a} \in \mathscr{D}(A) \cap \mathscr{D}(B)$ and outputs $\boldsymbol{b} \in \mathscr{T}_A(\boldsymbol{b})$ and $\boldsymbol{c} \in \mathscr{T}_B(\boldsymbol{a})$ such that:

1. A has a *length-conflict* with B: $|\boldsymbol{b}| \neq |\boldsymbol{c}|$.

2. A has a *position-conflict* with B: $|\boldsymbol{b}| = |\boldsymbol{c}|$ and, for some position i, $0 \leq i < |\boldsymbol{b}|$, $\boldsymbol{b}_i \neq \boldsymbol{c}_i$.

We introduce the following basic product construction of SFTs as a generalization of the product of SFAs [48]. The product construction is most effectively realized by using a DFS procedure. Note that the product of SFTs is a "2-output-SFT" (SFTs are not closed under product).

Definition 7: The *product* of SFTs A and B, denoted $A \times B$, is defined as the least fixpoint of pair states $Q \subseteq Q_A \times Q_B$ and rules under the following conditions:

- $(q_A^0, q_B^0) \in Q$,
- if $(p_1, p_2) \in Q$, $p_1 \xrightarrow{\varphi/\boldsymbol{f}}_A q_1$, and $p_2 \xrightarrow{\psi/\boldsymbol{g}}_B q_2$, then
  - $(q_1, q_2) \in Q$ and
  - $(p_1, p_2) \xrightarrow{\varphi\wedge\psi/(\boldsymbol{f},\boldsymbol{g})}_{A\times B} (q_1, q_2)$,

  provided that $\mathit{IsSat}(\varphi \wedge \psi)$.

All *deadends*, noninitial states from which $F_A \times F_B$ is not reachable, are eliminated from $A \times B$. ⊠

Example 4: Consider the SFT *GetTags* in Example 3. Then the product $\mathit{GetTags} \times \mathit{GetTags}$ has rules $(p, p) \xrightarrow{\varphi/(\boldsymbol{f},\boldsymbol{f})} (q, q)$ for all rules $p \xrightarrow{\varphi/\boldsymbol{f}}_{\mathit{GetTags}} q$ and no other rules due to elimination of deadends, e.g., (q_3, q_2) is a deadend because the guard of (the only possible rule)

$$(q_3, q_2) \xrightarrow{\lambda x.(x=\text{`>'}\wedge x\neq\text{`>'})/([\lambda x.x],\epsilon)} (q_0, q_0)$$

from (q_3, q_2) is unsatisfiable. ⊠

Let $\mathscr{D}(A \times B)$ denote the set of inputs that are accepted by the product. It follows from the product construction that:

$$\mathscr{D}(A \times B) = \mathscr{D}(A) \cap \mathscr{D}(B). \tag{1}$$

The reachability relation is lifted to $A \times B$ and the following holds for $p = (p_1, p_2), q = (q_1, q_2) \in Q_{A\times B}$:

$$p \xrightarrow{\boldsymbol{a}/(\boldsymbol{b},\boldsymbol{c})}\!\!\!\!\rightarrow_{A\times B} q \quad \Leftrightarrow \quad p_1 \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\rightarrow_A q_1 \wedge p_2 \xrightarrow{\boldsymbol{a}/\boldsymbol{c}}\!\!\!\!\rightarrow_B q_2. \tag{2}$$

We omit the index $A \times B$ when it is clear from the context. For $q \in Q_{A\times B}$, and $k \geq 0$, define the *offset relation*,

$$q \vartriangle k \stackrel{\text{def}}{=} \exists\, \boldsymbol{a}\, \boldsymbol{b}\, \boldsymbol{c}\, (q_{A\times B}^0 \xrightarrow{\boldsymbol{a}/(\boldsymbol{b},\boldsymbol{c})}\!\!\!\!\rightarrow q \wedge k = |\boldsymbol{b}| - |\boldsymbol{c}|) \tag{3}$$

The intuition behind $q \vartriangle k$ is that, for some common input $\boldsymbol{a}$, there exists an output $\boldsymbol{b}$ from A that is either ahead of an output $\boldsymbol{c}$ from B with k-positions at product state q when $k > 0$, or behind when $k < 0$. The following lemma is used to detect length-conflicts.

Lemma 1: If there exists $q \in Q_{A\times B}$ and $m \neq n$ such that $q \vartriangle m$ and $q \vartriangle n$ then A has a length-conflict with B. Moreover, A has a length-conflict with B iff there exists $q \in F_{A\times B}$ and $m \neq 0$ such that $q \vartriangle m$.

Lemma 1 suggests an efficient DFS algorithm to detect if a length-conflict exists and otherwise computes the fixed offset $\mathit{offs}(p)$ such that $p \vartriangle \mathit{offs}(p)$. In order to decide if a position-conflict exists between A and B, we first assume that A and B have no length-conflicts and assume that $\mathit{offs}(p)$ is defined and $\mathit{offs}(p) = 0$ for $p \in F_{A\times B}$. We say that $(\alpha, \beta) \in \Gamma^* \times \Gamma^*$ is a *promise* of a product state $p \in Q_{A\times B}$ when the following holds:

$$(\alpha = \epsilon \vee \beta = \epsilon) \wedge \exists\, \boldsymbol{a}\, \boldsymbol{b}\, \boldsymbol{c}\, (|\boldsymbol{b}| = |\boldsymbol{c}| \wedge q_{A\times B}^0 \xrightarrow{\boldsymbol{a}/(\boldsymbol{b}\cdot\alpha,\boldsymbol{c}\cdot\beta)}\!\!\!\!\rightarrow p)$$

It follows that

$$(|\alpha|, |\beta|) = \begin{cases} (\mathit{offs}(p), 0), & \text{if } \mathit{offs}(p) \geq 0; \\ (0, -\mathit{offs}(p)), & \text{otherwise.} \end{cases}$$

Lemma 2: If $A \stackrel{1}{=} B$ then each product state in $Q_{A\times B}$ has a fixed promise.

Proof. Assume $A \stackrel{1}{=} B$. Suppose, by way of contradiction, that there exists $p \in Q_{A\times B}$ with two distinct promises (α, β) and (α', β'). By Lemma 1 we know that $|\alpha| = |\alpha'|$ and $|\beta| = |\beta'|$ and either $\alpha = \epsilon$ or $\beta = \epsilon$. Suppose $\beta = \epsilon$ (the case $\alpha = \epsilon$ is symmetrical). Thus $\alpha \neq \alpha'$ (or else the promises are identical). By definition of promises there exist $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{a}', \boldsymbol{b}', \boldsymbol{c}'$ such that, $|\boldsymbol{b}| = |\boldsymbol{c}|$, $|\boldsymbol{b}'| = |\boldsymbol{c}'|$,

$$q_{A\times B}^0 \xrightarrow{\boldsymbol{a}/(\boldsymbol{b}\cdot\alpha,\boldsymbol{c})}\!\!\!\!\rightarrow p, \quad q_{A\times B}^0 \xrightarrow{\boldsymbol{a}'/(\boldsymbol{b}'\cdot\alpha',\boldsymbol{c}')}\!\!\!\!\rightarrow p.$$

Since p is not a deadend, there exist $\boldsymbol{a}'', \boldsymbol{b}'', \boldsymbol{c}''$ and $q^f \in F_{A\times B}$ such that

$$p \xrightarrow{\boldsymbol{a}''/(\boldsymbol{b}'',\boldsymbol{c}'')}\!\!\!\!\rightarrow q^f.$$

It follows from $A \stackrel{1}{=} B$ that

$$\left.\begin{array}{l} \boldsymbol{b}\cdot\alpha\cdot\boldsymbol{b}'' \in \mathscr{T}_A(\boldsymbol{b}\cdot\boldsymbol{b}'') \\ \boldsymbol{c}\cdot\boldsymbol{c}'' \in \mathscr{T}_B(\boldsymbol{a}\cdot\boldsymbol{a}'') \end{array}\right\} \Rightarrow \boldsymbol{b}\cdot\alpha\cdot\boldsymbol{b}'' = \boldsymbol{c}\cdot\boldsymbol{c}''$$

$$\left.\begin{array}{l} \boldsymbol{b}'\cdot\alpha'\cdot\boldsymbol{b}'' \in \mathscr{T}_A(\boldsymbol{a}'\cdot\boldsymbol{a}'') \\ \boldsymbol{c}'\cdot\boldsymbol{c}'' \in \mathscr{T}_B(\boldsymbol{a}'\cdot\boldsymbol{a}'') \end{array}\right\} \Rightarrow \boldsymbol{b}'\cdot\alpha'\cdot\boldsymbol{b}'' = \boldsymbol{c}'\cdot\boldsymbol{c}''$$

and, since $|\boldsymbol{b}| = |\boldsymbol{c}|$ and $|\boldsymbol{b}'| = |\boldsymbol{c}'|$, it follows that $\alpha\cdot\boldsymbol{b}'' = \boldsymbol{c}''$ and $\alpha'\cdot\boldsymbol{b}'' = \boldsymbol{c}''$, contradicting that $\alpha \neq \alpha'$. □

Lemma 2 can be used to check for non-1-equality as follows: In a depth-first manner compute for every state in the product $Q_{A\times B}$ the current promise (α, β). If the state gets revisited with a different promise, then $A \stackrel{1}{\neq} B$.

Example 5: The product $\mathit{GetTags} \times \mathit{GetTags}$ in Example 4 has trivially no length-conflicts because offsets of all product states are 0 and thus promises of all product states are (ϵ, ϵ). ⊠

Finally, assuming $A \times B$ has no length-conflicts and each product state $p \in Q_{A\times B}$ has a fixed promise $\mathit{prom}(p) = (\alpha, \beta)$, then p is *conflict-free*, when, for all rules $p \xrightarrow{\varphi/(\boldsymbol{f},\boldsymbol{g})} q$, the maximal prefixes of $\alpha\cdot\boldsymbol{f}$ and $\beta\cdot\boldsymbol{g}$ are equivalent relative to φ. (Note, when concatenating $\alpha \in \Gamma^*$ with a sequence $\boldsymbol{f}$ of λ-terms of type $\sigma \to \gamma$ we assume an implicit conversion of α to $\lambda x.\alpha$.) Let $k = \min(|\alpha\cdot\boldsymbol{f}|, |\beta\cdot\boldsymbol{g}|) - 1$,

$$\bigwedge_{j=0}^{k} (\forall a (a \in [\![\varphi]\!] \Rightarrow [\![(\alpha\cdot\boldsymbol{f})_j]\!](a) = [\![(\beta\cdot\boldsymbol{g})_j]\!](a))). \tag{4}$$

We say that p is a *conflict-state* if p is not conflict-free. Verifying absence of conflict-states is a linear search over rules in $A \times B$ that verifies the condition (4) for each rule.

Lemma 3: If $A \times B$ has fixed promises and no length-conflicts then $A \stackrel{1}{\neq} B \Leftrightarrow Q_{A\times B}$ contains a conflict-state.

Proof. Assume that $A \times B$ is as stated.

($\Leftarrow$): existence of a conflict-state p implies, by definition, that there exists a position-conflict, since p is both reachable and not a deadend.

($\Rightarrow$): Assume $A \stackrel{1}{\neq} B$. We show that there exists a conflict-state. Since A and B have no length-conflicts there exist $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}$ such that $\boldsymbol{b} \in \mathscr{T}_A(\boldsymbol{a})$, $\boldsymbol{c} \in \mathscr{T}_B(\boldsymbol{a})$, $|\boldsymbol{b}| = |\boldsymbol{c}|$, and there exists a position i, $0 \leq i < |\boldsymbol{b}|$, such that $\boldsymbol{b}_i \neq \boldsymbol{c}_i$. Fix i to be the smallest such position. So there exists $\boldsymbol{a}^1, a, \boldsymbol{b}^1, \boldsymbol{b}^2, \boldsymbol{c}^2, \alpha, \beta, p, q$ such that $\boldsymbol{a}^1\cdot a$ is a prefix of $\boldsymbol{a}$, $\boldsymbol{b}^1\cdot\alpha\cdot\boldsymbol{b}^2$ is a prefix of $\boldsymbol{b}$, $\boldsymbol{b}^1\cdot\beta\cdot\boldsymbol{c}^2$ is a prefix of $\boldsymbol{c}$, and

$$q_{A\times B}^0 \xrightarrow{\boldsymbol{a}^1/(\boldsymbol{b}^1\cdot\alpha,\boldsymbol{b}^1\cdot\beta)}\!\!\!\!\rightarrow p \xrightarrow{a/(\boldsymbol{b}^2,\boldsymbol{c}^2)} q$$

where $prom(p) = (\alpha, \beta)$ and $(\alpha \cdot \boldsymbol{b}^2)_j \neq (\beta \cdot \boldsymbol{c}^2)_j$ with $j = i - |\boldsymbol{b}^1|$. So there exists a rule $p \xrightarrow{\varphi/(\boldsymbol{f},\boldsymbol{g})} q$ such that $a \in [\![\varphi]\!]$ but $[\![(\alpha \cdot \boldsymbol{f})_j]\!](a) \neq [\![(\beta \cdot \boldsymbol{g})_j]\!](a)$. Thus, p is a conflict-state because (4) does not hold. □

The following theorem describes precisely the assumptions under which 1-equality of SFTs is decidable.

Theorem 1 (SFT-1-equality)**:** If A and B are SFTs over a decidable label theory then $A \overset{1}{=} B$ is decidable. Moreover, if the complexity of the label theory for instances of size m is $f(m)$ then the complexity of $A \overset{1}{=} B$ is $O(n^2 \cdot f(m))$ where n is the number of rules and m the size of the rules.

Proof. By using Lemmas 1, 2 and 3. Deciding satisfiability of φ is needed in the construction of $A \times B$. Deciding $f \equiv_\varphi g$ is needed for deciding validity of the formula (4). In Lemma 2, we need to decide if f is constant relative to a satisfiable formula φ: decide if $f \equiv_\varphi [\![f]\!](witness(\varphi))$.

Lemmas 1, 2 and 3 are combined into a single DFS algorithm, shown in Figure 3, that decides 1-equality of SFTs over a decidable label theory. Line 6 corresponds

```
Decide1equality(A, B) ≝
1  C := A × B; Q := {q_C^0 ↦ (ϵ, ϵ)}; S := stack(q_C^0);
2  while S ≠ ∅
3     p := pop(S); (α, β) := Q(p);
4     foreach (p, φ, (f, g), q) ∈ R_C(p)
5        (u, v) := (α · f, β · g);
6        if q ∈ F_C ∧ |u| ≠ |v| return f;
7        if |u| ≥ |v|
8           if ⋁_{i=0}^{|v|-1} u_i ≢_φ v_i return f;
9           w := [u_{|v|}, ..., u_{|u|-1}]; c := [w](witness(φ));
10          if w ≢_φ c ∨ (q ∈ Dom(Q) ∧ Q(q) ≠ (c, ϵ)) return f;
11          if q ∉ Dom(Q) push(q, S); Q(q) := (c, ϵ);
12       if |u| < |v| ... (symmetrical to the case |u| ≥ |v|)
13 return t;
```

Figure 3. 1-equality algorithm for SFTs.

to detection of a final state with non-zero offset by using Lemma 1. Line 8 corresponds to use of Lemma 3(⇐). Line 10 corresponds to use of Lemma 2. Line 13 corresponds to use of contraposition of Lemma 3(⇒).

The number of iterations of the loop as well as in the product construction is bounded by $|R_A| \cdot |R_B|$. The algorithm uses satisfiability checks during product construction in line 1, and in the loop in lines 8 and 10. In line 8 the number of checks is linear in the length of the output sequence v: decide if there exists i, $0 \leq i < |v|$, and $\varphi(x) \wedge u_i(x) \neq v_i(x)$ is satisfiable. Similarly for line 10. The complexity follows. □

Theorem 1 shows that complexity of 1-equality of SFTs depends on the complexity of the label theory. For example, if we use linear arithmetic with one free variable as the label theory, and guards are represented in normalized form as conjunctions of linear inequalities, then the Fourier-Motzkin elimination procedure [14] implies a polynomial worst-case complexity of 1-equality.

The algorithm for partial-equivalence of classical single-valued finite transducers, has complexity $O((|T|+|Q|)^2)$ [16], where T is the set of transitions. With a symbolic encoding we can replace $|T|$ by the decision complexity for the alphabet theory. Symbolic encodings also make expressing dependencies between input-output characters succinct. To

$$
\begin{array}{lcl}
\sigma, \tau, \gamma & ::= & \text{types} \\
sfa^\sigma & ::= & \text{explicit dfn of an SFA over } \sigma \\
sft^{\sigma/\gamma} & ::= & \text{explicit dfn of an SFT over } \sigma \to \gamma \\
A^\sigma & ::= & sfa^\sigma \mid A^\sigma - A^\sigma \mid A^\sigma \times A^\sigma \mid B^{\sigma/\gamma} \circ A^\gamma \\
B^{\sigma/\gamma} & ::= & sft^{\sigma/\gamma} \mid B^{\sigma/\tau} \circ B^{\tau/\gamma} \mid B^{\sigma/\gamma} \restriction A^\sigma \\
F & ::= & A^\sigma \subseteq A^\sigma \mid B^{\sigma/\gamma} \overset{1}{=} B^{\sigma/\gamma} \mid F \wedge F \mid \neg F
\end{array}
$$

Figure 4. Algebra of SFTs; A is a valid SFA expression; B is a valid SFT expression; F is a valid formula; label theories are assumed composable.

give the flavor, in a UTF-16 to UTF-8 encoder, we use transitions of the form

$$p \xrightarrow{\lambda x.\texttt{0xD800} \leq x \leq \texttt{0xDBFF}/[\lambda x.enc_1(x), \lambda x.enc_2(x), \lambda x.enc_3(x)]} q.$$

It succinctly represents 1024 transitions required by explicit finite transducer representations. Overall, the full SFT (over 16-bit bit-vector arithmetic) for the encoder uses 5 states and 16 rules, compared to 2^{16} concrete transitions required by an equivalent classical finite transducer. We also evaluate the benefits of SFAs in [24]. SFA equivalence is required for the domain equivalence check of SFTs.

Relation to One-Counter Automata. In closely related work, Alur et.al. [1] present *streaming transducers*, an extension of classical finite state transducers that is largely orthogonal to SFTs presented here. For example, streaming transducers allow reversing the input, which is not possible with SFTs, but require the character theory to be a total order, so that equivalence remains decidable. The authors prove the decidability of equivalence of streaming transducers by reducing it to reachability of *one-counter automata*. At a high level, the automaton is constructed so that it simulates the execution of the two given transducers in parallel, synchronized on the input tape, while using the counter to represent the length offset between outputs. If the one-counter automaton can reach a final state on a zero count, then the simulated transducers must have different output on some input.

While our main algorithm (1-equality algorithm in Figure 3) is similar in spirit, we do not make explicit use of the one-counter automaton construction. To do so would impose two restrictions on our approach: (1) the one-counter automaton construction would require satisfiability checking of conjunctions of formulae in the background theory; and (2) it would require special handling to deal with the fact that we allow elements in output sequences to be functions of the input symbols. While we are confident that the first requirement (1) can be readily accommodated, we are unaware of any way to circumvent the second restriction (2) without imposing additional limitations on the types of allowable output functions, and by adding a symbolic component to the one-counter automaton itself. Instead, we focus on a more ad-hoc construction that does not impose additional restrictions. With respect to the algorithm in Figure 3, (2) is reflected in the use of *witness or model generation* modulo the input condition and the output transformation functions, (2) is essentially the implementation of Lemma 2.

3.3 Algebra of SFTs

We introduce an algebra of SFTs, in Figure 4, that allows us to express several useful decision problems involving SFTs and SFAs. Note that $B \circ A$ of an SFT B with an SFA A is again an SFA because all the outputs of $B \circ A$ are empty. We

call $B \circ A$ the *inverse image of B under A*. The definition of $B \restriction A$ in our algebra is as follows.

Definition 8: Let B be an SFT and A an SFA. The *domain restriction of B for A*, denoted $B \restriction A$, is the SFT obtained from $B \times A$ by eliminating the second output component ϵ from all the rules. ⊠

The following property follows from (1) and (2).

$$\mathscr{T}_{B\restriction A}(\boldsymbol{b}) = \begin{cases} \mathscr{T}_B(\boldsymbol{b}), & \text{if } \boldsymbol{b} \in \mathscr{D}(A); \\ \emptyset, & \text{otherwise.} \end{cases} \quad (5)$$

We say that the SFT algebra in Figure 4 is *decidable* if validity of all the formulas F in the algebra is decidable.

Theorem 2 (SFT-algebra)**:** The algebra of SFTs is decidable if the label theories are decidable.

Proof. The SFA operations are effectively closed under intersection an complement and equivalence is decidable if satisfiability of the guards is decidable [48]. Decidability of 1-equality of SFTs is Theorem 1. Closure under composition is Proposition 1. Domain restriction is given in (5). □

The following corollary identifies a collection of practically relevant decision problems that follow from Theorem 2. *Subsumption* of SFTs, $A \sqsubseteq B$, is the problem of deciding if $\mathscr{T}_A(\boldsymbol{b}) \subseteq \mathscr{T}_B(\boldsymbol{b})$ for all $\boldsymbol{b}$. *Reachability* is the problem of existence of an input that is transformed to an output accepted by an SFA.

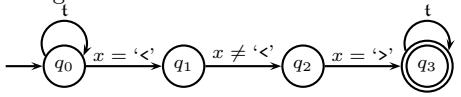
Corollary 1: The following decision problems over single-valued SFTs over a decidable label theory are decidable: Subsumption; Equivalence; Idempotence; Commutativity; Reachability.

Proof. Assume A and B are single-valued SFTs and recall Proposition 2. Subsumption, $A \sqsubseteq B$, is $\mathbf{d}(A) \subseteq \mathbf{d}(B) \wedge A \overset{1}{=} B$. Equivalence, $A \equiv B$, is $A \sqsubseteq B \wedge B \sqsubseteq A$. Idempotence is $A \equiv A \circ A$. Commutativity is $A \circ B \equiv B \circ A$. Reachability of a given output SFA D is $A \circ D \neq \emptyset$. □

The following example illustrates a use of the SFT algebra for reachability analysis of SFTs. The example is a digest behind security analysis of string sanitizers with respect to known XSS attack vectors.

Example 6: Consider the SFT $B = \mathit{GetTags}$ from Example 3. Is it possible that B does not detect all tags? In other words, does there exist an input $\boldsymbol{b}$ that matches the regex $P =$ `".*<[^<]>.*"` but $\mathscr{T}_B(\boldsymbol{b}) = \{\epsilon\}$?

Let A_ϵ and $A_\emptyset$ be the SFAs such that $\mathscr{D}(A_\epsilon) = \{\epsilon\}$ and $\mathscr{D}(A_\emptyset) = \emptyset$. Let A_P be the SFA that accepts all strings that match the regex P.



The question is equivalent to deciding if (6) fails,

$$\underbrace{(B \restriction A_P) \circ A_\epsilon}_{D} \equiv A_\emptyset \quad (6)$$

because, for $\boldsymbol{b} \in \Sigma^*$,

$$\begin{aligned} \boldsymbol{b} \in \mathscr{D}(D) \quad &\Leftrightarrow \quad \exists\, \boldsymbol{a}\, (\mathscr{T}_{B\restriction A_P}(\boldsymbol{b}) = \{\boldsymbol{a}\} \wedge \boldsymbol{a} \in \mathscr{D}(A_\epsilon)) \\ &\Leftrightarrow \quad \mathscr{T}_{B\restriction A_P}(\boldsymbol{b}) = \{\epsilon\} \\ &\Leftrightarrow \quad \mathscr{T}_B(\boldsymbol{b}) = \{\epsilon\} \wedge \boldsymbol{b} \in \mathscr{D}(A_P) \end{aligned}$$

It turns out that D (when minimized) is an SFA with 8 states, e.g., `"<a<a>"` $\in \mathscr{D}(D)$. (What is remarkable is that D can be effectively converted back to a regex that describes *all* inputs where tags are not detected.) By considering the witness, it can easily be traced back to the missing case in the `GetTags` program: line 8 of the C# code should be `q = (x == '<' ? 1 : 0);`. By verifying (6) for the SFT corresponding to the fixed code, we can verify the new code indeed detects all tags. ⊠

It also follows from Theorem 1 and Proposition 2, that we can decide single-valuedness of SFTs. This is a practically valuable result that lifts the burden of the semantic assumption of single-valuedness in decision problems that assume single-valuedness.

Corollary 2: Single-valuedness of SFTs over a decidable label theory is decidable.

3.4 Extension with Registers

We extend SFTs to symbolic transducers or STs by allowing the use of registers. This will provide a more succinct representation, and enable more efficient symbolic analysis methods to be used, by taking advantage of recent advances is SMT technology [15]. An ST uses a set of variables called *registers* as a symbolic representation of states. The rules of an ST are guarded commands with a symbolic input and output component. Since the finite state component of an SFT can be represented with a particular register of finite type, the explicit state component is omitted from STs. Moreover, by using Cartesian product types, we represent multiple registers with a single (compound) register.

Definition 9: A *Symbolic Transducer* or *ST* with input type σ, output type γ and register type τ is a tuple (q^0, ϑ, R), where $q^0 \in \mathcal{U}^\tau$ is the *initial state*, ϑ is a τ-predicate called the *final state condition*, and R is a finite set of *rules* $(\varphi, \boldsymbol{f}, g)$ where φ is a $(\sigma\times\tau)$-predicate, $\boldsymbol{f}$ is a sequence of λ-terms of type $(\sigma\times\tau) \to \gamma$, and g is a λ-term of type $(\sigma\times\tau) \to \tau$. ⊠

We write $A^{\sigma/\gamma;\tau}$ to indicate the input/output element type σ/γ and the register type τ of an ST A. When we write $A^{\sigma/\gamma}$, we assume τ to be implicit. A rule $(\varphi, \boldsymbol{f}, g) \in R_A$ denotes the following set of concrete transitions:

$$[\![(\varphi, \boldsymbol{f}, g)]\!] \overset{\text{def}}{=} \{q \xrightarrow{a/[\![\boldsymbol{f}]\!](a,q)} [\![g]\!](a,q) \mid (a,q) \in [\![\varphi]\!]\}$$

Although the formal definition omits finite states, it is often useful to explicitly include a separate finite state component, we do this in the examples below. Moreover, it is technically convenient to extend final states with *final outputs* by extending ϑ to be a finite set of *final output rules* $(\psi, \boldsymbol{g})$ where ψ is a τ-predicate and $\boldsymbol{g}$ is a sequence of λ-terms of type $\tau \to \gamma$. Intuitively, a final output is the output produced when the end of the input has been reached, often this is ϵ, but it need not be. When all final outputs are ϵ then ϑ is equivalent to being a τ-predicate as in Definition 9, i.e., $q \in [\![\vartheta_A]\!]$ then means that $q \xrightarrow{/\epsilon}_A$. Final outputs correspond to a restricted use of input-epsilon rules as used in classical finite transducers.

The reachability relation $p \xrightarrow{\boldsymbol{a}/\boldsymbol{b}}\!\!\!\!\!\rightarrow_A q$ for $\boldsymbol{a} \in \Sigma^*$, $\boldsymbol{b} \in \Gamma^*$, and $p, q \in \mathcal{U}^\tau$ is defined analogously to SFTs. The definition of $\mathscr{T}_A$ is lifted similarly, for $\boldsymbol{a} \in \Sigma^*$:

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

Example 7: Consider the C# code in Example 3. There is a direct mapping of the code to an ST A that uses the compound register $\langle q, c\rangle$. The initial state of A is $\langle 0, 0\rangle$, the

final state condition is $\mathfrak{t}$ and the rules are (we omit λ's):

$(q{=}0 \wedge x{=}\text{`<'}, \epsilon, \langle 1, x\rangle)$, $(q{=}0 \wedge x{\neq}\text{`<'}, \epsilon, \langle 0, x\rangle)$,
$(q{=}1 \wedge x{=}\text{`<'}, \epsilon, \langle 1, x\rangle)$, $(q{=}1 \wedge x{\neq}\text{`<'}, \epsilon, \langle 2, x\rangle)$,
$(q{=}2 \wedge x{=}\text{`>'}, [\text{`<'}, c, \text{`>'}], \langle 0, x\rangle)$, $(q{=}2 \wedge x{\neq}\text{`>'}, \epsilon, \langle 0, x\rangle)$

The register update $\langle r_q, r_c\rangle$ of a rule corresponds to the assignments $q := r_q$ and $c := r_c$. Since all assignments to c have the form $c := x$, c corresponds to the *previous* input character. A can be automatically transformed to the equivalent *GetTags* SFT; the register c is eliminated by using a new state and nondeterminism. ⊠

One can effectively construct a well-founded axiomatic theory $Th(A)$ of an ST $A^{\sigma/\gamma}$ over a background of *lists*, similarly to symbolic automata in [47]. $Th(A)$ defines a symbol T_A that provides a sound and complete axiomatization of $\mathscr{T}_A$, i.e., for any model $\mathfrak{A} \models_{\mathcal{U}} Th(A)$, $T_A^{\mathfrak{A}} = \mathscr{T}_A$.

Moreover, $Th(A)$ can be directly asserted as an *auxiliary theory* of any state-of-the-art SMT solver that supports lists. By deploying $Th(A)$ in this way, we obtain an *integrated decision procedure* for satisfiability and model generation for quantifier free formulas that may arbitrarily combine formulas over the background $\mathcal{U}$ with *transduction atoms* $T_A(u, v)$ where $u : \mathbb{L}\langle\sigma\rangle$ and $v : \mathbb{L}\langle\gamma\rangle$ are arbitrary list terms. A direct application, outlined in Figure 5, is a semi-decision proce-

Witness1disequality$(A^{\sigma/\gamma}, B^{\sigma/\gamma}) \stackrel{\text{def}}{=}$

1 **assert** $Th(A) \cup Th(B)$;
2 $(u, v, w) := (nil, NewVariable(\mathbb{L}\langle\gamma\rangle), NewVariable(\mathbb{L}\langle\gamma\rangle))$;
3 **while** $\mathfrak{t}$
4 **if** $\exists\,\mathfrak{A}\,(\mathfrak{A} \models_{\mathcal{U}} T_A(u, v) \wedge T_B(u, w) \wedge v{\neq}w)$ **return** $(u^{\mathfrak{A}}, v^{\mathfrak{A}}, w^{\mathfrak{A}})$;
5 **else** $u := cons(NewVariable(\sigma), u)$;

Figure 5. Given STs $A \not\stackrel{1}{=} B$, generates a witness $(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})$ such that $\boldsymbol{b} \in \mathscr{T}_A(\boldsymbol{a})$, $\boldsymbol{c} \in \mathscr{T}_B(\boldsymbol{a})$, and $\boldsymbol{b} \neq \boldsymbol{c}$.

dure for *1-disequality* of STs, where the auxiliary theories are asserted to the solver in line 1, and successively longer input-lists are used to invoke the solver to decide 1-disequality of the instance in line 4. The semi-decision procedure computes a shortest-input witness of 1-disequality.

4. Case Studies

We present four case studies for applications of SFTs. The first case study focuses on sophisticated string manipulation, that goes beyond our first case study of sanitizer analysis with Bek [4] (that is discussed further in Section 5), we want to emphasize that the utility of SFTs goes well beyond reasoning about string sanitizer processing. Figure 2 summarizes the essential features of each case study.

4.1 Representing `HTMLDecode`

To prevent injection attacks such as cross-site scripting (XSS) and SQL injection, Web applications employ *sanitizers*, which are string manipulation routines that remove or encode dangerous input characters. Many applications include their own sanitizer implementations. Recent work by Hooimeijer et al. [4] examines several such sanitizers, demonstrating that a subset of popular sanitizers can be modeled using transducers. Furthermore, they show that safety properties of Web sanitizers can be checked using transducer analyses.

We focus on the sanitizer `HTMLDecode` to evaluate the practical utility of the ST representation. Figure 6 outlines a

```
public String HTMLDecode( String input ) {
  StringBuffer sb = new StringBuffer();
  PushbackString pbs = new PushbackString( input );
  while ( pbs.hasNext() ) {
    Character c = decodeCharacter( pbs );
    if ( c != null ) { sb.append( c ); }
    else { sb.append( pbs.next() ); } }
  return sb.toString();
}

public Character decodeCharacter( PushbackString input ) {
  input.mark();
  Character first = input.next();
  if ( first == null ) { input.reset(); return null; }
  if ( first.charValue() != '&' ) {
    input.reset(); return null; }
  Character second = input.next(); if ( second == null ) {
    input.reset(); return null; }
  if ( second.charValue() == '#' ) {
    Character c = getNumericEntity( input );
    if ( c != null ) return c; }
  else if ( Character.isLetter( second.charValue() ) ) {
    input.pushback( second );
    Character c = getNamedEntity( input );
    if ( c != null ) return c; }
  input.reset();
  return null;
}

private Character getNumericEntity( PushbackString input ) {
  ...
  return parseNumber( input );
}

private Character parseNumber( PushbackString input ) {
  StringBuffer sb = new StringBuffer();
  while( input.hasNext() ) {
    Character c = input.peek();
    if ( Character.isDigit( c.charValue() ) ) {
      sb.append( c );
      input.next(); }
    else if ( c.charValue() == ';' ) {
      input.next();
      break; }
    else break; }
  try {
    int i = Integer.parseInt(sb.toString());
    return new Character( (char)i ); }
  catch( NumberFormatException e ) return null;
}
```

Figure 6. Excerpt `HTMLDecode` in Java (from OWASP 1.4.0). The code shown converts named entities (e.g., `<` to `<`) and numeric entities (e.g., `&#52;` to `4`). The numeric entity conversion is difficult to model efficiently using previous approaches.

real-world implementation, taken from the OWASP library. `HTMLDecode` transforms HTML *entities* back to the symbol they represent. Entities can be named (e.g., `<` maps to `<`), or numeric in decimal or hexadecimal representation (e.g., decimal entity `&#48;` maps to symbol `0`). For simplicity, we will restrict our attention to decimal entities.

Intuitively, `HTMLDecode` is difficult to cast as a transducer because it requires *lookahead*: a single output symbol may depend on a specific sequence of several characters. The full Unicode set consists of more than one million symbols. To decode a decimal entity, therefore, we need to inspect up to six digits. While that is possible using either SFTs or traditional transducers, it requires a large state space.

In contrast, the corresponding ST is quite succinct. Figure 7 shows a ST $Decode^{\mathbb{Z}/\mathbb{Z};Q\times(\mathbb{Z}\times\mathbb{Z})}$, that uses two registers to handle numeric entities with exactly two digits. The compound register is $\langle q, \langle y, z\rangle\rangle$ We illustrate explicitly the finite

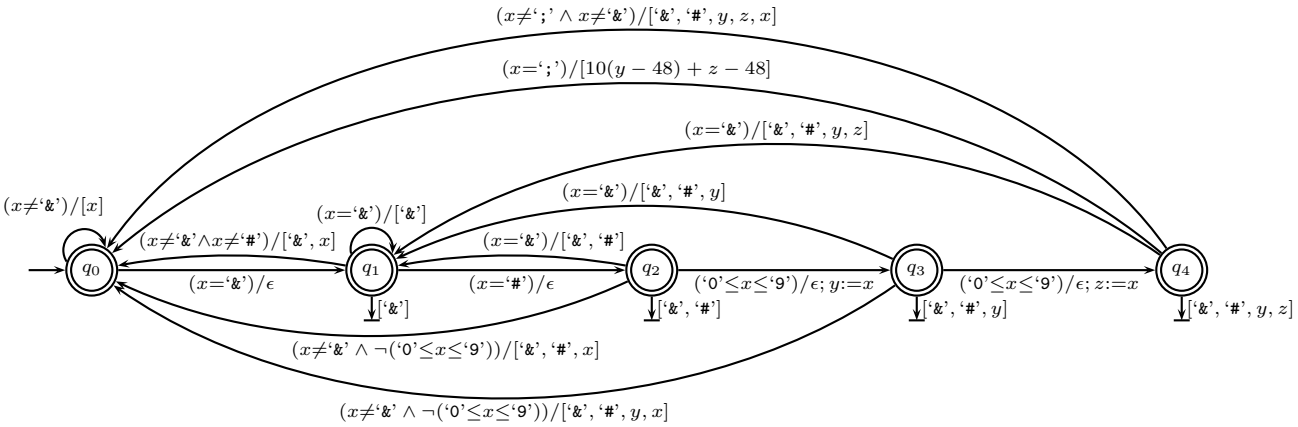


Figure 7. ST representation of the `HTMLdecode` code in Figure 6, restricted to decimal numeric entities of the form `&#[0-9][0-9];`. This ST uses two registers to remember one digit each, and uses integer-linear arithmetic to compute the corresponding code point. The λ-terms are implicit in the labels, e.g., the output $10(y-48)+z-48$ is short for $\lambda(x,(y,z)).10(y-48)+z-48$, where x is the input and (y,z) the register.

state component (that is the value of q). Final outputs, unless they are ϵ, are shown by labels on outgoing $\dashv$ arcs from final states. Characters correspond to their Unicode code points, e.g., `'0'` = 48. The actual decoding happens in the rule from state q_4 to q_0 with guard $x =$ `';'`, where the output $10 \cdot (y-48) + z - 48$ corresponds to invocation of `parseInt` in Figure 6. The states q_i roughly correspond to the control flow of the code in Figure 6.

Evaluation: We compare the ST representation to the equivalent SFTs, in terms of size and analysis speed. Let $\mathsf{DecodeST}_n$ denote an ST that models `HTMLDecode` for entities of the form `&#[0-9]{1,n};` (i.e., up to n decimals, inclusive). For each $i \in [2;6]$, we compute an SFT that is equivalent to $\mathsf{DecodeST}_i$ by concretizing the possible register values at each state. Figure 8(a) shows the number of both states and edges on the y-axis, for both representations; the x-axis denotes the number of digits modeled. The most prominent take-away is that the ST representation is drastically smaller. For example, for the 6-digit encoding, the SFT encoding has over $10,000\times$ as many states, and $135,000\times$ as many edges, as the equivalent ST encoding.

We consider the speed of two algorithms: composition and equivalence checking. The experimental task is to find a witness (i.e., an input string) w that demonstrates that $\mathsf{DecodeST}_i(w) \neq \mathsf{DecodeST}_i(\mathsf{DecodeST}_i(w))$, for some number of self-compositions. We consider three different algorithms for performing this task: ST.E uses an in-memory representation and conducts an *eager* search (computing entire transducers); ST.L uses a *lazy* approach by encoding the entire ST into the underlying SMT solver; and SFT represents an eager SFT-based approach.

Figure 8(b) shows the speed results. Compositions represents the number of times we compose each class of transducer with itself. Composition refers to the time taken to perform the composition; for the lazy ST.L this time is negligible (since the composition is simply asserted to the underlying solver). IdempotenceChecking refers to the time taken to find the actual witness, after the composition. Each column represents a single experimental run; we employed a 2-hour timeout per run. The label OM marks runs that ran out of memory, while TO marks cases that hit the 2-hour timeout.

These results show that STs, in particular using the lazy representation, are significantly more scalable for this task than SFTs. The SFT representation either exhausts memory or passes the timeout in the majority of cases. The eager ST representation outperforms the lazy representation only for the smallest two testcases. For larger runs (i.e., more compositions and more digits), the lazy SFT representation scales much more reliably, ranging from 1 to 20 seconds over the 2-composition range (compared to several minutes for the eager representation).

4.2 Malware Fingerprinting Code

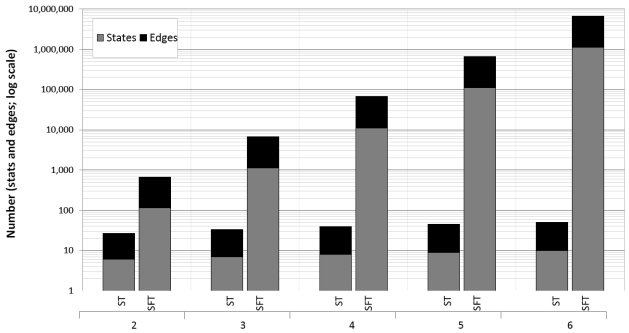
Millions of web pages today contain malicious JavaScript that attempts to take over a victim's web browser. An active research literature has proposed static and dynamic methods for detecting these attacks [9, 11, 13, 41]. A key finding of this work is that malware authors use *fingerprinting* techniques [17, 36] to decide which malware to deliver to the victim user.

Figure 9 shows an example of client-side browser fingerprinting. [2] The code iterates over the list of plugins installed in the browser and queries their version numbers. In some cases, version numbers are padded by optionally adding leading 0s to them. Finally, variables `quicktime_plugin` and `adobe_plugin` are combined to produce the final `fingerprint` value. This fingerprint value is then used to select a specific attack to run against the user.

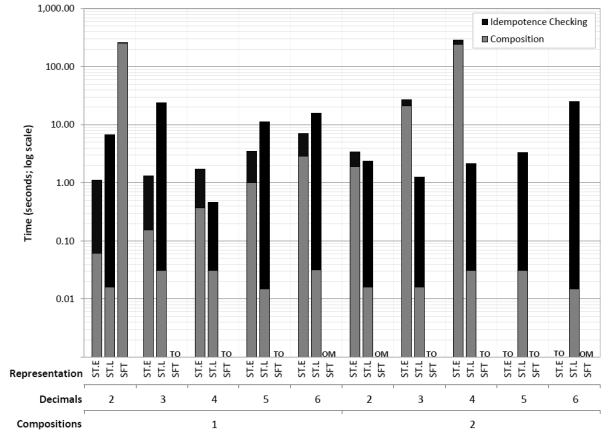
Figure 12 list several concrete fingerprint values from real browser setups. Note that QuickTime 7.6.6 has at least one known vulnerability, and may thus be of special interest to an attacker.

We consider a scenario in which we have acquired these fingerprints (e.g., through network sniffing), and want to find out the corresponding plugin names. At a higher level, the question is: "Can we find out interesting properties by com-

[2] This code is simplified for illustrative purposes: the original considers more plugin types, including Flash, etc.



(a) Initial transducer sizes.



(b) Running times.

Figure 8. `HTMLDecode` results. The task is to prove that `HTMLDecode` does not commute with itself, and to provide a witness that demonstrates this for a given number of Compositions. We evaluate three representations: ST.E (eager ST composition), ST.L (lazy ST composition using Z3), and SFT (SFT composition). We consider five distinct models (indicated by Decimals, based on how many digits the ST or SFT can handle.

puting string-related pre-conditions based on a postcondition?"

Our techniques can answer this question in the affirmative by modeling the code of Figure 9 using multiple SFTs. The key idea is conditional assignment translates into nondeterministic case splits inside the SFT. At a high level, each transducer corresponds to a split or a merge in the control flow of the fingerprinting code, relative to a single variable of interest. This is illustrated in Figure 10, which shows the transducer `QuicktimeSplitter` together with the path predicates modeled by each of its three main branches. The transducer reads both `quicktime_plugin` and `plugin_name`, separated by a special `#` symbol. Its output is guaranteed to start with `#` if and only if the branch `if(quicktime`$_{\text{p}}$`lugin == 0&&...` was taken.

For the sake of brevity, we do not display the remaining SFTs. Figure 11 lists the full set of SFTs and their statistics. `QuicktimeMerger` takes the output of `QuicktimeSplitter` and models the control flow join at the end of the first `if` statement. `QuickTimePadder` models the final while loop in Figure9. The manipulation of variable `adobe_plugin` is analogous.

```
var quicktime_plugin = "0", adobe_plugin = "00";

for(var i = 0; i < navigator.plugins.length; i++)
{
  var plugin_name = navigator.plugins[i].name;
  if (quicktime_plugin == 0 &&
    plugin_name.indexOf("QuickTime") != -1)
  {
     var helper = parseInt(plugin_name.replace(/\D/g,""));
     if (helper > 0) // not base 16
       quicktime_plugin = helper.toString(10)
  }
  if (adobe_plugin == "00" &&
    plugin_name.indexOf("Adobe Acrobat") != -1)
  {
    plugin_name = navigator.plugins[i].description;
    if(plugin_name.indexOf(" 5") != -1) adobe_plugin = "05";
    else if(plugin_name.indexOf(" 6") != -1) adobe_plugin = "06";
    else if(plugin_name.indexOf(" 7") != -1) adobe_plugin = "07";
    else adobe_plugin = "01"
  } else {
    // flash, java...
  }
}

while(quicktime_plugin.length < 8)
  quicktime_plugin = "0" + quicktime_plugin;

var fingerprint = "Q" + quicktime_plugin + "8" + adobe_plugin;
// ...
fetch_exploit(fingerprint);
```

Figure 9. Browser and plugin fingerprinting code found in JavaScript malware.

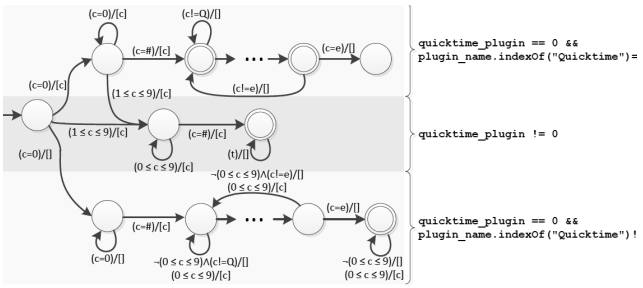


Figure 10. SFT `QuicktimeSplitter` with corresponding path predicates for the fingerprinting code.

Evaluation: We compute the pre-image of the composition transducers discussed above as follows. For each fingerprint w, we construct an SFA that accepts $\{w\}$ and corresponds to the postcondition `fingerprint ==` w. The pre-images of *QuicktimeComposed* and *AdobeComp* correspond to preconditions for a single iteration of the `for` loop in Figure 9. For example, for w = `Q00000769801`, we find a precondition that relates values of `quicktime_plugin` to values of `plugin_name`, as follows: (1) `quicktime_plugin` already had value `769`, possibly padded with up to five zeroes, or (2) `quicktime_plugin` consisted entirely of zeroes and `plugin_name` contained the substring $\widehat{\text{Quicktime}}$ together with digits `7`, `6`, and `9` in that relative order. Condition (1) represents the case where `quicktime_plugin` had a previously-assigned version number, while condition (2) represents the case in which a version number was extracted inside the `for` loop. For all real fingerprints we tried, our analysis took less than one second per fingerprint.

Next, we evaluate whether inverse image generation, as used above, scales to relatively large output values. Unlike most previous string constraint solvers [26, 30, 42], SFT-

SFT	States	Edges
Quicktime (variable `quicktime_plugin`)		
`QuicktimeSplitter`	25	60
`QuicktimeMerger`	6	9
`QuicktimePadder`	37	37
Composed	534	1,425
Adobe (variable `adobe_plugin`)		
`AdobeSplitter`	36	81
`AdobeMerger`	21	40
Composed	203	797

Figure 11. SFTs used for the malware fingerprinting example. Statistics for the Quicktime and Adobe components are shown. The composition SFTs take approximately one second to compute.

Browser/plugin combination	Fingerprint
FF: Acrobat 9.4.5.236; no quicktime	`Q00000000801`
FF: Acrobat 9.4.5.236; Quicktime 7.6.9	`Q00000769801`
IE: no plugins of interest installed	`Q00000000800`
FF: Acrobat 9.4.5.236; Quicktime 7.5.5	`Q00000755801`
FF: Acrobat 9.4.5.236; Quicktime 7.6.6	`Q00000766801`

Figure 12. Browser fingerpints. Using an SFT model, computing input values for `quicktime_plugin` and `adobe_plugin` takes less than one second per fingerprint.

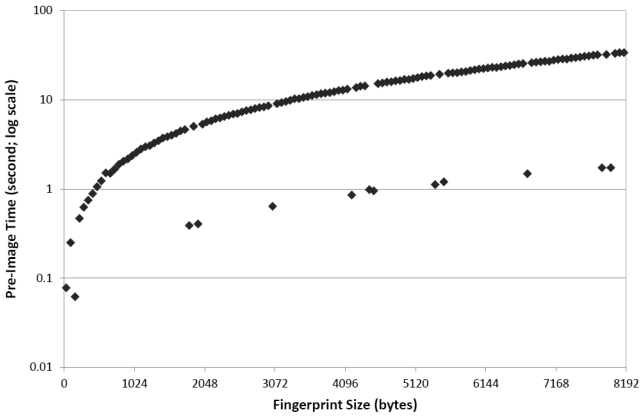


Figure 13. Inverse image generation time in seconds for fingerprint outputs up to 8192 bytes. The outputs were randomly generated from the language `Q[0-9]{n}801` over 16 bit characters.

based analysis does not impose length bounds on the strings under consideration. This is only beneficial if the approach actually scales to large strings. We show the approach *does* scale by generating random fingerprints of the form `Q[0-9]{n}801`, and measuring the time it takes to compute the inverse images for both the QuickTime and Adobe variables. Figure 13 shows encouraging results: in general, our approach takes less than half a minute to generate pre-images for up to *eight kilobytes* worth of output. In contrast, the Hampi solver was limited to finding pre-images up to fifty bytes worth of output.

Our malware case study demonstrates several important points. First, SFTs are well-suited for describing code by making use of non-determinism. The transducers needed can be large (on the order of hundreds of states and edges), but

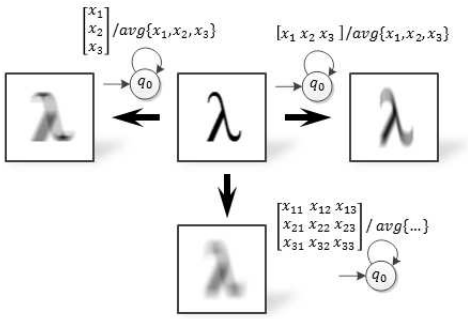


Figure 14. Blurring transformation illustrated.

we can construct them from much smaller transducers (tens of states and edges) through composition. The pre-image computation reveals interesting relations among mutually dependent variables. Finally, the pre-image computation is efficient: it can generate valid string inputs for outputs that measure several kilobytes in size, while we are unaware of any previous string constraint solver that can handle this order of magnitude.

Takeaways: From our first two case studies, we have the following key takeaways

- STs can be radically more succinct in representation than SFTs: we saw in our HTMLDecode example that our ST representation had 10,000 times fewer states and 150,000 times fewer edges than our SFT representation.
- Lazy ST encoding scales best for our `HTMLDecode` example, taking between 1 to 20 seconds for six characters and two compositions. Eager ST encoding is slower, and eager SFT encoding times out above two characters.
- SFTs can accurately model real examples of malicious Javascript fingerprinting code. Our analysis requires less than one second to recover plug-in versions from real examples of fingerprints generated by malicious code found "in the wild."
- Previous work in string constraint solving has focused almost exclusively on constraints that require fewer than 50 bytes; for example, the majority of Hampi experiments were conducted with length bounds of 15 bytes or fewer [30]. In contrast, our techniques can synthesize pre-images in excess of 8,000 bytes in roughly a minute.

4.3 Image Blurring

To illustrate the generality of SFTs, we look at image transformations. A clear advantage of representing image transformations in the form of transducers is the ability to do composition on transducers it gives us. In fact, image editors such as Google Picasa represent image contents as the original image as well as a series of image transformations, such as blurring, sharpening, black and white conversion, contrast enhancement, and the like.

In many cases, of course, editing a large-scale image that includes millions of 32-bit pixes before high-quality printing might involve a dozen of such transformations. Applying them one after another, in a sequence is often too time-consuming to be practical. A better alternative consists of composing the transformations together and applying them to the input image only a single time.

We focus on *image blurring*, which is a prototypical "textbook" image transformation [40]. Figure 14 illustrates

the two transducers for *horizontal* and *vertical* blurring of an image.

Measuring privacy via entropy: A fascinating feature of our analysis is that we can estimate a *privacy* metric for image blurring using our techniques. Our starting point is the observation that if an image has a *unique* pre-image given blurring, then the blurring does not hide the original image at all. Just consider a black square: no matter how many times we might attempt to blur it, the image will remain unchanged. On the other hand, after blurring a face, there may be multiple original images that yield the same blurred face.

Put another way, a blurred face image defines a set of potential candidate original images. If we assume that all candidate original images are equally likely, then we can define the *entropy* H of the original face after a blurring transformation β as follows: $H = -log(|\beta(\beta^{-1})|)$, in other words, we use the reverse mapping β^{-1} given to us by the inverse transducer, and take the negated logarithm of the size of its preimage. For the entirely black image example, $H = 0$ because there is only one element in the preimage. To increase privacy, our goal is to maximize the entropy. Note that given for images of a given rectangular size, we can always exhaustively check if two images result in the same image after blurring.

Our techniques allow us to write down a SMT formula where the number of solutions is equal to the number of preimages of β on a specific image. This is the first connection to our knowledge between SMT techniques and probabilistic definitions of privacy. Of course computing the exact number of solutions is $\#P$-complete, but we can employ approximation techniques to estimate this quantity. Then we can programmatically compare different methods of blurring by the entropy induced. We leave exploration of the impact of different approximation techniques as future work.

4.4 Location Privacy

GPS sensors located in most mobile devices constantly track our location [2, 32]. While the popularity of applications such as Foursquares, Gowalla, and Facebook check-ins show user demand for sharing this information, this also raises privacy concerns.

As a reaction to privacy concerns, Google's Latitude location sharing service started allowing users to only release the *city* they are in, not their precise location, so that a trace of a user's day might look like

```
Menlo Park, CA; Palo Alto, CA; Los Gatos, CA;
Mountain View, CA; Los Gatos, CA.
```

Clearly, given the sizes of these cities, tracking the user precisely might present some difficulty. To generalize, one approach that has emerged is context sensitive choice of granularity; intuitively, if I am located in a densely-populated location such as midtown Manhattan, block-level location is fine to reveal. If I am in a sparsely-populated area such as the Death Valley, we should only reveal a coarse location approximation. This can be encoded with a transducer that works on a stream of recorded location measurements. To summarize, given GPS coordinates ⟨latitude/longitude⟩:

1. Use a lookup list of world cities and their latitude and longitude values and nearset point calculation to compute the nearest city to the current location;
2. Determine the city population via a lookup table;
3. Map the population to a high or low density area (`H` or `L`).
4. Based on the last five GPS readings, enter a high- or low-density output state. Depending on that, the ⟨latitude,longitude⟩ pair is approximated with different precision.

This can be captured by a transducer with different output actions depending on the current state. For instance, for high-density areas, we can drop GPS location seconds, and for low-density areas we can drop GPS location minutes. Krumm et al consider additional trace obfuscation techniques such as adding noise or quantizing traces, which can also be represented using our SFT framework [8].

5. Related Work

In an applied setting [27] we introduced and applied SFTs to analysis of security sanitizers in the Bek project. This work relied on semi-decision procedures for SFT analysis and did not state or prove any algorithmic results on SFT decision procedures. In comparison, the current paper provides the missing formal foundation for Bek program analysis. It further develops new and more general algorithms, including the more general 1-equality algorithm that factors out the decision problem for single-valuedness. The new support for registers enable new, previously infeasible, application areas. These include HtmlDecode, even Example 7 and most of the case study section. Furthermore, the new support for nondeterminism allows elimination of registers, as illustrated in Example 7, without violating single-valuedness.

General equivalence of finite state transducers is undecidable [23], and already so for very restricted fragments [28]. Equivalence of decidability of single-valued GSMs was shown in [43], and extended to the finite-valued case (there exists k such that, for all v, $|\mathscr{T}_A(v)| \leq k$) in [12, 50]. The decidability of equivalence of the finite-valued case does not follow from the single-valued case. Corresponding decidability result of equivalence of finite-valued SFTs is shown in [6]. Unlike for the single-valued case that has a practical algorithm (Figure 3), the finite-valued case is substantially harder, the 1-equality algorithm does not generalize to this case because the satisfiability checks cannot be made locally: Lemma 2 does not imply violation of partial-equivalence in the finite-valued case.

In recent years there has been considerable interest in automata over infinite languages [44], starting with the work on *finite memory automata* [29], 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* [37] and *data automata* [7]. Several characterizations of logics with respect to different models of data word automata are studied in [5]. 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* [22] that are finite state automata whose transitions are labeled by elements of an atomic lattice with motivation coming from verification of symbolic communicating machines.

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, e.g., full linear arithmetic is not allowed.

Finite state automata with arbitrary predicates over labels, called *predicate-augmented finite state recognizers*, or

symbolic finite automata (SFAs) in the current paper, were first studied in the context of natural language processing [38]. While the work [38] views symbolic automata as a "fairly trivial" extension, the fundamental algorithmic questions are far from trivial. For example, it is shown in [24] that symbolic complementation by a combinatorial optimization problem called *minterm generation* leads to significant speedups compared to state-of-the-art automata algorithm implementations. The work in [38] introduces a different symbolic extension to finite state transducers called *predicate-augmented finite state transducers*. This extension is not expressive enough for describing SFTs. Besides identities, it is not possible to establish functional dependencies from input to output that are needed for example to encode transformations such as `HtmlEncode`.

We use the SMT solver Z3 [15] for solving label constraints that arise during composition and equivalence checking algorithms, as well as for witness search by model generation using auxiliary SFT axioms.

Finite state transducers have been used for dynamic and static analysis to validate sanitization functions in web applications in [3], 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 [35, 49]. Yu et al. show how multiple automata can be composed to model looping code [51].

Our work is complementary to previous efforts in using SMT solvers to solve problems related to list transformations. The tools HAMPI [30] and Kaluza [42] extend the STP solver to handle equations over strings and equations with multiple variables. The work in [25] shows how to solve subset constraints on regular languages. In contrast, we show how to combine any of these solvers with SFTs whose edges can take symbolic values in the theories understood by the solver.

Top-down tree transducers [21] provide another extension of finite state transducers: a finite state transducer is a top-down tree transducer over a *monadic* ranked alphabet. Similar to finite state transducers, decidability of equivalence of top-down tree transducers is known for the single-valued case [18, 20], including a specialized method for the *deterministic* case [10], and also for the finite-valued case [45]. Several non-symbolic extensions of top-down tree transducers have been studied, e.g., [19, 21, 31, 33, 34, 39]. Symbolic top-down tree transducers are studied in [46] where partial equivalence is shown to be decidable for the linear single-valued case.

6. Conclusion

We introduced a symbolic extension of the theory of classical finite transducers, where transitions are represented by terms modulo a given background theory. Our approach enables a range of analyses in combination with state-of-the-art constraint solving techniques. The core algorithms we presented are composition and equivalence checking of single-valued symbolic finite transducers, and we showed how to decide whether arbitrary symbolic transducers have the single-valuedness property. We demonstrated how our work directly applies to analysis of web string sanitizers, malware detection, image manipulation, and location privacy, and we expect more applications to follow. Our techniques can synthesize string pre-images in excess of $8,000$ bytes in roughly a minute, our ST representation had $10,000$ times fewer states than previous approaches, and we found lazy ST encoding for our `HTMLDecode` example took at most 20 seconds even in the most extreme cases. These algorithms make it possible to work with symbolic representations of transducers, just as traditionally done with finite state transducers, as first class citizens in designing new analyses and program transformation techniques by leveraging the continuous advances and improvements in constraint solvers and satisfiability modulo theories solvers.

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. E. Bakke, R. Parameswaran, D. M. Blough, A. A. Franz, and T. J. Palmer. Data obfuscation: Anonymity and desensitization of usable data sets. *IEEE Security and Privacy*, Apr. 2004.

[3] 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.

[4] Bek. http://research.microsoft.com/bek.

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

[6] N. Bjørner and M. Veanes. Symbolic transducers. Technical Report MSR-TR-2011-3, Microsoft Research, January 2011.

[7] 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.

[8] A. Brush, J. Krumm, and J. Scott. Exploring end user preferences for location obfuscation, location-based services, and the value of location. In *UbiComp*, September 2010.

[9] K. Z. Chen, G. Gu, J. Nazario, X. Han, and J. Zhuge. WebPatrol: Automated collection and replay of web-based malware scenarios. In *ASIACCS*, March 2011.

[10] B. Courcelle and P. Franchi-Zannettacchi. Attribute grammars and recursive program schemes. *Theoretical Computer Science*, 17:163–191, 1982.

[11] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *WWW Conference*, Raleigh, NC, April 2010.

[12] K. Culic and J. Karhumäki. The equivalence of finite-valued transducers (on HDTOL languages) is decidable. *Theoretical Computer Science*, 47:71–84, 1986.

[13] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static javascript malware detection. In *Proceedings of the Usenix Security Symposium*, Aug. 2011.

[14] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

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

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

[17] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18, 2010.

[18] J. Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book, editor, *Formal Language Theory*, pages 241–286. Academic Press, 1980.

[19] J. Engelfriet and S. Maneth. A comparison of pebble tree transducers with macro tree transducers. *Acta Informatica*, 39:2003, 2003.

[20] Z. Esik. Decidability results concerning tree transducers. *Acta Cybernetica*, 5:1–20, 1980.

[21] Z. Fülöp and H. Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers*. EATCS. Springer, 1998.

[22] 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.

[23] T. Griffiths. The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. *J. ACM*, 15: 409–413, 1968.

[24] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI'11*, LNCS. Springer, 2011.

[25] 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. ISBN 978-1-60558-392-1.

[26] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *ASE*, 2010.

[27] 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.

[28] 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.

[29] 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.

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

[31] N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *POPL*, pages 495–508. ACM, 2010.

[32] J. Krumm. A survey of computational location privacy. *Personal Ubiquitous Comput.*, 13:391–399, August 2009.

[33] A. Maletti, J. Graehl, M. Hopkins, and K. Knight. The power of extended top-down tree transducers. *SIAM J. Comput.*, 39:410–430, June 2009.

[34] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *Proc. 19th ACM Symposium on Principles of Database Systems (PODS'2000)*, pages 11–22. ACM, 2000.

[35] 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. ISBN 1-59593-046-9.

[36] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in javascript implementations. In *Proceedings of Web 2.0 Security and Privacy 2011 (W2SP)*, May 2011.

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

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

[39] C.-H. L. Ong and S. J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*, pages 587–598. ACM, 2011.

[40] J. R. Parker. *Algorithms for Image Processing and Computer Vision*. Wiley and Sons, 2006.

[41] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[42] 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.

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

[44] 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.

[45] H. Seidl. Equivalence of finite-valued tree transducers is decidable. *Math. Systems Theory*, 27:285–346, 1994.

[46] M. Veanes and N. Bjørner. Symbolic tree transducers. In *Perspectives of System Informatics (PSI'11)*, 2011.

[47] 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, 2010.

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

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

[50] A. Weber. Decomposing finite-valued transducers and deciding their equivalence. *SIAM Journal on Computing*, 22(1): 175–202, February 1993.

[51] 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.