String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS

Anthony W. Lin
Yale-NUS College, Singapore
anthony.w.lin@yale-nus.edu.sg

Pablo Barceló
Center for Semantic Web Research
& Dept. of Comp. Science, Univ. of Chile, Chile
pbarcelo@dcc.uchile.cl

Abstract

We study the fundamental issue of decidability of satisfiability over string logics with concatenations and finite-state transducers as atomic operations. Although restricting to one type of operations yields decidability, little is known about the decidability of their combined theory, which is especially relevant when analysing security vulnerabilities of dynamic web pages in a more realistic browser model. On the one hand, word equations (string logic with concatenations) cannot precisely capture sanitisation functions (e.g. htmlescape) and implicit browser transductions (e.g. innerHTML mutations). On the other hand, transducers suffer from the reverse problem of being able to model sanitisation functions and browser transductions, but not string concatenations. Naively combining word equations and transducers easily leads to an undecidable logic. Our main contribution is to show that the "straight-line fragment" of the logic is decidable (complexity ranges from PSPACE to EXPSPACE). The fragment can express the program logics of straight-line string-manipulating programs with concatenations and transductions as atomic operations, which arise when performing bounded model checking or dynamic symbolic executions. We demonstrate that the logic can naturally express constraints required for analysing mutation XSS in web applications. Finally, the logic remains decidable in the presence of length, letter-counting, regular, indexOf, and disequality constraints.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of Programs

Keywords String analysis, XSS, word equations, transducers

1. Introduction

The past decade has witnessed a significant amount of progress in constraint solving technologies, thanks to the emergence of highly efficient SAT-solvers (e.g. see [15, 42, 46]) and SMT-solvers (e.g. see [11, 24, 42]). The goal of SAT-solvers is to solve constraint satisfaction problem in its most basic form, i.e., satisfiability of propositional formulas. Nowadays there are numerous highly efficient solvers including Chaff, Glucose, Lingeling, and MiniSAT, to name a few (e.g. see [3] for others). Certain applications, however, require more expressive constraint languages. The framework of satisfiability modulo theories (SMT) builds on top of the basic constraint satisfaction problem by interpreting atomic propositions as a quantifier-free formula in a certain "background" logical theory, e.g., linear arithmetic. Today fast SMT-solvers supporting a multitude of background theories are available including Boolector, CVC4, Yices, and Z3, to name a few (e.g. see [4] for others). The backbones of fast SMT-solvers are usually a highly efficient SAT-solver (to handle disjunctions) and an effective method of dealing with satisfiability of formulas in the background theory.

In the past seven years or so there have been a lot of works on developing robust SMT-solvers for constraint languages over strings (a.k.a. *string solvers*). The following (incomplete) list of publications indicates the amount of interests in string solving (broadly construed): [7, 8, 16, 21, 23, 28–31, 34, 35, 38, 43, 48, 53, 55, 57, 65–69, 71–76]. One main driving force behind this research direction — as argued by the authors of [8, 23, 29, 35, 38, 38, 48, 48, 57, 65–69, 75, 76] among others — is the application to analysis of security vulnerabilities in web applications against code injections and cross-site scripting (XSS), which are typically caused by improper handling of untrusted strings by the web applications, e.g., leading to an execution of malicious JavaScript in the clients' browsers.

Despite the amount of recent progress in developing practical string solvers, little progress has been made on the foundational issues of string solving including *decidability*, which is particularly important since it imposes a fundamental limit of what we can expect from a solver for a given string constraint language (especially with respect to soundness and completeness). Perhaps the most important theoretical result in string solving is the decidability of satisfiability for *word equations* (i.e. string logic with the concatenation operator) by Makanin [45] (whose computational complexity was improved to PSPACE by Plandowski [51, 52]). It is known that adding regular constraints (i.e. regular-expression matching) preserves decidability without increasing the complexity [25, 52]. Very few decidability results extending this logic are known.

For such an application as detecting security vulnerabilities in web applications in a realistic browser model (e.g. see [70]), word equations with regular constraints alone are *insufficient*. Firstly, browsers regularly perform *implicit transductions*. For example, upon `innerHTML` assignments or `document.write` invocations, browsers mutate the original string values by HTML entity decoding, e.g., each occurrence of `&#34;` will be replaced by `"`. [Modern browsers admit some exceptions including the HTML entity names `&amp;`, `&lt;`, and `&gt;` (among others), which will not be decoded.] Since such transductions involve only conversions of one character-set encoding to another, they can be encoded as *finite-state (input/output) transducers*, as has already been noted in

[23, 35, 66, 70] (among others). Secondly, in an attempt to prevent code injection and XSS attacks, most web applications will first sanitise untrusted strings (e.g. obtained from an untrusted application) before processing them. Common sanitisation functions include JavaScript-Escape and HTML-Escape (an implementation can be found in The Closure Library [5]). HTML-Escape converts reserved characters in HTML such as `&`, `<`, and `'` to their respective HTML entity names `&amp;`, `&lt;` and `&#39;`. On the other hand, JavaScript-Escape will backslash-escape certain metacharacters, e.g., the character `'` and `"` are replaced by `\'` and `\"`. Again, such sanitisation functions can be encoded as finite-state transducers, as has been noted in [35] (among others).

Example 1. The following JavaScript code snippet adapted from a recent CACM article [37] is a simple example that uses *both* concatenations and finite-state transducers (both explicitly and implictly):

```
var x = goog.string.htmlEscape(cat);
var y = goog.string.escapeString(x);
catElem.innerHTML = '<button onclick=
    "createCatList(\'' + y + '\')">' + x + '</button>';
```

The code assigns an HTML markup for a hyperlink to the DOM element catElem. The hyperlink creates a category `cat` whose value is provided by an untrusted third party. For this reason, the code attempts to first sanitise the value of `cat`. This is done via The Closure Library [5] string functions `htmlEscape` and `escapeString` (implementing JavaScript-Escape). Inputting the value `Flora & Fauna` into `cat` gives the desired HTML markup:

```
<button onclick="createCatList('Flora &amp; Fauna')">
    Flora &amp; Fauna</button>
```

On the other hand, inputting the value `');alert(1);//` to `cat`, results in the HTML markup:

```
<button onclick="createCatList('&#39;);alert(1);//')">
    &#39;);alert(1);//')</button>
```

When this is inserted into the DOM via `innerHTML`, an implicit browser transduction will take place, i.e., first HTML-unescaping the value inside the `onclick` attribute and invoking the attacker's script `alert(1)` after `createCatList`. This subtle XSS bug (a type of *mutation XSS* [32]) is due to calling the appropriate escaping functions in the wrong order. □

It is well-known (e.g. see [8, 57, 75]) that string solving can be applied to detecting security vulnerabilities against a given injection and XSS attack pattern[1] P in the form of a regular expression. After identifying certain "hot spot" variables in the program where attacks can be performed (e.g. possibly via taint analysis), a string constraint will be generated that is satisfiable iff the program is vulnerable against a given attack pattern. In the above example, to analyse security vulnerabilities in the variable `catElem.innerHTML` against the following attack pattern (given in JavaScript regex notation; blank space inserted for readability):

```
e1 = /<button onclick=
        "createCatList\(' ( ' | [^']*[^'\\] ' ) \);
            [^']*[^'\\]' )">.*<\/button>/
```

one would express the program logic as a conjunction of:

- $x = R_1(\texttt{cat})$
- $y = R_2(x)$
- $z = w_1 \cdot y \cdot w_2 \cdot x \cdot w_3$ for some constant strings w_1, w_2, w_3, e.g., w_1 is `<button onclick="createCatList('`
- `catElem.innerHTML` $= R_3(z)$
- `catElem.innerHTML` matches `e1`.

Here, R_1 and R_2 are, respectively, transducers implementing `htmlEscape` and `escapeString`, while R_3 is a transducer implementing the implicit browser transductions upon `innerHTML` assignments. Note that the above string constraint cannot be written as word equations with regular constraints alone since the finite-state transducers replace *each occurrence* of a substring (e.g. `&#39;`) in a string by another string (e.g. the single character `"`). To the best of our knowledge, there is no known decidable logic which can express the above string constraint.

Contribution: We study the decidability of satisfiability over string logics with concatenations, finite-state transductions, and regex matching as atomic operations. Naively combining concatenations and transducers easily leads to an undecidable logic (e.g. see [10, 16]). In fact, it was shown in [10] that restricting to string constraints of the form $x = y \cdot z \wedge x = R(z)$, where R ranges over finite-state transducers, is undecidable. [Actually, R can be restricted to a relatively weak class of finite-state transducers that express only *regular relations* (a.k.a. *automatic relations* [18]).] Our main contribution is to show that the "straight-line fragment" of the logic is decidable (in fact, is EXPSPACE-complete, but under a certain reasonable assumption the complexity reduces to PSPACE). In fact, our decidability results provide an upper bound for the maximum size of solutions that need to be explored to guarantee completeness for bounded-length string solvers whenever the input constraint falls within our straight-line fragment. The fragment can express the program logics of straight-line string-manipulating programs with concatenations and transductions as atomic operations. This includes the program logic of Example 1. In fact, straight-line programs naturally arise when performing *bounded model checking*[2] or *dynamic symbolic executions*, which unrolls loops in the programs (up to a given depth) and converts the resulting programs into *static single assignment form* (i.e. each variable defined once). Please consult [26, 42, 65] for more details.

Example 1 is one example of analysis of mutation XSS vulnerabilities that can be expressed in our logic. In this paper, we provide three other mutation XSS examples that can be expressed in our logic (adapted from [32, 37, 62]).

Finally, the case study of [57] suggested that the use of length constraints (comparing lengths of different strings) and IndexOf constraints (IndexOf(x, y) outputs the position of an occurrence of a given string x in another string y) is prevalent in JavaScript programs. To this end, we show that our logic is still decidable (with the same complexity) when extended with:

1. *(linear) arithmetic constraints* whose free variables are interpreted as one of the following: integer variables in the program, length of a string variable, or the number of occurrences of a certain letter in a string variable.
2. IndexOf *constraints* of the form $h = \mathsf{IndexOf}(w, y)$, where w is a constant string and h is an integer variable. This is the most frequent usage of IndexOf operator in JavaScript.

All the examples in the benchmark examples of [57] were observed to belong to a class called *solved forms* [30], which is a subset of our string logic with linear arithmetic constraints. Lastly, we can also add unrestricted disequality constraints between string variables, while still preserving decidability.

Organisation: Section 2 fixes general mathematical notations and reviews the necessary concepts on automata and transducers that will be used throughout the paper. In Section 3, we define a

[1] Attack patterns are identified from previous vulnerabilities, some of which have been well-documented, e.g., see [2, 6].

[2] Note that this does *not* mean restrictions to strings of bounded length.

general string constraint language that combines concatenations, finite-state transductions, and regex matching. The language is undecidable even after imposing various restrictions that have been proposed in the literature. In Section 4, we recover decidability for the straight-line fragment. In Section 5, we show that decidability can be retained even when length and $\mathsf{IndexOf}$ constraints are incorporated. We conclude with the related work and possible future works in Section 6. Additional material can be found in the full version [44].

2. Preliminaries

General notations: Given two integers i, j, we write $[i, j]$ to denote the set $\{i, \ldots, j\}$ of integers in between i and j. For each integer k, we write $[k]$ to denote $[0, k]$. Given a binary relation $R \subseteq S \times S$ and a set $A \subseteq S$, we use $R(A)$ to denote the set $\{s' \in S \mid (s, s') \in R \text{ for some } s \in A\}$. In other words, $R(A)$ is the post-image of A under R. We use R^{-1} to denote the reverse relation, i.e., $(s, s') \in R^{-1}$ iff $(s', s) \in R$ for each $s, s' \in S$. Notice then that $R^{-1}(A)$ is the pre-image of A under R. The term *DAG* stands for *directed acylic graphs*.

Regular languages: Fix a finite alphabet Σ. Elements in Σ^* are interchangeably called words or strings. For each finite word $w = w_1 \ldots w_n \in \Sigma^*$, we write $w[i, j]$, where $1 \leq i \leq j \leq n$, to denote the segment $w_i \ldots w_j$. We use $w[i]$ to denote $w[i, i] = w_i$. In addition, the symbol $|w|$ denotes the length n of w, while the symbol $|w|_a$ ($a \in \Sigma$) denotes the number of occurrences of a in w.

Recall that a *nondeterministic finite-state automaton* (NFA) is a tuple $\mathcal{A} = (\Sigma, Q, \delta, q_0, F)$, where Q is a finite set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. A *run* of $\mathcal{A}$ on w is a function $\rho : \{0, \ldots, n\} \to Q$ with $\rho(0) = q_0$ that obeys the transition relation δ, i.e., $(\rho(i), w_i, \rho(i+1)) \in \delta$ for each $i \in \{0, \ldots, n-1\}$. We write $\mathcal{A}_{[q,q']}$ to denote $\mathcal{A}$ but the initial state (resp. set of final states) is replaced by q (resp. $\{q'\}$). We may also denote the run ρ by the word $\rho(0) \cdots \rho(n)$ over the alphabet Q. The run ρ is said to be *accepting* if $\rho(n) \in F$, in which case we say that the word w is *accepted* by $\mathcal{A}$. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of words in Σ^* accepted by $\mathcal{A}$. Such a language is said to be *regular*. Recall that regular languages are precisely the ones that can be defined by regular expressions. In the sequel, when the meaning is clear from the context, we will sometimes confuse an NFA or a regular expression with the regular language that it recognises/generates.

Transducers and rational relations: A transducer (short for "finite-state input output transducer") is a two-tape automaton that has two heads for the tapes and one additional finite control; at every step, based on the state and the letters it is reading, the automaton can enter a new state and move some (but not necessarily all) tape heads. Each transducer generates a binary relation over strings called rational relation.

We will now make this definition more precise. A *transducer* over the alphabet Σ is a tuple $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$, where $\Gamma := \Sigma_\epsilon^2$ and $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$, such that $\mathcal{A}$ is syntactically an NFA over Γ. The transducer $\mathcal{A}$ is said to be *synchronised* if $\mathcal{A}$ (viewed as an NFA) does not accept words $w = (a_1, b_1) \cdots (a_n, b_n) \in (\Sigma_\epsilon^2)^*$ such that there exist $i, j \in [n]$ with $i < j$ such that one of the following conditions holds: (1) $(a_i, b_i) \in \Sigma \times \{\epsilon\}$ and $b_j \in \Sigma$, (2) $(a_i, b_i) \in \{\epsilon\} \times \Sigma$ and $a_j \in \Sigma$. Intuitively, as soon as the two heads go out of sync, the head that is lagging behind is no longer allowed to move forward. The relation $R \subseteq (\Sigma^*)^2$ that $\mathcal{A}$ *recognises* consists of all tuples $\bar{w}$ for which there is a run

$$\pi := q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$$

of $\mathcal{A}$ (treated as an NFA) such that $\bar{w} = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$, where the string concatenation operator $\circ$ is extended to tuples over words component-wise (i.e. $(v_1, v_2) \circ (w_1, w_2) = (v_1 w_1, v_2 w_2)$). A relation is said to be *rational* if it is recognised by a transducer. A relation is said to be *synchronised rational* (a.k.a. *regular* or *automatic*; see [10, 16, 17]) if it is recognised by a synchronised transducer. Rational relations satisfy some nice properties (e.g. see [14, 54]): (1) closure under union and concatenation, and (2) the pre/post image of a regular language under a rational relation is regular. The transducers/automata witnessing the above two properties can also be constructed efficiently: taking union can be done in linear-time, while taking concatenation and pre/post image of a regular language can be done in quadratic time (e.g. see [12–14]). Synchronised rational relations are adequate for certain applications (e.g. see [9, 17, 64, 74]), while also satisfying effective closure under intersection and complementation (cf. [17]).

Example 2. The operator `replace-all` replaces all occurrences of subwords matched by a regular expression e by a word in a regular expression e', which in a Vim-like notation can be written as `s/`e`/`e'`/g`. There are various matching strategies that are used by real-world programming languages, e.g., first match, longest match, etc. They can all be encoded as transducers (e.g. see [55]). One particular use of `replace-all` is to replace words that match a regular language L by ϵ (i.e. an *erase* operation). Such usage of `replace-all` can be found in sanitisation of PHP scripts, e.g., see [8, 28, 29, 73, 75]. For example, to thwart XSS attack patterns of the form `<script>`Σ^*`</script>` from a string variable x, one could erase each occurrence of `<` from x (e.g. see [8, 73, 75]).

Let $y =$ `replace-all`$(x, \epsilon/A)$ denote the operation of erasing each occurrence of letters $a \in A$ (e.g. $A = \{$`<`$\}$) from x and assign it to y. The transducer T for this is simple. It has one state q, which is both an initial and final state. It has $|A|$ transitions, i.e., for each $a \in A$ the transducer T has the transition $(q, (a, \epsilon), q)$. □

Computational complexity: In this paper, we study not only decidability but also the *complexity* of string logics. Pinpointing the precise complexity of verification problems is not only of fundamental importance, but also it often suggests algorithmic techniques that are most suitable for attacking the problem in practice. In this paper, we deal with the following computational complexity classes (see [61] for more details): P (problems solvable in polynomial-time), PSPACE (problems solvable in polynomial space and exponential time), and $\mathsf{EXPSPACE}$ (problems solvable in exponential space and double exponential time). Verification problems that have complexity PSPACE or beyond — see [39, 60] for a few examples — have substantially benefited from techniques like symbolic model checking [47]. As we shall see later, our complexity upper bound also suggests the maximum lengths of words that need to be explored to *guarantee completeness*.

3. The Core Constraint Language

We start by defining a general string constraint language that supports concatenations, finite-state transducers, and regex matching. The language is a natural generalisation of three decidable string constraint languages: word equations, finite-state transducers, and regex matching. The generality of the language, however, quickly makes it undecidable. To delineate the border of undecidability, we shall show that the undecidability already holds for various restrictions that have been proposed in the literature (e.g. restricting finite-state transducers to `replace-all`). We will recover decidability in the next section.

3.1 Language Definition

We assume a finite alphabet Σ and countably many *string variables* $x, y, z, \ldots$ ranging over Σ^*. We start by defining *relational constraints*.

Definition 1 (Relational constraints)**.** *An* atomic relational constraint *over Σ is an expression φ defined by the following grammar:*

$$\varphi \ ::= \ y = x_1 \circ \cdots \circ x_n (n \in \mathbb{N}) \mid y = w \mid R(x, y)$$

where y, x_i are string variables, $w \in \Sigma^*$ is a constant word, and R is a rational relation over Σ given as a transducer. Here, $\circ$ is used to denote the string concatenation operator, which we shall often omit (or simply replace by $\cdot$) to avoid notational clutter. A relational constraint *is a conjunction of atomic relational constraints.*

In other words, an atomic relational constraint allows us to test equality of a string variable y with either a concatenation of string variables or a constant string, or whether the transducer R can transform x into y. Notice that the atomic constraint $y = x_1 \circ \cdots \circ x_n$ cannot be defined as a rational relation $R(x, y)$ (or in fact any binary relation) when $n > 1$. We now define regular constraints (i.e. regex matching constraints), which check whether a word belongs to a boolean combination of regular languages.

Definition 2 (Regular constraints)**.** *An* atomic regular constraint *over Σ is an expression of the form $P(x)$, for P a regular language over Σ given as an NFA and x a variable. A* regular constraint *over Σ is a boolean combination of atomic regular constraints over Σ defined by the following grammar:*

$$\varphi \ ::= \ P(x) \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$$

We finally define the class of string constraints by taking the conjunction of relational and regular constraints.

Definition 3 (String constraints)**.** *A* string constraint *over finite alphabet Σ is a conjunction of a relational constraint and a regular constraint over Σ.*

String constraints allow us to express general *word equations* (i.e., $x_1 \ldots x_n = y_1 \ldots y_n$ for not necessarily distinct variables) by a conjunction of $y = x_1 \ldots x_n$ and $y = y_1 \ldots y_n$. In addition, when the word equation asserts that one of the x_i's or y_j's is the constant string $w \in \Sigma^*$, we simply add a regular constraint that enforces the variable to belong to the language $\{w\}$.

An *assignment* for a string constraint φ over Σ is simply a mapping ι from the set of string variables mentioned in φ to Σ^*. It *satisfies* φ if the constraint φ becomes true under the substitution of each variable x by $\iota(x)$. We formalise this for atomic relational and regular constraints (boolean connectives are standard):

1. ι satisfies the relational constraint $y = x_1 \cdots x_n$, for string variables $y, x_1, \ldots, x_n$, if and only if $\iota(y) = \iota(x_1) \cdots \iota(x_n)$.
2. ι satisfies the relational constraint $y = w$, for string variable y and word $w \in \Sigma^*$, if and only if $\iota(y) = w$.
3. ι satisfies the relational constraint $R(x, y)$, for a rational relation R, if and only if the pair $(\iota(x), \iota(y))$ belongs to R.
4. ι satisfies the atomic regular constraint $P(x)$, for P a regular language, if and only if $\iota(x) \in P$.

A satisfying assignment for φ is also called a *solution* for φ. If φ has a solution, then it is said to be *satisfiable*.

Example 3. In Introduction, we have expressed the program logic of the script in Example 1 and the attack pattern as a conjunction of four atomic relational constraints and one regular constraint. □

3.2 The Satisfiability Problem

The definition of the problem is given as follows.

PROBLEM :	SATISFIABILITY.
INPUT :	A string constraint φ over Σ.
QUESTION :	Is φ satisfiable?

The generality of the constraint language makes it undecidable, even in very simple cases.

Proposition 1. SATISFIABILITY *is undecidable.*

This is because checking satisfiability of constraints of the form $R(x, x)$ is already undecidable by a simple reduction from the *Post Correspondence Problem* (PCP), e.g., see [49, Proof of Proposition 2.14]. For this reason, an *acyclicity* constraint is often imposed (e.g. see [10, 12, 13]) to obtain decidability for formulas that are conjunctions of constraints of the form $P(x)$ or $R(x, y)$, where P is a regular language and R is a rational constraint. This condition is defined as follows. Let φ be a formula of the form above and $\mathbb{G}(\varphi)$ the undirected graph whose nodes are the variables in φ and there is an edge $\{x, y\}$ if $R(x, y)$ is a constraint in φ. Further, let AC be the class of those formulas φ such that $\mathbb{G}(\varphi)$ is acyclic. Then:

Proposition 2 ([10])**.** *Checking satisfiability of formulas in* AC *is* PSPACE*-complete. In fact, if a formula $\varphi \in \mathsf{AC}$ is satisfiable, then it has a solution of size at most exponential in $|\varphi|$.*

The authors of [10] refer to this problem as *the generalised intersection problem with acyclic queries* (see Theorem 6.7 in [10]). [Decidability (in fact, in exponential time) of such a restriction already follows from the classic result by Nivat in the study of rational relations (e.g. see the textbook [14]) that the pre/post images of regular languages under a rational transducer is effectively regular and the complexity analysis in [12, 13].]

Unfortunately, the positive result in Proposition 2 cannot be easily extended to our constraint language, as taking the conjunction of a formula in AC (in particular, of a single rational constraint $R(x, y)$) with the very simple word equation $x = y$ turns the satisfiability problem undecidable. This is because the undecidable constraint $R(x, x)$ can be expressed as: $x = y \wedge R(x, y)$. Restricting R to synchronised rational relations does not help either: satisfiability of string constraints of the form $x = yz \wedge R(x, z)$, where R is a synchronised rational relation, is undecidable [10].

Another option is to restrict the use of finite-state transducers to the `replace-all` operators. As we have argued in Example 2 this might be sufficient to model some sanitisation functions that arise in practice. In fact, some string constraint languages that have been proposed in the literature (e.g. see [65, 75]) permit the use of the `replace-all` operator, but not finite-state transductions in general. It turns out that this restriction is still undecidable even for the very restricted use of `replace-all` of the form `replace-all`$(x, \epsilon/A)$ (defined in Example 2) which erase all occurrences of characters $a \in A$ in x. The proof (see full version) is via a simple but tedious reduction from PCP.

Proposition 3. *Checking satisfiability of a constraint of the form $x = yz \wedge \varphi$, where φ is a formula in* AC *that only mentions transducers $R(x, y)$ of the form* `replace-all`*$(x, \epsilon/A)$, is undecidable.*

4. The Straight-Line Fragment

In Section 3, we have explored various syntactic restrictions of the core constraint language that still permit both concatenation and transducers, and saw that undecidability still held. In this section, we will show that the "straight-line fragment" of the language is decidable (in fact, solvable in exponential space). This straight-line fragment captures the structure of straight-line manipulating programs with concatenations and finite-state transductions as atomic string operations. Note that straight-line programs naturally arise when verifying string-manipulating programs using bounded model checking and dynamic symbolic executions (e.g. see [26, 42, 65]), which unrolls loops in the programs (up to a given depth) and converts the resulting programs into *static single assign-*

ment form (i.e. each variable defined only once). For applications to detecting mutation XSS [32], we will see that the formula for analysing mutation XSS from Example 1 in Introduction can be expressed in the straight-line fragment. We will also see other such examples in this section.

Convention. *In the sequel, we will treat an atomic relational constraint of the form $y = w$, for a word $w \in \Sigma^*$, as an atomic regular constraint, i.e., simply treat w as regular expression and assert $y \in L(w)$.*

To define the straight-line fragment of the core constraint language, we first write $R(x, y)$ as $y = R(x)$. This notation is quite natural since R can be viewed as a string transformation from the input x to the output y. [However, a word of caution is necessary: it is important to remember that R is a relation that need not be a function in general.] We also say that a variable x is *a source variable* in the relational constraint φ if there is no conjunct in φ of the form $x = x_1 \cdots x_n$ or $x = R(y)$ for some transducer R.

Definition 4 (Straight-line constraints)**.** *A relational constraint φ is said to be* straight-line *if it can be rewritten (by reordering the conjuncts) as a relational constraint of the form $\bigwedge_{i=1}^{m} x_i = P_i$ such that:*

(SL1) $x_1, \ldots, x_m$ *are different variables*

(SL2) *Each P_i uses only source variables in φ or variables from $\{x_1, \ldots, x_{i-1}\}$.*

We say that a string constraint is straight-line if it is a conjunction of a straight-line relational constraint with a regular constraint. Let SL *be the set of all straight-line string constraints.*

An example of a straight-line string constraint is $y = R(x) \wedge z = yyz'$. Another example is the constraint from Example 1. Straight-line restrictions also rule out all the undecidable constraints from Section 3, e.g., formulas of the form $x = y \wedge R(x, y)$ and $x = yz \wedge R(z, x)$.

The literal definition of straight-line constraints does not give an efficient algorithm for checking whether a constraint is straight-line. This, however, can be done efficiently.

Proposition 4. *There is a linear-time algorithm for checking whether a relational constraint φ can be made a straight-line constraint (by reordering equations) and, if so, outputs the reordered constraint $\bigwedge_{i=1}^{m} x_i = P_i$ satisfying **(SL1)** and **(SL2)**.*

The proof of this proposition is standard. Straight-line relational constraints can be visualised by drawing a "dependency graph" of the variables in the constraints. Formally, given a relational constraint φ, the *dependency graph* $\mathcal{G}(\varphi)$ of φ is the *directed* graph whose nodes are the string variables appearing in φ and there is an edge from variable x to y iff (a) φ contains a conjunct of the form $R(x, y)$, for a rational relation R, or (b) an equation of the form $y = x_1 \ldots x_n$ for some string variables $x_1 \ldots x_n$ which include x. It is easy to see that straight-line constraints have acyclic dependency graphs. In fact, the converse is also true provided that the relational constraint φ is *uniquely definitional* (i.e. there are no two conjuncts $x = P$ and $x = P'$ in φ with the same left-hand side variable). A linear-time algorithm for Proposition 4, then, first checks if the given constraint φ is uniquely definitional by sequentially going through each conjunct while maintaining m bits in memory (one for each variable in the left-hand side of an equation). Once unique definitionality has been checked, the algorithm checks whether the directed graph $\mathcal{G}(\varphi)$ is acyclic and, if so, output a topological sort, which is well-known to be solvable in linear-time (e.g. see [22]). The topological sort corresponds to a reordering of the conjuncts in φ so that **(SL1)** and **(SL2)** are both satisfied.

Our main result states that satisfiability for SL is decidable.

Theorem 5. SATISFIABILITY *for the class* SL *is* EXPSPACE-*complete.*

Recall that EXPSPACE problems require double-exponential time algorithms in the worst case. An important corollary of the proof of Theorem 5 is a bounded model property.

Theorem 6. *If a string constraint φ in* SL *is satisfiable, then it has a solution with each word of length at most $2^{2^{p(|\varphi|)}}$, for some polynomial $p(x)$.*

In Theorem 10 and Theorem 11 below, we identify a natural restriction of SL that yields an upper bound for satisfiability — PSPACE (i.e., a single-exponential time) and satisfying models of size at most single-exponential — and seems sufficiently expressive in practice (e.g. it subsumes all our examples).

Remark. *The reader might be wondering whether Theorem 5 and Theorem 6 immediately follow from Proposition 2? This is not the case since* AC *does not permit equalities and concatenations (therefore, expressions of the form $x = y.z$ cannot be expressed). Similarly, the theorems also do not immediately follow from the effective closure of regular languages under (1) pre/post images under rational relations (see discussion below Proposition 2), and (2) concatenation. For example, consider the constraint $x = yy \wedge y \in L(a^* + b^*) \wedge x \in L(ab)$ over the alphabet $\Sigma = \{a, b\}$. This is unsatisfiable. Instead, naively applying the two aforementioned closure, we would deduce that x matches $(a^* + b^*).(a^* + b^*)$, which can be matched by ab, i.e., a false positive. As we shall see later, multiple occurrences of variables in an assignment yield constraints with "dimensions" > 1 (definition below), instances of which include all the mutation XSS examples in the paper.*

4.1 Application to Detecting Mutation XSS

Before proving Theorems 5 and Theorem 6, we will mention that the logic SL is sufficiently powerful for expressing string constraints that arise from analysis of mutation XSS vulnerability in web applications. By Theorem 5, such a vulnerability analysis can be performed automatically. We have seen one such example (i.e. Example 1). We will now provide other examples of mutation XSS attacks that can be expressed within the framework of SL.

Example 4. We have seen that the script in Example 1 contains an XSS bug. As suggested in [37], the corrected version of the script swaps the *order* of the sanitisation functions `escapeString` and `htmlEscape` resulting in the following script:

```
var x = goog.string.escapeString(cat);
var y = goog.string.htmlEscape(x);
catElem.innerHTML = '<button onclick=
    "createCatList(\'' + y + '\')">' + x + '</button>';
```

This script is no longer vulnerable to the attack pattern `e1` provided in Example 1. The program logic of the corrected script can be expressed in SL using the same formula as for Example 1 (see Introduction) except that the transducers R_1 and R_2 are swapped. The algorithm from Theorem 5 will be able to automatically point out that the script is secure against the attack pattern `e1`. □

Example 5. This example is an adaptation of vulnerable code patterns from [32, Listing 1.12] applied to the previous example. After the JavaScript from Example 1 has been corrected in Example 4, suppose now that a programmer wishes to introduce a new "title" HTML element into the HTML document in which the new catalogue category name will be displayed. The following code snippet contains two lines in this HTML file:

```
<h1>New catalogue category:
    <span id="node1">TBA</span></h1>
<div id="node2"></div>
```

The following JavaScript code snippet is a modification of the JavaScript from Example 4 which additionally puts the new catalogue category name in the title (i.e. ID `node1`):

```
var titleElem = document.getElementById("node1");
var catElem = document.getElementById("node2");

var x = goog.string.escapeString(cat);
titleElem.innerHTML = x;
var y = goog.string.htmlEscape(titleElem.innerHTML);
catElem.innerHTML = '<button onclick=
    "createCatList(\'' + y + '\')">' + x + '</button>';
```

This JavaScript now contains a subtle mXSS vulnerability. Consider the value `cat = '&#39;);alert(1);//'`. The value `x` will be the same as `cat` since the metacharacters `'` and `"` do not occur in `cat`. The value of `titleElem.innerHTML` is, however, `');alert(1);//` since an implicit browser transduction occurs upon writing into the DOM via `innerHTML`. HTMLescaping `');alert(1);//` results in the string `&#39;);alert(1);//`, which is the value of `y`. Another implicit browser transduction takes place when assigning a value to `catElem.innerHTML`. The DOM element `catElem` now contains the HTML markup:

```
<button onclick="createCatList('');alert(1);//')">
    ');alert(1);//')</button>
```

Upon clicking the button, the browser executes the attacker's script `alert(1)` after `createCatList` has been invoked. The XSS bug is due to the programmer's wrong assumption that the value of `titleElem.innerHTML` is the same as `x` after the assignment.

Let us now encode the program logic of the above JavaScript as a straight-line string constraint. Let `e1` be the attack pattern from Example 1. As in Example 1, let R_1 and R_2 be transducers implementing `htmlEscape` and `escapeString`, and let R_3 be the transducer implementing the implicit browser transductions upon `innerHTML` assignments. The desired formula can now be written as a conjunction of

- $x = R_2($`cat`$)$
- `titleElem.innerHTML` $= R_3(x)$
- $y = R_1($`titleElem.innerHTML`$)$
- $z = w_1 \cdot y \cdot w_2 \cdot x \cdot w_3$ for some constant strings w_1, w_2, w_3
- `catElem.innerHTML` $= R_3(z)$
- `catElem.innerHTML` matches `e1`.

Observe that this is a straight-line string constraint. Therefore, the algorithm from Theorem 5 will be able to automatically detect a vulnerability against the attack pattern `e1`. □

In the full version, we will provide another example of mXSS bug from adapted from [62] and show how it can be analysed within the framework of SL.

Remark. *At this stage, the reader might be wondering about the security implication when the above formulas φ are satisfiable or unsatisfiable. Unsatisfiability rules out specific vulnerability pattern. In the case of satisfiability, since attack patterns (including* `e1`*) generally* overapproximate *a set of bad strings, a solution to φ might not correspond to an actual attack. There are multiple proposals to address this problem (e.g. see [8, 75]). One method (e.g. see [8]) is to supply each attack pattern (e.g. Σ^**`<script>`*Σ^*) with a set of test cases in the form of actual strings (e.g.* `<script>alert(1);</script>`*). Replacing* `e1` *by a specific test case, φ can be checked again for satisfiability.*

4.2 Proofs of Theorem 5 and Theorem 6

We start by proving the upper bound of Theorem 5. Then we explain how Theorem 6 follows from it. Finally, we sketch the proof of the lower bound of Theorem 5.

4.2.1 Upper Bound of Theorem 5

Let SL_r denote the restriction of SL to formulas which do not involve any concatenation, i.e., constraints of the form $y = y_1 \ldots y_n$. The crux of the algorithm witnessing Theorem 5 is that we transform the input constraint φ in SL into one in the class SL_r. After this, in Step 3 we will apply a classic result in the theory of rational relations to obtain decidability (or a recent result [10] for a better complexity). We now provide the details of our algorithm for the satisfiability problem for SL.

Step 1: simplification of regular constraints Recall that a regular constraint is a boolean combination of constraints of the form $P(x)$, where P is a regular language given as an NFA over a finite alphabet Σ. Given a regular constraint ψ, there exists an equivalent constraint in disjunctive normal form (DNF) of exponential size, where each disjunct θ is a conjunction of literals involving the atoms from ψ of the form $P(x)$ (so θ has size linear in $|\psi|$). From propositional logic, we know that there is a standard enumeration of these disjuncts, i.e., enumerate satisfying assignments of ψ treated as a propositional formula (i.e. each atom $P(x)$ is now a proposition). Such an enumeration runs in polynomial space and exponential time.

Next, each disjunct θ in the aforementioned enumeration can now be converted into a conjunction

$$P_1(x_1) \wedge \cdots \wedge P_n(x_n) \tag{1}$$

of positive literals, where each string variable x_i is constrained only by precisely one atomic regular constraint P_i. This can be done by complementing each NFA that occurs as a negative literal in θ, and by computing a single NFA for the intersection of regular languages by a standard product automata construction in the case when a string variable is constrained by several literals in θ. This construction runs in exponential time.

Let φ be the given constraint in SL, consisting of the conjunction of a relational constraint χ and a regular constraint ψ. In order for φ to be satisfiable, it is sufficient and necessary that $\chi \wedge \theta$ is satisfiable, for some disjunct θ in the enumeration of disjuncts of ψ in DNF. In this step, our algorithm simply guesses such disjunct θ (which is of polynomial size). The remaining steps of the algorithm then check whether $\chi \wedge \theta$ is satisfiable. From our previous remarks, we may assume then that θ is of the form (1), where each string variable x_i is uniquely constrained by a literal $P_i(x_i)$. We also assume, without loss of generality, that each variable y in χ is constrained by some literal $P(y)$ in θ. Otherwise, we simply add to θ a literal which states that y belongs to Σ^*.

Step 2: Removing concatenation For this step, we will transform a given constraint φ in SL into a constraint φ' that uses only atomic string constraints of the form $x = R(y)$, i.e., all string constraints of the form $x = y_1 \ldots y_n$ are removed. Since word equations of the form $w = yy$ cannot in general be expressed as a transducer, our transformation cannot possibly express the same property that φ expresses, i.e., it is impossible that φ and φ' have the same set of satisfying assignments in general. However, as we shall see later, *by introducing extra variables and allowing both conjunctions/disjunctions for our string constraints* it is possible to produce the formula φ' without concatenation operators whose satisfying assignments can be easily transformed into satisfying assignments of φ and vice versa (see Lemma 7 below).

The structure of the straight-line relational constraint φ immediately gives us a topological sort $x_1 \prec \ldots \prec x_m$ on $\mathcal{G}(\varphi)$. We may assume without loss of generality that all the source nodes are at the beginning of the topological sort, i.e., it is not the case that $x_i \prec x_{i+1}$ for some non-source node x_i and some source node x_{i+1} (which might happen if they are incomparable in $\mathcal{G}(\varphi)$ construed as a partial order). Next, for each variable y in φ, we will define a relational constraint $\lambda_{\textbf{rel}}(y)$ involving only rational relations (but with conjunctions and disjunctions), a regular constraint $\lambda_{\textbf{reg}}(y)$, and fresh variables $y(1), \ldots, y(\max_y)$, where $\max_y$ is a positive integer. We will define these by induction on the position of y with respect to the order $\prec$.

Base cases: A source node y with a regular constraint $P(y)$. For this, we set $\lambda_{\textbf{rel}}(y) := \top$ and $\lambda_{\textbf{reg}}(y) := P(y)$. We also set $\max_y = 1$ and define a fresh variable $y(1)$.

First inductive case: A non-source node y with an assignment $y = y_1 \ldots y_n$ and a regular constraint $P(y)$, for some $y_1, \ldots, y_n \prec y$. By induction, we assume that $\max_{y_i} \in \mathbb{Z}_{>0}$ and new string variables $y_i(1), \ldots, y_i(\max_{y_i})$ have all been defined (for each $1 \leq i \leq n$). The main idea behind our construction is to interpret each assignment for y_i ($1 \leq i \leq n$) as the "concatenation" of an assignment for $y_i(1), \ldots, y_i(\max_{y_i})$, respectively.

Formally, let $\max_y = \sum_{j=1}^{n} \max_{y_j}$ and $S = \{1, \ldots, \max_y\}$. We define a *selector function* $\nu : S \to \mathbb{Z}^2_{>0}$ as follows: For each $k \in S$ it is the case that $\nu(k)$ is a pair of numbers (i, l), where i is the smallest number i such that $k \leq \sum_{j=1}^{i} \max_{y_j}$ and $l := k - \sum_{j=1}^{i-1} \max_{y_j}$. Intuitively, if $\nu(k) = (i, l)$, then ν "selects" the variable y_i for the fresh variable $y(k)$, and l further "refines" this selection to $y_i(l)$.

In the set $\lambda_{\textbf{rel}}(y)$ we "define" the fresh variables $y(k)$, for each $k \in S$. We do so by setting $\lambda_{\textbf{rel}}(y)$ to be the conjunction of all formulas of the form $y(k) = y_i(l)$, where $k \in S$ and $(i, l) = \nu(k)$. Notice that we can express each such conjunct as a transducer $R(y_i(l), y(k))$, where R is the identity transducer, i.e., the one that outputs its input without modification.

We now define $\lambda_{\textbf{reg}}(y)$. To this end, we first observe that an accepting run of the automaton P on a word $w = w_1 \cdots w_N$, can be split into N subruns. To make this notion more precise, we define an *N-splitting* σ of an automaton $\mathcal{A}$ with states Q to be a sequence $q_0, \ldots, q_N \in Q$ such that each state q_i ($i \in \{1, \ldots, N\}$) is reachable from the state q_{i-1} in $\mathcal{A}$. Recall that $\mathcal{A}_{[q,q']}$ is the NFA $\mathcal{A}$ whose initial (resp. set of final states) is replaced by q (resp. $\{q'\}$). Then $\lambda_{\textbf{reg}}(y)$ is defined as:

$$\bigvee_{\sigma = q_0, \ldots, q_{\max_y}} P_{[q_0,q_1]}(y(1)) \wedge \cdots \wedge P_{[q_{\max_y - 1}, q_{\max_y}]}(y(\max_y)),$$

where σ ranges over $\max_y$-splittings of P. That is, $\lambda_{\textbf{reg}}(y)$ states that there is some $\max_y$-splitting $q_0, \ldots, q_{\max_y}$ of P such that each $y(k)$ ($1 \leq k \leq \max_y$) is accepted by $P_{[q_{k-1}, q_k]}$. This amounts to y being accepted by P (since y is interpreted as the "concatenation" of $y(1), \ldots, y(\max_y)$, respectively).

Second inductive case: A non-source node y with an assignment $y = R(x)$ and a regular constraint $P(y)$, for some $x \prec y$. By induction, we assume that $\max_x \in \mathbb{Z}_{>0}$ and new string variables $x(1), \ldots, x(\max_x)$ have all been defined. The crux of the construction is to replace $y = R(x)$ by "splitting" it into $\max_x$ different constraints. This is achieved by splitting the transducer R (syntactically seen as an NFA over $\Sigma_\varepsilon \times \Sigma_e$). More precisely, let $\max_y = \max_x$. Then $\lambda_{\textbf{rel}}(y)$ is:

$$\bigvee_{\sigma = p_0, \ldots, p_{\max_y}} \bigwedge_{i=1}^{\max_y} y(i) = R_{[p_{i-1}, p_i]}(x(i)),$$

where σ ranges over all $\max_y$-splittings of R. We may define $\lambda_{\textbf{reg}}(y)$ precisely in the same way as in the first inductive case.

The output formula φ': We have now defined $\lambda_{\textbf{rel}}(y)$ and $\lambda_{\textbf{reg}}(y)$ for each node y in $\mathcal{G}(\varphi)$. The output of our transformation is the formula

$$\varphi' := \bigwedge_{y} (\lambda_{\textbf{rel}}(y) \wedge \lambda_{\textbf{reg}}(y)),$$

where y ranges over all variables in φ.

Notice that φ' is not necessarily a string constraint, as $\lambda_{\textbf{rel}}(y)$ might be a *disjunction* of relational constraints for some y's. The notion of satisfiability extends to this class of formulas in the standard way. In particular, an assignment ι for the variables in $\lambda_{\textbf{rel}}(y)$ satisfies this formula iff it satisfies one of its disjuncts. The formula φ' is satisfiable iff there is an assignment ι for the variables of φ' that satisfies $\lambda_{\textbf{rel}}(y) \wedge \lambda_{\textbf{reg}}(y)$ for each string variable y in φ.

Correctness: The following lemma shows that our transformation is satisfiability preserving; in fact, there is an easy way to obtain satisfying assignments for φ from those of φ' and vice versa.

Lemma 7 (Correctness). *φ is satisfiable iff φ' is satisfiable.*

The proof is done by induction on the position of nodes in the topological sort $\prec$ of $\mathcal{G}(\varphi)$. To this end, given a node y in $\mathcal{G}(\varphi)$, we define φ_y to be the set of conjuncts in φ involving only y and variables that precede y in $\prec$. That is, the conjuncts in φ_y are the ones that only mention variables in the set $Z_y := \{x : x \preceq y\}$. For example, if φ contains the constraint $z = yy$, then this constraint cannot be a conjunct of φ_y (but it is a conjunct of φ_z). Similarly, we define $\varphi'_y := \bigwedge_{x \in Z_y} \lambda_{\textbf{rel}}(x) \wedge \lambda_{\textbf{reg}}(x)$. That is, φ'_y is the set of conjuncts in φ' of the form $\lambda_{\textbf{rel}}(x) \wedge \lambda_{\textbf{reg}}(x)$ involving only variables in the set $Z'_y := \{x(i) : x \in Z_y, 1 \leq i \leq \max_x\}$. We will prove the following technical lemma, which is a stronger formulation of Lemma 7.

Lemma 8. *For each node y in $\mathcal{G}(\varphi)$ and every assignment $\iota : Z_y \to \Sigma^*$ of variables in φ_y, the following are equivalent:*

(i) *ι is a satisfying assignment for the formula φ_y.*
(ii) *There is a satisfying assignment $\iota' : Z'_y \to \Sigma^*$ for φ'_y such that $\iota(x) = \iota'(x(1)) \circ \cdots \circ \iota'(x(\max_x))$ for each $x \preceq y$.*

The proof of Lemma 8 is by induction on the position of y in the topological sort $\prec$ of $\mathcal{G}(\varphi)$. Due to lack of space we relegate this proof to the full version. Since Lemma 8 is a stronger formulation of Lemma 7, the correctness of our construction in Step 2 follows.

Step 3: Solving the final formula After applying the transformation from Step 2, the size of the resulting formula φ' could be exponential in $|\varphi|$ due to repeated applications of constraints of the form $y = y_1 \ldots y_n$, where some variable y_i occurs several times on the right-hand side of the equation. In particular, there are exponentially many conjuncts of the form $\lambda_{\textbf{rel}}(x) \wedge \lambda_{\textbf{reg}}(x)$ in φ', More precisely, the resulting formula φ' is a conjunction of multiple formulas of two types:

- conjunctions of atomic formulas of the form $R(x, y)$, for some transducer R, or a disjunction of several such formulas.
- atomic formulas of the form $P(x)$, for some regular language P, or a disjunction of several such formulas.

So, except for the disjunctions, the formula φ' satisfies the shape of the fragment SL_r of SL. In fact, it is not difficult to remove these disjunctions without too much additional computational overhead. Recall that disjunctions were caused by splitting automata or transducers. Although in this case a conjunct can have exponentially many disjuncts, we may simply nondeterministically guess one of the disjuncts (in effect, guessing one of the splittings of the automata/transducers). Nondeterministic algorithms can be determinised at the cost of quadratically extra space [56]. The resulting formula φ'' is now a conjunction of atomic formulas of the form $R(x, y)$ or $P(x)$. Moreover, it is easy to prove that the undirected graph $\mathbb{G}(\varphi'')$ defined in Section 3 is acyclic:

Lemma 9. $\mathbb{G}(\varphi'')$ *is acyclic. Thus,* $\varphi'' \in \mathsf{AC}$.

Proof. By construction of $\varphi' = \bigwedge_y (\lambda_{\textbf{rel}}(y) \wedge \lambda_{\textbf{reg}}(y))$, for each variable y in φ' there is at most one x such that $\lambda_{\textbf{rel}}(y)$ (and, therefore, φ') contains an atom of the form $y = R(x)$, for R a rational transducer. From this it is clear that $\mathbb{G}(\varphi'')$ is acyclic. $\square$

Decidability in EXPSPACE can now be easily obtained by Proposition 2, i.e., we apply Proposition 2 on the formula φ'' of size at most exponential in $|\varphi|$ and so our resulting algorithm runs in exponential space, as desired.

4.2.2 Proof of Theorem 6

Suppose that $\varphi \in \mathsf{SL}$ is satisfiable. The previous algorithm computes a formula φ'' in AC such that φ is satisfiable iff φ'' is satisfiable (cf. Lemma 7). This implies that φ'' is satisfiable and, so by the bounded model property from Proposition 2, φ'' has a solution of size at most exponential in $|\varphi''|$. Now, the size $|\varphi''|$ is $O(\mathrm{Dim}(\varphi) \times |\varphi|)$, where $\mathrm{Dim}(\varphi)$ is the *dimension* of φ which is defined to be the maximum $\max_x$ over all string variables x in φ. The value $\mathrm{Dim}(\varphi)$ is at most exponential in $|\varphi|$. This implies that φ'' has a solution of size at most $F(|\varphi|) := 2^{(\mathrm{Dim}(\varphi)\times|\varphi|)^{O(1)}} = 2^{2^{|\varphi|^{O(1)}}}$. Then, by Lemma 8, φ has a solution of size $\mathrm{Dim}(\varphi) \times F(|\varphi|) = 2^{(\mathrm{Dim}(\varphi)\times|\varphi|)^{O(1)}} = 2^{2^{|\varphi|^{O(1)}}}$, giving us the desired upper bound on the maximum solution size for φ that we need to explore.

4.2.3 Lower Bound of Theorem 5

In order to prove that checking satisfiability of string constraints in SL is EXPSPACE-hard, we reduce from the acceptance problem for a deterministic Turing machine $\mathcal{M}$ that works in space 2^{cn}, for $c > 1$. That is, we provide a polynomial time reduction that, given an input w to $\mathcal{M}$, it constructs a constraint $\varphi(w)$ in SL such that w is accepted by $\mathcal{M}$ if and only if $\varphi(w)$ is satisfiable. Due to lack of space, the complete proof is relegated to the full version. We only sketch the main ideas below.

The reduction starts by constructing a regular constraint of the form $P_1(x) \wedge \cdots \wedge P_m(x)$, in such a way that a word w satisfies this constraint if and only if it codifies a sequence of configurations of $\mathcal{M}$, the first such configuration corresponds to the initial configuration of $\mathcal{M}$ on input w, and the final such configuration corresponds to a final configuration of $\mathcal{M}$. The we only require to check that each non-initial configuration in w is obtained from its preceding configuration by applying the transition function of $\mathcal{M}$. This is done by adding a set of relational constraints that creates an exponential number of copies of x (with equations of the form $y = xx$), but deletes certain distinguished parts of each such copy (with suitable transducers).

4.3 A PSPACE Restriction

We mention a natural restriction of SL that yields a PSPACE upper bound and seems to be sufficiently expressive in practice. Recall that the *dimension* of a string constraint φ is the maximum $\max_x$ over all string variables x in φ. [Incidentally, this notion is closely related to the notion of dimension from the study of context-free grammars (e.g. see [27]).]

Theorem 10. *For any fixed $k \in \mathbb{N}$, satisfiability for the class of formulas in* SL *of dimension k is* PSPACE*-complete.*

The lower bound follows since our logic can easily encode in dimension one the PSPACE-complete problem of checking emptiness of the intersection of m regular languages given as NFA [41]. [In fact, when $k = 1$, the logic still subsumes SL_r.] An easy corollary of the proof of Theorem 6 is the following improved bound.

Theorem 11. *For any fixed $k \in \mathbb{N}$, if a formula φ of dimension k is satisfiable, then it has a solution with each word of length at most $2^{p(|\varphi|)}$ for some polynomial $p(x)$.*

This bound can be derived by noticing from the proof of Theorem 6 that the maximal solution size of φ that one needs to explore is $F(x) := 2^{(\mathrm{Dim}(\varphi)\times|\varphi|)^{O(1)}}$, which is exponential in $|\varphi|$ if $\mathrm{Dim}(\varphi)$ is a fixed constant. Note that the dimension of the scripts in all our examples is 2 (in fact the dimensions of the examples in the benchmark of [72] are all 1, except for one with dimension 4).

5. Adding Integer and Character Constraints

In this section we extend the language SL from Section 4 with *integer* and *character constraints*, and show that the satisfiability problem remains decidable in EXPSPACE. We also show that this bound continues to hold in the presence of two other important features: the $\mathsf{IndexOf}$ constraints and disequalities between strings.

Our language will use two types of variables, **str** and **int**. The type **str** consists of the string variables we considered in the previous sections. In particular, a constraint in SL only uses variables from **str**. On the other hand, a variable of type **int** (also called an *integer variable*) ranges over the set $\mathbb{N}$ of all natural numbers. The choice of omitting negative integers is only for simplicity, but our results easily extend to the case when **int** includes negative integers. For each $T \in \{\textbf{str}, \textbf{int}\}$, we use $\mathcal{V}(T)$ to denote the set of variables of type T.

We start by defining integer constraints, which allow us to express bounds for linear combinations of lengths or number of occurences of symbols in words.

Definition 5 (Integer constraints)**.** *An* atomic integer constraint *over Σ is an expression of the form*

$$a_1 t_1 + \cdots + a_n t_n \ \leq \ d,$$

where $a_1, \ldots, a_n, d \in \mathbb{Z}$ are constant integers (represented in binary) and each t_i is either (i) an integer variable $u \in \mathcal{V}(\textbf{int})$, (ii) $|x|$ for a string variable $x \in \mathcal{V}(\textbf{str})$, or (iii) $|x|_a$ for $x \in \mathcal{V}(\textbf{str})$ and some constant letter $a \in \Sigma$. Here, $|x|$ (resp. $|x|_a$) denotes the length of x (resp. the number of occurrences of a in x). An integer constraint *over Σ is a Boolean combination af atomic integer constraints over Σ.*

Character constraints, on the other hand, allow us to compare symbols from different strings. They are formally defined below.

Definition 6 (Character constraints)**.** *A* atomic character constraint *over Σ is an expression of the form $x[u] = y[v]$, where: (1) x and y are either a variable in $\mathcal{V}(\textbf{str})$ or a word in Σ^*, and (2) u and v are either integer variables in $\mathcal{V}(\textbf{int})$ or constant positive integers. Here, the interpretation of the symbol $x[u]$ is consistent with our notation from Section 2, i.e., the u-th letter in x. A* character constraint *over Σ is a Boolean combination af atomic character constraints over Σ.*

Next, we define the extension of the class SL with integer and character constraints.

Definition 7 (The class SL^e)**.** *The class SL^e consists of all formulas $\varphi \wedge \theta_{\textbf{int}} \wedge \theta_{\textbf{char}}$ such that (i) φ is a constraint in SL, (ii) $\theta_{\textbf{int}}$ is an integer constraint, and (iii) $\theta_{\textbf{char}}$ is a character constraint.*

Since constraints in SL^e are two-sorted, we have to slightly refine the notion of *assignment*. Formally, an assignment for a constraint φ in SL^e is a mapping ι from each variable $x \in \mathcal{V}(T)$ in φ to an object of type T (i.e. either a string or an integer). We also assume for safety that for each term of the form $x[u]$ in φ it is the case that $\iota(u) \leq |\iota(x)|$ (i.e., $\iota(u)$ is in fact a position in $\iota(x)$). [If this assumption is not met, we can simply define that the assignment does not satisfy the formula φ.] As before, ι *satisfies* φ if the constraint φ becomes true under the substitution of each variable x by $\iota(x)$. We formalise this for atomic integer and character constraints (as Boolean connectives are standard):

1. ι satisfies the atomic integer constraint $\sum_{i=1}^{n} a_i t_i \leq d$ if and only if $\sum_{i=1}^{n} a_i \iota(t_i) \leq d$, where for each $1 \leq i \leq n$ we have that (i) $\iota(t_i) = |\iota(x)|$, if $t = |x|$ for $x \in \mathcal{V}(\mathbf{str})$, and (ii) $\iota(t_i) = |\iota(x)|_a$, if $t = |x|_a$ for $x \in \mathcal{V}(\mathbf{str})$ and $a \in \Sigma$.
2. ι satisfies the atomic character constraint $x[u] = y[v]$ if and only if $\iota(x)[\iota(u)] = \iota(y)[\iota(v)]$, where (i) $\iota(x) = x$ (resp., $\iota(y) = y$), if x (resp., y) is a constant word over Σ, and (ii) $\iota(u) = u$ (resp., $\iota(v) = v$), if u (resp., v) is a positive integer.

The constraint φ is *satisfiable* if there exists a satisfying assignment for it. The *satisfiability problem for* SL^e is the problem of deciding if φ is satisfiable, for a given constraint φ in SL^e.

5.1 The Satisfiability Problem for SL^e

In this section, we will show that our EXPSPACE upper bound for the satisfiability of SL extends to SL^e.

Theorem 12. *The satisfiability problem for the class* SL^e *is solvable in* EXPSPACE*. Furthermore, if* SL^e *has a solution, then it has a solution of size at most* $2^{2^{p(x)}}$*, for some polynomial* $p(x)$*.*

The rest of the section is dedicated to proving the theorem. Let SL^e_r be the extension of SL_r with integer constraints and character constraints. Given a formula $\varphi \in \mathsf{SL}^e$, we first transform φ into a constraint φ' in SL^e_r by using (an extension of) the satisfiability-preserving transformation from Section 4.2. We will then show that the formula φ' has a bounded model property (cf. Lemma 13 below). More precisely, if φ' is satisfiable, then it has a satisfying assignment of size at most exponential in $|\varphi'|$. This immediately provides a decision procedure for checking satisfiability of φ, though a naive algorithm only yields a triple-exponential procedure. We will show, however, that this yields a polynomial-space procedure for checking satisfiability of φ', and hence a single exponential space procedure for checking satisfiability for φ.

5.1.1 Transforming SL^e into SL^e_r

Suppose that $\varphi = \varphi_{\mathbf{str}} \wedge \varphi_{\mathbf{int}} \wedge \varphi_{\mathbf{char}} \wedge \varphi_{\mathbf{reg}} \in \mathsf{SL}^e$, where $\varphi_{\mathbf{str}}$ is a relational constraint in SL, $\varphi_{\mathbf{reg}}$ a regular constraint, $\varphi_{\mathbf{int}}$ an integer constraint, and $\varphi_{\mathbf{char}}$ a character constraint.

We apply the transformations from Step 1 and 2 in Section 4.2 on the formula $\psi := \varphi_{\mathbf{str}} \wedge \varphi_{\mathbf{reg}} \in \mathsf{SL}$, yielding an acyclic string/regular constraint $\psi' = \bigwedge_y \lambda_{\mathbf{rel}}(y) \wedge \lambda_{\mathbf{reg}}(y)$ with no concatenation (but possibly some disjunctions), where each variable x in ψ is replaced by several variables $x(1), \ldots, x(\max_x)$ in ψ'. Recall that Lemma 7 states that ψ is satisfiable iff ψ' is satisfiable. In fact, following the notation in Step 2 of Section 4.2, Lemma 8 shows that for each node y in $\mathcal{G}(\psi)$ and every assignment $\iota : Z_y \to \Sigma^*$ of (string) variables in the constraint ψ_y (associated with the node y in $\mathcal{G}(\psi)$), the following two conditions are equivalent:

- ι is a satisfying assignment for the formula ψ_y.
- There exists a satisfying assignment $\iota' : Z'_y \to \Sigma^*$ for the formula $\psi'_y := \bigwedge_{y \in Z_y} \lambda_{\mathbf{rel}}(y) \wedge \lambda_{\mathbf{reg}}(y)$ such that for each $x \preceq y$:
$$\iota(x) = \iota'(x(1)) \circ \cdots \circ \iota'(x(\max_x)). \qquad (*)$$

Handling the integer constraint is now easy. Because of (*), we simply replace each occurrence of $|y|$ (resp. $|y|_a$, where $a \in \Sigma$) in $\varphi_{\mathbf{int}}$ by $\sum_{i=1}^{\max_y} |y(i)|$ (resp. $\sum_{i=1}^{\max_y} |y(i)|_a$). Let $\psi'_{\mathbf{int}}$ be the resulting formula. [Note that y is *not* a variable in ψ'.] This transformation preserves satisfiability, even in the above stronger sense.

Let us now show how to deal with the character constraint $\varphi_{\mathbf{char}}$. Without loss of generality, we may assume that the term t which occurs on left/right hand side of atomic character constraint is of the form $x[u]$ (for an integer variable u), which denotes the u-th character in x. [If t is a string variable x, we can replace x by $x[u]$, where u is a fresh **int** variable, and add the integer constraint $|x| = 1 \wedge u = 1$. Similarly, if t is of the form $x[c]$, where c is an integer constant, we could simply replace this by $x[u]$, for a fresh **int** variable u, and add the integer constraint $u = c$.] Now the u-th character $x[u]$ in x must fall within precisely one of the word segments $x(1), \ldots, x(\max_x)$. Therefore, we simply make a nondeterministic guess on which segment $x(i)$ the position $x[u]$ belongs to, and replace every occurrence of $x[u]$ by $x(i)[u']$, where u' is a fresh **int** variable, and add an integer constraint of the form $u = u' + \sum_{j=1}^{i-1} |x(j)|$. Observe that the constraint θ that we generate from $\varphi_{\mathbf{char}}$ involves both integer constraint and character constraints. Then, the formula φ is satisfiable iff, for some formula θ obtained from the aforementioned nondeterministic construction, the formula $\psi' \wedge \psi'_{\mathbf{int}} \wedge \theta$ is satisfiable.

Note, however, that ψ' still has some disjunctions and so strictly speaking it is not a formula in SL_r. So, to complete our transformation of φ into SL^e_r, we use Step 3 from Section 4.2 on ψ' to make further nondeterministic guesses to eliminate the disjunctions. Let us call a possible resulting formula ψ''. Therefore, φ is satisfiable iff, for some ψ'' and θ, the formula $\psi'' \wedge \psi'_{\mathbf{int}} \wedge \theta$ is satisfiable.

5.1.2 Bounded Model Property for SL^e_r

We will prove a bounded model property for SL^e_r.

Lemma 13 (Bounded Model)**.** *Given a formula* φ *in* SL^e_r*, if it is satisfiable, then there exists a satisfying assignment of whose strings are of length at most exponential in* $|\varphi|$ *and whose integers are of size (in binary) at most polynomial in* $|\varphi|$*.*

Before proving this lemma, we will first show how this can be used to obtain Theorem 12.

5.1.3 Lemma 13 Implies Theorem 12

As mentioned in Section 5.1.1, the problem of checking whether an SL^e formula φ is satisfiable can be reduced in nondeterministic exponential time to checking whether an SL^e_r formula φ' is satisfiable. Next we construct an algorithm that solves this problem in PSPACE in the size of φ', and, therefore, in EXPSPACE in the size of φ (since φ' might be exponentially bigger than φ). Theorem 12 then follows from the fact that nondeterministic exponential time is contained in EXPSPACE and EXPSPACE computable functions are closed under composition.

In order to do this, the first step is to restate a stronger form of Proposition 2. To this end, we first recall the standard generalisation of the notion of transducers to allow an arbitrary number of tracks (e.g. see [20]). An *m-track (rational) transducer* over the alphabet Σ is a tuple $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$, where $\Gamma := \Sigma^m_\epsilon$ and $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$, such that $\mathcal{A}$ is syntactically an NFA over Γ. In addition, we define the m-ary relation $R \subseteq (\Sigma^*)^m$ that $\mathcal{A}$ *recognises* to consist of all tuples $\bar{w}$ for which there is an accepting run

$$\pi := q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_n} q_n$$

of $\mathcal{A}$ (treated as an NFA) such that $\bar{w} = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_n$, where the string concatenation operator $\circ$ is extended to tuples over words component-wise (i.e. $(v_1, \ldots, v_k) \circ (w_1, \ldots, w_k) = (v_1 w_1, \ldots, v_k w_k)$). An m-ary relation is said to be *rational* if it is recognised by an m-ary transducer. To avoid notational clutter, we shall confuse an m-ary transducer and the m-ary relation that it recognises. In the following proposition, the set of solutions (i.e. satisfying assignments) to a formula φ in SL^e — i.e. mappings from variables x in φ to strings, integers, or characters depending on the type of x — is interpreted as a relation (i.e. a set of tuples) by fixing any ordering to the variables occuring in φ.

Proposition 14 ([10])**.** *There exists an exponential-time algorithm for computing an m-track transducer $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$ for the set of solutions of an input formula $\varphi \in \mathsf{SL}_r$ with variables $x_1, \ldots, x_m$, where each state in Q is of size polynomial in $|\varphi|$. Furthermore, there exists a polynomial space algorithm for:*

1. *Computing q_0.*
2. *Checking whether a string is a state of Q.*
3. *Checking whether a state q belongs to F.*
4. *Checking whether $(q, \bar{a}, q') \in \delta$, for some given states q and q' and a symbol $\bar{a} \in \Gamma$.*

This proposition is a stronger version of Proposition 2, which follows from the proof of Theorem 6.7 in [10].

Obtaining a PSPACE algorithm for checking satisfiability of a given formula φ' in SL_r^e is now almost immediate. Our nondeterministic algorithm guesses an assignment to each character variable in φ, whose size is linear in $|\varphi|$. By virtue of Lemma 13, our algorithm needs to guess an assignment to each integer variable in φ of size polynomial in $|\varphi|$ (i.e. numbers represented in binary). In effect, if $|x|$ (resp. $|x|_a$) appears in the integer constraint of φ, our algorithm also guesses the length of (resp. number of occurrences of a in) the string variable x. Our algorithm now checks that the integer constraints and character constraints in φ are satisfied. Next our algorithm guesses assignments to the string variables in φ. However, since they are of exponential size in the worst case, our algorithm will have to construct the assignment on the fly. By using Proposition 14, we will have to *simultaneously* construct the string assignments to all string variables in φ. To this end, for each string variable x, we keep track of $|\Sigma| + 1$ extra integer counters (counting in binary) $l(x)$ and $l_a(x)$, respectively, for each $a \in \Sigma$. The counter $l(x)$ (resp. l_a) keeps track of the length of (resp. number of occurrences of a in) the partially constructed string assignment for x. Putting this all together, if $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$ is the m-track transducer for the set of solutions for φ (as in Proposition 14), our algorithm first computes q_0 and let $q = q_0$. It then repeats the following step until (i) $q \in F$, and (ii) $l(x) = |x|$ and $l_a(x) = |x|_a$, for each string variable x and letter $a \in \Sigma$:

1. Guess a state q' of Q and a symbol $\bar{a} \in \Gamma = \Sigma_\epsilon^m$ of $\mathcal{A}$, and check that $(q, \bar{a}, q') \in \delta$ in polynomial space.
2. If $\bar{a} = (a_1, \ldots, a_m)$, then set $l(x_i) := l(x_i) + 1$ and $l_a(x_i) := l_a(x_i) + 1$ for each $i \in \{1, \ldots, m\}$ and $a \in \Sigma$ satisfying $a_i = a$.
3. Set $q := q'$

This is a nondeterministic algorithm that uses polynomial space. Nondeterministic algorithms can be determinised at the cost of quadratically extra space [56]. As for the bounded model property part of Theorem 12, it can be derived in precisely the same way as Theorem 6.

5.1.4 Proving Lemma 13

Our proof idea goes as follows. Given a satisfiable formula $\varphi \in \mathsf{SL}_r^e$ with m variables, we suppose that ι is a satisfying assignment of φ. It assigns each character variable $x[u]$ to a certain character $\iota(x[u]) \in \Sigma$. So, we will only have to ensure the existence of a satisfying assignment that assigns each string (resp. integer) variable to a small enough string (resp. integer). To this end, Proposition 14 gives an m-track transducer $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$ that recognises the set of solutions of φ. The number of states in $\mathcal{A}$ is exponential in $|\varphi|$. Next we erase the input tape of $\mathcal{A}$, while equipping it with nonnegative integer-valued counters that *cannot be decremented*. The resulting machine is a *monotonic (Minsky's) counter machine*, whose set of final configurations captures the set of solutions for φ. Monotonic counter machines are restrictions of reversal-bounded counter machine [36] (i.e. counter machines whose counters can switch between non-incrementing and non-decrementing modes only for a fixed $r \in \mathbb{N}$ number of times). [In the case of monotonic counter machines, we have $r = 0$]. Such a computation model is *not* Turing-complete. In fact, their sets of reachable configurations are effectively semi-linear, as was first shown in [36]. We will use a recent result from [40, 63] to analyse the size of the smallest reachable configuration, which will give us the bounded model property of SL_r^e.

We now formalise the notion of monotonic counter machine with $k \in \mathbb{N}$ counters. A *monotonic counter machine* is a tuple $\mathcal{A} = (Q, \delta, q_0, F, P)$, where: (1) Q is a finite set of states, (2) $q_0 \in Q$ is an initial state, (3) $F \subseteq Q$ is a set of final states, (4) $P = \{p_1, \ldots, p_k\}$ is a set of k counters, and (5) $\delta \subseteq (Q \times \mathrm{Cons}_P) \times (Q \times \{0,1\}^k)$ is the transition relation, where Cons_P is the set of counter tests of the form $\bigwedge_{i=1}^k p_i \ \sim_i \ 0$ such that $\sim_i \in \{=, >\}$ for each $1 \leq i \leq k$. A vector $\bar{v} \in \{0,1\}^k$ such that $(q, \psi, q', \bar{v}) \in \delta$, for $q, q' \in Q$ and $\psi \in \mathrm{Cons}_P$, is called an *update vector*. In the sequel we shall also denote this vector by its characteristic set, i.e., the one which consists of all counters $p_i \in P$ such that $\bar{v}[i] = 1$.

A *configuration* of $\mathcal{A}$ is a pair $(q, \bar{v})$, where $q \in Q$ and $\bar{v} \in \mathbb{N}^k$. A *run* π of $\mathcal{A}$ is a sequence of the form

$$(q_0, \bar{v}_0),\ (q_1, \bar{v}_1),\ \ldots,\ (q_n, \bar{v}_n)$$

such that:

- $q_i \in Q$ for each $1 \leq i \leq n$,
- $\bar{v}_0 = (0, \ldots, 0)$ (i.e., counters are initially empty),
- for each $1 \leq i \leq n$ there exists a transition $(q_{i-1}, \psi(\bar{x}), q_i, \bar{c}) \in \delta$ such that $\psi(\bar{v}_{i-1})$ is true, and $\bar{v}_i = \bar{v}_{i-1} + \bar{c}$.

The configuration $(q_n, \bar{v}_n)$ is said to be *accepting* if $q_n \in F$. The set of accepting configurations of $\mathcal{A}$ is denoted by $\mathcal{L}(\mathcal{A})$.

From our transducer $\mathcal{A} = (\Gamma, Q, \delta, q_0, F)$, we construct our monotonic counter machine $\mathcal{B} = (Q, \delta', q_0, F, P)$, where P consists of the following counters:

- $c_{|x|_a}$ for each string variable x in φ and each letter $a \in \Sigma$;
- c_u for each integer variable u in φ;
- $y_{x[u]}$ and $z_{x[u]}$ for each $x[u]$ occuring in a character constraint of φ.

The counter $c_{|x|_a}$ records the number of a's seen so far in in the transducer's track corresponding to the variable x. The counter c_u records the guessed value for the variable u. To avoid notational clutter, we shall confuse $c_{|x|_a}$ (resp. c_u) with $|x|_a$ (resp. u). The counter $y_{x[u]}$ records a guess for the position of u in x, which has to be recorded separately due to the different tracks in $\mathcal{A}$. The value of $z_{x[u]}$ is a boolean variable (i.e., either 0 or 1) that acts as a flag whether the guess for the variable $y_{x[u]}$ is complete.

We now specify the transitions in δ'. In doing so, we will ensure that once a variable of the form $z_{x[u]}$ is set to 1, the value of $y_{x[u]}$ can no longer be incremented. Let W be the set of all character variables $x[u]$ in φ. For each $Y \subseteq W$, let ψ_Y denote the formula of the form $\bigwedge_{x[u] \in Y} z_{x[u]} = 0 \wedge \bigwedge_{x[u] \in W \setminus Y} z_{x[u]} > 0$. Then:

1. If $(q, \bar{a}, q') \in \delta$, where $\bar{a} = (a_1, \ldots, a_m)$, then for each subset $Y \subseteq W$ we add to δ' *each* transition of the form

   $$(q, \psi_Y, q', Z),$$

   where Z consists of: (1) each $|x|_{a_i}$ with $a_i \neq \epsilon$, (2) $y_{x[u]}$ for each $x[u] \in Y$, and (3) if $z_{x_i[u]} = 0$ and $a_i = \iota(x_i[u])$, the set Z may (nondeterministically) contain $z_{x_i[u]}$.
2. $(q, \top, q, \{u\})$ for each integer variable u in φ.

In other words, the first transition above simply simulates a transition of $\mathcal{A}$, while the second transition nondeterministically increments the integer counter u.

Recall that φ is our initial formula in SL^e_r and ι is a satisfying assignment for it. Now, let $\varphi_{\textbf{int}}$ be the conjunct of φ containing the integer constraint. We use $\varphi'_{\textbf{int}}$ to denote a conjunction of (i) the constraint $\varphi_{\textbf{int}}$ but substituting every occurrence of $|x|$ by $\sum_{a\in\Sigma}|x|_a$, (ii) a conjunction of constraints of the form $u = y_{x[u]}$ for each $x[u] \in W$ (i.e. all positions $y_{x[u]}$ equal u), and (iii) a conjunction of constraints of the form $z_{x[u]} = 1$ for each $x[u] \in W$ (i.e. all $y_{x[u]}$ have been completely guessed). The following lemma is immediate from our construction and Proposition 14.

Lemma 15. *Given a configuration $(q, \bar{v})$ of $\mathcal{B}$, the following are equivalent:*

- *$(q, \bar{v}) \in \mathcal{L}(\mathcal{B})$ and $\bar{v}$ satisfies $\varphi'_{\textbf{int}}$.*
- *There exists a satisfying assignment ι' of φ whose integer values agree with $\bar{v}$ and whose character values agree with ι.*

We now use the following proposition, which is a result of [40] (see the proof of [63, Proposition 7.5.5]):

Proposition 16. *Given a monotonic k-counter machine $\mathcal{A}$ with n states, the set of reachable configurations can be represented as a disjunction of existential Presburger formulas, each of size polynomial in $k + \log(n)$ and at most $O(k)$ variables.*

Indeed, in the above proposition the number of disjuncts is polynomial in n, but this is not important for our purpose. It is now easy to obtain a satisfying assignment for φ given our original satisfying assignment ι. By the above proposition, the set of reachable configurations of the monotonic counter machine $\mathcal{B}$ that we constructed is a disjunction of existential Presburger formulas each of size polynomial in $k + \log(n)$ and with $O(k)$ variables, where $k = O(|\varphi|)$ and $n = 2^{|\varphi|^{O(1)}}$. By Lemma 15 and our assumption that φ is satisfiable with assignment ι, it follows that one of these disjuncts ψ is satisfiable. Scarpellini [59, Theorem 6(a)] proved that a satisfiable existential Presburger formula θ with c variables has solutions where each variable is assigned a number that is at most $2^{(|\theta|+c)^{O(1)}}$ (and thus can be represented with at most $(|\theta|+c)^{O(1)}$ bits). Applying Scarpellini's result on ψ now gives us a satisfying assignment of φ which assigns numbers of polynomially many bits to integer variables of φ and lengths of string variables. This completes the proof of Lemma 13.

5.2 Extensions with Disequalities and IndexOf

Finally, we show that two important features can be added to the language while retaining decidability in EXPSPACE: Disequalities between strings and IndexOf constraints.

Disequalities: Assume that constraints in SL^e are now extended with disequalities of the form $x \neq y$, for $x, y \in \mathcal{V}(\textbf{str})$, which state that x and y are interpreted as different strings. The disequality relation is regular, and thus can be expressed as a transducer $y = R(x)$. The problem is that the addition of this transducer may yield a constraint that is no longer uniquely definitional. To solve this, we use integer and character constraints; in fact, the disequality $x \neq y$ is equivalent to $(|x| \neq |y|) \vee (x[u] \neq y[u])$, for a fresh variable $u \in \mathcal{V}(\textbf{int})$. More formally, if φ is a constraint in SL^e and $x \neq y$ is a disequality between string variables, then $\varphi \wedge (x \neq y)$ is satisfiable if and only if either the SL^e constraint $\varphi \wedge (|x| \neq |y|)$ or the SL^e constraint $\varphi \wedge (x[u] \neq y[u])$ is satisfiable. Checking if any of these constraints is satisfiable can be solved in EXPSPACE from Theorem 12. Clearly, adding more disequalities does not increase the computational cost if we use a nondeterministic algorithm that chooses to check either $|x| \neq |y|$ or $x[u] \neq y[u]$ for each disequality $x \neq y$.

Expressing the IndexOf method: One reason we introduced the character constraints is, besides the use of the JavaScript string method `charAt` (which is used rather frequently in JavaScript according to the benchmark [57]), they can also be used to define $\mathsf{IndexOf}(w, x)$ for any word $w \in \Sigma^*$, which is the most standard usage of IndexOf method in practice. We consider both the *first-occurrence* semantics (i.e., for an integer variable u, the constraint $u = \mathsf{IndexOf}(w, x)$ says that u is the first position in x where w occurs), or the *anywhere* semantics (i.e., $u = \mathsf{IndexOf}(w, x)$ says that u is any position in x where w occurs).

Formally, an IndexOf constraint is a conjunction of expressions of the form $u = \mathsf{IndexOf}(w, x)$, where $w \in \Sigma^*$, x is a string variable/constant, and u is either an integer variable or a positive integer. The satisfaction of an expression of this form (under any of the two semantics) with respect to an assignment of the variables is the expected one (following the intuition given in the previous paragraph). The next proposition states that IndexOf constraints do not increase the "expressiveness" of SL^e.

Proposition 17. *Let φ be the conjunction of a constraint in SL^e and the IndexOf constraint $u = \mathsf{IndexOf}(w, x)$. The satisfiability of φ can be checked in* EXPSPACE.

Proof. Let $w = a_1 \ldots a_p \in \Sigma^*$. For the anywhere semantics, every occurrence of $u = \mathsf{IndexOf}(w, x)$ in φ is replaced by the formula

$$x[u_1] = a_1 \wedge \cdots \wedge x[u_p] = a_p$$
$$u = u_1 \wedge u_2 = u_1 + 1 \wedge \cdots \wedge u_p = u_{p-1} + 1,$$

where $u_1, \ldots, u_p$ are fresh integer variables. The resulting formula is an SL^e constraint whose satisfiability can be checked in EXPSPACE from Theorem 12.

The first-occurrence semantics could be handled by replacing $u = \mathsf{IndexOf}(w, x)$ with a relational constraint $x = x_1x_2x_3$ and regular constraints stating that w does not occur in x_1 and $w = x_2$. This would allow us to use the anywhere semantics to express $u = \mathsf{IndexOf}(w, x_2)$, which can be done as before. The problem with this approach is that the introduction of the word equation $x = x_1x_2x_3$ may yield a constraint that is no longer uniquely definitional. To avoid this, we make a nondeterministic guess (as we did for formula $\varphi_{\textbf{char}}$ in Section 5.1.1) as to how x_1 overlaps with $x(1), \ldots, x(\max_x)$, e.g., it might overlap with $x(1)x(2)x(3)$. We will then simply assert that $x(1)x(2)x(3) \in L$, where L is the regular language of words v that contains no w as a (contiguous) subword. We then have to apply the splitting technique for the regular constraint just as in Step of Section 4 to express $x(1)x(2)x(3) \in L$ as $\bigwedge_{i=1}^{3} x(i) \in L_i$. This can all be done by nondeterministic guesses while incurring only a polynomial blowup. □

6. Related Work and Future Work

In this section, we mention a few related works and discuss their connections with our work in more detail. Roughly speaking, they can be classified into three categories: (1) decidability results, (2) heuristics and string solver implementations, (3) benchmarking examples. We shall also mention a few possible research avenues in passing.

Decidability results: In §Introduction, we have mentioned the results of Makanin's and Plandowski's [45, 50–52] on the decidability and complexity of satisfiability for word equations (a conjunction of equations of the form $v = w$, where v and w are a concatenation of string constants and variables). We should also remark that the decidability (with the same PSPACE complexity) extends to quantifier-free first-order theory of strings with concatenations and regular constraints [19]. Since extending word equations with finite-state transducers yields undecidability (see Section 3), the straight-line fragment SL of our core string constraint language is incomparable to word equations with regular constraints (neither subsumes the other). The fragment SL is, in a sense, more complex since its computational complexity is EXPSPACE, though for constraints of small dimensions the complexity reduces to PSPACE (cf. Theorem 10). In addition, it is still a long-standing open problem whether word equations with length constraints is decidable, though it is known that letter-counting (i.e. counting the number of occurrences of 0s and the number of occurrences of 1s separately) yields undecidability [19]. On the other hand, the extension SL^e of our straight-line fragment SL is decidable (with the same complexity) and yet admits general letter-counting.

In our decidability proof of Theorem 5, we have also used the result of Barcelo *et al.* [10] (see Proposition 2) that acyclic conjunctions of rational relation constraints (with regular constraints) is decidable in PSPACE. Their logic AC, however, supports neither string concatenations nor letter-counting constraints. In fact, it is easy to show by a standard pumping lemma argument that the constraint $x = y \cdot y$ cannot be expressed in AC. For this reason, our logics SL and SL^e are not subsumed in AC.

Abdulla *et al.* [7] studied acyclic constraints over systems of word equations with a length predicate (without transducers) and disequality constraints, for which they showed decidability. Our decidable logics are incomparable to their logic. On the one hand, SL^e supports finite-state transducers, letter-counting, and `IndexOf` constraints, which are not supported by their logic. Our logic also supports unrestricted disequality constraints, whereas their logic supports only restricted (acyclic) disequalities. On the other hand, the string logic of [7] supports string equations of the form $x \cdot y = z \cdot z'$ (i.e. both sides of the equations contain different variables), which they showed could be reduced to a boolean combination of regular constraints. Using this reduction, we can incorporate this feature into SL^e, yielding a more expressive decidable string logic.

Heuristics and string solver implementation: In §Introduction, we have mentioned the large amount of works in the past seven years or so towards developing practical string solvers (e.g. [7, 8, 16, 21, 23, 28–31, 34, 35, 38, 43, 48, 53, 55, 57, 65–69, 71–76]). We are not aware of existing string solvers that support both concatenations and finite-state transductions. However, string solvers that support concatenations and the `replace-all` operator (i.e. a subset of finite-state transductions) are available, e.g., [16, 65, 75].

Since the focus of our work is on the fundamental issue of decidability, we consider our work to be complementary to these works. In fact, our results may be construed as providing some *completeness guarantee* for existing string solvers. Practical string solvers do not implement Makanin's or Plandowski's algorithms, but instead rely on certain heuristics (e.g. bounding the maximum length k of solutions [16, 38, 57]). For this reason, none of the above solvers have a completeness guarantee for the entire class of word equations. However, when the input string constraint φ falls within the logic SL^e, the bounded model properties of SL and SL^e (e.g. Theorem 6 and Theorem 12) imply that string solvers need only look for solutions of size at most $2^{2^{p(|\varphi|)}}$ for some polynomial $p(x)$. Double exponential size is of course only an extremely crude estimate, so in practice one could devise an algorithm for computing a better estimate $f(\varphi) \leq 2^{2^{p(|\varphi|)}}$ by looking at the structure of the formula φ. A rough estimate of $f(\varphi)$ could, for example, be obtained by first computing the dimension of φ (which could be computed quickly); as we have seen in Theorem 11, when the dimension is *small* the double exponential bound actually reduces to exponential size.

Future Work 1. *Give a better algorithm for computing a better estimate $f(\varphi)$ of the maximum size of the solutions for straight-line formulas φ that need to be explored.*

Veanes *et al.* (e.g. see [23, 33, 35, 66]) have observed that, in practice, the number of transitions of finite-state transducers for encoding web sanitisation functions could become large fairly quickly (due to large alphabet size, e.g., `utf-8`). For this reason, they introduce extensions of finite-state transducers that allow succinct representations by allowing transitions to take an arbitrary formula in a decidable logical theory, while taking advantage of state-of-the-art SMT solvers for the theory. As a simple example, consider the finite-state transducer that converts a sequence of digits (over the alphabet $\Sigma = \{\mathtt{0}, \ldots, \mathtt{9}\}$) to its HTML character numbers (over the alphabet $\Sigma' = \{\mathtt{\&}, \mathtt{;}, \mathtt{\#}, \mathtt{0}, \mathtt{9}\}$). The general formula for this is that a digit `i` is converted to `&#(48+i);`. Using the standard finite-state transducers, we would require about $2 + 10 \times 3 = 32$ states, whereas representing using symbolic finite-state transducers only ~ 4 states and ~ 4 transitions are required. There are real-world examples where this compression would be enormous (e.g. see the encoding of `HTMLdecode` in [66] as a symbolic transducer). For this reason, for future work, it would make sense to consider an extension of our work that uses symbolic finite-state transducers (or extensions thereof) both from practical and theoretical viewpoints.

Future Work 2. *Study the extension of* SL *and* SL^e *with symbolic (finite-state) transducers.*

In order to be able to apply string solvers to analyse injection and XSS vulnerabilities, it is paramount to develop realistic browser models, which would model implicit browser transductions. Preliminary works in this direction are available (e.g. see [58, 70]). We believe that an interesting (but perhaps extremely challenging) line of future work is to develop a formal and precise browser transductions and their transductions (e.g. for a particular version of Firefox) as finite-state transducers (or extensions thereof). Although some such transducers are already available (e.g. see [1, 23, 35, 66]), much work remains to be done to develop full-fledged browser models that can capture all the subtlety of browser behaviors (e.g. those that can be found in [6, 32]).

Benchmarking examples: In this paper, we have provided four examples of analysis of mutation-based XSS vulnerabilities that can be expressed in SL. A few other interesting XSS vulnerability examples from [32, 62] and [6] can actually be expressed in our logic, though the vulnerabilities only exist in older browsers (e.g. IE8). We also note that the five benchmarking examples of PHP programs from [72] that exhibit SQL injection and XSS vulnerabilities (cf. `http://www.cs.ucsb.edu/~vlab/stranger/`) can also be expressed in SL. As we have argued in §Introduction, the benchmarking examples from Kaluza [57] are in *solved forms*, and therefore expressible in SL^e. To the best of our knowledge, the benchmarking examples from [72] and [57] do not contain mutation XSS test cases.

Acknowledgments

We thank Leonid Libkin and anonymous referees for their helpful feedback. We also thank Lukas Holik and Joxan Jaffar for the fruitful discussion. Lin was supported by Yale-NUS College through the MoE Tier-1 grants R–607–265–056–121 and IG15–LR001. Barceló is funded by the Millenium Nucleus Center for Semantic Web Research under grant NC120004.

References

[1] BEK website (referred in Nov 2015). http://research.microsoft.com/en-us/projects/bek/.

[2] OWASP XSS cheat sheet (referred in Nov 2015). https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet.

[3] SAT competition (referred in Nov 2015). http://www.satcompetition.org/.

[4] SMT competition (referred in Nov 2015). http://www.smtcomp.org/.

[5] Google Closure Library (referred in Nov 2015). https://developers.google.com/closure/library/.

[6] HTML5 Security cheat sheet (referred in Nov 2015). http://html5sec.org/.

[7] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV*, pages 150–166, 2014.

[8] 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 *S&P*, pages 387–401, 2008.

[9] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31, 2012.

[10] P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations. *Logical Methods in Computer Science*, 9(3), 2013. .

[11] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In Biere et al. [15], pages 825–885. .

[12] W. Bekker and V. Goranko. Symbolic model checking of tense logics on rational Kripke models. In *Infinity in Logic and Computation, International Conference, ILC 2007, Cape Town, South Africa, November 3-5, 2007, Revised Selected Papers*, pages 2–20, 2007. .

[13] W. Bekker and V. Goranko. Symbolic model checking of tense logics on rational Kripke models. *CoRR*, abs/0810.5516, 2008.

[14] J. Berstel. *Transductions and Context-Free Languages*. Teubner-Verlag, 1979.

[15] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 2009. IOS Press.

[16] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.

[17] A. Blumensath and E. Grädel. Automatic structures. In *LICS*, pages 51–62, 2000. .

[18] A. Blumensath and E. Grädel. Finite Presentations of Infinite Structures: Automata and Interpretations. *Theory Comput. Syst.*, 37(6):641–674, 2004.

[19] J. R. Büchi and S. Senger. Definability in the existential theory of concatenation and undecidable extensions of this theory. In *The Collected Works of J. Richard Büchi*, pages 671–683. Springer, 1990.

[20] O. Carton, C. Choffrut, and S. Grigorieff. Decision problems among the main subfamilies of rational relations. *ITA*, 40(2):255–275, 2006.

[21] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[23] L. D'Antoni and M. Veanes. Static analysis of string encoders and decoders. In *VMCAI*, pages 209–228, 2013.

[24] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[25] V. Diekert. Makanin's Algorithm. In M. Lothaire, editor, *Algebraic Combinatorics on Words*, volume 90 of *Encyclopedia of Mathematics and its Applications*, chapter 12, pages 387–442. Cambridge University Press, 2002.

[26] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[27] J. Esparza, P. Ganty, S. Kiefer, and M. Luttenberger. Parikh's theorem: A simple and direct automaton construction. *Inf. Process. Lett.*, 111 (12):614–619, 2011.

[28] X. Fu and C. Li. Modeling regular replacement for string constraint solving. In *NFM*, pages 67–76, 2010.

[29] X. Fu, M. C. Powell, M. Bantegui, and C. Li. Simple linear string constraints. *Formal Asp. Comput.*, 25(6):847–891, 2013.

[30] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. Word equations with length constraints: whatâĂŹs decidable? In *Hardware and Software: Verification and Testing*, pages 209–226. Springer, 2013.

[31] C. Gould, Z. Su, and P. T. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654, 2004.

[32] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mxss attacks: attacking well-secured web-applications by using innerhtml mutations. In *CCS*, pages 777–788, 2013.

[33] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *VMCAI*, pages 248–262, 2011.

[34] P. Hooimeijer and W. Weimer. StrSolve: solving string constraints lazily. *Autom. Softw. Eng.*, 19(4):531–559, 2012.

[35] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *USENIX Security Symposium*, 2011. URL http://static.usenix.org/events/sec11/tech/full_papers/Hooimeijer.pdf.

[36] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *J. ACM*, 25(1):116–133, 1978.

[37] C. Kern. Securing the tangled web. *Commun. ACM*, 57(9):38–47, Sept. 2014.

[38] A. Kiezun et al. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4):25, 2012.

[39] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(04):571–586, 2002.

[40] E. Kopczynski and A. W. To. Parikh images of grammars: Complexity and applications. In *LICS*, 2010.

[41] D. Kozen. Lower bounds for natural proof systems. In *FOCS*, pages 254–266, 1977.

[42] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[43] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662, 2014.

[44] A. W. Lin and P. Barceló. String Solving with Word Equations and Transducers: Towards a Logic for Analysing Mutation XSS (Full Version). http://arxiv.org/abs/1511.01633 (cited in 2015).

[45] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Sbornik: Mathematics*, 32(2):129–198, 1977.

[46] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM*, 52(8):76–82, 2009.

[47] K. L. McMillan. *Symbolic model checking*. Kluwer, 1993.

[48] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW*, pages 432–441, 2005.

[49] C. Morvan. On rational graphs. In *FoSSaCS*, pages 252–266, 2000.

[50] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. In *FOCS*, pages 495–500, 1999.

[51] W. Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.

[52] W. Plandowski. An efficient algorithm for solving word equations. In *STOC*, pages 467–476, 2006.

[53] G. Redelinghuys, W. Visser, and J. Geldenhuys. Symbolic execution of programs with strings. In *SAICSIT*, pages 139–148, 2012.

[54] J. Sakarovitch. *Elements of automata theory*. Cambridge University Press, 2009.

[55] Y. Sakuma, Y. Minamide, and A. Voronkov. Translating regular expression matching into transducers. *J. Applied Logic*, 10(1):32–51, 2012.

[56] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970.

[57] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *S&P*, pages 513–528, 2010.

[58] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, pages 601–614, 2011.

[59] B. Scarpellini. Complexity of subcases of presburger arithmetic. *Trans. of AMS*, 284(1):203–218, 1984.

[60] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technischen Universität München, 2002.

[61] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[62] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise client-side protection against dom-based cross-site scripting. In *USENIX Security*, pages 655–670, 2014.

[63] A. W. To. *Model Checking Infinite-State Systems: Generic and Specific Approaches*. PhD thesis, LFCS, School of Informatics, University of Edinburgh, 2010.

[64] A. W. To and L. Libkin. Algorithmic metatheorems for decidable LTL model checking over infinite systems. In *FOSSACS*, 2010.

[65] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, pages 1232–1243, 2014.

[66] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. Symbolic finite state transducers: algorithms and applications. In *POPL*, pages 137–150, 2012.

[67] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.

[68] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[69] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.

[70] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, E. C. R. Shin, and D. Song. A systematic analysis of XSS sanitization in web application frameworks. In *ESORICS*, pages 150–171, 2011.

[71] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *TACAS*, pages 322–336, 2009.

[72] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, pages 154–157, 2010. Benchmark can be found at `http://www.cs.ucsb.edu/~vlab/stranger/`.

[73] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *ICSE*, pages 251–260, 2011.

[74] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. *Int. J. Found. Comput. Sci.*, 22(8):1909–1924, 2011.

[75] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[76] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In *ESEC/SIGSOFT FSE*, pages 114–124, 2013.