Integrating String Reasoning in Symbolic Execution of C Programs

Rachel Cleaveland and Clark Barrett

Stanford University, Stanford, CA, USA
{rcleavel,barrettc}@stanford.edu

Abstract. Symbolic execution is a powerful program analysis technique which explores new paths through a program by generating inputs that are guaranteed to exercise those paths. However, there is a lack of modern research on symbolic execution for programs that manipulate strings, especially for C programs. In this work, we develop an approach for symbolically executing string-manipulating C programs and implement it in a proof-of-concept tool called SymCC-str. SymCC-str is built on top of the concolic execution engine SymCC and the `smt-switch` SMT solving API, enabling it to make use of modern advancements in both symbolic execution and string solving. SymCC-str's novelty lies in its *hybrid* data representation, expressing symbolic data *both* as SMT bitvectors *and* strings in order to balance the performance and expressivity of these respective theories. Through this novel approach, SymCC-str achieves a median 9.9% increase in branch coverage (up to 990% in the best case) compared to SymCC on 6 real-world benchmark programs.

Keywords: Symbolic execution · Theory of Strings · SMT.

1 Introduction

Symbolic execution is a program analysis technique which enables the simultaneous exploration of multiple program paths by representing user-defined inputs symbolically. Constraints are added to these symbolic inputs to represent program execution and control flow, and these constraints can be solved to produce new inputs that are guaranteed to explore particular paths through the program.

The efficacy of a symbolic execution engine is contingent upon its ability to accurately convert a program's entire semantics into constraints. As such, any symbolic execution engine aiming to completely encode the space of program behaviors must be able to accurately represent strings, which are a fundamental datatype in modern programming languages. Strings are often used to represent user-provided program data [4]. After being read into the program, these inputs may be subject to string-specific manipulations and comparisons, including regular expression membership checks, comparison of individual characters or substrings, and deletion or replacement of substrings.

Vulnerabilities enabled by the use of such string manipulations can have major consequences for the security of the programs that employ them. Notable

S. Junges and G. Katz (Eds.): TACAS 2026, LNCS 16506, pp. 151–172, 2026.
https://doi.org/10.1007/978-3-032-22749-2_8

string-enabled vulnerabilities include XSS attacks [23] and SQL injection attacks [25], both of which arise from malicious string inputs to a program that allow untrusted users to execute code or perform database queries, respectively. Symbolic execution engines that are overly conservative when reasoning about strings can miss finding such critical vulnerabilities during their analyses.

Although reasoning about strings in symbolic execution is not a new topic [5, 7, 21, 33, 39, 43, 45, 48, 49], previous work suffers from several limitations.

First, the symbolic execution engines in previous work were mainly designed for the analysis of Java programs, meaning they are not applicable to programs in other languages such as C. The importance of analyzing C code in particular is evidenced by large-scale projects like Google's OSS-Fuzz [47], which performs continuous fuzzing on open-source C/C++ projects.

Second, much of this work is now out-of-date, in that it was done when specialized solvers for string constraints were either nonexistent or much more primitive than they are now [27]. In fact, a main focus of prior work is the creation (often from scratch) of basic string solving capabilities needed for handling the symbolic execution task [8, 21, 39, 45, 49]. However, since then, there have been vast improvements in general-purpose string solvers [15–17, 29–32, 40–43]. In particular, many SMT solvers now include efficient and powerful string theory solvers. These advancements have been crucial to large-scale deployments of string solving in industrial applications [6, 44].

Third, prior tools have mostly been evaluated on small test programs limited to a few hundred lines of code [5,33,39,49]. Such evaluations provide little insight on the applicability of these tools to real-world programs, which may contain tens to hundreds of thousands of lines of code. Thus, the question of scalability has largely been left unanswered.

Motivated by these limitations, this paper revisits the topic of symbolic execution for string-manipulating programs in order to answer the following questions: What are the modern bottlenecks to a string-based symbolic execution approach, and what are ways to address these? We develop our approach into a proof-of-concept symbolic execution engine, which we call SymCC-str, targeting string-manipulating C programs. SymCC-str uses `smt-switch` [35], a generic SMT solving API that allows it to take advantage of modern advancements in string solving. Furthermore, our design of SymCC-str is motivated by the following two performance-related insights that enable SymCC-str to scale to larger benchmarks.

Our first insight is that programs often perform operations that do not require the full expressivity of the theory of strings to encode. Thus, in SymCC-str we implement a novel "hybrid" data representation, whereby input data is represented as *both bitvectors and strings*. SymCC-str always first attempts to solve problems using the bitvector representation, and only if that fails to produce a new program input does it attempt to solve the problem using the string representation. This enables significantly better performance than a string-only representation would, while generating more new inputs than would be possible using only a bitvector representation.

Our second insight is that back-end string solvers, despite benefiting from vast improvements in the past decade, are still a limiting factor during execution. To address this, we first develop and implement novel string assertion rewrites and a custom string inequality solving pass within SymCC-str to bolster the solver. However, even with these contributions, the assertions often eventually become too large and complex for the string solver to solve. Therefore, we enhance this with a mechanism that simply turns off the string solver when it stops generating new inputs.

These innovations enable SymCC-str to analyze large benchmarks—with sizes ranging between 550 and 68,000 lines of code—while improving branch coverage by a median of 9.9% (up to 990% in the best case) over SymCC. We summarize our contributions as follows:

- We develop SymCC-str, the first symbolic execution engine for string-manipulating C programs that uses a novel hybrid data representation, representing symbolic data as both bitvectors and strings.
- We perform a case study on 6 real-world programs from the OSS-Fuzz project suite, demonstrating SymCC-str's performance on large benchmarks.
- We identify new string assertion rewrites and implement a novel string inequality-solving pass within SymCC-str in order to performantly handle constraints containing string comparisons.
- We package the solver queries generated from our case study into a challenging new set of string benchmarks for the SMT community.

2 Background

2.1 Symbolic Execution

Symbolic execution is a technique for representing and reasoning about multiple possible program executions by: (*i*) encoding inputs as symbolic variables; (*ii*) generating constraints over these variables to represent the execution of the program; and (*iii*) solving these constraints to produce real inputs to the program that follow the execution path encoded by the constraints.

Symbolic execution engines come in many varieties, differing, for example, in their execution model (are programs-under-test interpreted [13, 18, 50] or dynamically run [14, 22, 37, 38]), their memory model (are symbolic addresses concretized [12]), or the types of programs they accept (source code [13, 38] or binaries [14, 18, 22, 50]). One thing that all engines have in common, however, is that they suffer from the *path explosion problem*, which limits their scalability. A number of approaches have been explored to address this challenge.

One such technique is concolic execution, in which a concrete input is dynamically run through a program, and a symbolic query is solved for each branch encountered to try to generate an input that takes the opposite branch from the one taken during the execution. In this way, concolic execution does not explore the entire program at once, and new inputs can be fed back into the engine to

```
1   mov   %r14,%rdi
2   mov   0x20(%rsp),%rsi
3   call  401180 <_sym_build_add@plt> # Symbolically add one to x
4   mov   %rax,-0xc8(%rbp)
5   add   %r12,1 # Concretely add one to x
6   mov   %r12,-0xcc(%rbp)
7   mov   $0x2,%rdi
8   call  401130 <_sym_build_integer@plt>
9   mov   -0xc8(%rbp),%rdi
10  mov   %rax,%rsi
11  call  4010f0 <_sym_build_signed_greater@plt> # Make branch constraint
12  mov   %rax,-0xe8(%rbp)
13  mov   -0xcc(%rbp),%eax
14  cmp   $0x2,%eax # Evaluate concrete branch condition
15  setg  %al
16  mov   %al,-0xd9(%rbp)
17  mov   -0xe8(%rbp),%rdi
18  mov   %al,%esi
19  mov   $0x1bfe9f0,%edx
20  call  4010b0 <_sym_push_path_constraint@plt> # Generate new input
21  mov   -0xd9(%rbp),%al
22  test  $0x1,%al
23  jne   40171a <main+0x48a> # Take concrete branch
```

Fig. 1: (Simplified) Assembly code after SymCC instrumentation of one line of C code (`if (x + 1 > 2) {...}`). Black instructions are original program instructions, and blue instructions are those inserted by SymCC. Register %`r14` initially holds the symbolic expression for `x`, %`r12` the concrete expression for `x`, and `0x20`(%`rsp`) a symbolic expression representing the constant `1`.

iteratively explore more paths. This technique requires fewer resources than the purely symbolic approach and easily couples with fuzz testing [51].

Another optimization, introduced in the concolic engine SymCC [38], is to compile the symbolic execution functionality directly into the program-under-test. SymCC is implemented as an LLVM pass and runtime library. It instruments the LLVM IR of a program-under-test to directly insert calls to functions within its runtime library that construct and solve symbolic path constraints.

To understand this instrumentation process, consider Fig. 1, which shows the original assembly for the line of code `if (x + 1 > 2)` in black and the assembly added by SymCC in blue. Each time a concrete computation is performed (e.g., the addition of 1 to register %`r12`, which contains `x`, on line 5), a corresponding symbolic operation is done (e.g., the call to function `_sym_build_add` on line 3). Each time a concrete conditional branch is executed (e.g., line 23), the corresponding symbolic condition is generated (e.g., line 11). Its negation is then sent to the solver, and the corresponding query is solved to obtain a new input that takes the opposite path from the concrete input, if one exists (e.g., line 20). SymCC exhibits superior performance compared to engines that interpret, rather than natively execute, their programs-under-test.

Data Representations Developers of symbolic execution engines make different decisions about how to represent symbolic data and memory. Most engines represent symbolic data either as bitvectors (e.g., [38,50]) or arrays of bitvectors (e.g., [13,19]). Both representations have drawbacks when it comes to encoding strings. Bitvectors are easily mutable, but must be of a fixed length. Strings

in C are not required to be any particular length, so it is suboptimal to encode them using fixed-length variables. Arrays do not have a fixed length; however, repeated array updates result in prohibitively large assertions. Both data representations lack native support for important string-specific functions like `indexOf` and `strcmp` (i.e., lexicographic comparison) [11].

2.2 String Solvers

String solvers are used to determine the satisfiability of constraints involving string variables and constants [3]. If a string solver returns `sat`, then the set of constraints has a solution, i.e., an assignment of concrete strings to variables that makes each constraint evaluate to true. On the other hand, an `unsat` result means that no such assignment exists.

Early developments in string solving were largely motivated by the need for test-case generation and verification in the realm of web security [3, 26, 36, 39, 45, 48]. However, these initial solvers suffered from various limitations. Many supported only strings with fixed lengths [28, 45, 46], and many did not support common operations (e.g. `contains`, `indexof`). Moreover, solvers capable of reasoning about unbounded-length strings often relied on reductions to automata, which historically limited their performance or expressiveness [4]. These weaknesses limited the ability of such early solvers to reason about real string-manipulating programs.

However, in the past decade, string solvers have improved significantly, both in their performance and in their expressivity [15, 17, 20, 30, 52]. Many of these improvements were motivated by benchmarks generated from symbolic execution applications [29, 40, 41, 43] (e.g., the addition of extended string functions like `indexof`, as well as the support for string-to-code point conversion [40, 41, 43]). Despite the fact that support for a full set of program-level string operations leads to undecidability [11], modern string solvers are often capable of solving constraints generated by complex real-world programs. This is evidenced by applications like Zelkova, which calls string solvers billions of times a day to secure cloud data at Amazon Web Services [6, 34, 44].

3 Related Work

As mentioned, string-based attacks like SQL injection [25] and XSS [23] motivated earlier work on symbolic execution of string-manipulating Java/JavaScript programs [8, 27, 36, 39, 48, 49]. However, our work is (to the best of our knowledge) the first to address string-manipulating C programs. Reasoning about strings in C presents unique challenges. For one, strings in C are not a built-in type, instead being represented as pointers to null terminated memory regions. Accounting for this null terminator complicates the representation of symbolic memory. Another complication is that C is not a strongly-typed language, meaning the same data can be interpreted as multiple types (e.g., data can be read into the program

```
1 char buffer[10000];
2 memset(buffer,0,10000);
3 read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
4 if (strlen(buffer) == 9999) { /* Code A */ }
5 if (buffer[3] == 'a') { /*  Code B */ }
```

Listing 1.1: Code snippet showing branches that SymCC-str can cover, given an empty initial input to `stdin`, but SymCC cannot.

as a string, and then converted to an integer via the `atoi` function). Addressing this required minimizing type conversions, which we discuss in §5.1.

Another difference is that the engines in [8, 21, 39, 45, 48, 49] all implement custom string solvers in their back ends. In other words, the string solver itself (or an extension to a custom string solver from prior work) is a primary contribution. In contrast, SymCC-str uses `smt-switch` [35] as its back-end solver API, allowing it to easily connect to any SMT solver that supports the SMT-LIB [10] standard. Thus, where prior work focused on developing new string solvers, this work seeks to scale symbolic execution of string programs to real-world software, uncovering the limitations of modern string solvers in the process.

Finally, on the topic of scalability, most prior work was limited to handling benchmarks of only tens or hundreds of lines of code [5, 21, 33, 39, 45, 49]. In contrast, our benchmarks range from 550 to 68,000 lines of code, representing a major advancement in scalability.

4 SymCC-str Overview

In this section, we introduce SymCC-str, a concolic execution engine for string-manipulating C programs designed to answer two main research questions: what bottlenecks still exist for symbolic execution using modern string solvers, and how can these be addressed? SymCC-str is derived from SymCC [38], with the key change that symbolic data is represented *both* as strings *and* bitvectors. We explain the rationale for this below.

4.1 Drawbacks of a Bitvector-Only Approach

The theory of fixed-size bitvectors includes many bitwise logical and arithmetic operations, like `bvand` and `bvmul`, making this theory a natural choice to represent low-level data during concolic execution. However, there are drawbacks to the use of bitvectors when it comes to symbolically representing strings.

Symbolic Input Lengths SymCC replaces the concrete inputs during execution with symbolic bitvector variables, whose length must be provided upon their creation. SymCC simply assigns these variables the length of their corresponding concrete inputs. As a result, it is unable to generate test cases with inputs that are different in length from the original concrete input.

For example, consider the code in Listing 1.1, and suppose the concrete input provided is the empty string. SymCC cannot generate a test case that explores

`Code B` on line 5 because it only considers symbolic inputs of length 0. It is possible to get around this limitation by manually setting the size of a concrete input, but manually exploring the large space of potential sizes is expensive, and in general, it is impossible to explore all of it. In contrast, string variables can have any length—the size does not have to be chosen up front. By using a string representation, new inputs can be generated whose lengths are different from the concrete input (e.g., the input `AAAa` could be generated to explore `Code B`).

Hooked Functions To efficiently execute certain `libc` functions symbolically, SymCC replaces calls to these functions during compilation (e.g., `malloc`) with custom runtime functions, or *hooked functions*, which efficiently encode the execution of the original `libc` functions. However, SymCC cannot do this for `libc` string functions.

Consider again Listing 1.1. SymCC does not hook the `strlen` on line 4 because the bitvector encoding of a string always has a fixed length. Thus, SymCC has no useful symbolic expression to assign as its symbolic result. The execution can proceed in one of two ways. If `libc` is not compiled with SymCC, then `strlen` will be executed fully concretely, and its return value will always match the concrete execution value (i.e., it will not be symbolic). Alternatively, if `libc` is compiled with SymCC, then the source code of `strlen` will be symbolically executed. The code returns the value of a counter that is incremented in a loop until the input's null terminator is reached. Within the loop, SymCC can generate test cases with different placements of the null terminator that thus produce different values for `strlen`. However, this choice is guided by the branches of the loop inside the `strlen` function, not by the branch condition on line 4. Thus, it could take many runs of SymCC to eventually generate an input correctly formatted to explore `Code A`.

With a string representation of symbolic data, however, calls to `strlen` can be hooked by simply applying the string length operator to the symbolic input (i.e., `len(buffer)`). Solving the query generated by the branch condition on line 4 could then produce a new input exploring `Code A` in a single run.

4.2 Hybrid Data Representation

Our preliminary testing of SymCC-str revealed that the string-only representation of data also has significant drawbacks, mostly in terms of prohibitively slow performance (despite modern string solving advancements—see §6). We thus propose a "hybrid" approach, wherein *both* bitvector *and* string representations of data are tracked for each symbolic variable.

SymCC-str maintains two separate sets of symbolic expressions and assertions during execution. Wherever SymCC would create a symbolic expression, SymCC-str creates two expressions: one based on bitvectors and one based on strings. Similarly, wherever SymCC would assert a formula, SymCC-str asserts two formulas, one using bitvectors and the other using strings, within two separate SMT solver instances.

The advantages of our hybrid approach are two-fold. First, in practice, many queries do not require the additional expressivity of the theory of strings to be

	sat	unsat	unknown	timeout	Time (s)
cvc5	**112**	**95**	0	**93**	**1220**
z3	65	92	0	143	1540
ostrich	35	81	1	183	3418
z3-noodler	29	90	0	181	1683

Table 1: String solver comparison on the first 300 benchmarks of our benchmark set [1] (see §7), 50 per case study.

solved. For example, the condition `buffer[3] == 'a'` on line 5 of Listing 1.1 can easily be satisfied *if* the initial input is at least four bytes long. Therefore, in our hybrid approach, we always let the bitvector solver make the first solving attempt. Only if the bitvector solver cannot generate a new input (i.e., returns `unsat`) do we invoke the string solver. This enables the best of both worlds: the performance of the bitvector solver in many cases, and the additional expressivity of the string solver when necessary.

The second advantage of the hybrid approach is that it enables SymCC-str to selectively disable the string solver. We observed that the string solver is very useful early on in larger programs, but it eventually becomes unable to solve the assertions in a reasonable amount of time. SymCC-str thus disables the string solver once its utility diminishes beyond a certain point (see §6.3 for details).

4.3 String Solver Extensions

One of the goals of this work was to assess the capabilities of modern string solvers and find opportunities for their improvement. By analyzing the SMT queries generated by SymCC-str, we identified several specific opportunities for solver improvement, and implemented workarounds for them in SymCC-str directly. In our work and evaluation (see §6), we used cvc5 [9] as a back-end solver, as it showed the most promising performance in preliminary testing. This was confirmed after running cvc5, z3 [20], ostrich [16], and z3-noodler [17] on a subset of our benchmark set [1] (see §7), showing cvc5 to be the best performer for our application (Tab. 1). We hope the developers of cvc5 will incorporate these optimizations in future versions of the solver. We expect them to be useful for other string solvers as well, though we leave that evaluation to future work.

Our first optimization focuses on patterns of assertions like the following:

$$\texttt{substr}(\texttt{S}, 0, 9) < \text{“}\texttt{Document}\text{”} \quad \wedge \quad \text{“}\texttt{Feature}\text{”} < \texttt{substr}(\texttt{S}, 0, 8)$$

Because it is always true that $\texttt{substr}(\texttt{S}, 0, 8) \leq \texttt{substr}(\texttt{S}, 0, 9)$, these assertions create a cycle of inequalities, two of which are strict, and are thus `unsat`. However, cvc5 times out on these assertions (with the 10-second timeout we used in SymCC-str). Further investigation revealed two limitations of cvc5's string solver. First, cvc5 cannot quickly deduce the relationship between “`Document`” and “`Feature`”, meaning that even when the substrings in the two assertions are exactly the same, cvc5 still cannot solve the problem in a reasonable amount of time. Second, cvc5 cannot deduce that for any trio of strings $\texttt{S}_1$, $\texttt{S}_2$, and $\texttt{S}_3$ such that $\texttt{S}_1 = \texttt{S}_2 +\!+ \texttt{S}_3$, $\texttt{S}_2 \leq \texttt{S1}$. That is, a string is always greater than or equal (lexicographically) to any of its prefixes.

$$
\begin{array}{rcll}
\texttt{indexOf}(\texttt{con}(A,B),c,0) & \rightarrow & -1 & \texttt{if } \texttt{len}(c)=1 \\
 & & & \quad \wedge\ \texttt{indexOf}(A,c,0)=-1 \\
 & & & \quad \wedge\ \texttt{indexOf}(B,c,0)=-1 \\
\texttt{indexOf}(\texttt{con}(A,B),c,0) & \rightarrow & \texttt{indexOf}(A,c,0) & \texttt{if } \texttt{indexOf}(A,c,0)\neq -1 \\
\texttt{indexOf}(\texttt{con}(A,B),c,0) & \rightarrow & \texttt{len}(A)+\texttt{indexOf}(B,c,0) & \texttt{if } \texttt{len}(c)=1 \\
 & & & \quad \wedge\ \texttt{indexOf}(A,c,0)=-1 \\
 & & & \quad \wedge\ \texttt{indexOf}(B,c,0)\neq -1 \\
\texttt{substr}(A,x,y) & \rightarrow & A & \texttt{if } x=0 \wedge \texttt{len}(A)=y \\
\texttt{substr}(\texttt{con}(A,B),x,y) & \rightarrow & \texttt{substr}(A,x,y) & \texttt{if } x+y\leq \texttt{len}(A) \\
\texttt{substr}(\texttt{con}(A,B),x,y) & \rightarrow & \texttt{substr}(B,x-\texttt{len}(A),y) & \texttt{if } x\geq \texttt{len}(A) \\
\texttt{substr}(\texttt{substr}(A,x,y),a,b) & \rightarrow & \texttt{substr}(A,x+a,b) & \texttt{if } a\geq 0\wedge x\geq 0\wedge a+b\leq y \\
\texttt{substr}(\texttt{substr}(A,x,y),a,b) & \rightarrow & \texttt{substr}(A,x+a,y-a) & \texttt{if } a\geq 0\wedge x\geq 0\wedge y\leq a+b \\
\texttt{len}(\texttt{substr}(A,x,y)) & \rightarrow & y & \texttt{if } \texttt{len}(A)\geq x+y\wedge y\geq 0\wedge x\geq 0 \\
\texttt{len}(\texttt{ITE}(cond,A,B)) & \rightarrow & \texttt{len}(A) & \texttt{if } \texttt{len}(A)=\texttt{len}(B) \\
A==B & \rightarrow & \texttt{false} & \texttt{if } \texttt{len}(B)=1\wedge !\texttt{contains}(A,B)
\end{array}
$$

Fig. 2: String-based simplifications implemented in SymCC-str.

Based on these observations, we implemented a novel string inequality refutation procedure. Our implementation is inspired by a similar procedure used in CVC4's bitvector solver [24]. Each time a string comparison constraint is asserted, it is added to a graph, where each string expression is a node, and each less than (less than or equal to) inequality is represented as a strong (weak) edge between the corresponding nodes. We also add tautological edges between nodes as follows: we add edges between all string constants based on their concrete lexicographical ordering (e.g., "`Document`" < "`Feature`"), and we add edges between strings and their prefixes. We immediately return `unsat` (without calling the solver) if we detect a cycle with at least one strong edge. Otherwise, we call the back-end solver as usual.

Our second optimization is based on the observation that keeping assertions simple allows the string solver to be useful for longer. However, there are many simplifications that can be performed only if some additional information is deducible from the existing assertions. For example, $\texttt{len}(\texttt{ITE}(cond,A,B))$ can be rewritten to $\texttt{len}(A)$, for strings A and B, if $\texttt{len}(A)=\texttt{len}(B)$ is known to be true. To implement such *conditional rewrites*, SymCC-str queries the solver with the negation of the rewrite condition using a very small timeout (we call this a "mini-check"). If the solver returns `unsat`, then the condition holds, and the rewrite can be performed.

Fig. 2 shows the conditional rewrites we have implemented in SymCC-str. Each has been separately verified as a sound lemma using cvc5. We also implemented several arithmetic simplifications to reduce the number of `ITE`s needed when checking for overflow and implementing sign extension.

5 Implementation

SymCC-str is implemented on top of SymCC. It consists of an LLVM compiler pass, written in 2,500 lines of code, and a runtime library, written in 10,000

```
1  #define isspace(c) __isctype((c), _ISspace)
2  #define __isctype(c, type) \
3    ((*__ctype_b_loc ())[(int) (c)] & (unsigned short int) type)
4
5  /* Strip whitespace chars off end of string. */
6  static char* ini_rstrip(char* s)
7  {
8    char* p = s + strlen(s);
9    while (p > s && isspace((unsigned char)(*--p))) *p = '\0';
10   return s;
11 }
12
13 int ini_parse_stream(...) {
14   ...
15   start = ini_rstrip(start);
16   if (strchr(";#", *start)) { /* Line comment */ }
17   ...
18 }
```

Listing 1.2: Code snippet taken from the `inih` benchmark program, in which data is treated both as a string and as a bitvector.

lines of code. 1,500 lines of the compiler and 2,000 lines of the runtime come from SymCC, and the rest implement our unique approach. The runtime is split into the bitvector and string runtimes, which treat symbolic expressions as bitvectors and strings, respectively. The new code in the runtime comes primarily from string-based implementations of the entire SymCC symbolic execution framework (creating expressions, pushing to the solver, solving constraints, and generating output files, 3,000 LOC), the symbolic memory store (see §5.2, 1,300 LOC), and new hooks and helper functions for common string functions plus string-based implementations of existing function hooks (1,500 LOC). In this section, we discuss some noteworthy implementation choices within SymCC-str.

5.1 Mixed String-Integer-Bitvector Constraints

In C programs, the same data may be interpreted in different ways. Consider Listing 1.2, a (condensed) code snippet from `inih`, a small `.ini` file parser used in our evaluation (see §6). On line 15, the input string `start` is stripped of any whitespace characters as defined by `libc`'s `__isctype` function, and then `strchr` is called on the result. However, `__isctype` performs a bitwise-and operation on its input, which is not natively defined in the theory of strings. Thus, we require multiple symbolic representations of the input to encode all of its occurrences.

In general, expressions may mix bitvector, string, and integer operations. Converting back and forth between different representations hurts performance because solvers struggle with such conversions. Therefore, we limit the number of conversions we perform in two ways. First, we replace calls to these `__isctype`-based functions, which introduce unavoidable type conversions, with implementations that do not. In Listing 1.2, for example, we would replace `isspace` with an implementation that uses integer comparison of the input character. This can be symbolically represented using code-point conversion, natively defined in

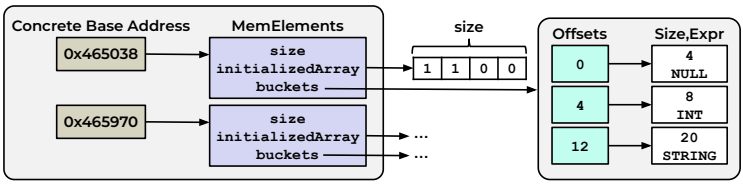


Fig. 3: SymCC-str's symbolic memory store.

the theory of strings. This process should be easily automatable using front-end tools; we leave such automation to future work. Second, when building symbolic expressions themselves, we attempt to use operations from the same theory as the inputs whenever possible. For example, consider a scenario where bit-shifting is performed on an integer-typed symbolic expression. A naive approach would convert the integer to a bitvector, and then bit-shift the converted expression. However, in SymCC-str, we first check if the second argument is a constant, and if so, we perform the bit-shift using integer multiplication or division to avoid the type conversion. Future work could explore more ways to reduce the number of conversions. For example, taint tracking could be used to determine whether an input is used in more string or arithmetic expressions in the program and assign its type at runtime accordingly.

5.2 Symbolic Memory Store

SymCC does not support symbolic addressing. If an array is accessed at a symbolic offset, SymCC will use the concrete array entry accessed instead of considering all possible entries that could be accessed. In SymCC-str, however, we deemed symbolic addressing to be important in order to leverage the fact that symbolic strings do not have a fixed length. For example, performing a function like `strcat` on strings A and B could require placing string B at a symbolic memory address, as its location depends on the (possibly symbolic) length of A.

Our symbolic memory store maps base memory addresses to symbolic memory elements as is done in KLEE [13]. Our symbolic memory elements are designed to support symbolic offsetting. Specifically, when memory regions are represented using strings, reading the memory at a symbolic offset is as simple as returning the substring starting at the symbolic offset of the string in the memory store. Writing to the memory store involves replacing the portion of the stored string starting at the symbolic offset with the new string.

As mentioned above, we wish to avoid conversions when possible. Thus, we use a memory model that supports fine-grained heterogeneity. Each memory region is broken up into a map of concrete offsets, each of which can be of type NULL (i.e., concrete values only—no symbolic representation), String, Int, or Bitvector. However, if a read or write to a symbolic offset is done on a region that is split in this way, then we convert the entire region into a single string (by converting each subregion to a string and then concatenating all subregions) and then index into that string.

inih	550 LOC	INI parser	libspectre	4670 LOC	Postscript parser
libtasn1	30450 LOC	ASN.1 codec	libyaml	6760 LOC	YAML parser
openjpeg	68415 LOC	JPEG codec	speex	2817 LOC	Audio file codec

Table 2: Lines of code (LOC) and description of each benchmark.

5.3 Representing Uninitialized Memory

When SymCC-str encounters calls to C functions like `read`, `fread`, and `fgets`, it creates new symbolic variables to represent the input string. Typically, these strings are placed in a buffer that may be much larger than the value read. The rest of the buffer often remains uninitialized. This could be modeled by assigning a string variable to the full buffer and then writing over it. However, there are several drawbacks to this approach. For one, this introduces extra variables, which makes the problem harder to solve. Second, if the program ever does access uninitialized memory, the solver will try to use these variables to satisfy the path constraints, but since uninitialized memory is not actually controllable, this can produce false positives. This can be avoided by checking whether memory reads access uninitialized memory, but keeping track of which memory in a region is initialized versus uninitialized is difficult if the region is a concatenation of variables, some of which are implicitly uninitialized.

Due to these drawbacks, SymCC-str treats uninitialized memory as concrete, and does not assign it symbolic values. When an input string is read into an uninitialized buffer, this input string alone is placed into the memory store, regardless of whether it takes up the whole memory region. We thus avoid unnecessarily tracking extra symbolic variables and can easily check whether a memory read accesses uninitialized memory.

6 Evaluation

We evaluate SymCC-str on 6 programs from the OSS-Fuzz project suite: `inih`, `libspectre`, `libtasn1`, `libyaml`, `openjpeg`, and `speex`, summarized in Table 2. Each was selected for its use of common string library functions, like `strcmp` and `strlen`. Our evaluation is designed to answer the following research questions:

- **RQ1:** How does "hybrid" SymCC-str compare to running the bitvector back end only in terms of branch coverage, a customary evaluation metric for symbolic execution engines?
- **RQ2:** How do "pure-bv", "pure-string", and "hybrid" SymCC-str compare on a single run in terms of runtime and number of new test cases generated?
- **RQ3:** At what point is the string back end no longer useful (i.e., what is a reasonable trigger for disabling the string back end)?
- **RQ4:** How effective are the inequality solver and the conditional rewrites?

Recall that SymCC-str performs concolic execution, so it runs a single concrete test case and uses it to generate new test cases that diverge from the given test case at a single branch. In our experiments, we run SymCC-str either once

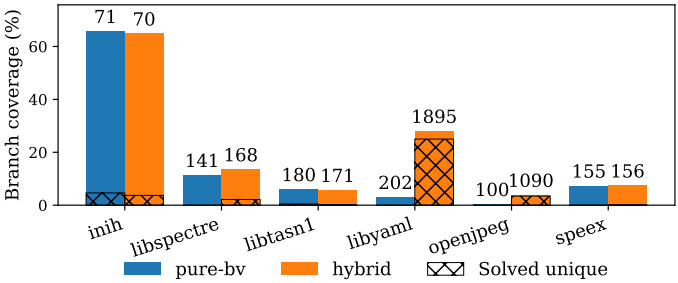


Fig. 4: Branch coverage of pure-bv and hybrid SymCC-str, repeatedly running each benchmark with the newly generated inputs for 12 hours. The absolute number of branches covered is given above each bar.

(i.e., a "single-run"), or in a loop which repeatedly feeds the newly generated test cases of each run back into the tool for 12 hours (i.e., a "multi-run").[1]

Note that we made the following minor changes to some benchmarks: we manually replaced re-implementations of common string functions with their `libc` counterparts (e.g., `dsc_strncmp` in `libspectre` with `strncmp`), and calls to the `__isctype` macros described in §5.1 with implementations that do not introduce bitwise operators. This was done in order to leverage our special hooked functions (§4.1) in the former case, and to minimize type conversions (§5.1) in the latter case so that SymCC-str can better demonstrate its potential. The automatic replacement of such functions and macros is a topic for future work.

6.1 RQ1: Branch Coverage of Pure-BV and Hybrid SymCC-str

In this experiment, we performed multi-runs of pure-bv SymCC-str and hybrid SymCC-str. We gave each single run a timeout of 5 minutes, and we disabled the string solver when 20% or fewer mini-checks succeeded (see §6.3).

The results are shown in Fig. 4. In most of the benchmarks, hybrid SymCC-str covers more branches than pure-bv SymCC-str, increasing the branch coverage by up to 990%, with a median increase of 9.9%. While four of the benchmarks show either a modest increase or a modest decrease in branch coverage, in the remaining two benchmarks, about 10x more branches are covered by hybrid than pure-bv SymCC-str. In these latter two cases, hybrid SymCC-str reaches a section of the code, after parsing, that the pure-bv engine was unable to reach. We conclude that hybrid SymCC-str, enabled by the string back end, was able to symbolically represent and solve enough branch conditions to successfully get through the parsing code, while pure-bv SymCC-str was not. This demonstrates that, at least in some cases, our hybrid approach can significantly improve on the pure-bv representation in terms of finding new paths to explore.

[1] We evaluate SymCC-str this way as opposed to coupling it with a fuzzer to isolate the work that SymCC-str is doing from any work that would be done by the fuzzer.

	pure-bv		pure-string		hybrid	
	`sat`	Runtime (s)	`sat`	Runtime (s)	`sat`	Runtime (s)
inih	701	7.3	106	17796	764	11364
libspectre	16911	17002	14	TO	386	TO
libtasn1	2741	102	140	TO	2763	TO
libyaml	2785	11105	56	TO	1636	TO
openjpeg	49	MO	63	TO	63	TO
speex	69	3.1	17	3962	80	3438

Fig. 5: Pure-bitvector, pure-string, and hybrid SymCC-str results on a single run with a representative input.

6.2 RQ2: Bitvector, String, and Hybrid Single-Run Comparisons

Next, we performed single runs of pure-bv, pure-string, and hybrid SymCC-str with a 12 hour timeout to better understand the tradeoffs of each version. The results are in Fig. 5. Pure-bv SymCC-str performs significantly better than pure-string SymCC-str due to its overall speed advantage. Pure-string SymCC-str times out on 4 benchmark programs, and on the other 2 benchmarks, it exhibits a 2,092x average increase in runtime. It also finds fewer `sat` instances, since it times out on many more queries than the pure-bv version. This is consistent with our observation that reasoning in the theory of strings is generally less efficient than in the theory of bitvectors.

The results for hybrid SymCC-str, however, are more encouraging. Across comparable benchmark results (i.e., those in which the pure-string and hybrid versions did not time out), hybrid SymCC-str finds 9.6% more `sat` instances on average than pure-bv SymCC-str and achieves a 32.0% average speedup in runtime compared to pure-string SymCC-str. These results are a promising first step toward trying to achieve the best of both approaches.

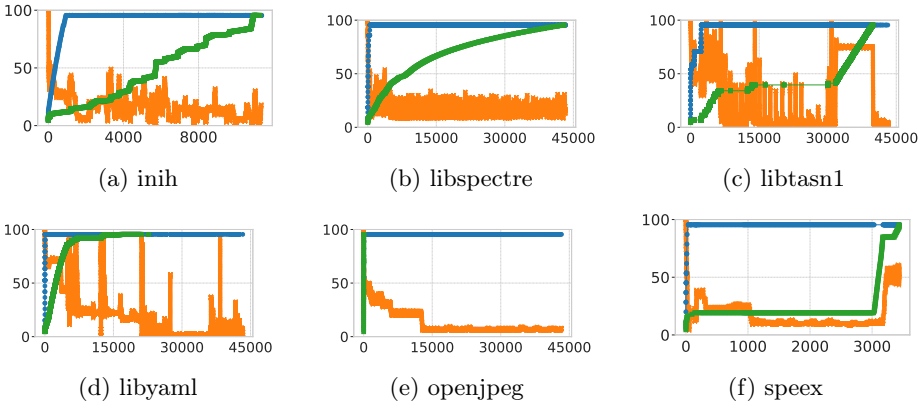
6.3 RQ3: Disabling SymCC-str's String Side

Despite its improvement in runtime over pure-string SymCC-str, the runtime of hybrid SymCC-str in the experiment above is still impractical. We next report on a set of experiments designed to answer two questions: first, is the percentage of mini-checks being solved a useful heuristic for deciding when to turn off the string solver (described in §4.2), and if so, what is the performance impact of disabling the string solver at the optimal mini-check percentage?

For the first experiment, we recorded the percentage of mini-checks being solved along with the percentage of total `sat` instances found by the string and bitvector sides over time. These results (Fig. 7) show that for most benchmarks, all of the `sat` instances found by the string solver are found very early in the execution. Also, we typically see a sharp decline in the mini-check percentage solved similarly early, demonstrating a promising correlation between the mini-check solved percentage and the ability of the string solver to find new `sat` instances. Based on this analysis, we decided to use the 20% mark as a trigger for disabling the string solver.

	Baseline		20%	
	`sat`	Runtime (s)	`sat`	Runtime (s)
inih	764	11364	764	1089
libspectre	386	TO	16928	27876
libtasn1	2763	TO	2763	8296
libyaml	1636	TO	2802	16719
openjpeg	63	TO	57	MO
speex	80	3438	79	30

Fig. 6: Number of `sat` instances and runtime for hybrid SymCC-str, with the string side disabled once less than 20% of the mini-checks are being solved.



(a) inih (b) libspectre (c) libtasn1

(d) libyaml (e) openjpeg (f) speex

Fig. 7: Percentage of mini-checks being solved (orange), percentage of string-side `sat` instances found so far (blue), and percentage of bitvector-side `sat` instances found so far (green) over time.

The next experiment measures the impact of this decision. The results are shown in Fig. 6. In terms of sat instances found, there is almost no cost to disabling the string solver, and in some cases there is a large benefit. Moreover, there is a significant speedup: we observed up to a 99% reduction in runtime. Thus, this optimization is essential to keep the runtime of hybrid SymCC-str practical, enabling the branch coverage improvements we saw in §6.1.

6.4 RQ4: Inequality Solver and Conditional Rewrites

In the last set of experiments, we compare the effectiveness of the inequality solver and the conditional rewrites. The results for disabling the conditional rewrites are shown in Fig. 8. In most of the cases, enabling the rewrites results in a tradeoff: many more solutions at the cost of a longer runtime. This makes sense, as the rewrites add thousands of extra checks with a small timeout, adding up over the course of the execution. However, the rewrites also make the assertions easier to solve. Excluding `libspectre`, we observe a 63.5% increase in `sat` instances and a 9.8% increase in `unsat` instances.

`libspectre` is a clear outlier—with rewrites enabled, it finds significantly fewer `sat` and `unsat` instances, skewing the averages to a 53.7% increase and a

	No conditional rewrites			With conditional rewrites		
	`sat`	`unsat`	Runtime (s)	`sat`	`unsat`	Runtime (s)
inih	35	928	15972	106	937	17796
libspectre	24	2072	TO	14	1337	TO
libtasn1	68	2315	TO	140	2556	TO
libyaml	50	1924	TO	56	2172	TO
openjpeg	63	14	TO	62	14	TO
speex	17	73	3617	17	88	3962

Fig. 8: Ablation study for conditional rewrites.

	No inequality solver		With inequality solver	
	`unsat`	Runtime (s)	`unsat`	Runtime (s)
libspectre	246	3008	247	3007
libtasn1	2508	TO	2566	TO

Fig. 9: Ablation study for the inequality solver.

3.0% decrease in `sat` and `unsat` instances, respectively. An investigation revealed that this is partially due to a particular internal cvc5 rewrite that gets applied when our conditional rewrites are enabled, but otherwise does not. The rewrite in question significantly degrades the performance of cvc5 on this benchmark. To confirm this, we ran the same experiment with this cvc5 rewrite disabled and found that `libspectre` finds 26 `sat` instances and 1361 `unsat` instances. Unfortunately, disabling this cvc5 rewrite has a mixed effect overall, so we ultimately decided not to disable it for our other experiments. However, it does suggest that there is room for improvement in cvc5's rewrite heuristics.

The results for disabling the inequality solver are in Fig. 9. We only ran this experiment on `libspectre` and `libtasn1`, as only these benchmarks produced assertions with string inequalities. For `libspectre`, this involved using a different initial input from the previous experiments to guide execution down a path that produced these inequalities. On average, the version with the inequality solver enabled finds 2.1% more `unsat` instances compared to the version without it.

7 Discussion and Future work

This work sheds light on the current state of string solvers and their usefulness in symbolic execution. While our results demonstrate much potential, there is also much room for improvement. As seen in Fig. 7, the string side of hybrid SymCC-str can only generate new inputs early on, and then mostly times out as the set of assertions grows. To make use of the string solver for whole programs, we need solvers that can handle the number and complexity of assertions generated by symbolic execution. To help with this goal, we have created a new set of string benchmarks generated by SymCC-str that we will submit for inclusion in SMT-LIB. These include many benchmarks that are highly challenging for current solvers [1]. We hope this benchmark set will motivate improvements in string solvers that will ultimately benefit the application of symbolic execution.

Our work also has already identified several promising directions for string solver developers. In particular we identified a set of conditional rewrites and a

custom string inequality solver that are both useful for symbolic execution and that could be implemented directly within string solvers.

Another observation regards *incremental solving*. Recall that each time a branch is encountered, its condition is negated and solved. In practice, this process involves repeatedly pushing, solving, and popping these negated branch conditions—i.e., incremental solving. During our experimentation, we noticed that cvc5 would sometimes time out on incremental assertions, but could quickly solve these same assertions non-incrementally. As a workaround, we ended up frequently resetting the solver instead of solving incrementally. However, this repeated resetting of the solver significantly increases the runtime; it would be better to improve the incremental mode of the solver to reduce such mismatches between incremental and non-incremental solving.

A final observation regards conversions. As discussed in §5.1, conversions often lead to prohibitively slow performance in SymCC-str. This suggests that improvements to how such conversions are handled could have a significant impact on the usefulness of solvers for symbolic execution.

While there is clearly a need for improving string solvers, future work should also explore ways to reduce the burden on the string solver within SymCC-str. For one, as discussed in §5.1, we would like to explore more optimal data representation strategies (e.g., using taint tracking to determine, for each symbolic variable, the representation choice that will minimize conversions). Additionally, we would like to improve our solver selection strategy. Currently, we query the bitvector solver first, and if that returns `unsat`, we always query the string solver after (§4.2). However, static analysis could be used to determine if the string solver is unlikely to produce a useful result for a certain branch, allowing its invocation to be skipped. If many queries to the string solver can be skipped, then hybrid SymCC-str's runtime could be greatly reduced.

8 Conclusion

We have presented SymCC-str, a concolic execution engine for string programs that builds on top of SymCC. SymCC-str uses a novel hybrid bitvector-string data representation in order to achieve the best of both worlds: the strong performance of the bitvector solver, with the additional expressivity of the string solver. With this hybrid representation and other optimizations, we achieve a 318% increase in branch coverage on real-world benchmarks over pure-bv SymCC-str.

Data-Availability Statement. Our artifact is available on Zenodo [2]. It contains the SymCC-str source code, experimental benchmarks and scripts, and instructions on regenerating the data, statistics, and graphs from the experiments.

References

1. https://doi.org/10.5281/zenodo.18262988
2. https://doi.org/10.5281/zenodo.18225511

3. Amadini, R.: A survey on string constraint solving. ACM Computing Surveys (CSUR) **55**(1), 1–38 (2021)
4. Amadini, R.: A survey on string constraint solving. ACM Comput. Surv. **55**(2), 16:1–16:38 (2023). https://doi.org/10.1145/3484198, https://doi.org/10.1145/3484198
5. Amadini, R., Andrlon, M., Gange, G., Schachte, P., Søndergaard, H., Stuckey, P.J.: Constraint programming for dynamic symbolic execution of javascript. In: Rousseau, L., Stergiou, K. (eds.) Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11494, pp. 1–19. Springer (2019). https://doi.org/10.1007/978-3-030-19212-9_1, https://doi.org/10.1007/978-3-030-19212-9_1
6. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: Bjørner, N.S., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018. pp. 1–9. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8602994, https://doi.org/10.23919/FMCAD.2018.8602994
7. Ball, T., Daniel, J.: Deconstructing dynamic symbolic execution. In: Dependable Software Systems Engineering, pp. 26–41. IOS Press (2015)
8. Bang, L., Aydin, A., Phan, Q., Pasareanu, C.S., Bultan, T.: String analysis for side channels with segmented oracles. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016. pp. 193–204. ACM (2016). https://doi.org/10.1145/2950290.2950362, https://doi.org/10.1145/2950290.2950362
9. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
10. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard – version 2.0. In: Proceedings of the 8^{th} International Workshop on Satisfiability Modulo Theories (SMT '10) (Jul 2010), http://theory.stanford.edu/~barrett/pubs/BST10.pdf, edinburgh, Scotland
11. Bjørner, N.S., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5505, pp. 307–321. Springer (2009). https://doi.org/10.1007/978-3-642-00768-2_27, https://doi.org/10.1007/978-3-642-00768-2_27
12. Borzacchiello, L., Coppa, E., Cono D'Elia, D., Demetrescu, C.: Memory models in symbolic execution: key ideas and new thoughts. Software Testing, Verification and Reliability **29**(8), e1722 (2019)
13. Cadar, C., Dunbar, D., Engler, D.R., et al.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
14. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: 2012 IEEE Symposium on Security and Privacy. pp. 380–394. IEEE (2012)

15. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Rümmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. Proc. ACM Program. Lang. **6**(POPL), 1–31 (2022). https://doi.org/10.1145/3498707, https://doi.org/10.1145/3498707
16. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362, https://doi.org/10.1145/3290362
17. Chen, Y., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síc, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 14570, pp. 24–33. Springer (2024). https://doi.org/10.1007/978-3-031-57246-3_2, https://doi.org/10.1007/978-3-031-57246-3_2
18. Chipounov, V., Georgescu, V., Zamfir, C., Candea, G.: Selective symbolic execution. In: Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep) (2009)
19. David, R., Bardin, S., Ta, T.D., Mounier, L., Feist, J., Potet, M.L., Marion, J.Y.: Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 653–656. IEEE (2016)
20. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
21. Ghosh, I., Shafiei, N., Li, G., Chiang, W.: JST: an automatic test generation tool for industrial java applications with strings. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. pp. 992–1001. IEEE Computer Society (2013). https://doi.org/10.1109/ICSE.2013.6606649, https://doi.org/10.1109/ICSE.2013.6606649
22. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: NDSS. vol. 8, pp. 151–166 (2008)
23. Grossman, J.: XSS attacks: cross site scripting exploits and defense. Syngress (2007)
24. Hadarean, L., Barrett, C., Jovanović, D., Tinelli, C., Bansal, K.: A tale of two solvers: Eager and lazy approaches to bit-vectors. In: Biere, A., Bloem, R. (eds.) Proceedings of the 26^{th} International Conference on Computer Aided Verification (CAV '14). Lecture Notes in Computer Science, vol. 8559, pp. 680–695. Springer (Jul 2014), http://theory.stanford.edu/~barrett/pubs/HBJ+14.pdf, vienna, Austria
25. Halfond, W.G.J., Viegas, J., Orso, A.: A classification of SQL injection attacks and countermeasures. In: Jr., S.T.R. (ed.) 2006 IEEE International Symposium on Secure Software Engineering, ISSSE 2006, Arlington, VA, USA, March 16 -17, 2006 (2006)
26. Hooimeijer, P., Weimer, W.: A decision procedure for subset constraints over regular languages. In: Hind, M., Diwan, A. (eds.) Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI

2009, Dublin, Ireland, June 15-21, 2009. pp. 188–198. ACM (2009). https://doi.org/10.1145/1542476.1542498, https://doi.org/10.1145/1542476.1542498

27. Kausler, S., Sherman, E.: Evaluation of string constraint solvers in the context of symbolic execution. In: Crnkovic, I., Chechik, M., Grünbacher, P. (eds.) ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 259–270. ACM (2014). https://doi.org/10.1145/2642937.2643003, https://doi.org/10.1145/2642937.2643003
28. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: A solver for word equations over strings, regular expressions, and context-free grammars. ACM Transactions on Software Engineering and Methodology (TOSEM) **21**(4), 1–28 (2013)
29. Li, G., Ghosh, I.: PASS: string solving with parameterized array and interval automaton. In: Bertacco, V., Legay, A. (eds.) Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings. Lecture Notes in Computer Science, vol. 8244, pp. 15–31. Springer (2013). https://doi.org/10.1007/978-3-319-03077-7_2, https://doi.org/10.1007/978-3-319-03077-7_2
30. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A dpll (t) theory solver for a theory of strings and regular expressions. In: Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. pp. 646–662. Springer (2014)
31. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient smt solver for string constraints. Formal Methods in System Design **48**, 206–234 (2016)
32. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: International Symposium on Frontiers of Combining Systems. pp. 135–150. Springer (2015)
33. Loring, B., Mitchell, D., Kinder, J.: Expose: practical symbolic execution of standalone javascript. In: Erdogmus, H., Havelund, K. (eds.) Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 10-14, 2017. pp. 196–199. ACM (2017). https://doi.org/10.1145/3092282.3092295, https://doi.org/10.1145/3092282.3092295
34. Lotz, K., Goel, A., Dutertre, B., Kiesl-Reiter, B., Kong, S., Nowotka, D.: Solving string constraints with concatenation using SAT. In: Narodytska, N., Rümmer, P. (eds.) Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024. pp. 29–38. IEEE (2024). https://doi.org/10.34727/2024/ISBN.978-3-85448-065-5_9, https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_9
35. Mann, M., Wilson, A., Zohar, Y., Stuntz, L., Irfan, A., Brown, K., Donovick, C., Guman, A., Tinelli, C., Barrett, C.W.: Smt-switch: A solver-agnostic C++ API for SMT solving. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 377–386. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_26, https://doi.org/10.1007/978-3-030-80223-3_26

36. Pasareanu, C.S., Visser, W., Bushnell, D.H., Geldenhuys, J., Mehlitz, P.C., Rungta, N.: Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. Autom. Softw. Eng. **20**(3), 391–425 (2013). https://doi.org/10.1007/S10515-013-0122-2, https://doi.org/10.1007/s10515-013-0122-2
37. Poeplau, S., A.Francillon: Symqemu: Compilation-based symbolic execution for binaries. In: NDSS 2021, Network and Distributed System Security Symposium. Internet Society (2021)
38. Poeplau, S., Francillon, A.: Symbolic execution with {SymCC}: Don't interpret, compile! In: 29th USENIX Security Symposium (USENIX Security 20). pp. 181–198 (2020)
39. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Kroeze, J.H., de Villiers, R. (eds.) 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012. pp. 139–148. ACM (2012). https://doi.org/10.1145/2389836.2389853, https://doi.org/10.1145/2389836.2389853
40. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: High-level abstractions for simplifying extended string constraints in SMT. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 23–42. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_2, https://doi.org/10.1007/978-3-030-25543-5_2
41. Reynolds, A., Nötzli, A., Barrett, C.W., Tinelli, C.: A decision procedure for string to code point conversion. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12166, pp. 218–237. Springer (2020). https://doi.org/10.1007/978-3-030-51074-9_13, https://doi.org/10.1007/978-3-030-51074-9_13
42. 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
43. Reynolds, A., Woo, M., Barrett, C.W., Brumley, D., Liang, T., Tinelli, C.: Scaling up DPLL(T) string solvers using context-dependent simplification. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 453–474. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_24, https://doi.org/10.1007/978-3-319-63390-9_24
44. Rungta, N.: A billion SMT queries a day (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13371, pp. 3–18. Springer (2022). https://doi.org/10.1007/978-3-031-13185-1_1, https://doi.org/10.1007/978-3-031-13185-1_1
45. 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, SP 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

46. Scott, J.D., Flener, P., Pearson, J.: Bounded strings for constraint programming. In: 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013. pp. 1036–1043. IEEE Computer Society (2013). https://doi.org/10.1109/ICTAI.2013.155, https://doi.org/10.1109/ICTAI.2013.155
47. Serebryany, K.: {OSS-Fuzz}-google's continuous fuzzing service for open source software (2017)
48. Shannon, D., Ghosh, I., Rajan, S., Khurshid, S.: Efficient symbolic execution of strings for validating web applications. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). pp. 22–26 (2009)
49. Shannon, D., Hajra, S., Lee, A., Zhan, D., Khurshid, S.: Abstracting symbolic execution with string analysis. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). pp. 13–22. IEEE (2007)
50. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al.: Sok:(state of) the art of war: Offensive techniques in binary analysis. In: 2016 IEEE symposium on security and privacy (SP). pp. 138–157. IEEE (2016)
51. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
52. 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). https://doi.org/10.1145/2491411.2491456, https://doi.org/10.1145/2491411.2491456