A Solver for a Theory of Strings and Bit-vectors

Sanu Subramanian*, Murphy Berzish*, Omer Tripp†, Vijay Ganesh*
*University of Waterloo
†Google, USA

In this paper, we address the problem of automated security verification of C/C++ programs. We focus in particular on overflow vulnerabilities (buffer overflow, integer overflow, etc.) which unfortunately still occur in production-level systems and are difficult to reason about effectively (i.e. both efficiently and accurately) due to the need to model simultaneously both string values input by the user and integral properties such as their length. We observe that in the context of automated reasoning tools for the theory of strings, existing string solvers are not efficient at modeling the combined theory over strings and bit-vectors. Current approaches either model such combination of theories by a reduction of strings to bit-vectors and then use a bit-vector solver as a backend, or model bit-vectors as natural numbers and use a backend solver for the combined theory of strings and natural numbers. This leads either to missing important vulnerabilities, such as buffer overflows, or to serious performance issues for symbolic analysis of such applications. Hence, there is a clear need for a solver that models strings and bit-vectors natively.

We present an SMT solver for a many-sorted first-order quantifier-free theory $T_{w,bv}$ of string equations, string length represented as bit-vectors, and bit-vector arithmetic. Our solver Z3strBV is such a decision procedure for the theory $T_{w,bv}$ that combines solvers for bit-vectors and string equations, targeted towards security analysis of string-handling applications. We demonstrate experimentally that Z3strBV is significantly more efficient than a reduction of string/bit-vector constraints to strings/natural numbers followed by a solver for strings/natural numbers or modeling strings as bit-vectors. Additionally, we prove decidability for the theory $T_{w,bv}$. We also propose two optimizations, which can be adapted to other contexts. The first accelerates convergence on a consistent assignment of string lengths, and the second — dubbed *library-aware SMT solving* — fixes summaries for built-in string functions directly in Z3strBV (e.g., `strlen` in C/C++). Finally, we demonstrate experimentally that Z3strBV is able to detect nontrivial overflows in real-world system-level code, as confirmed against 7 security vulnerabilities from CVE and Mozilla database.

I. Introduction

Despite years of effort to prevent or mitigate the risk of security defects related to the use of string functions in low-level languages such as C and C++, buffer overflow and integer overflow errors remain one of the leading causes of security vulnerabilities. A large number of recent, high-profile entries in the CVE database [1], such as the Android Stagefright vulnerabilities and the Linux kernel SCSI ioctl exploit, can be traced back to improper use of string functions leading to an exploitable buffer overflow or integer overflow. This motivates the need for stronger software analysis tools to detect such vulnerabilities or demonstrate their absence.

Constraint solvers are one such class of software analysis tool, and have been shown to be effective in a wide variety of applications. These programs have increasingly become the basis of many tools for hardware verification [13], program analysis [43], [42], [32], [20] and automated testing [7], [6], [28], [29], [17]. The key idea is to model behaviors of interest of the subject system as logical constraints, and then discharge the resulting constraints to a SAT or SMT solver such that solutions generated by the solver serve as test inputs for the system under verification. In particular, these solvers have enjoyed applications in automated analysis and verification of security properties, and automated test generation for programs.

Naturally, the ability to carry out reasoning in this fashion is dependent on the expressive power and efficiency of the solver, which has motivated significant effort in developing useful theories, integrate them into solvers, and optimize their ability to solve such rich classes of constraints. Examples include the quantifier-free (QF) first-order theory of bit-vectors, which is effective in modeling machine arithmetic [13], [11]; the QF theory of arrays, which enables modeling of machine memory [13], [11]; and the theory of uninterpreted functions and integers to model abstractions of program state [6].

We consider the application of constraint solvers to the problem of detecting buffer overflow and integer overflow vulnerabilities in C/C++ applications. This requires the solver to be able to reason not only about bit-level operations, but about the relatively high-level string data structure as well, and the operations that can be performed on it.

a) **Existing Solutions, and the need for a New Solver:** There are several powerful tools to reason about string-manipulating code [20], [32], [33], [42], [23]. All these tools support the theory of string equations, where the length function — applied to a string — returns an arbitrary-precision (or unbounded) natural number. (HAMPI [20] is an exception since it only deals with bounded-length string variables.) While effective, these solvers are not adequate for reasoning about low-level C/C++ programs that use a fixed-precision machine representation for integers rather than an arbitrary-precision natural number datatype. The reason is that modeling bit-vectors as natural numbers complicates reasoning about potential overflows (or underflows), which is an artifact of the fixed-precision machine (bit-vector) representation of numeric values. Precise modeling of arithmetic overflow/underflow is a key reason to model numeric values in terms of bit-vectors and not integers, and the motivation for a large body of work on bit-vector solvers[14], [11].

Another approach to solving $L_{w,bv}$-formulas is to represent strings as bit-vectors. In fact many symbolic execution engines like KLEE [6] and S2E [8] perform reasoning at this level. They collect constraints as bit-vectors by solving branch conditions using STP [14] or Z3 [11]. However, these engines perform poorly on programs that make heavy use of string functions, as the low-level bit-vector representation of program data fails to efficiently capture the high-level semantics of the string data type [9].

As this brief survey highlights, currently there is a disturbing gap. The existing solutions as illustrated above are all either inefficient or imprecise. On the one hand, reasoning about the lengths of strings as though they were natural numbers is inadequate for modelling integer overflow, underflow, bit-wise operations, and pointer casting. On the other hand, modelling strings as bit-vectors makes it difficult to perform direct reasoning on strings efficiently [30]. Furthermore, this approach cannot handle strings that are not explicitly bounded in length, and therefore is not appropriate for verification of the absence of overflow vulnerabilities.

Hence, we were motivated to build Z3strBV, which solves $L_{w,bv}$-formulas by treating strings and bit-vectors natively. We did so by combining a solver for strings, augmented with a bit-vector-sorted length function, and a solver for bit-vectors within the Z3 SMT solver combination framework.

b) **Motivating Example:** This paper follows the same general motivation as that of a typical SMT solver design and implementation. Specifically, we address the increasing need of *efficient* solver supports to reason about security errors due to improper string manipulations, which occur frequently in C/C++ system-level code [25]. Improper handling of string values carries serious ramifications, including crashes, unintended program behaviors, and exposure to security threats [35], [12], [36], [34]. A constraint solver for such analyses must be able to model not only string values but also machine-level constraints in bit-vector arithmetic, and in particular the potential for arithmetic overflow.

```
1  bool check_login(char* username, char* password
       ) {
2    if (! validate_password(password)) {
3      invalid_login_attempt(); exit(-1);
4    }
5    const char *salt = get_salt8(username);
6    unsigned short len = strlen(password) +
         strlen(salt) + 1;
7    if (len > 32) {
8      invalid_login_attempt(); exit(-1);
9    }
10   char *saltedpw = (char*)malloc(len);
11   strcpy(saltedpw, password);
12   strcat(saltedpw, salt);
13   ...
14 }
```

Fig. 1. A login function with an integer overflow vulnerability

As an illustration, we consider an example (Figure 1) inspired by real-world vulnerabilities we analyzed. It uses a combination of string manipulations, the string length function, and bit-vector arithmetic, and is interesting to analyze from the perspective of this theory combination. The `validate_password` call at line 2 abstracts a method that utilizes string operators (e.g., `strstr` and `strcmp`) and makes sure the input is safe, for example, that it does not contain any non-printable characters, but does not perform any checking of the input length. The `get_salt8` method maps a username to a pseudorandom 8-character string, which is concatenated with the password to strengthen it against certain types of password cracking attacks. Although the input length is indirectly checked at line 7, there is still the threat of an overflow. If the input consists of $65535 - 8 = 65527$ characters, an overflow occurs when the variable `len`, which is of type `unsigned short` and thus ranges over $[0, 2^{16} - 1 = 65535]$, is assigned `strlen(password) + strlen(salt) + 1`. This leads to the allocation of a buffer of size 0, and consequently to heap corruption due to copying of `password` into the empty buffer.

Analyses of such programs based on symbolic executions with a traditional string solver (without bit-vector support) or a purely bit-vector solver can be *highly inefficient* [30]. A source of complexity is the handling of integer modulo operations required to simulate the fixed-precision calculations and overflows. Additionally, other string operations listed require the use of a solver that supports strings as well as bit-vectors, as simple bounds checking alone is insufficient to capture the relevant semantics of this method.

c) **Summary of Contributions:** The key novelty and contribution of this paper is a solver algorithm for a combined theory of string equations, string length modelled using bit-vectors, and bit-vector arithmetic, its theoretical underpinnings, implementation, and evaluation over several sets of benchmarks. The contributions, in more detail, are:

1) **(Solver Algorithm)** We specify a practical solver algorithm for the combined theory of strings and bit-vectors that is efficient for a large class of verification, testing, analysis and security applications. Additionally, we formally prove that our solving algorithm is sound.
2) **(Binary Search Heuristic)** We propose a "binary search" heuristic, which allows fast convergence on consistent string lengths across the string and bit-vector solvers. This heuristic can be of value in other theory combinations, such as the combination of strings and natural numbers.
3) **(Library-Aware Heuristic)** We introduce a second heuristic for our solver that adds summaries for common string functions at the SMT solver level, such that both precision and performance are improved. The solver can reason about certain C/C++ string library functions natively at the contract (or summary) level, rather than having to (re)analyze their actual code (and corresponding constraints) each time the symbolic analysis encounters them.

4) **(Formal Characterization)** We formally characterize the theory of string equations, string length as bit-vectors, and linear arithmetic over bit-vectors, and provide a (constructive) proof of decidability for this theory. In particular, we want to stress that the decidability of such a theory is non-trivial. At first glance it may seem that all models for this theory have finite universes since bit-vector arithmetic has a finite universe. However, this is misleading since the search space over all possible strings remains infinite even when string length is a fixed-width bit-vector due to the wraparound semantics of bit-vector arithmetic. (For example, for any fixed length N, there are infinitely many strings whose length is some constant k modulo N.)
5) **(Implementation and Evaluation)** Finally, we describe the implementation of our solver, Z3strBV, which is an extension of the Z3str2 string solver. We present experimental validation for the viability and significance of our contributions, including in particular (i) the ability to detect overflows in real-world systems using our solver, as confirmed via reproduction of several security vulnerabilities from the CVE vulnerability database, and (ii) demonstration of orders-of-magnitude improvements in solver performance by applying the two optimizations we described above.

II. Syntax and Semantics

A. The Syntax of the Language $L_{w,bv}$

We define the sorts and constant, function, and predicate symbols of the countable first-order many-sorted language $L_{w,bv}$.

Sorts: The language is many-sorted with a string sort str and a bit-vector sort bv. The language is parametric in k, the width of bit-vector terms (in number of bits). The Boolean sort $Bool$ is standard. When necessary, we write the sort of an $L_{w,bv}$-term t explicitly as $t : sort$.

Finite Alphabet: We assume a finite alphabet Σ of characters over which all strings are defined.

String and Bit-vector Constants: We define a disjoint two-sorted set of constants $Con = Con_{str} \cup Con_{bv}$. The set Con_{str} is a subset of Σ^*, the set of all finite-length string constants over the finite alphabet Σ. Elements of Con_{str} will be referred to as *string constants*, or simply *strings*. ϵ denotes the empty string. Elements of Con_{bv} are binary constants over k digits. As necessary, we may subscript bit-vector constants by bv to indicate that their sort is "bit-vector".

String and Bit-vector Variables: We fix a disjoint two-sorted set of variables $var = var_{str} \cup var_{bv}$. var_{str} consists of string variables, denoted $X, Y, \ldots$ that range over string constants, and var_{bv} consists of bit-vector variables, denoted $a, b, \ldots$ that range over bit-vectors.

String Function Symbols: The string function symbols include the concatenation operator $\cdot : str \times str \rightarrow str$ and the length function $strlen_{bv} : str \rightarrow bv$.

Bit-vector Arithmetic Function Symbols: The bit-vector function symbols include binary k-bit addition (with overflow) $+ : bv \times bv \rightarrow bv$. Following standard practice in mathematical logic literature, we allow multiplication by constants as a shorthand for repeated addition.

String Predicate Symbols: The predicate symbols over string terms include equality and inequality: $=, \neq$: $str \times str \rightarrow Bool$.

Bit-vector Predicate Symbols: The predicate symbols over bit-vector terms include $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ (with their natural meaning), all of which have signature $bv \times bv \rightarrow Bool$.

B. Terms and Formulas of $L_{w,bv}$

Terms: $L_{w,bv}$-terms may be of string or bit-vector sort. A string term t_{str} is inductively defined as an element of var_{str}, an element of Con_{str}, or a concatenation of string terms. A bit-vector term t_{bv} is inductively defined as an element of var_{bv}, an element of Con_{bv}, the length function applied to a string term, a constant multiple of a length term, or a sum of length terms. (For convenience we may write the concatenation and addition operators as n-ary functions, even though they are defined to be binary operators.)

Atomic Formulas: The two types of atomic formulas are (1) word equations (A_w) and (2) inequalities over bit-vector terms (A_{bv}).

QF Formulas: We use the term "QF formula" to refer to any Boolean combination of atomic formulas, where each free variable is implicitly existentially quantified and no explicit quantifiers may be written in the formula.

Formulas and Prenex Normal Form: $L_{w,bv}$-formulas are defined inductively over atomic formulas. We assume that formulas are always represented in prenex normal form (i.e., a block of quantifiers followed by a QF formula).

Free and Bound Variables, and Sentences: We say that a variable under a quantifier in a formula ϕ is bound. Otherwise we refer to variables as free. A formula with no free variables is called a sentence.

C. Semantics and Canonical Model over the Language $L_{w,bv}$

We fix a string alphabet Σ and a bit-vector width k. Given $L_{w,bv}$-formula ϕ, an *assignment* for ϕ w.r.t. Σ is a map from the set of free variables in ϕ to $Con_{str} \cup Con_{bv}$, where string (*resp.* bit-vector) variables are mapped to string (*resp.* bit-vector) constants. Given such an assignment, ϕ can be interpreted as an assertion about Con_{str} and Con_{bv}. If this assertion is true, then we say that ϕ itself is *true* under the assignment. If there is some assignment s.t. ϕ is true, then ϕ is *satisfiable*. If no such assignment exists, then ϕ is *unsatisfiable*.

For simplicity we omit most of the description of the canonical model of this theory, choosing to use the intuitive combination of well-known models for word equations and bit-vectors. We provide, however, semantics for the $strlen_{bv}$ function, since it is not a "standard" symbol of either separate theory.

Semantics of the $strlen_{bv}$ Function: For a string term w, $strlen_{bv}(w)$ denotes an unsigned, fixed-precision bit-vector representation of the "precise" integer length of w, truncating the arbitrary-precision bit-vector representation of that integer to its lowest k bits, for fixed bit-vector width k. The bit-vector addition and (constant) multiplication operators produce a result of the same width as the input terms and treat both operands as though they represent unsigned integers. Of particular note is that both of these operators have the potential to overflow (that is, to produce a result that is smaller than either operand). This is a consequence of the fixed precision of bit-vectors. Furthermore, the $strlen_{bv}$ function itself may also "overflow", because it is a fixed-width representation of an arbitrary-precision integer. More precisely, bit-vector arithmetic has the semantics of integer arithmetic modulo 2^k, and the value represented by $strlen_{bv}(w)$ is the bit-vector representation of the value in the field of integers modulo 2^k that is congruent to the "precise" integer length of w.

For example, if the number of bits used to represent bit-vectors is $k = 3$, a string of precise length 1 and another string of precise length 9 both have bit-vector width of "001". Although the complete bit-vector representation of 9 as an arbitrary-precision bit-vector would be "1001", the semantics of $strlen_{bv}$ specify that all but the $k = 3$ lowest bits are omitted.

Note that the search space with respect to strings is (countably) infinite despite the fixed-width representation of string lengths as bit-vectors. This is because the bit-vector length of a string is only a view of its precise length, i.e. the integer number of characters in the string. This integer length may be arbitrarily finitely large. The semantics of fixed-width integer overflow is, in essence, applied to the integer length in order to obtain the bit-vector length. In fact, there are infinitely many strings that have the same bit-vector length. For example, if eight bits are used to represent string length, strings of length 0, 256, 512, ... would all appear to have a bit-vector length of "00000000".

III. Decidability of QF String Equations, String Length, and Bit-Vector Constraints

In this section, we prove the decidability of the theory $T_{w,bv}$ of QF word equations and bit-vectors. Towards this goal, we first establish a conversion from bit-vector constraints to regular languages. For regular expressions (regexes), we use the following standard notation: AB denotes the concatenation of regular languages A and B. $A|B$ denotes the alternation (or union) of regular languages A and B. A^* denotes the Kleene closure of regular language A (i.e., 0 or more occurrences of a string in A). For a finite alphabet $\Sigma = \{a_1, a_2, \ldots, a_l\}$, $[a_1 - a_l]$ denotes the union of regex $a_1|a_2|\ldots|a_l$. Finally, A^i, for nonzero integer constants i and regex A, denotes the expression $AA\ldots A$, where the term A appears i times in total.

Lemma 1. *Let k be the width of all bit-vector terms. Suppose we have a bit-vector formula of the form $len_{bv}(X) = C$, where X is a string variable and C is a bit-vector constant of width k. Let i_C be the integer representation of the constant C, interpreting C as an unsigned integer. Then the set $M(X)$ of all strings satisfying this constraint is equal to the language L described by the regular expression $([a_1 - a_l]^{2^k})^* \, [a_1 - a_l]^{i_C}$.*

Proof. In the forward direction, we show that $M(X) \subseteq L$. Let $x \in M(X)$. x satisfies the constraint $len_{bv}(x) = C$, which means that the integer length z of x modulo 2^k is equal to i_C. Additionally, $z \geq 0$ as strings cannot have negative length. Then there exists a non-negative integer n such that $z = n2^k + i_C$. We decompose x into strings u, v such that $uv = x$, the length of u is $n2^k$, and the length of v is i_C. Now, $u \in ([a_1 - a_l]^{2^k})^*$ because its length is a multiple of 2^k, and $v \in [a_1 - a_l]^{i_C}$ because its length is exactly i_C. By properties of regex concatenation, $uv \in ([a_1 - a_l]^{2^k})^* \, [a_1 - a_l]^{i_C}$ and therefore $uv \in L$. Since $uv = x$, we have $x \in L$.

In the reverse direction, we show that $L \subseteq M(X)$. Let $x \in L$. By properties of regex concatenation, there exist strings u, v such that $uv = x$, $u \in ([a_1 - a_l]^{2^k})^*$, and $v \in [a_1 - a_l]^{i_C}$. Suppose u was matched by n expansions of the outer Kleene closure for some non-negative integer n. Then the integer length of u is $n2^k$. Furthermore, the integer length of v is i_C. This implies that the integer length of x is $n2^k + i_C$, which means that the integer length of x is equal to i_C modulo 2^k, from which it directly follows that $len_{bv}(x) = C$. Hence $x \in M(X)$ as required. This completes both directions of the proof and so we have equality between the sets $M(X) = L$. □

Theorem 1. *The satisfiability problem for the QF theory of word equations and bit-vectors is decidable.*

Proof Idea for the Decidability Theorem 1: Intuitively, the decision procedure proceeds as follows. The crux of the proof is to convert bit-vector constraints into regular languages (represented in terms of regexes) relying on the lemma mentioned above, and correctly capture overflow/underflow behavior. In order to capture the semantics of unsigned overflow, each regex we generate has two parts: The first part, under the Kleene star, matches strings of length a multiple of 2^k that cause an k-bit bit-vector to overflow and wrap around to the original value; the second part matches strings of constant length i_C, corresponding to the part of the string that is "visible" as the bit-vector length representation. By solving the bit-vector fragment of the equation first we can generate all of finitely many possible solutions, and therefore check each of finitely many assignments to the bit-vector length terms. For each bit-vector solution, we solve the word-equation fragment separately under regular-language constraints, which guarantee that only strings that have the expected bit-vector length representation will be allowed as solutions. It is easy to see that this algorithm is sound, complete, and terminating, given a decision procedure for word equations and regex.

Proof. We demonstrate a decision procedure by reducing the input formula to a finite disjunction of subproblems in the theory of QF word equations and regular language constraints. This theory is known to be decidable by Schulz's extension of Makanin's algorithm for solving word equations [27].

Suppose the input formula ϕ has the form $W_1 = W_2 \wedge A_1 = B_1 \wedge A_2 = B_2 \wedge \ldots \wedge A_n = B_n$, where W_1, W_2 are terms in the theory of word equations and $A_1 \ldots A_n, B_1 \ldots B_n$ are terms in the theory of bit-vectors. Let k be the width of all bit-vector terms. For each term of the form $len_{bv}(X_i)$ in $A_1 \ldots A_n, B_1 \ldots B_n$, replace it with a fresh bit-vector variable v_i and collect the pair (v_i, X_i) in a set $\mathcal{S}$ of substitutions. Suppose there are m such pairs. Then the total number of bits among all variables introduced this way is mk. This means that there are 2^{mk} possibilities for the values of $v_1 \ldots v_m$. Because the theory of QF bit-vectors is decidable and because there are finitely many possible values for $v_1 \ldots v_m$, we can check the satisfiability of the bit-vector fragment of the input formula $A_1 = B_1 \wedge A_2 = B_2 \wedge \ldots \wedge A_n = B_n$ for all possible substitutions of values for $v_1 \ldots v_m$ in finite time. For each assignment $A = \{(v_1, C_1), (v_2, C_2), \ldots, (v_m, C_m)\}$, where each v_i is a variable and each C_i is a bit-vector constant, if the bit-vector constraints are satisfiable under that assignment, collect A in the set $\mathcal{A}$ of all satisfying assignments. If the set $\mathcal{A}$ is empty, then the bit-vector constraints were not satisfiable under any assignment to $v_1 \ldots v_m$. In this case we terminate immediately and decide that the input formula is UNSAT, as the bit-vector constraints must be satisfied for satisfiability of the whole formula. Otherwise, construct the formula $R'(\phi)$ as follows. For each assignment $A \in \mathcal{A}$, for each term $(v_i, C_i) \in A$, we find the pair $(v_i, X_i) \in \mathcal{S}$ with corresponding v_i. Because each variable v_i corresponds to a term $len_{bv}(X_i)$, and since we have $v_i = C_i$, we have the constraint $len_{bv}(X_i) = C_i$. This allows us to apply Lemma 1 and generate a regular language constraint $X_i \in L_i$. After generating each such regular language constraint, we collect $R'(\phi) := R(\phi) \vee (W_1 = W_2 \wedge X_1 \in L_1 \wedge X_2 \in L_2 \wedge \ldots \wedge X_m \in L_m)$. We repeat this for each assignment $A \in \mathcal{A}$. The resulting formula $R(\phi) = (W_1 = W_2) \wedge R'(\phi)$ is a conjunction of the original word equation from ϕ and a finite disjunction of regular-language constraints over variables in that word equation. We now invoke Schulz's algorithm to solve this formula. If the word equation and any disjunct are satisfiable, then we report that the original formula ϕ is SAT; otherwise, ϕ is UNSAT. Finally, it is easy to show that the reduction is sound, complete, and terminating for all inputs. □

A. Proof of Soundness and Completeness of the Reduction used in Theorem 1

We demonstrate that the reduction from bit-vector constraints to regular language constraints, as performed in the proof for Theorem 1, is sound and complete. We do so by showing equisatisfiability between ϕ and $R(\phi)$.

Theorem 2. *ϕ is satisfiable iff $R(\phi)$ is satisfiable.*

Proof. In the forward direction, we show that if ϕ is satisfiable then $R(\phi)$ is satisfiable. Let M be a satisfying assignment of all variables in ϕ. Because ϕ and $R(\phi)$ share the same constraint $W_1 = W_2$, M is a satisfying assignment for the word equation fragment of $R(\phi)$ as well. It remains to show that at least one of the terms in $R'(\phi)$, the disjunction of regular language constraints, is satisfiable. The algorithm described in Theorem 1 generates one group of regex constraints for each satisfying assignment to the bit-vector fragment that produces a distinct model for all bit-vector length constraints. In particular, the algorithm generates regular language constraints for the particular model described by M of bit-vector length constraints. Because the string variables in M satisfy these constraints, we apply Lemma 1 to find that the regular language constraints that were generated with respect to this model M are satisfied by the assignment of all string variables in M. Therefore M is also a model of $R(\phi)$, and hence $R(\phi)$ is satisfiable.

In the reverse direction, we show that if $R(\phi)$ is satisfiable then ϕ is satisfiable. Let M be a satisfying assignment of all variables in $R(\phi)$. Because $R(\phi)$ and ϕ share the same constraint $W_1 = W_2$, M is a satisfying assignment for the word-equation fragment of ϕ as well. It remains to show that the bit-vector constraints in ϕ are satisfiable under this assignment to the string variables. Let r be a regular-language constraint in $R'(\phi)$ (the disjunction of regular-language constraints), such that r evaluates to true under the assignment M. We know that such an r must exist because the formula $R(\phi)$ is satisfiable, and therefore at least one of the terms in the disjunction $R'(\phi)$ must evaluate to true. By applying Lemma 1 "backwards", we can derive an assignment of constants to bit-vector length terms in ϕ corresponding to each regular-language constraint in r that is consistent with the lengths of the string variables. We also know that the bit-vector constraints are satisfiable under this assignment of constants to strings and bit-vector length terms because, by Lemma 1, a precondition for the appearance of any term r in $R'(\phi)$ is that the bit-vector fragment of ϕ is satisfiable under the partial assignment to bit-vector length terms that yielded r. Therefore, by solving the remaining bit-vector constraints, which must be satisfiable, M can be extended to a model of ϕ and hence ϕ is satisfiable. □

It may appear that the decidability result is trivial as the domain of bit-vectors is finite for fixed width k, and therefore the formula could be solved by trying all 2^k possible assignments for each bit-vector length term and all finitely many strings whose length is equal to each given bit-vector length. However, this is incorrect, as the bit-vector length of a string is only a representation of its "true length". As the semantics of bit-vector arithmetic specify that overflow is possible under this interpretation, strings of integer length 1, 5, 9, etc. – indeed, infinitely many strings – will all satisfy a constraint asserting that the 2-bit bit-vector length of a string term is 1. Therefore, it is not sufficient to search over, for example, only the space of strings with length between 0 and $2^k - 1$. Hence the decidability of this theory is non-trivial, and this motivates the need for a stronger argument, such as given in Theorem 1.

IV. Z3strBV Solver Algorithm

The Z3strBV algorithm is informed by the decision procedure presented in the previous section, but embodies several important design choices to guarantee its efficiency.

A. Pseudocode Description

The main procedure of the Z3strBV solver, which is similar to the Z3str2 procedure [43], [42], is summarized as Algorithm 1. It takes as input sets $\mathcal{Q}_w$ of word equations and $\mathcal{Q}_l$ of bit-vector (length) constraints, and its output is either SAT or UNSAT or UNKNOWN. UNKNOWN means that the algorithm has encountered overlapping arrangements and pruned those arrangements (thereby potentially missing a SAT solution), and a SAT solution could not be found in remaining parts of the search space.

Algorithm 1 High-level description of the Z3strBV main algorithm.

```
Input: sets $\mathcal{Q}_w$ of word equations and $\mathcal{Q}_l$ of bit-vector (length) constraints
Output: SAT / UNSAT / UNKNOWN
1:  procedure SOLVESTRINGCONSTRAINT($\mathcal{Q}_w$,$\mathcal{Q}_l$)
2:      if all equations in $\mathcal{Q}_w$ are in solved form then
3:          if $\mathcal{Q}_w$ is UNSAT or $\mathcal{Q}_l$ is UNSAT then
4:              return UNSAT
5:          if $\mathcal{Q}_w$ and $\mathcal{Q}_l$ are SAT and mutually consistent then
6:              return SAT
7:      $\mathcal{Q}_a \longleftarrow$ Convert $\mathcal{Q}_w$ into equisatisfiable DNF formula
8:      for all disjunct $D$ in $\mathcal{Q}_a$ do
9:          $\mathbb{A} \longleftarrow$ all possible arrangements of equations in $D$
10:         for all arrangement $A$ in $\mathbb{A}$ do
11:             $l_A \longleftarrow$ length constraints implied by $A$
12:             if $l_A$ is inconsistent with $\mathcal{Q}_l$ then
13:                 $\mathbb{A} \longleftarrow \mathbb{A} \setminus \{ A \}$
14:         for all string variable $s$ incident in $D$ do
15:             $G(s) \longleftarrow$ merge per-equation arrangements involving $s$
16:         for all merged arrangements $a \in G(s)$ with no overlaps do
17:             $\mathcal{Q}'_w \longleftarrow$ refine variables in $\mathcal{Q}_w$ per $a$
18:             $\mathcal{Q}'_l \longleftarrow$ update length constraints $\mathcal{Q}_l$ per $\mathcal{Q}'_w$
19:             $r \longleftarrow$ SOLVESTRINGCONSTRAINT($\mathcal{Q}'_w$,$\mathcal{Q}'_l$)
20:             if $r$=SAT then
21:                 return SAT
22:     if overlapping variables detected at any stage then
23:         return UNKNOWN
24:     else
25:         return UNSAT
```

The input to the procedure is a conjunction of constraints. Any higher-level Boolean structure is handled by the SMT core solver, typically a SAT solver. The first part of the procedure (lines 2-9) check whether (i) either $\mathcal{Q}_w$ or $\mathcal{Q}_l$ is UNSAT or (ii) both are SAT and the solutions are consistent with each other. If neither of these cases applies, then arrangements that are inconsistent with the length constraints are pruned (lines 12-21). Finally, the surviving arrangements $G(s)$ guide refinement of the word equations $\mathcal{Q}_w$, and so also of the length constraints $\mathcal{Q}_l$, and for each $G(s)$ the solving loop is repeated for the resulting sets $\mathcal{Q}'_w$ and $\mathcal{Q}'_l$ (lines 22-29). A SAT answer leads to a SAT result for the entire procedure. If no solution is found, but overlapping variables have been detected at some point, then the procedure returns UNKNOWN. (Note that all current practical string solvers suffer from both incompleteness and potential non-termination.)

Notice that during the solving process, the string plug-in (potentially) derives additional length constraints incrementally (line 14). These are discharged to the bit-vector solver on demand, and are checked for consistency with all the existing length constraints (both input length constraints and constraints added previously during solving).

More generally, during the solving process the string and bit-vector solvers each generate new assertions in the other domain. Inside the string theory, candidate arrangements are constrained by the assertions on string lengths, which are provided by the bit-vector theory. In the other direction, the string solver derives new length assertions as it progresses in exploring new arrangements. These assertions are provided to the bit-vector theory to prune the search space.

Basic Length Rules: Given strings $X, Y, Z, W, \ldots$, we express their respective lengths $l_X, l_Y, l_Z, \ldots$ as $strlen\_bv(X,n), strlen\_bv(Y,n), strlen\_bv(Y,n), \ldots$ respectively in the constraint system, where n is the bit-vector sort. The empty string is denoted by ϵ. Two rules govern the reasoning process: (i) $X = Y \implies l_X = l_Y$ and (ii) $W = X \cdot Y \cdot Z \cdot \ldots \implies l_W = l_X + l_Y + l_Z + \cdot \ldots$.

As an example, consider the word equation $X \cdot Y = M \cdot N$, where X, Y, M, N are nonempty string variables. The are three possible arrangements [42], as shown below, where T_1 and T_2 are temporary string variables:

$(X = M \cdot T_1) \wedge (N = T_1 \cdot Y)$
$(X = M) \wedge (N = Y)$
$(M = X \cdot T_2) \wedge (Y = T_2 \cdot N)$

The respective length assertions, derived from these three arrangements, are as follows:

$(l_X = l_M + l_{T_1}) \wedge (l_N = l_{T_1} + l_Y)$
$(l_X = l_M) \wedge (l_N = l_Y)$
$(l_M = l_X + l_{T_2}) \wedge (l_Y = l_{T_2} + l_N)$

B. Proof of the Soundness of Algorithm 1

We use the standard definition of soundness for decision procedures from the SMT literature [42], whereby a solver is sound if whenever the solver returns UNSAT, the input formula is indeed unsatisfiable.

Theorem 3. *Algorithm 1 is sound, i.e., when Algorithm 1 reports UNSAT, the input constraint is indeed UNSAT.*

Proof. First, line 4 returns an UNSAT if either the string or the bit-vector constraints are determined to be UNSAT. For string constraints, we use the algorithms described in [15] to decide the satisfiability of word (dis)equations. Soundness follows from the soundness of the procedure [15] and the (Z3) bit-vector solver.

For the UNSAT returned at line 25, we show that transformations impacting it are all satisfiability-preserving. In particular, the transformations at (i) line 7 (ii) lines 9-13 (iii) line 14-15 and (iv) line 17 are satisfiability-preserving: (i) The DNF conversion at line 7 is obviously satisfiability-preserving. (ii) Line 9 is a variant of the sound arrangement generation method mentioned in Makanin's paper [24]. It is satisfiability-preserving because each arrangement is a finite set of equations implied by the input system of equations. Besides, we extract length constraints from arrangements. If they conflict with the existing bit-vector constraints, we drop the corresponding arrangements. As we assume the bit-vector theory is sound, this step is also satisfiability-preserving. (iii) Lines 14-15 systematically enumerate all feasible options to further split word equations [42]. This step is satisfiability-preserving. (iv) Line 17 derives simpler equations by a satisfiability-preserving rewriting [42]. $\Box$

We conclude by noting that the Z3str2 algorithm for string constraints is terminating, as shown in [42], and that it is easy to see that integrating the theory of bit-vectors with this algorithm preserves this property, as the space of bit-vector models is finite.

C. Binary Search Heuristic

As explained above, length assertions are added to the Z3 core, then processed using the bit-vector theory. For efficiency, we have developed a binary-search-based heuristic to fix a value for the length variables in the bit-vector theory. To illustrate the heuristic, and the need for it, we consider the example

$$"a" \cdot X = Y \cdot "b"$$
$$bv8000[16] < strlen\_bv(X, 16) < bv9000[16]$$

where $bv8000[16]$ ($bv9000[16]$) denotes the constant 8000 (9000). The constraint $"a" \cdot X = Y \cdot "b"$ is discharged to the string theory, whereas $bv8000[16] < strlen\_bv(X, 16) < bv9000[16]$ is discharged to the bit-vector solver. For the string constraint, a (non-overlapping) solution is

$$X = "b"$$
$$Y = "a"$$
$$strlen\_bv(X, 16) = strlen\_bv(Y, 16) = bv1[16]$$

but this solution is in conflict with the bit-vector constraints. Thus, the (overlapping) arrangement $X = T \cdot "b" \quad Y = "a" \cdot T$ is explored, which leads to length constraints

$$strlen\_bv(v, 16) = strlen\_bv(T, 16) + bv1[16] : v \in \{X, Y\}$$
$$strlen\_bv(T, 16) > bv0[16]$$

Now the need arises to find consistent lengths for X, Y, T. Iterating all possibilities one by one, and checking these possibilities against the bit-vector theory, is slow and expensive. Instead, binary search is utilized to fix lower and upper bounds for candidate lengths.

The first choice is for a lower bound of 0 and an upper bound of 2^{16} (where 16 is the bit-vector width, as indicated above). This leads to the first candidate being $strlen(X, 16) = bv32767[16]$ (where $32767 = 2^{15} - 1$). This fails, leading to an update to the upper bound to be 2^{15}, and consequently the next guess is $strlen(X, 16) = bv16383[16]$. This too falls outside the range $(8000, 9000)$, and so the upper bound is updated again, this time becoming 2^{14}, and the next guess is $strlen(X, 16) = bv8191[16]$. This guess is successful, and so within 3 (rather than 8000) steps the search process converges on the following consistent length assignments:

$$l_v = strlen(v, 16) = bv8191[16] \quad : \quad v \in \{X, Y\}$$
$$l_T = strlen(T, 16) = bv8190[16]$$

As the example highlights, in spite of the tight interaction between the string and bit-vector theories, large values for length constraints are handled poorly by default, since the process of converging on consistent string lengths is linearly proportional to those values. Pleasingly, bit-vectors, expressing a finite range of values, enable safe lower and upper bounds. More concretely, given a bit-vector of width n, the value of the length variable is in the range $[0, 2^n - 1]$. Our heuristic iteratively adds length assertions to the bit-vector theory following a binary-search pattern until convergence on consistent length assignments. This process is both sound and efficient.

A similar binary search-inspired heuristic can also be applied in the case where string lengths are represented as natural numbers, as is done by Z3str2. However, as the set of natural numbers is not finite, a complication arises when choosing an upper bound over which to perform the search. We therefore implemented a slightly modified heuristic which performs binary search over a dynamic window whose size can be increased or decreased by the core solver according to the satisfiability of the binary search constraints we generate. Briefly, we choose an (initially arbitrary) upper bound for the window size, and if the solver asserts that the length of the string is greater than the midpoint of the window, we double the upper bound and search again. By doing this we can quickly converge on a satisfiable upper bound and then perform binary search in the window that was discovered this way.

D. Library-aware Solving Heuristic

We introduce as well a further optimization, by which we encode into the SMT solver a class of library functions f in popular programming languages like C/C++ or Java, such that (i) f is commonly used by programmers, (ii) uses of f are a frequent source of errors (due to programmer mistakes), and (iii) symbolic analysis of f is expensive due to the many paths it defines.

More precisely, by library-aware SMT solvers we mean that the logic of traditional SMT solvers is extended with declarative summaries of functions such as `strlen` or `strcpy`, expressed as global invariants over all behaviors of such functions. The merit of declaratively modeling such functions is that, unlike the real code implementing these functions, the summary is free of downstream paths to explore. Instead, the function is modeled as a set of logical constraints, thereby offsetting the path explosion problem. Through this method we are able to encode summaries of library functions from the C/C++ string library, including `IndexOf`, `Substring`, `Contains`, and `Replace`. Observe, importantly, that our summary-based optimization is complementary to summary-based symbolic execution. To fully exploit library-aware SMT solving, one has to modify the symbolic execution engine as well to generate summaries or invariants upon encountering library functions.

V. Experimental Results

In this section, we describe our evaluation of Z3strBV. The experiments were performed on a MacBook computer, running OS X Yosemite, with a 2.0GHz Intel Core i7 CPU and 8GB of RAM. We have made the Z3strBV code, as well as the experimental artifacts, publicly available [2].

A. Experiment I: Buffer Overflow Detection

To validate our ability to detect buffer overflows using Z3strBV, we encoded 7 well-known buffer overflow vulnerabilities from the CVE database [1] as two semantically equivalent sets of constraints — in the string/natural number theory and in the string/bit-vector theory — to compare between Z3strBV and Z3str2. The Z3str2 tool is one of the most efficient implementations of the string/natural number theory. The solvers only differ in whether string length is modelled as an integer or as a bit-vector.

The vulnerabilities examined here include several recent and high-profile security issues, including two Google Stagefright vulnerabilities, an OpenSSH remote code execution vulnerability, and a Linux kernel SCSI ioctl information disclosure bug. The common element to all of these vulnerabilities is that an overflow or underflow in an arithmetic computation on the length of a string (or buffer) directly leads to an exploitable region of code. This necessitates the ability of the tool to reason efficiently about arithmetic overflow in the context of various operations on strings in order to construct an input that triggers each vulnerability.

We encoded the vulnerable regions of the code by hand. For the bit-vector solver, we encoded all operations natively as expressions over bit-vector terms; for the integer solver, we added additional constraints to model the possibility of overflow on numeric terms. We set the solver timeout for each test case at 1 hour. Figure 2 presents the results. Z3strBV is able to detect all vulnerabilities, and further generate concrete input values that synthesize an exploit. Z3str2, by contrast, provides limited support for arithmetic overflow/underflow. Unfortunately, correctly modeling overflow/underflow using linear arithmetic over natural numbers is inefficient, and thus it fails within the prescribed time budget of 1 hour. Without the ability to perform overflow modelling, Z3str2 cannot detect overflow bugs at all, since arbitrary-precision integers cannot overflow. This experiment, therefore, shows that Z3strBV can detect real-world buffer overflow vulnerabilities that Z3str2 does not detect (due to timeouts).

B. Experiment II: Library-aware SMT Solving

We evaluated the library-aware solving heuristic atop the example shown in Fig. 1 by applying both our technique and KLEE, a state-of-the-art symbolic execution engine, to this code. The goal was to detect the heap corruption threat in that code. We faithfully encoded the program snippet in check_login() as string/bit-vector constraints. We then checked whether the buffer pointed-to by _username is susceptible to overflow.

Fig. 2. CVE Buffer Overflow Detection and Exploit Synthesis (See [31], [2] for details.)

Vulnerability	Z3strBV	Z3str2
CVE-2015-3824	0.079s	TO (1h)
CVE-2015-3826	0.108s	TO (1h)
CVE-2009-0585	0.031s	TO (1h)
CVE-2009-2463	0.279s	TO (1h)
CVE-2002-0639	0.116s	TO (1h)
CVE-2005-0180	0.029s	TO (1h)
FreeBSD Bugzilla #137484	0.038s	TO (1h)

Fig. 3. Vulnerability Detection using KLEE and Library-aware Solving.

Prec.	Lib-aware Solving		KLEE
	Z3strBV	Z3str2	
8-bit	0.507s	167s	300s
16-bit	0.270s	TO (7200s)	TO (7200s)

Notice that len is an unsigned short variable, and thus ranges from 0 to $2^{16} - 1 (65,535)$. As it represents the buffer size, it determines the number of concrete execution paths KLEE has to enumerate, as well as the search space for library-aware solving. By contrast, the constraints generated by library-aware SMT solver declaratively model `strlen` as part of the SMT solver logic.

To characterize performance trends, we consider two different precision settings for string length: 8-bit and 16-bit. We used 120 minutes as the timeout value. There was no need to go beyond 16 bits since KLEE was already significantly slower at 16 bits relative to the library-aware SMT solver. Note that KLEE is slow because it has to explore a large number of paths, and not because the individual path constraints are difficult to solve.

The results are provided in Figure 3. Under both precision settings, KLEE is consistently and significantly slower than the library-aware solving technique. In particular, if we represent numeric values using 16 bits, then KLEE is not able to identify the problem in 120 minutes, while Z3strBV can solve the problem in 0.27 seconds.

The benefit thanks to library-aware solving is clear. The analysis is as follows. Suppose both username and _username are symbolic string variables. In Figure V-B(a), as KLEE forks a new state for each character, an invocation of `strlen` on a symbolic string $S1_{sym}$ of size $|S1_{sym}|$ will generate and check $|S1_{sym}| + 1$ path constraints (one per each possible length value between 0 and $|S1_{sym}|$). In Figure V-B(b), in contrast, the constraint encoding enabled by library-aware solving essentially captures the semantics of the program without explicitly handling the loop in `strlen`. Only one query is needed to check whether the length $S2_{sym}$ can be smaller than the length $S1_{sym}$.

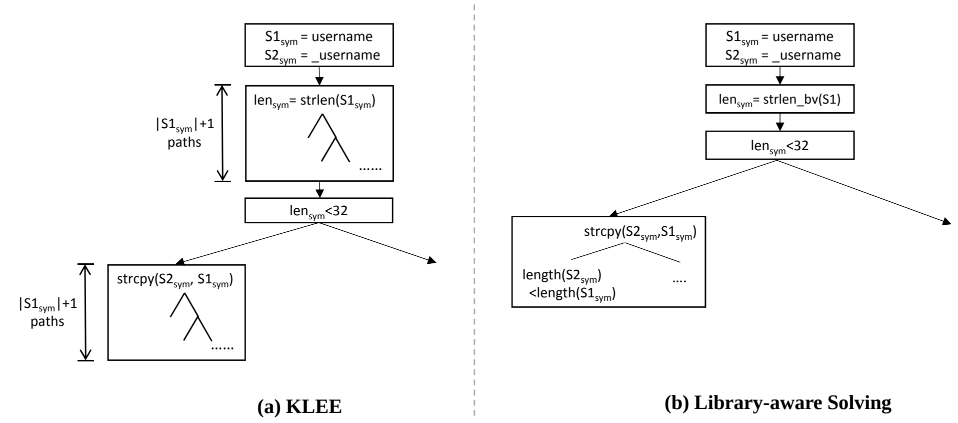

Fig. 4. Comparison between KLEE and Library-aware Solving

C. Experiment III: Binary Search Heuristic

In the case of unconstrained string variables, both Z3str2 and Z3strBV negotiate with the Z3 core to converge on concrete length assignments. Z3str2 does so via a linear length search approach. We evaluate this approach against the binary search heuristic. For that, we have implemented a second version of Z3strBV that applies linear search. We adapted benchmarks used to validate Z3str2 [3], resulting in a total of 109 tests which make heavy use of string and bit-vector operators. Timeout was set at 20 seconds per benchmark. The comparison results are presented in Table 5. We group the instances by the solver result: SAT, UNSAT, TIMEOUT or UNKNOWN.

The Z3strBV solver is able to complete on all instances in 17.7 seconds, whereas its version with linear search requires 548.1 seconds. This version can solve simple SAT cases, but times out on 26 of the harder SAT cases, whereas Z3strBV has zero timeouts. Z3strBV is able to detect overlapping arrangements in 2 cases, on which it returns UNKNOWN. The linear-search version, in contrast, can only complete on one of the UNKNOWN instances. These results lend support to the idea that binary search heuristic is significantly faster than linear search.

We additionally evaluated the performance of the binary search heuristic for the integer solver (Z3str2). We designed a benchmark suite containing 205 handcrafted test-cases involving operations on large strings. The constraints involved use various high-level string operators supported by Z3str2, such as `Length`, `Concat`, `IndexOf`, `Substring`, `StartsWith`, `Replace`, etc. We also translated the constraints to equivalent versions in the CVC4 input language and compared the performance against CVC4 version 1.5.

The results of the comparison are presented in Figure 6. The SAT and UNSAT rows denote the number of (correct) SAT and UNSAT results from each solver. Notably, Z3str2 did not time out on any instances when the binary search heuristic was used, whereas the linear search timed out on 31 instances and CVC4 timed out on 60 instances. In both search modes, Z3str2 is able to detect overlapping arrangements and report UNKNOWN, preventing a potential infinite loop. CVC4 timed out on both of these instances. The results demonstrate that the binary search heuristic is useful for integer-domain string solvers as well, providing a speedup of over two orders of magnitude compared to CVC4 and the linear-search version of Z3str2.

VI. Related Work

While we are unaware of existing solver engines for a *combined* QF first-order many-sorted theory of strings and bit-vectors, considerable progress has been made in developing solvers that model strings either natively or as bit-vectors. We survey some of the main results in this space.

String solvers: Zheng et al. [42] present a solver for the QF many-sorted theory T_{wlr} over word equations, membership predicate over regular expressions, and length function, which consists of the string and numeric sorts. The solver algorithm features two main heuristics: (i) sound pruning of arrangements with overlap between variables, which guarantees termination, and (ii) bi-directional integration between the string and integer theories. S3 [33] is another solver with similar capabilities. S3 reuses Z3str's word-equation solver, and handles regex membership predicates via unrolling. CVC4 [23] handles constraints over the theory of unbounded strings with length and regex membership. It is based on multi-theory reasoning backed by the DPLL(T) architecture combined with existing SMT theories. The Kleene operator in regex membership formulas is dealt with via unrolling as in Z3str2. Unlike Z3strBV, these techniques all model string length as an integer, which makes it difficult to reason about potential overflow. In particular, none of these approaches combines strings and bit-vectors into a unified theory.

Another approach is to represent string variables as a regular language or a context-free grammar (CFG). JSA [10] computes CFGs for string variables in Java programs. Hooimeijer et al. [18] suggest an optimization, whereby automata are built lazily. Other heuristics, to eliminate inconsistencies,

Z3strBV	SAT				UNSAT				TIMEOUT (20s)				UNKNOWN			
	#	T_{min}	T_{avg}	T_{max}	#	T_{min}	T_{avg}	T_{max}	#	T_{min}	T_{avg}	T_{max}	#	T_{min}	T_{avg}	T_{max}
Binary Search	98	0.060	0.172	2.667	9	0.047	0.081	0.320	0	0	0	0	2	0.051	0.085	0.118
Linear Search	72	0.060	0.097	0.618	9	0.061	0.111	0.415	27	20	20	20	1	0.072	0.072	0.072

Z3strBV	Total	
	#	Time(s)
Binary Search	109	17.7 (**1x**)
Linear Search	109	548.1 (**31x**)

Fig. 5. Performance Comparison of Search Heuristics.

	Z3str2-Bin	Z3str2-Lin	CVC4
SAT	169	138	126
UNSAT	34	34	19
UNKNOWN	2	2	0
Timeout	0	31	60
Total #	205	205	205
Total time (s)	42 (**1x**)	9570 (**229x**)	12015 (**264x**)

Fig. 6. Performance on string-integer benchmark suite. The columns "Z3str2-Bin" and "Z3str2-Lin" differentiate between two versions of Z3str2, one with the binary search heuristic and the second with the default linear search procedure.

are introduced as part of the Rex algorithm [38], [37]. To overcome the challenge faced by automata-based approaches of capturing connections between strings and other domains (e.g. to model string length), refinements have been proposed. JST [16] extends JSA. It asserts length constraints in each automaton, and handles numeric constraints after conversion. PISA [32] encodes Java programs into M2L formulas that it discharges to the MONA solver to obtain path- and index-sensitive string approximations. PASS [22], [21] combines automata and parameterized arrays for efficient treatment of UNSAT cases. Stranger extends string automata with arithmetic automata [40], [41]. For each string automaton, an arithmetic automaton accepts the binary representations of all possible lengths of accepted strings. Norn [4] relates variables to automata. Once length constraints are addressed, a solution is obtained by imposing the solution on variable languages. These solutions, similarly to Z3str2, offer model string length as an integral value, thereby failing to directly capture the notion of overflow. The S-Looper tool [39] addresses the specific problem of detecting buffer overflows via summarization of string traversal loops. The S-Looper algorithm combines static analysis and symbolic analysis to derive a constraint system, which it discharges to S3 to detect whether overflow conditions have been satisfied. While S-Looper is effective, it operates under a set of assumptions that limit its applicability (e.g. no loop nesting and only induction variables in conditional branches). Z3str2+BV, in contrast, is a general solution for system-level programs.

Bit-vector-based Solvers: Certain solvers convert string and other constraints to bit-vector constraints. HAMPI [20] is an efficient solver for string constraints, though it requires the user to provide an upper bound on string lengths. The bit-vector constraints that it generates are discharged to STP [13]. Kaluza [26] extends both STP and HAMPI to support mixed string and numeric constraints. It iteratively finds satisfying length solutions and converts multiple versions of fixed-length string constraints to bit-vector problems. A similar approach powers Pex [5] to address the path feasibility problem, though strings are reduced to integer abstractions. The main limitation of solvers like HAMPI is the requirement to bound string lengths. In our approach, there is no such limitation.

VII. Conclusion and Future Work

We have presented Z3strBV, a solver for a combined quantifier-free first-order many-sorted theory of string equations, string length, and linear arithmetic over bit-vectors. This theory has the necessary expressive power to capture machine-level representation of strings and string lengths, including the potential for overflow. We motivate the need for such a theory and solver by demonstrating our ability to reproduce known buffer-overflow vulnerabilities in real-world system-level software written in C/C++. We also establish a foundation for unified reasoning about string and bit-vector constraints in the form of a decidability result for the combined theory.

Future Work: While summary-based symbolic execution has been studied (e.g. as part of the S-Looper tool [39]), we are not aware of any previous work where SMT solvers directly support programming-language library functions declaratively as part of their logic. One recent application of a similar concept is discussed in [19], where models of design patterns are abstracted into a symbolic execution engine. Being able to perform a similar analysis at the level of individual library methods as part of a library-aware SMT solver can enhance library-aware symbolic execution such as demonstrated in that work. We intend to explore this idea further in the future to broaden its applicability beyond the current context.

VIII. Acknowledgements

We are deeply grateful to Yunhui Zheng of IBM Research for his collaboration on this paper and in the creation of the Z3strBV tool, and who due to submission limitations could not be named as a co-author on this work.

References

[1] Common Vulnerabilities and Exposures Database. https://cve.mitre.org/.

[2] Z3strBV constraint solver project. https://sites.google.com/site/z3strsolver/.

[3] Z3str2 Testcase Suite. https://github.com/z3str/Z3-str/tree/master/tests/.

[4] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.

[5] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 307–321, 2009.

[6] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[7] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In A. Juels, R. N. Wright, and S. D. C. di Vimercati, editors, *ACM Conference on Computer and Communications Security*, pages 322–335. ACM, 2006.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. 46(3):265–278, Mar. 2011.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, 2003.

[11] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, 2008.

[12] P. Ferrara, O. Tripp, and M. Pistoia. Morphdroid: Fine-grained privacy verification. In *Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, CA, USA, December 7-11, 2015*, pages 371–380, 2015.

[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, 2007.

[14] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.

[15] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. Rinard. Word equations with length constraints: what's decidable? In *HVC'12*, 2012.

[16] I. Ghosh, N. Shafiei, G. Li, and W.-F. Chiang. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, 2013.

[17] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[18] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 377–386, 2010.

[19] J. Jeon, X. Qiu, J. Fetter-Degges, J. S. Foster, and A. Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 156–167, New York, NY, USA, 2016. ACM.

[20] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.

[21] G. Li, E. Andreasen, and I. Ghosh. SymJS: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014.

[22] G. Li and I. Ghosh. PASS: String solving with parameterized array and interval automaton. In *9th International Haifa Verification Conference*, HVC '13, pages 15–31. 2013.

[23] 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 *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. 2014.

[24] G. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English transl. in Math USSR Sbornik 32 (1977).

[25] X. Qu and B. Robinson. A case study of concolic testing tools and their limitations. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*, pages 117–126. IEEE Computer Society, 2011.

[26] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.

[27] K. U. Schulz. Makanin's algorithm for word equations - two improvements and a generalization. In *Proceedings of the First International Workshop on Word Equations and Related Topics*, IWWERT '90, pages 85–150, London, UK, UK, 1992. Springer-Verlag.

[28] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[29] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[30] A. C. Sima. Efficient string processing in symbolic execution. Master's thesis, École Polytechnique Fédérale de Lausanne, 2015.

[31] S. Subramanian. Bit-vector Support in Z3-str2 Solver and Automated Exploit Synthesis. Master's thesis, University of Waterloo, 2015.

[32] T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, Oct. 2013.

[33] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, 2014.

[34] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 49–59, 2014.

[35] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97, 2009.

[36] O. Tripp, O. Weisman, and L. Guy. Finding your way in the testing jungle: a learning approach to web security testing. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 347–357, 2013.

[37] M. Veanes and N. Bjørner. Symbolic automata: the toolkit. In *Proceedings of the 18th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 472–477, 2012.

[38] M. Veanes, P. d. Halleux, and N. Tillmann. Rex: Symbolic regular expression explorer. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, pages 498–507, 2010.

[39] X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 188–198, New York, NY, USA, 2015. ACM.

[40] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: an automata-based string analysis tool for php. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157, 2010.

[41] F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 322–336, 2009.

[42] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 235–254, 2015.

[43] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, New York, NY, USA, 2013. ACM.