Solving Not-Substring Constraint with Flat Abstraction *

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Yu-Fang Chen[2], Bui Phi Diep[2], Lukáš Holík[3], Denghang Hu[4], Wei-Lun Tsai[2], Zhillin Wu[4], and Di-De Yen[2]

[1] Uppsala University
{parosh,mohamed_faouzi.atig,bui.phi-diep}@it.uu.se
[2] Academia Sinica
yfc@iis.sinica.edu.tw,alan23273850@gmail.com,bottlebottle13@gmail.com
[3] Brno University of Technology
holik@fit.vutbr.cz
[4] State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences
{hudh,wuzl}@ios.ac.cn

Abstract. Not-substring is currently among the least supported types of string constraints, and existing solvers use only relatively crude heuristics. Yet, not-substring occurs relatively often in practical examples and is useful in encoding other types of constraints. In this paper, we propose a systematic way to solve not-substring using based on flat abstraction.In this framework, the domain of string variables is restricted to flat languages and subsequently the whole constraints can be expressed as linear arithmetic formulae. We show that non-substring constraints can be flattened efficiently, and provide experimental evidence that the proposed solution for not-substring is competitive with the state of the art string solvers.

Keywords: String Constraints · Not-substring relation · Flat abstraction · Formal Verification.

1 Introduction

Due to the fast growth of web applications, string data type plays an increasingly important role in computer software. Many software security vulnerabilities, such as cross-site scripting and injection attack, are caused by careless treatment of strings in programs [1], which jeopardize the end-users' trust in digital technology. There is therefore a crucial need of rigorous engineering techniques to ensure the correctness of string manipulating programs. Such techniques (e.g. (bounded) model checking [10, 15, 21], symbolic execution techniques [11, 19],

* This work has been supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the FIT BUT internal project FIT-S-20-6427, Guangdong Science and Technology Department grant (No. 2018B010107004), the NSFC grants (No. 61872340), and the INRIA-CAS joint research project VIP.

and concolic testing [17, 24]) are highly based on efficient symbolic encodings of executions into a formula, and rely on highly performing constraint solvers for computing on such encodings. The types of constraints needed depend on the types of program expressions to be analyzed. In the case of scripting languages, constraint solvers need to support different combinations of string operations.

Thus, *string constraint solvers* such as [3–5, 7, 8, 12, 18, 20, 25] are the engine of modern web program analysis techniques. Due to the high demand, there is a boosting amount of publications on this subject in recent years (e.g., [3,9,13,22]). Implementing a constraint solver to cover all standard string libraries in programming languages is a challenging task. One can choose to develop a specialized solving procedure for each string operation, but it requires enormous maintenance efforts. A more feasible solution is to define a minimal set of core constraints that is expressive to encode all others and develop solving procedures only for these core constraints. A common set of such constraints includes: (1) *equality constraints*, e.g., $x.y = y.z$, which says the concatenation of string variables x and y equals the concatenation of y and z, (2) *membership constraints*, e.g., $x \in L(ab^*)$, which says the value of x is the character a followed by a sequence of b's, and (3) *length constraints*, e.g., $|x| = |y| + 3$, which says the length of x is the length of y plus 3, and (4) *"not substring constraints"*, e.g., $\texttt{NotSubstr}(x, y)$, which says x is not a substring of y. Most of the early works focused on the first three types of constraints. The "not substring" constraints have not been systematically studied before[5]. In fact, many common string operations, such as, indexOf and replace, cannot be precisely expressed without using "not substring constraints". Previous study [23] suggests that those operations are among the most commonly used string operation in the applications they studied. The same observation holds while checking existing string constraint benchmarks [2].

More concretely, for two string variables x, y, the operations indexOf(x, y) should return the first occurrence of y in x. We can use the equality and length constraints to encode the position y in x as follows. We need two extra free variables p and s. Then we can use $x = p.y.s$ to express that y is a substring of x and in this case, $|p|$ is the position of y in x. However, there is no guarantee that this y is the first occurrence in x. To do so, we still need to make sure y never occurs in $p.y'$ where y' is the prefix of y with only the last character removed, i.e., $y'.z = y \wedge |z| = 1$ for some z. This can be guaranteed using the constraint NotSubStr(y, $p.y'$). In fact, this is exactly how the Z3 SMT solver encodes indexOf constraint [20].

Observe that the positive version $x \sqsubseteq y$ (i.e., x is a substring of y) can be easily encoded as $y = p.x.s$ using two extra variables. However, the negated version of $x \sqsubseteq y$ is not equivalent to $y \neq p.x.s$. For example, $(x, y, p, s) = (\text{"}a\text{"}, \text{"}ab\text{"}, \epsilon, \epsilon)$ is a model for this latter formula, but x is a substring of y in this case. To capture the not-substring relation precisely, we need to establish that,

[5] More precisely, *"replace all" constraints* [5] and string-integer conversion constraints [3] are not covered by these common set of constraints. Nevertheless, both have been systematically discussed in recent years.

for all strings p and s, y does not equal $p.x.s$; or more formally $\forall p, s : y \neq p.x.s$. Unfortunately, it is known that equality constraint with universal quantifiers is undecidable [16]. Although state-of-the-art solvers, such as Z3, reduce the indexOf and replace constraints by the not-substring constraint, their procedures for solving the latter do not provide much guarantee regarding completeness. It is not hard to find instances with not-substring constraints that the most advanced solvers like CVC4 and Z3 fail to solve (see Figure 1).

$$v.u = u.v \wedge u = p_1.\text{“123456”}.s_1 \wedge v = p_2.\text{“12345”}.s_2 \wedge |u| = 21 \wedge u.\text{“}a\text{”}.v \not\sqsubseteq v.\text{“}a\text{”}.y$$

Fig. 1. An example that both CVC4 and Z3 fail to solve in 3 minutes

In this work, we extend the framework of flat underapproximation [3–5] to handle not substring. The framework has been shown efficient and easily extensible to a rich set of string constraints. It was for instance one of the first approaches to handle string-to-int constraints [3] and it is competitive in efficiency with the best solvers. It relies on construction of so called *flattening* as an underapproximation of string constraints. Namely, it restricts domains of string variables to flat languages of the form $w_1^* w_2^* \cdots w_n^*$, where n and the length of the words $w_1, \ldots, w_n$ are parameters controlling the balance between the cost and the precision of the underapproximation. Under this restriction, string constraints are losslesly translated into quantifier-free linear integer arithmetic formula. The formula is then efficiently handled by the state-of-art SMT solvers. The flat underapproximation is then combined with an overapproximation module capable of proving unsatisfiability. In this work, we are using particularly the method of [13, 14], implemented in the tool OSTRICH, to prove the unsatisfiability. The overapproximation module either solves the constraint as is, if it fits the straight-line fragment of OSTRICH, or it solves an overapproximation of the constraints that fits the fragment. Namely, not-substring constraints of the form $t_1 \not\sqsubseteq t_2$ are first overapproximated as disequalities $t_1 \neq t_2$, and if the straight-line restriction is broken after that, it is recovered by replacing certain occurrences of variables by fresh variables. String-integer conversion constraints, that are not handled by OSTRICH, are simply removed.

A main contribution of this work is a construction that allows to flatten also non-substring constraints. Our solution is efficient despite that the final arithmetic formula for not-substring is not entirely quantifier free—it contains a single universal quantifier. The SMT solver Z3 apparently solves the formulae generated by our implementation fast.

We have evaluated our implementation of the extended framework on a large set of benchmarks from the literature, and a new benchmark collected from executions of the symbolic executor Py-Conbyte[6] on three GitHub projects. Our experimental results show that our prototype, STR, is among the best tools for

[6] `https://github.com/alan23273850/py-conbyte`

solving basic string constraints and outperforms all other tools on benchmarks with not substring constraints.

2 Preliminaries

We use $\mathbb{N}$ (resp., $\mathbb{Z}^+$) to denote the set of non-negative integers (resp., positive integers). For $m, n \in \mathbb{Z}^+$, we write $[n]$ (resp. $[m, n]$) to denote the set $\{1, \ldots, n\}$ (resp. $\{m, m+1, \ldots, n\}$). We use $\mathbb{x}, \mathbb{y}, \ldots$ to denote the integer variables.

In this paper, we assume that Σ is a finite alphabet satisfying $\Sigma \subseteq \mathbb{N}$. The elements of Σ are called *characters*. We use $a, b, \ldots$ to denote the characters. A *string* u over Σ is a sequence $a_1 \ldots a_n$ where $a_i \in \Sigma$ for all i. We use ε to represent the empty string. For two strings u and v, u is said to be a *substring* of v if there exist strings w and w' such that $v = wuw'$. For a string $u = a_1 \ldots a_n$, $|u|$ represents the *length* of u, that is n; moreover, for $i \in [n]$, $u(i) = a_i$ represents the i-th character of u. In addition, for a string u and $a \in \Sigma$, we use $|u|_a$ to denote the number of occurrences of a in u. We use Σ^* to denote the set of strings over Σ, and $\Sigma^{\leq n}$ to denote the set of strings in Σ^* of length at most n. For convenience, we assume that ε is encoded by a fixed natural number from $\mathbb{N} \setminus \Sigma$, and let Σ_ε denote $\Sigma \cup \{\varepsilon\}$.

We use $x, y, \ldots$ to denote *string variables* ranging over Σ^* and we use X to denote the set of string variables. A *string term* is a sequence over $\Sigma_\varepsilon \cup X$.

A *linear integer arithmetic* (LIA) formula is defined by $\phi ::= t \ \mathfrak{o} \ 0 \mid t \equiv c \bmod c \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \mid \exists \mathbb{x}.\ \phi \mid \forall \mathbb{x}.\ \phi$ and $t ::= c \mid \mathbb{x} \mid t + t \mid t - t$, where $\mathfrak{o} \in \{=, \neq, <, >, \leq, \geq\}$ and $\mathbb{x}, c$ are integer variables and constants respectively. A *quantifier-free* LIA (QFLIA) formula is an LIA formula containing no quantifiers. The set of free variables of ϕ, denoted by $\mathsf{Var}(\phi)$, is defined in a standard manner. Given an LIA formula ϕ, and an integer interpretation of $\mathsf{Var}(\phi)$, i.e. a function $I : \mathsf{Var}(\phi) \to \mathbb{Z}$, we denote by $I \models \phi$ that I satisfies ϕ (which is defined in the standard manner), and call I a *model* of ϕ. We use $[\![\phi]\!]$ to denote the set of models of ϕ.

Finally, the *Parikh image* of a word $w \in \Sigma^*$ maps each Parikh (integer) variable $\#a$, where $a \in \Sigma$, to the number of occurrences of a in w. Let $\#\Sigma = \{\#a \mid a \in \Sigma\}$. The Parikh image of w is a function $\mathbb{P}(w) : \#\Sigma \to \mathbb{N}$ such that $\mathbb{P}(w)(\#a) = |w|_a$, for each $a \in \Sigma$. The Parikh image of a language L is defined as $\mathbb{P}(L) = \{\mathbb{P}(w) \mid w \in L\}$. It is well known that the Parikh image of a regular language can be characterized by an LIA formula.

3 String Constraints

In this paper, we extend the class of atomic constraints handled in the framework of [3–5] with not-substring. We focus especially on the core contraints, conjunctions of the atomic constraints of the following forms:

- a string equality constraint $t_1 = t_2$, where t_1, t_2 are string terms,
- a not-substring constraint $t_1 \not\sqsubseteq t_2$, where t_1, t_2 are string terms,

- a QFLIA formula over the integer variables $|x|$ for string variables x,
- a regular constraint $x \in \mathcal{A}$, where x is a string variable and $\mathcal{A}$ is an FA.

We note that presented extension is compatible also with the other types of constraints handled by [3–5], especially context-free membership, transducer constraints, string-integer conversions, negated membership, and disequality. We omit these for simplicity of presentation. For a string constraint ϕ, let us use $\mathsf{StrVar}(\phi)$ and $\mathsf{LenVar}(\phi)$ to denote the set of string variables and the set of length (integer) variables respectively.

While the semantics of linear integer constraints and regular constraints are standard, let us explain the semantics of the string equality constraints and string not-substring constraints:

- A string equality constraint $t_1 = t_2$ has a solution iff there is a homomorphism h from $(X \cup \Sigma)^*$ to Σ^* such that $h(u) = u$ for all $u \in \Sigma^*$ and $h(t_1) = h(t_2)$. For instance, let $t_1 = abxc$, $t_2 = yc$, and $t_3 = ya$. Then $h(x) = \epsilon$ and $h(y) = ab$ is a solution of $t_1 = t_2$. However, for all homomorphisms h, $h(t_1) \neq h(t_3)$, thus $t_1 = t_3$ is not satisfiable.
- A not-substring constraint $t_1 \not\sqsubseteq t_2$ has a solution iff there is a homomorphism h from $(X \cup \Sigma)^*$ to Σ^* such that $h(u) = u$ for all $u \in \Sigma^*$ and $h(t_1)$ is *not* a substring of $h(t_2)$. For instance, let $t_1 = ax$, $t_2 = abxc$, and $t_3 = bx$. Then $h(x) = a$ is a solution of $t_1 \not\sqsubseteq t_2$. However, $t_3 \not\sqsubseteq t_2$ is not satisfiable since $t_3 = bx$ is a subterm of $t_2 = abxc$.

We would like to remark that although the aforementioned class of string constraints does not include explicitly the constraint that t_1 is a substring of t_2, they can be encoded by the string equality constraint $t_2 = xt_1y$, where x, y are the freshly introduced string variables.

4 Solving String Constraints with Flattening

In this section, we will recall the principles of the flattening approach to string solving used in the works [3–5], which we will then extend with not-substring constraints in Section 5.

4.1 (Parametric) Flat Languages

We will present our underapproximations in terms of *flat languages*, which are used to restrict the domain of string variables, and *parametric flat languages*, that are used to specify them.

For integers k and ℓ and a string variable x, we define the family of indexed *character variables* $\mathsf{CharVar}_{k,\ell}(x) = \{x^i_j \mid 1 \leq i \leq k, 1 \leq j \leq \ell\}$. A *parametric flat language* (PFL) with the *period* ℓ and the *cycle count* k is the language $\mathrm{PFL}_{k,\ell}$ of strings over the alphabet $\mathsf{CharVar}_{k,\ell}(x)$ that conform to the regular expression

$$(x^1_1 \dots x^1_\ell)^* \cdot \dots \cdot (x^k_1 \dots x^k_\ell)^*$$

That is, the words of $\mathrm{PFL}_{k,\ell}$ consist of k consecutive parts, each created by iterating a *cycle*, a string $x_1^i \dots x_\ell^i$ of ℓ unique character variables.

A word $w = x_1 \dots x_n \in \mathrm{PFL}_{k,\ell}$ will be interpreted respective to an interpretation of the character variables $I_{\mathsf{Char}} : \mathsf{CharVar}_{k,\ell}(x) \to \Sigma_\varepsilon$ as a string $I_{\mathsf{Char}}(w) = I_{\mathsf{Char}}(x_1) \cdots I_{\mathsf{Char}}(x_n)$ over Σ.

The property of a PFL that is central in our approach is that every PFL is fully characterised by its Parikh image. Let $\mathsf{ParVar}_{k,\ell}(x) = \{\#x_i^j \mid 1 \leq i \leq \ell, 1 \leq j \leq k\}$ be the set of Parikh variables for $\mathsf{CharVar}_{k,\ell}(x)$. Their interpretation $I_{\mathsf{Par}} : \mathsf{ParVar}_{k,\ell}(x) \to \mathbb{N}$ can be unambiguously decoded as a word from the language $\mathrm{PFL}_{k,\ell}$:

Proposition 1. *There is a function* $\mathbb{P}_{k,\ell}^{-1} : (\mathsf{ParVar}_{k,\ell}(x) \to \mathbb{N}) \to \mathrm{PFL}_{k,\ell}$ *which acts as the inverse function of* $\mathbb{P}$*, namely,* $\mathbb{P}_{k,\ell}^{-1}(\mathbb{P}(w)) = w$ *for each* $w \in \mathrm{PFL}_{k,\ell}$.

Intuitively, the function $\mathbb{P}_{k,\ell}^{-1}$ computes the word $w \in \mathrm{PFL}_{k,\ell}$ by repeating each cycle several times, the number of repetitions of the i-the cycle being $\mathbb{P}(w)(\#x_1^i)$ (note that $\mathbb{P}(w)(\#x_j^i)$ is the same for all $x_j^i, j \in [\ell]$).

Hence, an interpretation of Parikh variables $I_{\mathsf{Par}} : \mathsf{ParVar}_{k,\ell}(x) \to \mathbb{N}$ together with an interpretation of character variables $I_{\mathsf{Char}} : \mathsf{CharVar}_{k,\ell}(x) \to \Sigma_\varepsilon$ encode a word over Σ, namely the word $I_{\mathsf{Char}}(\mathbb{P}_{k,\ell}^{-1}(I_{\mathsf{Par}}))$.

The set of all strings over Σ that can be encoded as such pair of interpretations I_{Char}, I_{Par} is the *flat language* with the cycle count k and period ℓ:

$$\mathrm{FL}_{k,\ell} = \{I_{\mathsf{Char}}(\mathbb{P}_{k,\ell}^{-1}(I_{\mathsf{Par}})) \mid I_{\mathsf{Char}} : \mathsf{CharVar}_{k,\ell}(x) \to \Sigma_\epsilon, I_{\mathsf{Par}} : \mathsf{ParVar}_{k,\ell}(x) \to \mathbb{N}\}$$

We note that we implement parametric flat languages as parametric flat automata. A parametric flat automaton is a finite automaton with a restricted structure—a sequence of cycles, each representing a cycle of the parametric language, as illustrated on Fig. 2. The automata form is needed for computing flattening of regular, context free, and other constraints (presented in [3, 4]). Flattening of non-substring constraints, the subject of this paper, can be however explained using the simpler language view, hence we can abstract from the technicalities of automata in the current paper.

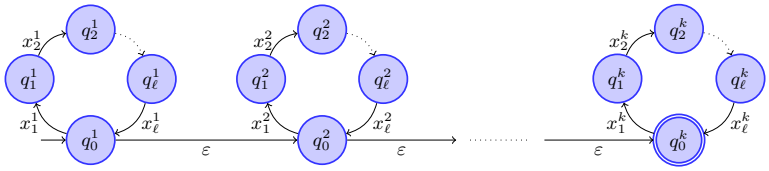

Fig. 2. The flat automaton $\mathcal{A}$ accepting the language $\mathrm{PFL}_{k,\ell}$.

4.2 Flattening of String Constraints

Let us now formalise the notion of flattening, a construction of LIA formulas that encode string constraints restricted to the domain of flat languages.

A *flat semantics* for a string constraint ϕ is obtained from the semantics of ϕ by restricting the domain of each string variable to the language $\mathrm{FL}_{k,\ell}$, for chosen k and ℓ. Let k and ℓ be fixed for the rest of this section.

An assignment I of $\mathsf{Var}(\phi)$ is called k,ℓ-*flat* if for each $x \in \mathsf{StrVar}(\phi)$, $I(x) \in \mathrm{FL}_{k,\ell}$. The flat semantics of ϕ is then defined as

$$[\![\phi]\!]_{k,\ell} = \{I \in [\![\phi]\!] \mid I \text{ is } k,\ell\text{-flat}\}$$

Our approach to string solving is built on that the flat semantics of the string constraint can be precisely encoded by a QFLIA formula in which every string variable $x \in \mathsf{StrVar}(\phi)$ is represented by the character and Parikh variables $\mathsf{CharVar}_{k,\ell}(x)$ and $\mathsf{ParVar}_{k,\ell}(x)$, respectively, and which inherits the integer variables.

A flat solution of ϕ, $I \in [\![\phi]\!]_{k,\ell}$, is encoded as an assignment $I' = I_{\mathsf{Flat}} \cup I_{\mathsf{Int}}$. I_{Flat} is the assignment of flattening variables that encodes the values of the original string variables. In other words, I' is the union of assignments $I_{\mathsf{CharVar}_{k,\ell}(x)} : \mathsf{CharVar}_{k,\ell}(x) \to \Sigma_\varepsilon$ and $I_{\mathsf{ParVar}_{k,\ell}(x)} : \mathsf{ParVar}_{k,\ell}(x) \to \mathbb{N}$ for every $x \in \mathsf{StrVar}(\phi)$ satisfying that

$$I(x) = I_{\mathsf{CharVar}_{k,\ell}(x)}(\mathbb{P}^{-1}_{k,\ell}(I_{\mathsf{ParVar}_{k,\ell}(x)}))$$

The encoding is not unique (a string can often be k,ℓ-encoded in multiple ways), hence the encoding function returns the set of all encodings of I, denote $\mathit{encode}_{k,\ell}(I)$.

Decoding is the inverse of encoding, though, due to Proposition 1 it is unambiguous, as stated by this lemma:

Lemma 1. *If* $\mathit{encode}_{k,\ell}(I) \cap \mathit{encode}_{k,\ell}(J) \neq \emptyset$*, then* $I = J$*.*

Hence we can define the decoding as a function that returns an unique interpretation of variables (not a set, as in the case of encoding):

$$\mathit{decode}_{k,\ell}(I') = I \text{ iff } I' \in \mathit{encode}_{k,\ell}(I)$$

We can now specify the required properties of the flattening QFLIA of ϕ. It is formula $\mathit{flatten}_{k,\ell}(\phi)$ that encodes the flat semantics of ϕ, that is

$$[\![\phi]\!]_{k,\ell} = \mathit{decode}_{k,\ell}([\![\exists \mathsf{AuxVar} : \mathit{flatten}_{k,\ell}(\phi)]\!]) \tag{1}$$

The existential quantification is above used to abstract away additional auxiliary variables AuxVar, variables other than $\mathsf{FlatVar}(\phi)$ and $\mathsf{LenVar}(\phi)$, which the formula $\mathit{flatten}_{k,\ell}(\phi)$ is sometimes constructed with.

The formula $\mathit{flatten}_{k,\ell}(\phi)$ is constructed inductively by following the structure of ϕ: $\mathit{flatten}_{k,\ell}(\phi_1 \wedge \phi_2) = \mathit{flatten}_{k,\ell}(\phi_1) \wedge \mathit{flatten}_{k,\ell}(\phi_2)$. Therefore, it is sufficient to show how to construct $\mathit{flatten}_{k,\ell}(\phi)$ for atomic constraints ϕ. Later on in Section 5, we will show how to construct $\mathit{flatten}_{k,\ell}(t_1 \not\sqsubseteq t_2)$. The construction of $\mathit{flatten}_{k,\ell}(\phi)$ for the other atomic constraints is discussed in [3].

For the inductive construction to work, flattening of the atomic constraint has to satisfy a stronger condition than Equation 1. Namely, the obtained QFLIA

formula must capture *all* encodings of the solutions of the string constraint, not only *some* of them. Otherwise for instance the inductive construction of flattening of a conjunction from flattenings of its conjuncts could be incorrect (the intersection of solution encodings could be empty while the sets of solutions themselves do intersect). Formally, the flattening of the atomic constraints must satisfy that:

$$\mathit{encode}_{k,\ell}([\![\phi]\!]_{k,\ell}) = [\![\exists \mathsf{AuxVar} : \mathit{flatten}_{k,\ell}(\phi)]\!] \tag{2}$$

where, again, AuxVar contains auxiliary variables of $\mathit{flatten}_{k,\ell}(\phi)$ other than $\mathsf{FlatVar}(\phi)$ and $\mathsf{LenVar}(\phi)$). A major point of this paper is a construction of flattening of non-substring constraints satisfying Equation 2, as presented in Section 5.

4.3 String Constraint Solving Algorithm

We now shortly recall the whole string solving algorithm. It uses an underapproximation module based on the flat abstraction and an overapproximation module. The two modules are run in parallel. The main loop is summarised in Algorithm 1.

The underapproximation module tries to prove satisfiability of a flat underapproximation, gradually incrementing both the period and cycle count, until the underapproximation is SAT or a limit is reached.

The overapproximation can use any algorithm capable of proving UNSAT. We do not claim any contribution in the overapproximation part, but to demonstrate that such combination indeed yields a capable tool, we combine our underapproximation technique with the method of [13,14] implemented in the tool OSTRICH. It is a complete method for the so called straight-line fragment of string constraints that supports regular and transducer constraints, replace-all, word-equations and other constraints. The straight-line restriction imposes, intuitively, that the constraint must have been obtained from a program in a single static assignment form in which every string variables is assigned at most once and is not used on the right side of an assignment before it is itself assigned. The length constraints are unrestricted (we refer the reader to [13, 14] for the precise definition). The overapproximation module either solves the constraint as is, if it fits the straight-line fragment, or it solves an overapproximation of the constraints that fits the fragment. Namely, not-substring constraints of the form $t_1 \not\sqsubseteq t_2$ are first overapproximated as disequalities $t_1 \neq t_2$, and if the straight-line restriction is broken after that, it is recovered by replacing certain occurrences of variables by fresh variables. String-integer conversion constraints, that are not handled by OSTRICH, are simply removed.

5 Flattening of Not-Contains Constraints

We will now describe the construction of the flattening formula for not-substring, the formula $\mathit{flatten}_{(1,\ell)}(t_1 \not\sqsubseteq t_2)$, for given terms t_1, t_2.

Algorithm 1: String solving via flattening

Input: string constraint ϕ, initial period k_0 and cycle count ℓ_0, flattening limit $flim$

do in parallel
  for *i from 0 to* $flim$ **do**
    if $\mathit{flatten}_{k_0+i,\ell_0+i}(\phi)$ *is SAT* **then return** *SAT*;
  if *Overapproximate*(ϕ) *is UNSAT* **then return** *UNSAT*;
return *UNKNOWN*

5.1 Simplifying Assumptions

To simplify the presentation, we will consider flat domain restrictions with the cycle count $k = 1$ only. This is without loss of generality since using a flat domain restriction $\mathrm{FL}_{k,\ell}$ with $k > 1$ is equivalent to replacing every substituting string variable x by the concatenation $x_1 \cdots x_k$ and using $\mathrm{FL}_{1,\ell}$. The assumption of $k = 1$ makes the upper index 1 of the character variables x_i^1 superfluous, hence we will omit it and write only x_i.[7]

We also make the following simplifying assumptions on the input string constraint ϕ:

1. We assume that $\mathsf{StrVar}(t_1) \cap \mathsf{StrVar}(t_2) = \emptyset$. Note that any string constraint can be made to satisfy this by replacing one of the occurrences of a string variable $y \in \mathsf{StrVar}(t_1) \cap \mathsf{StrVar}(t_2)$ by its fresh primed variant y' and introducing an additional constraint $y = y'$.
2. We assume that t_1 and t_2 do *not* contain constant strings, that is, they are concatenations of string variables. Every equality or not substring constraint can be transformed into this form by replacing each occurrence of a constant a with a fresh variable x_{oc} (each occurrence with a unique fresh variable) together with the regular constraint $x_{oc} \in \{a\}$. Such modification does not influence the constraints membership in the decidable fragment used for overapproximation (whereas replacing all occurrences of a by a single fresh variable could).

5.2 Construction of the Flattening Formula

The construction of $\mathit{flatten}_{(1,\ell)}(t_1 \not\sqsubseteq t_2)$ is based on the following observation.

Observation 1 *For every two strings* $u, v \in \Sigma^*$, $u \not\sqsubseteq v$ *iff either* $|u| > |v|$ *or* $|u| \leq |v|$ *and for every shift* $\in [0, |v| - |u|]$, *there exists* $pos \in [|u|]$, $u(pos) \neq v(\mathit{shift} + pos)$.

Intuitively, either t_1 is longer than t_2, or, as illustrated on Figure 3, for any position *shift* where we try to fit t_1 inside t_2, we can find a position *pos* in t_1 which will not match the corresponding position $\mathit{shift} + pos$ in t_2.

The core of the flattening formula will be constructed as a disjunction of two formulae that express the two cases of Observation 1, $\psi_{|t_1|>|t_2|}$ and $\psi_{|t_1|\leq|t_2|}$.

[7] Our implementation however handles cycle counts k larger than one directly.

String lengths and effective periods. To express the two cases of Observation 1, we will speak about *effective periods* of string variables and about their *lengths*, for which we introduce auxiliary constants and variables.

First, the *effective period* of a string variable z is the number ℓ_z of character variables in $\mathsf{CharVar}_{(1,\ell)}(z)$ that are assigned a non-ε value (a character from Σ) under a given assignment of character variables. Whenever a constant representing an effective period is used, we must ensure that it is indeed the effective period.

To make this test easier and the formula testing this more compact, we restrict the space of encodings of strings by requiring that the interpretations of the variables $x \in \mathsf{StrVar}(\phi)$ to ones that are *sorted*. That means that character variables x_j assigned ε appear only *at the end* of the cycle, namely *after* all character variables that are assigned letters of Σ. This is achieved by conjoining the flattening of ϕ with the formula $\psi_{\varepsilon\text{-end}}$:

$$\psi_{\varepsilon\text{-end}} = \bigwedge_{x \in \mathsf{StrVar}(\phi)} \bigwedge_{j \in [\ell-1]} (x_j = \varepsilon \to x_{j+1} = \varepsilon)$$

Note that this only restricts the encodings of the solutions of ϕ but not the set of solutions itself, since every solution has among its encodings one with aligned ε's.

In sorted interpretations, it holds that $\ell_z \in [\ell]$ if and only if z_{ℓ_z} is the last character variable assigned a non-ε character, and $\ell_z = 0$ if and only if z_1 is assigned ε. This is checked by the formula $\psi^{(z,\ell_z)}_{\text{period}}$:

$$\psi^{(z,\ell_z)}_{\text{period}} = \begin{cases} z_{\ell_z} \neq \varepsilon \wedge z_{\ell_z+1} = \varepsilon & \text{if } \ell_z \in [\ell-1] \\ z_\ell \neq \varepsilon & \text{if } \ell_z = \ell \\ z_1 = \varepsilon & \text{if } \ell_z = 0 \end{cases}$$

The length len_z of z is then determined from ℓ_z and the value of the Parikh variable $\#z_1$ as $len_z = \ell_z * \#z_1$. Indeed, $\#z_1$, the number of occurrences of z_1, is the number of iterations of the cycle $z_1 \cdots z_\ell$, and each iteration of the cycle produces a string of the length equals to the effective period ℓ_z. Additionally, we also need to ensure that $\#z_1, \ldots, \#z_\ell$ are the same, since $z_1 \cdots z_\ell$ is iterated as a whole, which is captured by $\bigwedge_{i \in [\ell-1]} \#z_i = \#z_{i+1}$. Put together, we create the formula ψ_{len}:

$$\psi_{len} = \psi_{\varepsilon\text{-end}} \wedge \bigwedge_{z \in \mathsf{StrVar}(t_1 \sqsubseteq t_2)} \psi^{(z,\ell_z)}_{\text{period}} \wedge len_z = \ell_z * \#z_1 \wedge \bigwedge_{i \in [\ell-1]} \#z_i = \#z_{i+1}$$

Since ℓ_z is a constant, $len_z = \ell_z * \#z_1$ is not a multiplication of two integer variables, but only an abbreviation of ℓ_z-fold addition of $\#z_1$.

Formula for Observation 1, case t_1 longer than t_2. Using the effective periods and lengths, the first case of Observation 1 with t_1 longer than t_2 can now be expressed as the formula $\psi_{|t_1|>|t_2|}$.

Suppose that $t_1 = x_1 \cdots x_m$ and $t_2 = y_1 \cdots y_n$. The case $|t_1| > |t_2|$ is specified simply by the formula

$$\psi_{|t_1|>|t_2|} = \sum_{i\in[m]} len_{x_i} > \sum_{i\in[n]} len_{y_i}$$

Formula for Observation 1, case t_1 not longer than t_2. The second case of Observation 1, with t_1 no longer than t_2, is expressed as the formula $\psi_{|t_1|\leq|t_2|}$. It is more complicated than the previous case.

Recall that it states that $|t_1| \leq |t_2|$ and for every $shift \in [0, |t_2| - |t_1|]$, there exists $pos \in [|t_1|]$, $t_1(pos) \neq t_2(shift + pos)$, as shown on Figure 3. The formula, that allows to check this, is constructed as follows:

1. For every positions *shift* where t_1 could fit into t_2 ($|t_1| + shift \leq |t_2|$),
2. find a position *pos* in t_1 such that t_2 could fit inside t_1 at this position,
3. find a string variable $(x_{m'})_\alpha$ and a string variable $(y_{n'})_\beta$ of t_2 which appear at positions *pos* and $pos + shift$, respectively, and
4. verify $(x_{m'})_\alpha \neq (y_{n'})_\beta$ (hence t_1 at *pos* differs from t_2 at $pos + shift$).

First, t_1 shifted by *shift* must still "fit" inside t_2, that is:

$$\psi_{shift} = (0 \leq shift \leq \sum_{i\in[n]} len_{y_i} - \sum_{i\in[m]} len_{x_i})$$

Second, there are *conflict variables* $x_{m'}$ and $y_{n'}$, $m' \in [m]$ and $n' \in [n]$ such that $t_1(pos)$ corresponds to some character of $x_{m'}$ and such that $t_2(shift + pos)$ corresponds to some character of $y_{n'}$. This is formally expressed by the formulas

$$\psi_{x_{m'}} = \sum_{i\in[m'-1]} len_{x_i} < pos \leq \sum_{i\in[m']} len_{x_i}$$
$$\psi_{y_{n'}} = \sum_{i\in[n'-1]} len_{y_i} < shift + pos \leq \sum_{i\in[n']} len_{y_i}$$

Third, $t_1(pos)$ and $t_2(shift+pos)$ must correspond to the values of some character variables $(x_{m'})_\alpha$ in $(x_{m'})_1 \cdots (x_{m'})_{\ell_{x_{m'}}}$ and $(y_{n'})_\beta$ in $(y_{n'})_1 \cdots (y_{n'})_{\ell_{y_{n'}}}$. The indices α and β are specified as follows:

– The following formula checks that $(x_{m'})_\alpha$ is indeed at the position *pos* in t_1. The formula assumes an effective period of $x_{m'}$ and also verifies that assumption:

$$\psi_{x_{m'}}^{(\ell_{x_{m'}},\alpha)} = \psi_{\text{period}}^{(x_{m'},\ell_{x_{m'}})} \wedge (pos - \sum_{i\in[m'-1]} len_{x_i}) \equiv \alpha \bmod \ell_{x_{m'}}$$

– Similarly, the following formula checks that $(y_{n'})_\beta$ is indeed at the position $shift + pos$ in t_2. It assumes an effective period of $y_{n'}$ and verifies that assumption:

$$\psi_{y_{n'}}^{(\ell_{y_{n'}},\beta)} = \psi_{\text{period}}^{(y_{n'},\ell_{y_{n'}})} \wedge (shift + pos - \sum_{i\in[n'-1]} len_{y_i}) \equiv \beta \bmod \ell_{y_{n'}}$$

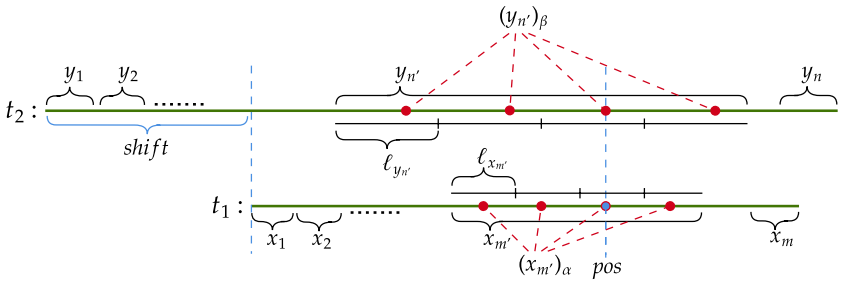

Fig. 3. An overview of the construction of $\psi_{|t_1|\geq|t_2|}$.

Fourth, having the indices α, β of character variables specified as above, $t_1(pos) \neq t_2(shift + pos)$ can be expressed as $(x_{m'})_\alpha \neq (y_{n'})_\beta$. The entire formula $\psi_{|t_1|\leq|t_2|}$ then is:

$$\psi_{|t_1|\leq|t_2|} = \sum_{i\in[m]} len_{x_i} \leq \sum_{i\in[n]} len_{y_i} \ \wedge \ \forall shift\, \exists pos.\ \psi_{shift} \rightarrow \bigvee_{\substack{m'\in[m],\\ n'\in[n]}} \left(\begin{array}{l} \psi_{x_{m'}} \wedge \psi_{y_{n'}} \ \wedge \\ \bigvee_{\ell_{x_{m'}},\ell_{y_{n'}}\in[\ell]} \ \bigvee_{\substack{\alpha\in[\ell_{x_{m'}}],\\ \beta\in[\ell_{y_{n'}}]}} \left(\begin{array}{l} \psi_{x_{m'}}^{(\ell_{x_{m'}},\alpha)} \wedge \psi_{y_{n'}}^{(\ell_{y_{n'}},\beta)} \wedge \\ (x_{m'})_\alpha \neq (y_{n'})_\beta \end{array} \right) \end{array} \right)$$

Finally, we construct the flattening of the not substring constraint as:

$$flatten_{(1,\ell)}(t_1 \not\sqsubseteq t_2) = \psi_{len} \wedge (\psi_{|t_1|>|t_2|} \vee \psi_{|t_1|\leq|t_2|})$$

Theorem 1 states that the construction is indeed correct in the sense that it satisfies Equation 2, only with modified, primed, variant of $encode_{(1,\ell)}$, restricted only to sorted interpretations (satisfying $\psi_{\varepsilon\text{-end}}$). This is still enough for Equation 2 to be true for conjunctive constraints that contain non-substring atomic predicates. AuxVar are variables other than string and length variables x and $|x|$, $x \in \mathsf{Var}(t_1 \not\sqsubseteq t_2)$.

Theorem 1. $encode'_{(1,\ell)}([\![t_1 \not\sqsubseteq t_2]\!]_{(1,\ell)}) = [\![\exists \mathsf{AuxVar} : flatten_{(1,\ell)}(t_1 \not\sqsubseteq t_2)]\!]$

6 Implementation and Evaluation

We compare STR[8] with the other state-of-the-art string solvers, namely, CVC4 (version 1.8)[9] [8] and Z3 (version 4.8.9)[10] [20]. For these tools, the versions we

[8] The github link will be made available after the double blind review process
[9] `https://github.com/CVC4/CVC4/releases/tag/1.8`
[10] `https://github.com/Z3Prover/z3/releases/tag/z3-4.8.9`

used are the latest release version. Observe that CVC4 and Z3 are DPLL(T)-based string solvers. We do not compare with Sloth [18] since it does not support length constraints, which occur in most of our benchmarks. Moreover, we do not compare with ABC [7] (a model counter for string constraints) and Trau+ [4–6] as well, because they do not support many string functions occurring in our benchmarks, especially those containing the "not contains" functionality.

We performed the experiments on two benchmark suites. The first benchmark suite is new and obtained by running the symbolic executor Py-Conbyte[11] on the following three GitHub projects,

- biopython[12]: freely available Python tools for computational molecular biology and bioinformatics,
- django[13]: a high-level Python Web framework that encourages rapid development and clean, pragmatic design,
- thefuck[14]: an app that corrects errors in previous console commands, inspired by a @liamosaur tweet.

The symbolic exeuctor Py-Conbyte produces files in the SMT2 format. We only keep those SMT2 files where the function "(*str.contains x y*)" or "(*str.indexof x y n*)" with a non-constant first or second argument occurs.

The second benchmark suite contains sets of standard benchmarks from [3] that have been used previously in the comparison of existing string solvers.

We carry out the experiments on a PC with an Intel Core i7-10700 (2.90 GHz) processor with 8 cores and 16 threads, a 48 GB of RAM, and a 1.8TB, 7200 rpm hard disk drive running the Ubuntu 20.04.1 LTS operating system. The timeout was set to 10s for each SMT file. In the implementation of STR, we modify the SAT handling component of Trau to the version described in this paper, and use it to handle SAT instances. Then STR run Ostrich and the modified Trau in parallel, and terminate when Ostrich reports UNSAT or Trau reports SAT. The experimental results are summarized in Table 1. Columns with heading SAT (resp. UNSAT) show the number of SAT (resp. UNSAT) test cases for which the solver returns correct answers. Column with heading FAILED indicates the number of test cases for which the solver returns UNKNOWN or cannot finished with 10 seconds.

From Table 1, we can see that overall STR is better than Z3 and has a similar performance to CVC4 in handling SAT instances. The handling of UNSAT instances is worse than the others, but this is mainly due to the use of OSTRICH. Observe that the over-approximation module is not the main focus of this paper since our main goal is to address the major weakness of the flattening framework of handling not-substring constraint and to provide an under-approximation technique which has at least as good, and in many cases better, performance than the state-of-the-art tools.

[11] `https://github.com/alan23273850/py-conbyte`
[12] `https://github.com/biopython/biopython`
[13] `https://github.com/django/django`
[14] `https://github.com/nvbn/thefuck`

	SAT			UNSAT			FAILED		
BENCHMARK	z3	cvc4	STR	z3	cvc4	STR	z3	cvc4	STR
biopython (77222)	5180	5707	5770	70190	70518	62435	1852	997	9017
django (52645)	8404	9297	9487	41871	42161	33471	2370	1187	9687
thefuck (19872)	1883	2313	2194	17545	17530	16018	444	29	1660
Leetcode (2666)	880	881	876	1785	1785	1658	1	0	132
PyEx (25421)	16656	20651	21420	3775	3857	3316	4990	913	685
aplas (600)	122	54	132	100	205	1	378	341	467
cvc4-str (1880)	22	18	25	1802	1841	184	56	21	1671
full-str-int (21571)	2875	4379	4433	16708	16985	12234	1988	207	4904
slog (3391)	1296	1309	1290	2082	2082	2054	13	0	47
stringfuzz (1065)	429	716	534	208	243	62	428	106	469

Table 1. Results on new and existing benchmarks

7 Conclusion and Future Work

We have proposed an extension of the flattening techniques for string constraints that handles constraints of the type not-substring. Our techniques generates flattening formulae that express the flat semantics of string constraints precisely. Although they do contain a single universal quantifier, they can still be handled efficiently by existing solvers. Our experimental results show that our prototype can solve not-substring constraints better than other tools (especially SAT cases) and it is competitive on the other types of constraints.

An interesting possibility for future is to solve string logic with not substring constraint precisely, not only under the flat abstraction. A possibility of flat abstraction of not substring which would be fully quantifier free is also not closed and is worth further investigation.

References

1. OWASP top ten web application security risk, https://owasp.org/www-project-top-ten (2017), `https://owasp.org/www-project-top-ten`
2. Trauc string constraints benchmark collection (2020), `https://github.com/plfm-iis/trauc_benchmarks`
3. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Dolby, J., Janku, P., Lin, H., Holík, L., Wu, W.: Efficient handling of string-number conversion. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 943–957. ACM (2020). https://doi.org/10.1145/3385412.3386034, `https://doi.org/10.1145/3385412.3386034`
4. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Cohen, A., Vechev, M.T. (eds.) Proceedings of the 38th ACM SIGPLAN Conference

on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017. pp. 602–617. ACM (2017)
5. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–5. IEEE (2018)
6. Abdulla, P.A., Atig, M.F., Diep, B.P., Holík, L., Janku, P.: Chain-free string constraints. In: Chen, Y., Cheng, C., Esparza, J. (eds.) Automated Technology for Verification and Analysis - 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28-31, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11781, pp. 277–293. Springer (2019)
7. Aydin, A., Eiers, W., Bang, L., Brennan, T., Gavrilov, M., Bultan, T., Yu, F.: Parameterized model counting for string and numeric constraints. In: Leavens, G.T., Garcia, A., Pasareanu, C.S. (eds.) Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018. pp. 400–410. ACM (2018)
8. Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011)
9. Berzish, M., Kulczynski, M., Mora, F., Manea, F., Day, J.D., Nowotka, D., Ganesh, V.: A length-aware regular expression SMT solver. CoRR **abs/2010.07253** (2020), `https://arxiv.org/abs/2010.07253`
10. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS. vol. 1579, pp. 193–207. Springer (1999)
11. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
12. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. PACMPL **2**(POPL), 3:1–3:29 (2018)
13. Chen, T., Hague, M., He, J., Hu, D., Lin, A.W., Rümmer, P., Wu, Z.: A decision procedure for path feasibility of string manipulating programs with integer data type. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12302, pp. 325–342. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_18, `https://doi.org/10.1007/978-3-030-59152-6\_18`
14. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. PACMPL **3**(POPL), 49:1–49:30 (2019)
15. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981. LNCS, vol. 131, pp. 52–71. Springer (1981)
16. Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of extended word equations: The boundary between decidability and undecidability. CoRR **abs/1802.00523** (2018), `http://arxiv.org/abs/1802.00523`
17. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI. pp. 213–223. ACM (2005)

18. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)
19. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
20. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings. Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer (2008)
21. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer (1982)
22. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: Reductions for strings and regular expressions revisited. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 225–235. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_30, `https://doi.org/10.34727/2020/isbn.978-3-85448-042-6\_30`
23. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 513–528. IEEE Computer Society (2010). https://doi.org/10.1109/SP.2010.38, `https://doi.org/10.1109/SP.2010.38`
24. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE. pp. 263–272. ACM (2005)
25. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: a z3-based string solver for web application analysis. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 114–124. ACM (2013)