Computing Summaries of String Loops in C for Better Testing and Refactoring

Timotej Kapus
Imperial College London
United Kingdom

Oren Ish-Shalom
Tel Aviv University
Israel

Shachar Itzhaky
Technion
Israel

Noam Rinetzky
Tel Aviv University
Israel

Cristian Cadar
Imperial College London
United Kingdom

Abstract

Analysing and comprehending C programs that use strings is hard: using standard library functions for manipulating strings is not enforced and programs often use complex loops for the same purpose. We introduce the notion of memoryless loops that capture some of these string loops and present a counterexample-guided synthesis approach to summarise memoryless loops using C standard library functions, which has applications to testing, optimisation and refactoring.

We prove our summarisation is correct for arbitrary input strings and evaluate it on a database of loops we gathered from thirteen open-source programs. Our approach can summarise over two thirds of memoryless loops in less than five minutes of computation time per loop. We then show that these summaries can be used to (1) improve symbolic execution (2) optimise native code, and (3) refactor code.

CCS Concepts • **Software and its engineering** → *Automatic programming*; *Semantics*; *Software testing and debugging*; *Extra-functional properties.*

Keywords Loop Summarisation, Synthesis, Symbolic Execution, Refactoring, Strings, Optimisation

ACM Reference Format:
Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing Summaries of String Loops in C for Better Testing and Refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314610

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6712-7/19/06...$15.00
https://doi.org/10.1145/3314221.3314610

1 Introduction

Strings are perhaps the most widely-used datatype, at the core of the interaction between humans and computers, via command-line utilities, text editors, web forms, and many more. Therefore, most programming languages have strings as a primitive datatype, and many program analysis techniques need to be able to reason about strings to be effective.

Recently, the rise of string solvers [5, 24, 28, 50, 53, 54] has enabled more effective program analysis techniques for string-intensive code, e.g. Kuduzu [37] for JavaScript or Haderach [40] and JST [15] for Java. However, these techniques operate on domains where strings are well-defined objects (i.e. the *String* class in Java). Unlike Java or similar languages, strings in C are just arbitrary portions of memory terminated by a null character. That means interacting with strings does not have to go through a well-defined API as in other programming languages. While there are string functions in the C standard library (e.g. *strchr*, *strspn*, etc. defined in *string.h*), programmers can—and as we will show often do—write their own equivalent loops for the functionality provided by the standard library.

In this paper, we focus on a particular set of these loops, which we call *memoryless loops*. Essentially, these are loops that do not carry information from one iteration to another. To illustrate, consider the loop shown in Figure 1 (taken from *bash* v4.4). This loop only looks at the current pointer and skips the initial whitespace in the string *line*. The loop could have been replaced by a call to a C standard library function, by rewriting it into `line += strspn(line, "␣\t")`.[1]

Replacing loops such as those of Figure 1 with calls to standard string functions has several advantages. From a software development perspective, such code is often easier to understand, as the functionality of standard string functions is well-documented. Furthermore, such code is less error-prone, especially since loops involving pointer arithmetic are notoriously hard to get right. Our technique is thus useful for refactoring such loops into code that calls into the C standard library. As detailed in §4, we submitted several

[1]We remind the reader that `strspn(char *s, char *charset)` computes the length of the prefix of `s` consisting only of characters in `charset`.

```
1 #define whitespace(c) (((c) == '␣') || ((c) == '\t'))
2 char* loopFunction(char* line) {
3     char *p;
4     for (p = line; p && *p && whitespace (*p); p++)
5       ;
6     return p;
7 }
```

Figure 1. String loop from the *bash v4.4* codebase, extracted into a function.

such refactorings to popular open-source codebases, with some of them now accepted by developers.

From a program analysis perspective, reasoning about calls that use standard library functions is easier because the semantics is well-understood; conversely, understanding custom loops involving pointers is difficult. Moreover, code that uses standard string functions can benefit from recent support for string constraint solving. Solvers such as HAMPI [24] and Z3str [5, 53, 54] can effectively solve constraints involving strings, but constraints have to be expressed in terms of a finite vocabulary that they support. Standard string functions can be easily mapped to this vocabulary, but custom loops cannot. In this paper, we show that program analysis via dynamic symbolic execution [8] can have significant scalability benefits by using a string solver, with a median speedup of 79x for the loops we considered.

Finally, translating custom loops to use string functions can also impact native execution, as such functions can be implemented more efficiently, e.g. to take advantage of architecture-specific hardware features.

In §2 we first define a vocabulary that can express memoryless string loops similar to the one in Figure 1. We then present an algorithm for counterexample-guided synthesis within a single (standard) symbolic execution run, which we use to synthesise memoryless string loops in our vocabulary and show they are equivalent up to a small bound. In §3 we formally prove that showing equivalence up to a small bound extends to strings of arbitrary length for memoryless loops, thus showing our summarisation is sound. Finally, we build a database of suitable loops in §4.1 based on popular open-source programs, which we use to comprehensively analyse the impact of time, program size and vocabulary on summarising loops in §4.2.

2 Technique

The goal of our technique is to translate[2] loops such as the one in Figure 1 into calls to standard string functions. Our technique uses a counterexample-guided inductive synthesis (CEGIS) algorithm [44] inspired by Sharma et al.'s work on synthesising adaptors between C functions with overlapping functionality but different interfaces [42]. This CEGIS approach is based on dynamic symbolic execution, a program analysis technique that can automatically explore multiple paths through a program by using a constraint solver [8]. In this section, we first discuss the types of loops targeted by our approach (§2.1), then present the vocabulary of operations to which loops are translated (§2.2), and finally introduce our CEGIS algorithm (§2.3).

2.1 Loops Targeted

Our approach targets relatively simple string loops which could be summarised by a sequence of standard string library calls (such as `strchr`) and primitive operations (such as incrementing a pointer). More specifically, our approach targets loops that take as input a pointer to a C string (so a `char *` pointer), return as output a pointer into that same string, and have no side effects (e.g., no modifications to the string contents are permitted). Such loops are frequently coded in real applications, and implement common tasks such as skipping a prefix or a suffix from a string or finding the occurrence of a character or a sequence of characters in a larger string. Such loops can be easily synthesised by a short sequence of calls to standard string functions and primitive operations (see §4 which shows that most loops in our benchmarks can be synthesised with sequences of at most 5 such operations). While our technique could be extended to other types of loops, we found our choice to provide a good performance/expressiveness trade-off.

More formally, we refer to the two types of loops that we can synthesise as *memoryless forward loops* and *memoryless backward loops*, as defined below.

Definition 1 (Memoryless Forward Loop)**.** Given a string of length len, and a pointer p into this buffer, a loop is called a forward memoryless loop with cursor p iff:

1. The only string location being read inside the loop body is $p_0 + i$, where p_0 is the initial value of p and i is the iteration index, with the first iteration being 0
2. Values involved in comparisons can only be one of the following: 0, i, len for integer comparisons; p_0, $p_0 + i$, $p_0 + len$ for pointer comparisons; and $*p$ (i.e. $p_0[i]$) and constant characters for character comparisons;
3. The conceptual return value (i.e. what the loop computes) is $p_0 + c$, where c is the number of completed iterations.[3]

Definition 2 (Memoryless Backward Loop)**.** Such loops are defined similarly to forward memory loops, except that the only string location being read inside the loop body is $p_0 + (len - 1) - i$ and the return value is $p_0 + (len - 1) - c$.

[2]In this paper, we use the terms translate, summarise and synthesise interchangeably to refer to the process of translating the loop into an equivalent sequence of primitive and string operations.

[3]I.e., the number of times execution reached the end of the loop body and jumped back to the loop header [2].

Table 1. The vocabulary employed by our technique.

Gadget	Regexp	Effect
rawmemchr	M(.)	result = rawmemchr(result, $1)
strchr	C(.)	result = strchr(result, $1)
strrchr	R(.)	result = strrchr(result, $1)
strpbrk	B(.+)\0	result = strpbrk(result, $1)
strspn	P(.+)\0	result += strspn(result, $1)
strcspn	N(.+)\0	result += strcspn(result, $1)
is nullptr	Z	skipInstruction = z != NULL
is start	X	skipInstruction = result != *s*
increment	I	result++
set to end	E	result = *s* + strlen(*s*)
set to start	S	result = *s*
reverse	^V	reverses the string (see §2.2)
return	F	return result and terminate

2.2 Vocabulary

Given a string loop, our technique aims to translate it into an equivalent program consisting solely of primitive operations (such as incrementing a pointer) and standard string operations (such as *strchr*). For convenience, we represent the synthesised program as a sequence of characters, each character representing a primitive or string operation. To make things concrete, we show the elements (called *gadgets*) of our working vocabulary in Table 1.

To summarise the loop in Figure 1, we need to first represent the call to `strspn`. We choose the character `P` to be the opcode for calls to `strspn`. We do not have to represent the first argument in `strspn` as it is implicitly the string the loop is operating on. The second argument is a string containing all the characters `strspn` counts in its span. We represent this using the actual list of characters passed as the second argument, followed by the \0 terminator character. Thus, the call `strspn(line, "␣\t")` would be encoded by `P␣\t\0`.

Extended regular expressions provide a concise way of presenting the vocabulary. A string matching the regular expression `P(.+)\0` represents a call to `strspn`, where the second argument consists of the characters in the capture group `(.+)` of the regular expression. In Table 1, we refer to the first capture group as `$1`.

To formally define the meaning of a program in our language (e.g. of a sequence like `P␣\t\0`), we define an interpreter that operates on such sequences of instructions (characters) and returns an offset in the original string. The execution of the interpreter represents the synthesised program.

A partial interpreter that can handle three of our vocabulary gadgets (*strspn*, *is nullptr* and *return*) is shown in Algorithm 1. The interpreter has an immutable input pointer register (*s*, the original string pointer on which the loop operates), a return pointer register (*result*) and a skip instruction flag (*skipInstruction*).

The interpreter loops through all the instructions in the *program* (lines 4–20). If the skip instruction flag is set, the interpreter skips the next instruction and resets the flag

Algorithm 1 Program interpreter for a vocabulary of three gadgets: *strspn*, *is nullptr* and *return*

```
1:  function Interpreter(s, program)
2:      result ← s
3:      skipInstruction ← false
4:      foreach instruction ∈ program do
5:          if skipInstruction then
6:              skipInstruction ← false
7:              continue
8:          end if
9:          arguments ← InstructionArgs(instruction)
10:         switch OpCode(instruction) do
11:             case 'P' // strspn
12:                 result ← result+ strspn(result, arguments)
13:             case 'Z' // is nullptr
14:                 skipInstruction ← result ! = NULL
15:             case 'F' // return
16:                 return result
17:             default
18:                 return invalidPointer
19:         end switch
20:     end foreach
21:     return invalidPointer
22: end function
```

(lines 5–8). Otherwise, it reads the next instruction and interprets each gadget type accordingly, by either updating the *result* register or the *skipInstruction* flag (lines 9–16). If the loop runs out of instructions or the opcode is unknown, we return an invalid pointer (lines 17–18). This ensures that malformed programs never have the same output as the original loop and are therefore not synthesised.

Consider running the interpreter on the program `P␣\t\0F`. The interpreter initialises `result` to the string pointer `s` and `skipInstruction` to `false`. It then reads the next instruction, `P`, and thus updates `result` to `result + strspn(result, "␣\t")`. Finally, it reads the next instruction, `F`, and returns the `result`. Therefore, the synthesised program is:

```
result = s;
skipInstruction = false;
if (!skipInstruction)
  result = result + strspn(result, "␣\t");
else skipInstruction = false;
if (!skipInstruction)
  return result;
else skipInstruction = false;
```

which is equivalent to `return = s + strspn(result, "␣\t")`.

To also illustrate the use of the *is nullptr* gadget covered by Algorithm 1, consider the slightly enhanced program `ZFP␣\t\0F`. In this case, the interpreter will synthesise the following code for the added `ZF` prefix, which returns `NULL` if the string is `NULL`:

```
result = s;
if (!skipInstruction)
```

```
    skipInstruction = result != NULL;
  else skipInstruction = false;
  if (!skipInstruction)
    return result;
  else skipInstruction = false;
```

Table 1 summarises the full set of vocabulary we devised to express the loops we wanted to target. The first column in the table gives the name of the gadget, while the second shows the extended regular expression that describes the gadget. The third column shows the effect this gadget has on the INTERPRETER. Note that `$1` gets replaced with the characters from the regex capture group.

Meta-characters. To more easily synthesise gadgets that take sets of characters as arguments, such as `strspn`, we introduce the notion of meta-characters. These are single symbols in our synthesised programs that expand into sets of characters. By reducing the program size, this helps us synthesise loops that contain calls to macros/functions such as `isdigit` in a more scalable way. For example, instead of having to synthesise 10 characters (`"0123456789"`), we can synthesise just a single meta-character (which we chose to be `'\a'`). Similarly, we also use a whitespace meta-character which represents `"␣\t\n"`). These are the only two meta-characters that we found beneficial in our experiments, but more could be added if needed. At the same time, we note that meta-characters are not strictly necessary, i.e. the synthesis algorithm would work without them, but would take longer.

Reverse instruction. The *reverse* instruction can only occur as the first instruction of a synthesised program. Its purpose is to facilitate the summarisation of backward loops. For example, *reverse* and *strchr* should be equal to *strrchr*. However, not all functions have a reverse version, so the purpose of the *reverse* instruction is to enable similar functionality for other functions such as *strspn*. For instance, such function can be easily expressed in terms of the vocabulary offered by string constraint solvers.

The *reverse* instruction takes the string *s* and copies it into a new buffer in reverse order. It then sets *result* to this new buffer. It also enhances the behaviour of the *return* instruction: instead of simply returning *result*, which now points to a completely different buffer, it maps the offset in the new buffer back into the original buffer.

2.3 Counterexample-Guided Synthesis

Our synthesis approach is inspired by Sharma et al.'s work on synthesising adapters between different C library implementations [42]. Their vocabulary gadgets are able to translate the interface of one library into that of another. Examples of gadgets include swapping arguments or introducing a new constant argument to a function.

Their approach uses counterexample-guided inductive synthesis (CEGIS) based on symbolic execution to synthesise the adapters in that vocabulary. In brief, this means they first take a symbolic adapter, constrain it so that it correctly translates all the currently known examples, then concretize the adapter and try to prove the two implementations are the same up to a bound. If that is successful, the synthesis terminates, otherwise they obtain a counterexample, add it to the list of known examples and repeat.

We took their approach further to synthesise whole functions instead of just adapters between function interfaces.

Before describing the algorithm in detail, we give a short introduction to symbolic execution (in its modern *dynamic symbolic execution* or *concolic execution* variant). Symbolic execution [7, 8, 16, 25] is a program analysis technique that systematically explores paths in a program using a constraint solver. Symbolic execution executes the program on *symbolic input*, which initially is allowed to take any value. Then, statements depending on the symbolic input (directly or indirectly) add constraints on the input. In particular, branches depending on the symbolic input lead symbolic execution to check the feasibility of each side of the branch using a constraint solver. If both sides are feasible, execution is forked to follow each side separately, adding appropriate constraints on each side. For instance, if the symbolic input is `x` and the branch `if x > 0` is the first statement of the program, then both sides are feasible so execution is forked into two paths: one path adds the constraint that $x > 0$ and follows the *then* side of the branch, while the other path adds the constraint that $x \leq 0$ and follows the *else* side of the branch. Once a path terminates, a constraint solver can be used to generate concrete solutions to the set of constraints added on that path. For instance, if the path constraints are $x > 0$ and $x < 100$, the constraint solver might return solution $x = 30$, which represents a concrete input that is guaranteed to execute the path on which the constraints were gathered. In some contexts, including CEGIS, such inputs are referred to as *counterexamples*.

Our approach focuses on the types of loops described in §2.1, which can be extracted into functions with a `char* loopFunction(char* s)` signature. However, the technique should be easy to adapt to loops with different signatures. Our vocabulary of gadgets is the one described in §2.2.

Algorithm 2 presents our approach in detail. The algorithm is a program that when executed under symbolic execution performs CEGIS and returns the synthesised program representing the loop. This program is just a sequence of characters which can be interpreted as discussed in §2.2.

The algorithm makes use of the following symbolic execution operations:

- SYMBOLICMEMOBJ(N) creates a symbolic memory object of size N bytes.
- ASSUME(COND) adds the condition COND to the current path constraints.
- CONCRETIZE(x) makes symbolic input x concrete, by asking the constraint solver for a possible solution.

Algorithm 2 Synthesis algorithm, which is run under symbolic execution.

```
1:  counterexamples ← ∅
2:  while Timeout not exceeded do
3:      prog ←SymbolicMemObj(MAX_PROG_SIZE)
4:      foreach cex ∈ counterexamples do
5:          Assume(Original(cex) = Interpreter(cex, prog))
6:      end foreach
7:      KillAllOthers( )                  ▷ Only need a single path
8:      prog ← Concretize(prog)
9:
10:     example ←SymbolicMemObj(MAX_EX_SIZE)
11:     example[MAX_EX_SIZE − 1] ← '\0'
12:     StartMerge( )
13:     originalOutput ← Original(example)
14:     synthesizedOutput ← Interpreter(example, prog)
15:     isEq ← originalOutput = synthesizedOutput
16:     EndMerge( )
17:
18:     if IsAlwaysTrue(isEq) then
19:         return prog                            ▷ We are done
20:     end if
21:
22:     Assume(!isEq)
23:     cex ← Concretize(example)
24:     counterexamples ← counterexamples ∪ {cex}
25: end while
```

- KillAllOthers() prunes all the paths, except the path first reaching this call.
- IsAlwaysTrue(cond) returns *true* iff the condition cond can only be true.
- StartMerge() and EndMerge() merge all the paths between these two calls into a single path, by creating a big disjunction of all the path constraints of the merged paths.

The target loop we are trying to synthesise is represented by the Original(s) function, with s the input string. Interpreter(s, program) executes the program on string s as discussed in Algorithm 1. Our aim is to synthesise a program such that: ∀s.Interpreter(s, program) = Original(s).

Algorithm 2 starts by initialising the set of all counterexamples encountered so far to the empty set (line 1). Counterexamples in this context are strings C on which the original loop and the synthesised program were found to behave differently, i.e. Interpreter(C, program) ≠ Original(C).

The algorithm is centered around a main loop (lines 2–25), which ends either when a synthesised program is found or a timeout is exceeded. On each loop iteration, we create a new symbolic sequence of characters representing our program (line 3) and constrain it such that its output on all current counterexamples matches that of the original function (lines 4-6).

When the program is run through the Interpreter function, there might be multiple paths on which the program is equivalent to the Original function for the current counterexamples. However, we are only interested in one of them. The computation spent on exploring other paths at this point is better used in the next iteration of the main loop, which will have more counterexamples to guide the synthesis. Therefore we only keep a single path by calling KillAllOthers (line 7). We also concretize the program (line 8) to make the next step computationally easier.

The remainder of the loop body then focuses on finding a new counterexample, if one exists. Lines 10–15 attempt to prove the Original function and program have the same effect on a fresh symbolic string of up to length max_ex_size on all possible paths. To be able to reason about all these paths at once, we merge them using StartMerge() and EndMerge() (lines 12–16).

Variable isEq on line 15 is a boolean variable that encodes whether the original loop and the synthesised program are equivalent on strings up to length max_ex_size. Line 18 checks whether isEq can only take value *true*. If so, we know that the synthesised program behaves the same as the original loop on all strings of up to length max_ex_size (and based on the proof in §3 on strings of arbitrary length) and we can successfully return the program.

Otherwise, we need to get a new counterexample and repeat the process in a next iteration of the main loop. For this, we first prune the paths where isEq is *true* by assuming it is *false* (line 22). Then, we obtain a counterexample by asking for a concrete solution for the example string that meets the current path constraints (where isEq is false) (line 23) and add it to the set of counterexamples (line 24).

3 Equivalence of Memoryless Programs

In §2, we presented a CEGIS-based approach for translating a class of string-manipulating loops into programs consisting solely of primitive operations (such as incrementing a pointer) and string operations (such as *strchr*). However, the synthesised programs were only symbolically tested to have the same effect as the original loop for all strings up to a given bound. In this section, we show how we can lift these bounded checks into a proof of equivalence.

Intuitively, we show that the loops we target cannot distinguish between executing on a "long" string and executing on its suffix. In both executions, the value returned by the loop is uniquely determined by the number of iterations it completed, and every time the loop body operates on a character c, it follows the same path, except, possibly, when scanning the first or last character.

Technically, we define a syntactic class of *memoryless specification* (Definition 3), which scans the input string either forwards or backwards, and terminates when it reaches a character which belongs to a given set X. The choice of this class of specification is a pragmatic one: We observed that all the programs we synthesise can be specified in this manner.

We then prove that if an arbitrary loop respects a memoryless specification on all strings up to length 3 and the original loop also adheres to certain easy-to-verify mostly-syntactic restrictions, the loop respects the specification on arbitrary strings. Essentially, we prove a small-model theorem: We show that any divergence from the specification on a string of length $k + 1$, for $3 \leq k$, can be obtained on a string of length k.

The proof goes in two stages. First, we prove a small-model theorem (Theorem 3.4) for a class of programs (memoryless loops) which is defined semantically. For example, the definition restricts the *values* the loop may use in comparisons. Second, §3.3 shows most of our programs respect certain easy-to-check properties. The combination of the aforementioned techniques allows us to provide a conservative technique for verifying that the synthesised program is equivalent to the original loop.

Notations. We denote the domain of characters by C and use $C_0 = C \cup \{null\}$ for the same domain extended with a special *null* character. We denote (possibly empty) sequences of non-*null* characters by C^*. We refer to a *null*-terminated array of characters as a *string buffer* (*string* for short). We denote the set of strings by $s \in \mathcal{S}$. We write constant strings inside quotation marks. For example, the string $s =$ "abc" is an array containing *four* characters: 'a', 'b', 'c', and *null*. Note that if $\omega =$ abc then $s =$ "ω".[4] We use "" to denote an *empty string*, i.e., one which contains only the *null* character. The *length* of a string s, denoted by strlen(s), is the number of characters it contains excluding the terminating *null* character. For example, strlen("abc") = |abc| = 3 and strlen("") = 0. We denote the ith character of a string s by $s[i]$. Indexing into a string is zero-based: $s[0]$ is the first character of s and s[strlen(s)] is its last. Note that the latter is always *null*. We write "$c\omega$" resp. "ωc" to denote a string whose first resp. penultimate character is c. We denote the complement of a set of characters X by $\overline{X} = C_0 \setminus X$.

3.1 Memoryless Specifications

Definition 3 (Memoryless Specification)**.** A *memoryless specification* of a string operation is a function whose definition can be instantiated from the following schema by specifying the missing code parts (*start*, *end*, *R*, and *X*):

```
char* func(char *input) {
  int i, len = strlen(input);
  for (i = start to end)
    if (input[i] ∈ X)
      return input + i;
  return R;
}
```

The schema may be instantiated to a function that traverses the input buffer either *forwards* or *backwards*: In a forward traversal, *start* = 0, *end* = len - 1, and *R* = input + len. In a *backward* traversal, *start* = len - 1, *end* = 0, and *R* = input. *X* is a set of characters.

Example 3.1. It is easy to see that many standard string operations respect a memoryless specification. For example, the loop inside strchr(p,c) resp. strrchr(p,c) can be specified using a forward resp. backward memoryless specification with X set to $\{c\}$. The loop inside strspn(p,s) has a memoryless forward specification in which X contains all the characters *except* the ones in s.

In this section we focus exclusively on memoryless forward loops. The definitions and proofs for memory backward loops are analogous, and are omitted for space reasons.

We use $[\![P]\!]$ for the semantic function of P, that is, $[\![P]\!]$ is the value returned by P when its string buffer is initialised to $s \in \mathcal{S}$. We refer to the character that a memoryless forward loop may observe in the ith iteration as the *current character of iteration i*, and omit the *iteration index i* when clear from context.

Definition 4 (Iteration Counter)**.** Let P be a memoryless forward loop, as per Definition 1. We define $\Delta_P(s)$ as the number of iterations that P completes when its input is a string s. If P does not terminate on s, then $\Delta_P(s) = \infty$.

Note that the semantic restrictions imposed on memoryless loops ensures that $[\![P]\!]$ and Δ_P have a one-to-one mapping. Thus, in the rest of the paper, we use these notations interchangeably.

3.2 Bounded Verification of Memoryless Equivalence

Theorem 3.2 (Memoryless Truncate)**.** *Let P be a memoryless forward loop, and let $\omega, \omega' \in C^*$.*

1. *If $\Delta_P(\text{"}\omega\omega'\text{"}) < |\omega|$, then $\Delta_P(\text{"}\omega\omega'\text{"}) = \Delta_P(\text{"}\omega\text{"})$.*
2. *If $\Delta_P(\text{"}\omega\omega'\text{"}) \geq |\omega|$, then $\Delta_P(\text{"}\omega\text{"}) \geq |\omega|$.*

Proof. 1. Since P performs fewer than $|\omega|$ complete iterations, Definition 1 ensures that P can only observe the prefix $0..(|\omega| - 1)$ of its input buffer "$\omega\omega'$",[5] which are all characters of ω. Moreover, all comparisons between integers $0, i, len$ or pointers $p_0, p_0 + i, p_0 + len$ must return identical values whether $len = |\omega|$ or $len = |\omega\omega'|$, since $i < |\omega|$ (and thus $i < |\omega\omega'|$) in all of these iterations. Therefore P behaves the same when executing on ω and on $\omega\omega'$, and thus it must be that $[\![P]\!](\text{"}\omega\text{"}) = [\![P]\!](\text{"}\omega\omega'\text{"})$.

2. Similarly, the first $|\omega|$ iterations are identical between $[\![P]\!](\text{"}\omega\omega'\text{"})$ and $[\![P]\!](\text{"}\omega\text{"})$. Since the former carried these $|\omega|$ iterations to completion, so must the latter perform at least as many iterations. □

[4] ω is a mathematical sequence of characters, s is a string buffer.

[5] When P performs k complete iterations, then it can read at most $k + 1$ (rather than k) characters from the input string. The last one occurs in the (incomplete) iteration that exits the loop.

Note that if P is *safe*, that is, never reads past the end of the string buffer, then $\Delta_P(\text{"}\omega\text{"}) \geq |\omega|$ in fact implies $\Delta_P(\text{"}\omega\text{"}) = |\omega|$. At this point we make no such assumptions; later we will see that if P is tested against a specification which is itself safe, then indeed it is guaranteed that P is also safe.

Let $Q_0(c)$ denote whether P completes (at least) the first iteration of the loop when it executes on some single-character string "c", i.e., $Q_0(c) \triangleq (\Delta_P(\text{"}c\text{"}) > 0)$.

If $Q_0 = \mathit{false}$, i.e., $\forall c \in C.\, \neg Q_0(c)$, then obviously P never gets to the second iteration of the loop. Otherwise, let $a \in C$ be a character such that $Q_0(a)$ is true. We can continue to define Q_1 as:

$$Q_1(c) \triangleq (\Delta_P(\text{"}ac\text{"}) > 1) \tag{1}$$

It is important to note that the choice of a does not affect the decisions made at the second iteration: again, based on the restrictions imposed by Definition 1, the program cannot record the current character in the first iteration and transfer this information to the second iteration; hence (1) is well defined. When there is no corresponding a, let $Q_1(c) = \mathit{false}$.

We now define a family of predicates Q_i over the domain of characters C_0 (including *null*) that describe the decision made at iteration i of the loop based on the current character, $c \in C_0$. We use these predicates to show that the decisions taken at iteration i are always the same as those taken at iteration 1 (which is the *second* iteration), and depend solely on the current character. The definition is done by induction on i:

$$\begin{array}{l} Q_{i+1}(c) \triangleq (\Delta_P(\text{"}\omega c\text{"}) > i+1) \\ \quad \text{for some } \omega = a_0 \cdots a_i \text{ such that } \bigwedge_{j=0..i} Q_j(a_j) \end{array} \tag{2}$$

As before, the choice of ω is insignificant, and if no such ω exists, let $Q_{i+1}(c) = \mathit{false}$.

Claim 1. $Q_i(c) = Q_1(c)$ for any $i \in \mathbb{N}^+$ and $c \in C_0$.

Proof. The definition of Q_i is based on the choice of P at iteration i when running on a string of length $i + 1$. At that point, the situation is that $0 < i < \mathit{len}$. Therefore, again, the observations of P at iteration i are no different from its observations at iteration 1, assuming the same current character c; therefore $Q_i(c) = Q_1(c)$. □

The reason Claim 1 states that $Q_i(c) = Q_1(c)$ instead of $Q_i(c) = Q_0(c)$ is that according to our restrictions, the behaviour of P when operating on the first character of the string might differ from its behaviour on all other characters (this is because P can compare the iteration index to zero). Note that a similar issue does not occur concerning the last character of the string as the latter is always *null*.

Theorem 3.3 (Memoryless Squeeze)**.** *Let P be a memoryless forward loop. We construct a buffer "$a\omega b$" where $a, b \in C$ and $\omega \in C^*$.*

1. *If $\Delta_P(\text{"}a\omega b\text{"}) = 1 + |\omega|$, then $\Delta_P(\text{"}ab\text{"}) = 1$.*
2. *If $\Delta_P(\text{"}a\omega b\text{"}) > 1 + |\omega|$, then $\Delta_P(\text{"}ab\text{"}) > 1$.*

Proof of Theorem 3.3. Let $a\omega b = a_0 a_1 \cdots a_{|\omega|+1}$ be the characters of $a\omega b$ (in particular, $a_0 = a$, $a_{|\omega|+1} = b$).

1. Assume $\Delta_P(\text{"}a\omega b\text{"}) = 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \leq i \leq |\omega|$, and $\neg Q_{|\omega|+1}$. Therefore, $Q_0(a)$ (since $a_0 = a$), and $\neg Q_{|\omega|+1}(b)$. From Claim 1, also $\neg Q_1(b)$. Hence $[\![P]\!](\text{"}ab\text{"})$ completes the first iteration and exits the second iteration; so $\Delta_P(\text{"}ab\text{"}) = 1$.

2. Assume $\Delta_P(\text{"}a\omega b\text{"}) > 1 + |\omega|$, then $Q_i(a_i)$ for all $0 \leq i \leq |\omega| + 1$. In this case we get $Q_0(a)$ and $Q_{|\omega|+1}(b)$. Again from Claim 1, $Q_1(b)$. Hence $[\![P]\!](\text{"}ab\text{"})$ completes at least two iterations, and $\Delta_P(\text{"}ab\text{"}) > 1$. □

Theorem 3.4 (Memoryless Equivalence)**.** *Let F be a memoryless specification with forward traversal and character set X, and P a memoryless forward loop. If for every character sequence $\omega \in C^*$ of length $|\omega| \leq 2$ it holds that $[\![P]\!](\text{"}\omega\text{"}) = F(\text{"}\omega\text{"})$, then for any string buffer $s \in \mathcal{S}$ (of any length), $[\![P]\!](s) = F(s)$.*

Proof. Assume by contradiction that there exists a string $s \in \mathcal{S}$ on which P and F disagree, i.e., $[\![P]\!](s) \neq F(s)$. We show that we can construct a string s' such that $[\![P]\!](s') \neq F(s')$ and $|s'| \leq 2$, which contradict our hypothesis.

We define $\Delta_F(s)$ as the number of iterations the specification F performs before returning. Definition 1 ensures that $0 \leq \Delta_F(s)$ and $\Delta_F(s) \leq \texttt{strlen}(s)$. By assumption, F is a forward loop, i.e., *start* = 0 and *end* = *len*. Thus, $\Delta_F(s)$ is the length of the *longest prefix* τ of s such that $\tau \in \overline{X}^*$.

Since $[\![P]\!](s) \neq F(s)$, we know that $\Delta_P(s) \neq \Delta_F(s)$. If $\texttt{strlen}(s) \leq 2$, we already have our small counterexample. Otherwise, we consider two cases.

Case 1: $\Delta_P(s) < \Delta_F(s)$. If $\Delta_P(s) = 0$, let $s' = \text{"}a\text{"}$ where a is the first character of s. According to Theorem 3.2, $\Delta_P(s') = 0$. However, $a \notin X$ (otherwise $\Delta_F(s) = 0 = \Delta_P(s)$, which we assumed is false), therefore $\Delta_F(s') = 1$. ◦

If $\Delta_P(s) > 0$, we decompose s into "$a\omega b\omega'$", such that $\Delta_P(s) = |\omega| + 1$. We have $|a\omega b| > |\omega| + 1$, hence by Theorem 3.2, $\Delta_P(\text{"}a\omega b\text{"}) = \Delta_P(\text{"}a\omega b\omega'\text{"}) = |\omega| + 1$. Let $s' = \text{"}ab\text{"}$; by Theorem 3.3, we obtain $\Delta_P(s') = 1$. We know that $\Delta_F(\text{"}a\omega b\omega'\text{"}) \geq |\omega| + 2$, so $a\omega b \in X^*$, in particular $a, b \in X$. Therefore $\Delta_F(s') = 2$. ◦

Case 2: $\Delta_P(s) > \Delta_F(s)$. If $\Delta_F(s) = 0$, let $s' = \text{"}a\text{"}$ where a is the first character of s. Since $\Delta_P(s) \geq |s'| = 1$, and according to Theorem 3.2, we get $\Delta_P(s') \geq 1$. ◦

If $\Delta_F(s) > 0$, we again decompose s into "$a\omega b\omega'$", this time such that $\Delta_F(s) = |\omega| + 1$. From this construction we get $a\omega \in X^*$ (in particular $a \in X$) and $b \notin X$. Since $\Delta_P(s) \geq |\omega| + 2 = |a\omega b|$, and according to Theorem 3.2, we get $\Delta_P(\text{"}a\omega b\text{"}) \geq |\omega| + 2 > |\omega| + 1$. Let $s' = \text{"}ab\text{"}$, and we know from Theorem 3.3 that $\Delta_P(s') > 1$. In contrast, from $a \in X, b \notin X$ established earlier, $\Delta_F(s') = 1$. ◦

In both cases (each with its two sub-cases, tagged with ∘), the end result is some $|s'| = \{1, 2\}$ for which $\Delta_P(s') \neq \Delta_F(s')$. This necessitates that $[\![P]\!](s') \neq F(s')$. □

We note that it is easy to see that we can allow simple loops to start scanning the string from the nth character of the string instead of the first one provided we test that the program is memoryless for strings up to length of $n + 3$.

Unterminated Loops. Some library functions (e.g. `rawmemchr`) do not terminate on a null character, and even potentially perform unsafe reads. Still, we want to be able to replace loops in the program with such implementations as long as they agree on all safe executions and do not introduce new unsafe executions. This is done with a small adjustment to Definition 3 which allows for unsafe specifications. The details are mostly mundane and are described in the online appendix.[6]

3.3 Bounded Verification of Memorylessness

We implemented the bounded verification as an LLVM pass. The input LLVM bitcode was instrumented with `assert` commands that check the memorylessness conditions. The instrumented bitcode was then fed to KLEE, which verified no assertion violations occur on strings of length three and under. For example, whenever two integer values are compared, an instrumentation is inserted before the comparison to check that the values compared are either i and len or i and zero.

We provide in the online appendix a detailed proof that this bounded verification is sufficient to show a loop is memoryless for all lengths provided that the program respects certain easily-check-able syntactic properties. These condition pertains to the way live variables are used as well as, effectively, that in every iteration a variable either increases its value by one or that its value is not changed in any iteration. We show that if the program presents such a uniform behaviour in its first three iterations, it is bound to do so in any iteration. Thus, it suffices to check these properties on strings of length up to three.

Using our technique, we could prove that 85 loops out of the 115 meet the necessary conditions, spending on average less than three seconds per loop. Invalid loops typically contain constants other than zero, or change the read value by some constant offset (e.g., in `tolower` and `isdigit`).

4 Evaluation

We implemented our synthesis algorithm on top of the popular KLEE symbolic execution engine [6]. We built on top of commit 4432580, where KLEE already supports creating symbolic memory objects, adding path constraints, concretizing symbolic objects and merging of paths. Thus, we only needed to add support for KillAllOthers and IsAlwaysTrue, which required under 20 LOC.

[6]Available at https://srg.doc.ic.ac.uk/projects/loop-summaries/

Table 2. Loops remaining after each additional filter.

	Initial loops	Inner loops	Pointer calls	Array writes	Multiple ptr reads
bash	1085	944	438	264	45
diff	186	140	60	40	14
awk	608	502	210	105	17
git	2904	2598	725	495	108
grep	222	172	72	42	9
m4	328	286	126	78	12
make	334	262	129	102	13
patch	207	172	88	67	20
sed	125	104	35	19	1
ssh	604	544	227	84	12
tar	492	432	155	106	33
libosip	100	95	39	30	25
wget	228	197	115	83	14
Total	7423	6448	2419	1515	323

We split our evaluation into three parts. First we describe how we gathered a large set of loops from real programs (§4.1). We then explore how many programs we can synthesise from this database with respect to time, vocabulary and maximum synthesised program size (§4.2). Finally, we evaluate the applications of loop synthesis in scaling up symbolic execution (§4.3), speeding up native execution (§4.4), and refactoring code (§4.5). All the experiments were run on a modern desktop PC with Intel i7-6700 CPU on Ubuntu 16.04 with 16GB of RAM.

4.1 Loop Database

We perform our evaluation on loops from 13 open-source programs: *bash*, *diff*, *awk*, *git*, *grep*, *m4*, *make*, *patch*, *sed*, *ssh*, *tar*, *libosip* and *wget*. These programs were chosen because they are widely-used and operate mostly on strings.

The process for extracting loops from these programs was semi-automatic. First, we used LLVM passes to find 7,423 loops in these programs and filter them down to 323 candidate memoryless loops. Then we manually inspected each of these 323 loops and excluded the ones still not meeting all the criteria for memoryless loops. The next sections describe in detail these two steps.

4.1.1 Automatic Filtering

After compiling each of the programs to LLVM IR, we apply LLVM's *mem2reg* pass. This pass removes load and store instructions operating on local variables, and is needed in our next step. LLVM's *LoopAnalysis* was then used to iterate through all the loops in the program, and filter out loops which are not memoryless. We automatically prune loops that have inner loops and then loops with calls to functions that take pointers as arguments or return a pointer.

Then, we filter out loops containing writes into arrays. We assume that due to *mem2reg* pass, any remaining store instructions write into arrays and not into local variables. Therefore we miss loops where this assumption fails, as they

get excluded based on containing a store instruction. Finally, we remove loops with reads from multiple pointer values. This ensures that we only keep loops with reads of the form $p_0 + i$ as per Definitions 1 and 2 of memoryless loops.

Table 2 shows, for each application considered, how many loops are initially selected (column *Initial loops*) and how many are left after each of the four filtering steps described above (e.g., column *Pointer calls* shows how many loops are left after both loops with inner loops and loops with calls taking or returning pointers are filtered out). In the end, we were left with between 9 and 108 loops per application, for a total of 323 loops.

4.1.2 Manual Filtering

We manually inspected the remaining 323 loops and manually excluded any loops that still did not meet the memoryless loops criteria from §2.1.

Two loops had `goto` in them, which meant they jumped to some other part of the function unrelated to the loops. Three loops had I/O side effects, such as outputting characters with `putc` (note that the automatic *pointer calls* filter removed most of the other I/O related loops).

A total of 74 loops did not return a pointer, and an additional 70 loops had a return statement in their body. 28 loops had too many arguments. For example, incrementing a pointer while it is smaller than another pointer would belong into this category, as the other pointer is an “argument” to the loop. Finally, 31 loops had more than one output, e.g. both a pointer and a length.

Note that some of these loops could belong into multiple categories, we just record the reason for which they were excluded during our manual inspection. In total, we manually excluded 208 loops, so we were left with 323 - 208 = 115 memoryless loops on which to apply our synthesis approach.

As part of this manual step, we also extracted each loop into a function with a `char* loopFunction(char*)` signature. While this could be automated at the LLVM level, we felt it is important to be able to see the extracted loops at the source level.

4.2 Synthesis Evaluation

We evaluate our synthesis algorithm in several ways. First, we report its effectiveness using a generous timeout (§4.2.1), then we analyse how results vary with the size of the synthesised program and the timeout (§4.2.2), and finally how they vary with the size and shape of the vocabulary (§4.2.3).

4.2.1 Results with a Large Timeout

We first aim to understand how well our synthesis approach performs within a fixed and generous budget. We choose to run the synthesis with a 2-hour timeout, which is a reasonably long timeout for symbolic execution. We use the full vocabulary, as we want to capture as many loops as possible. Finally, we choose a maximum synthesised program size (MAX_PROG_SIZE in Alg. 2) of 9 characters based on some exploratory runs, where going above this number made synthesis very slow. The bound for checking program equivalence (MAX_EX_SIZE in Alg. 2) was set to length 3, which was sufficient for our proof that shows equivalence for all string lengths.

Table 3. Successfully synthesised loops in each program and the time taken by the synthesis process (with all gadgets, MAX_PROG_SIZE=9, MAX_EX_SIZE=3 and TIMEOUT=2h).

	% synthesised	Average (min)	Median (min)
bash	12/14	5.7	5.3
diff	3/5	5.1	5.3
awk	3/3	2.1	0.2
git	18/33	3.1	2.6
grep	1/3	4.4	4.4
m4	1/5	4.4	4.4
make	0/3	n/a	n/a
patch	9/13	1.8	0.2
sed	0/0	n/a	n/a
ssh	2/2	2.7	2.7
tar	10/15	10.2	4.5
libosip	12/13	12.6	5.3
wget	6/6	6.3	1.5
Total	77/115	6.1	4.5

Table 3 summarises the time it took to synthesise loops in each program. In total, we successfully synthesised 77 out of 115 loops, most under 10 minutes, well below our 2h timeout. Note that the median is sometimes significantly smaller that the average, indicating that a few loops were particularly difficult to synthesise. For example, in *libosip* 10 loops are synthesised within 10 minutes, while another 2 take well over an hour to complete. These 2 loops are summarised by a `strspn` with a four character argument, whereas most other loops have just 1 or 2 characters as arguments. *strpbrk*, *is start* and *reverse* gadgets were not synthesised during this experiment.

4.2.2 Influence of Program Size and Timeout

We next investigate how our synthesis approach performs with respect to the synthesised program size and the chosen timeout value. To do so, we use an iterative deepening approach where we gradually increase the program size from 1 to 9 and plot the number of programs we can synthesise. We run each experiment with timeouts of 30 seconds, 3 minutes, 10 minutes and 1 hour.

More generally, we advocate using such an iterative deepening approach when summarising loops, as this strategy gives the smallest program we can synthesise for each loop, with reasonable additional overhead.

Figure 2 shows the results. Unsurprisingly, we cannot synthesise any programs with program size 1 as such a program cannot express anything but identity in our vocabulary. However, even with size 2 we can synthesise one program, namely

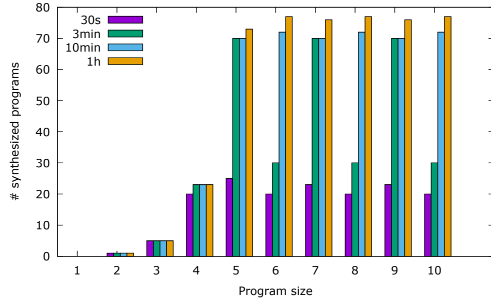

Figure 2. Number of programs synthesised as we increase program size, with different timeouts.

EF, which iterates until the null character is encountered and then returns the pointer.

Figure 2 also shows that we can synthesise most loops with program size 5 and a timeout of only 3 minutes, which is encouraging for using this technique in practice. Increasing the program size or the timeout further provides only marginal benefits.

Another interesting observation is that there are some dips in the graph, e.g. when we increase the program size from 5 to 6 while keeping a 3-minute timeout. This is of course because increasing the program size has an impact on performance, as the search space of our synthesis algorithm increases with each additional character in the synthesised program.

4.2.3 Optimising the Vocabulary

We next explore how the performance of the synthesis approach depends on the gadgets included in the vocabulary, and propose a method for optimising the vocabulary used. Before we go into discussing results, let us first formalise our investigation.

Let N be the number of gadgets in our universe, and $G_1, G_2, \ldots G_N$ be the actual gadgets. In our concrete instantiation presented in Table 1, $N = 13$ and the gadgets are *rawmemchr*, *strchr*, etc.

We represent the vocabulary employed by the synthesis algorithm using a bitvector $v \in \{0,1\}^N$, where bit v_i of the vector is set if and only if the gadget G_i is used. For instance, a vocabulary allowing only the *rawmemchr* instruction would be represented as 1000000000000, a full vocabulary would be 1111111111111, and the vocabulary with three gadgets whose interpreter is shown in Algorithm 1 would be represented as 0000101000001.

The number of programs we can synthesise in a given time can be seen as a function of the vocabulary used, let us call it the success function $s\colon \{0,1\}^N \rightarrow \mathbb{N}$.

Understanding the influence of the vocabulary and choosing the best possible vocabulary is then equivalent to exploring the behaviour of s. While we could exhaustively evaluate s, it would take a long time, as we would need to $2^{13} = 8192$ experiments. Therefore, we use instead Gaussian Processes [29] to effectively explore s.

Table 4. The 7 vocabularies that perform better than the 2-hour experiment of §4.2.1.

Vocabularies	Synthesised programs
MPNIFV	81
MPNBIFV	80
PNIFV, MPNIFVS, MPNBXIFV	78

While describing Gaussian Processes (GPs) in detail is beyond the scope of this paper, we will give an intuition as to why they are useful in this case. GPs can be thought of as a probability distribution over functions. Initially, when we know nothing, they can represent any function (including s) albeit with small probability. The main idea is to get a real evaluation of s and refine the GP to better approximate s. Let's say we evaluate $s(v) = n$. We can now refine the GP to have the value n at v with 0 variance. The variances around v also decrease, while variances far away from v do not.

We can repeat this refinement for other values of v to get an increasingly more precise model of s with the GP. The GP also tells us where to evaluate next. We can use GPs to optimise s—i.e. to optimise the vocabulary with respect to the number of programs we can synthesise—by looking at points where s is likely to have a large value [46].

We used GPyOpt [4] for an implementation of GPs. We use an expected improvement acquisition function, which is well suited for optimisation.

In our experiments, we chose to optimise the vocabulary by using a maximum program size of 7 and a timeout of 5 minutes per loop. As we will show, this is enough to beat the results of §4.2.1 that used a maximum program size of 9 and a much larger 2-hour timeout per loop. Our optimised vocabulary is obviously optimised for our benchmarks and might not generalise to a different set of benchmarks, however the optimisation technique should generalise.

The optimisation process evaluated s 40 times, and to our surprise, found 5 vocabularies that achieve better results in 5 minutes/loop than those achieved in 2 hours/loop in §4.2.1. These vocabularies are shown in Table 4. The largest number of loops synthesised was 81, for a vocabulary containing *rawmemchr*, *strspn*, *strcspn*, *increment*, *return* and *reverse*. The smallest vocabulary which still beats the results of §4.2.1 (in just 5 minutes) consists of only 5 gadgets: *strspn*, *strcspn*, *increment*, *return* and *reverse*. We note that the *reverse* gadget is never synthesised in the 2-hour experiment, because it is too expensive in the context of a full vocabulary. But as the GP discovered significantly smaller vocabularies, *reverse* was now used several times.

We also note that the programs synthesised by the best GP-discovered vocabulary (MPNIFV) do not subsume the programs synthesised by the full vocabulary used in §4.2.1. For instance, there were 11 loops synthesised by MPNIFV and

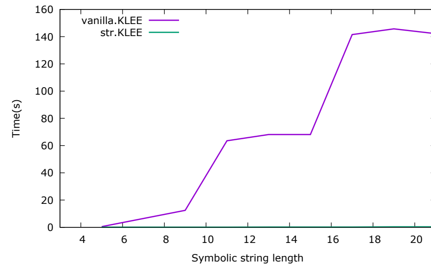

Figure 3. Mean time to execute all loops with str.KLEE and vanilla.KLEE, as we increase the length of input strings.

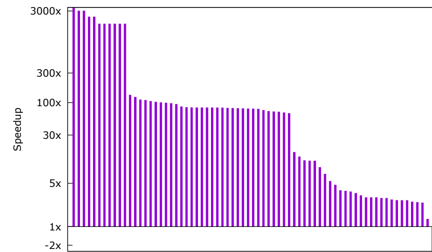

Figure 4. The speedup for each loop by str.KLEE over vanilla.KLEE for inputs of length 13, sorted by speedup value.

not the full vocabulary. These were mostly loops that look for a *strspn* from the end of the string. Conversely, there were 7 loops synthesised by the full vocabulary but not the `MPNIFV` vocabulary. Three of these were loops that required 3 arguments to *strspn*, one was a *strchr*, and 3 were a *strrchr* loop. Overall, the vocabularies in Tables 3 and 4 synthesised 88 out of the 115 loops.

4.3 Loop Summaries in Symbolic Execution

The next three sections discuss various scenarios that can benefit from our loop summarisation approach: symbolic execution (this section), compiler optimisations (§4.4) and refactoring (§4.5).

Recent years have seen the development of several efficient constraint solvers for the theory of strings, such as CVC4 [28], HAMPI [24] and Z3str [5, 53, 54]. These solvers can directly benefit symbolic execution, a program analysis technique that we also use in our approach. However, to be able to use these solvers, constraints have to be expressed in terms of a finite vocabulary that they support; standard string functions can be easily mapped to this vocabulary, but custom loops cannot.

To measure the benefits of using a string solver instead of directly executing the original loops, we wrote an extension to KLEE [6] (based on KLEE revision 9723acd) that can translate our loop summaries into constraints over the theory of strings and pass them to Z3 version 4.6.1. We refer to this extension as *str.KLEE*, and to the unmodified version of KLEE as *vanilla.KLEE*.

Figure 3 shows the average time difference across all loops between these two versions of KLEE when we grow the symbolic string length. We use a 240-second timeout and show the average execution time across all loops. For small strings of up to length 8 the difference is negligible, but then it skyrockets until we hit a plateau where some loops start to time out with vanilla.KLEE. In contrast, str.KLEE's average execution time increases insignificantly, with an average execution time under 0.36s for all string lengths considered.

Figure 4 shows the speedup achieved for each loop by str.KLEE over vanilla.KLEE for symbolic input strings of length 13. For over half of the loops, str.KLEE achieves a speedup of more than two orders of magnitude, with several loops experiencing speedups of over 1000x (these include loops where str.KLEE times out, and where the actual speedup might be even higher). For others, we see smaller but still significant speedups. There is a single loop where str.KLEE does worse by a factor of 2.5x; this is a *strlen* function where it takes 1.4s, compared to 0.5s for vanilla.KLEE.

4.4 Loop Summaries for Optimisation

Compilers usually provide built-in functions for standard operations such as *strcat*, *strchr*, *strcmp*, etc. for optimisation purposes[7,8]. The reason is that such functions can often be implemented more efficiently, e.g. by taking advantage of certain hardware features such as SIMD instructions. In addition, compilers often try to identify loops that can be translated to such built-in functions. E.g., `LoopIdiomRecognize.cpp` in LLVM 8 "implements an idiom recogniser that transforms simple loops into a non-loop form. In cases that this kicks in, it can be a significant performance win."[9]

However such loop recognises are typically highly specialised for certain functions, so we wanted to understand if our more general technique could be helpful to compiler writers in recognising more loops. To do so, we built a simple compiler that translates our vocabulary back to C. We then benchmarked the compiled summary against the original loop function.

We ran each function for 10 million times on a workload of four strings about 20 characters in length. Picking the strings has a large impact on the execution time and it is difficult to choose strings that are representative across all loops, since they come from different programs and modules within those programs. Therefore, we make no claim that this optimisation should always be performed, rather we try to highlight that there are indeed cases where summarising loops into standard string functions can be beneficial.

The programs were compiled with GCC version 5.4.0 using -O3 -march optimisation level. Figure 5 shows the results of this experiment. The bars going up show cases in which the

[7] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html
[8] https://clang.llvm.org/docs/LanguageExtensions.html
[9] http://llvm.org/doxygen/LoopIdiomRecognize_8cpp_source.html

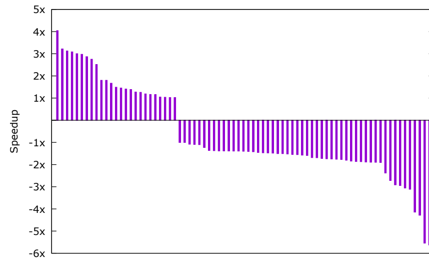

Figure 5. Relative time between running the original loop vs. running the synthesised program for each loop.

summarised version is faster, and bars going down cases in which it is slower. Whether the summary achieves a speedup or slowdown is mostly due to the function being synthesised. For instance, loops summarised by `strrchr` see important speedups because the glibc implementation of `strrchr` uses hand-crafted assembly based on fast vectorized instructions. Replacing the loop with a call to `strrchr` is beneficial here as the compiler does not optimise the loop enough to compete with handcrafted assembly.

By contrast, loops summarised by `strspn` are slower, because the glibc implementation of `strspn` builds a lookup table of the characters it needs to span over, which is expensive. This is because `strspn` is optimised for cases where one wants to span over a large set of characters and the span is large. In our example we had loops that e.g. simply skip whitespace, which does not fit this pattern.

Therefore, our data suggests that it is always beneficial to replace `strrchr`, whereas for `strspn` we should only replace when the accept string is long. A similar analysis can be performed for the other string functions in our vocabulary.

4.5 Loop Summaries for Refactoring

Our technique could also be incorporated into a refactoring engine. We envision an IDE that would highlight certain loops and suggest to developers a change that would replace them with more readable and less error-prone calls to standard string functions.

We decided to evaluate the refactoring potential of our loop summarisation approach by manually submitting some patches that summarise loops in five open-source projects, selected from those used to build our loop database (see §4.1).

We submitted patches to five different applications, three of which accepted some or all of our proposed changes. Figure 6 shows three loop summarisation patches that were accepted in *patch*, *libosip* and *wget*.

The responses we received show that depending on the functionality involved, some developers prefer to have loops, while others prefer functions. For example, developers who rejected our `strspn` summaries cited as reasons `strspn`'s relative obscurity and the performance implications that we also discovered in §4.4. On the other hand, the developers who accepted our patches found the standard string functions more readable. In *wget*, we also noticed prior patches from developers that did similar replacements.

5 Limitations

Our approach, as presented, is limited by the single pointer input/output interface to which loops have to conform. This restriction could be relaxed. For example, allowing an integer output instead of a pointer could be achieved with minor engineering effort. Allowing for loops that take two strings as input would be a larger effort. It would require both moderate engineering effort and a new small-model theorem. The synthesis will also require new gadgets conforming to the two-pointer interface. The new small-model theorem could be difficult to prove because the loop traverses two lists, but could exploit the fact that the loops are traversed in-sync.

While the synthesis algorithm supports any code conforming to the single pointer interface as gadgets, gadgets are limited by the scalability of symbolic execution. The gadgets we chose are either constant (i.e. *isnull*), linear (i.e. *strchr*) or quadratic (i.e. *strspn*) in terms of their input size. Gadgets of higher than quadratic complexity would likely make the synthesis impractical.

We recognise that some of the loops we summarise could be recognised by more lightweight approaches such as source code pattern matching or scalar evolution approaches such as the one used by LLVM's LoopIdiomRecognize. However, it would be difficult to apply these approaches for complex loops that require additional modifications, such as conditionally incrementing a pointer after loop exit, setting it to the end of the string etc. More generally, pattern matching approaches require great manual effort in finding the patterns and then encoding them. In fact, during development we found it difficult to manually synthesise loops because equivalence checking kept finding incorrectly-handled edge cases, so we preferred to tweak the synthesis parameters rather than attempting to manually synthesise the loops.

In §4.3, we show large increases in the scalability of symbolic execution with our summaries. This does not directly imply the same speed-ups would be observed when running whole programs with loops summaries, however we believe this work is an important step towards scaling symbolic execution to large strings.

6 Related Work

Similar to our work, S-Looper [52] automatically summarises loops with the aim of improving program analysis. Their technique uses static analysis to enhance buffer-overflow detection. Our work is more general in that it is applicable to any analysis that operates on C directly, generating human-readable summaries that can even be used for refactoring.

Godefroid and Luchaup [17] use partial loop summarisation to enable concolic execution to reason about multiple

Program	Before	After
patch	`while (*s != '\n')` `  s++;`	`s = rawmemchr(s, '\n');`

libosip	`while ((' ' == *pbeg) || ('\r' == *pbeg)`
	`    || ('\n' == *pbeg) || ('\t' == *pbeg))`
	`  pbeg++;`
		`pbeg += strspn(pbeg, " \r\n\t");`

wget	`p = path + strlen (path);`
	`for (; *p != '/' && p != path; p--)`
	`  ;`
		`p = strrchr(path, '/');`
		`p = p == NULL ? path : p;`

Figure 6. Examples of loop summarisation patches accepted by developers.

paths through a loop at once. Their summaries consist of pre- and post-conditions, which they automatically infer during concolic execution. Similarly, *loop-extended symbolic-execution* [38] uses a combination of symbolic execution and static analysis to summarise loops in order to speed up symbolic execution. As for S-Looper, these two approaches are intertwined with their analysis, unlike our approach which can be immediately used in any technique.

STOKE [39] is an assembly level superoptimizer that speeds up loop-free code segments. With its recent extension to loops [10] their work is similar in spirit. They also use bounded verification to aid synthesis, but instead of a small-world theorem they use a sound verifier to generalise to arbitrary bounds. Their work focuses on optimising libc functions, whereas our work focuses on summarising loops in arbitrary programs, therefore we believe the work is complementary.

Srivastava et al. [47] present an approach synthesising loops from pre- and post-conditions using a verifier. While more precise, they require user-specified annotations, making it inapplicable as an automatic summarisation technique.

LLVM's LoopIdiomRecognize pass attempts to replace loops that match `memset` or `memcpy` patterns and is quite specific to these functions (other compilers, such as GCC, have similar passes that recognise patterns). It detects induction variables from which it can recognise stride load and store instructions. Their to-do includes functions like `strlen` for over 6 years, showing that such passes require significant expertise to implement. By contrast, our approach is more general and can easily be extended by adding a gadget.

More generally, program synthesis has seen renewed interest in recent years [1, 13, 20–22, 31, 43, 47]. Our synthesis approach is based on CEGIS [44], with the synthesizer and verifier both based on dynamic symbolic execution [42].

Program equivalence may be considered one of the most important problems in formal verification and has been the subject of decades of research [48]. Due to the vast literature on the topic and space, we only briefly review the subject.

Proving program equivalence is useful in many domains ranging from translation validation [26, 30, 34, 36, 41], regression verification [18, 19], automatic merging [45], semantic differencing [14], and cross-version verification [23, 27].

One common approach for attacking the problem, e.g., [51], is establishing a simulation invariant between the states of two programs. Tracking the simulation enables defining a so-called correlating semantics which allows reasoning about correlated (interleaved) execution of two programs [3, 14, 49]. In contrast to these techniques, our approach focuses on establishing the equivalence of programs without co-executing them, but instead examines their input/output behaviour on bounded examples using symbolic execution.

Symbolic execution-based methods [6, 9, 11, 12, 32, 35] often focus on practical equivalence verification up to a certain input bound. In contrast, we speculatively search for a synthesised program that agrees with the investigated loop on bounded inputs, and develop a small model theorem [33] which allows us to lift symbolic execution validated bounded equivalence to full equivalence.

7 Conclusion

In this paper, we presented a novel approach for summarising loops in C code. This approach uses counterexample-guided synthesis to generate the loop summaries, Gaussian processes to optimise the vocabulary used, and a formal proof to show that the summaries are correct for unbounded loop lengths if they are correct for the first two iterations.

We evaluated our approach on a large loop database extracted from popular open-source systems and assessed its utility in several contexts: symbolic execution, where we recorded speedups of several orders of magnitude; compiler optimisations, where several summaries resulted in significant performance improvements; and refactoring, where some of our summary patches were accepted by developers.

Acknowledgements

We thank Frank Busse for his help in debugging our synthesis implementation, the PLDI reviewers for their valuable feedback, and Daniel Grumberg and Martin Nowack for proofreading our paper. This research was sponsored by the UK EPSRC via grant EP/N007166/1 and a PhD studentship, the Len Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, the Pazy Foundation, and the Israel Science Foundation (ISF) grant No. 1996/18.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample Guided Inductive Synthesis Modulo Theories. In *Proc. of the 30th International Conference on Computer-Aided Verification (CAV'18)*.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley.

[3] Daphna Amit, Noam Rinetzky, Tom Reps, Mooly Sagiv, and Eran Yahav. 2007. Comparison under abstraction for verifying linearizability. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)*.

[4] The GPyOpt authors. 2016. GPyOpt: A Bayesian Optimization framework in python. http://github.com/SheffieldML/GPyOpt.

[5] M. Berzish, V. Ganesh, and Y. Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. In *Proc. of the 17th Formal Methods in Computer-Aided Design (FMCAD'17)*.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[7] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*.

[8] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90.

[9] Maria Christakis and Patrice Godefroid. 2015. Proving Memory Safety of the ANI Windows Image Parser Using Compositional Exhaustive Testing. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*.

[10] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*.

[11] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-Point and SIMD Code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*.

[12] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Testing of OpenCL Code. In *Proc. of the Haifa Verification Conference (HVC'11)*.

[13] Cristina David, Pascal Kesseli, Daniel Kroening, and Matt Lewis. 2018. Program Synthesis for Program Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 40, 2 (May 2018), 5:1–5:45.

[14] Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*.

[15] Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. 2013. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*.

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*.

[17] Patrice Godefroid and Daniel Luchaup. 2011. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*.

[18] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proc. of the 46th Design Automation Conference (DAC'09)*.

[19] Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Software Testing Verification and Reliability (STVR)* 23, 3 (2013), 241–258.

[20] Sumit Gulwani. 2010. Dimensions in Program Synthesis. In *Proc. of the 12th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'10)*.

[21] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-output Examples. In *Proc. of the 38th ACM Symposium on the Principles of Programming Languages (POPL'11)*.

[22] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11)*.

[23] Chris Hawblitzel, Shuvendu K. Lahiri, Kshama Pawar, Hammad Hashmi, Sedar Gokbulut, Lakshan Fernando, Dave Detlefs, and Scott Wadsworth. 2013. Will You Still Compile Me Tomorrow? Static Crossversion Compiler Validation. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.

[24] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: A Solver for String Constraints. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*.

[25] James C. King. 1976. Symbolic execution and program testing. *Communications of the Association for Computing Machinery (CACM)* 19, 7 (1976), 385–394.

[26] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. 2009. Proving Optimizations Correct Using Parameterized Program Equivalence. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'09)*.

[27] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proc. of the 24th International Conference on Computer-Aided Verification (CAV'12)*.

[28] Tianyi Liang, Andrew Reynolds, Nestan Tsiskaridze, Cesare Tinelli, Clark Barrett, and Morgan Deters. 2016. An Efficient SMT Solver for String Constraints. *Formal Methods in System Design (FMSD)* 48, 3 (June 2016), 206–234.

[29] David J. C. MacKay. 2002. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press.

[30] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'00)*.

[31] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. 2014. Test-driven Synthesis. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'14)*.

[32] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 2008. Differential symbolic execution. In *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'08)*.

[33] Amir Pnueli, Yoav Rodeh, Ofer Strichman, and Michael Siegel. 2003. The Small Model Property: How Small Can It Be? *Information and Computation* 184, 1 (2003), 227.

[34] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proc. of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*.

[35] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*.

[36] Hanan Samet. 1976. Compiler Testing via Symbolic Interpretation. In *Proc. of the 1976 Annual Conference (ACM'76)*.

[37] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'10)*.

[38] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended Symbolic Execution on Binary Programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'09)*.

[39] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proc. of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*

(ASPLOS'13).

[40] Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. 2007. Abstracting Symbolic Execution with String Analysis. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*.

[41] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven Equivalence Checking. In *Proc. of the 28th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'13)*.

[42] V. Sharma, K. Hietala, and S. McCamant. 2018. Finding Substitutable Binary Code for Reverse Engineering by Synthesizing Adapters. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'18)*.

[43] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*.

[44] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proc. of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*.

[45] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified Three-way Program Merge. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 165 (Oct. 2018), 29 pages.

[46] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *Proc. of the 27th International Conference on International Conference on Machine Learning (ICML'10)*.

[47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proc. of the 37th ACM Symposium on the Principles of Programming Languages (POPL'10)*.

[48] Ofer Strichman. 2018. Special issue: program equivalence. *Formal Methods in System Design* 52, 3 (01 Jun 2018), 227–228.

[49] Tachio Terauchi and Alexander Aiken. 2005. Secure Information Flow as a Safety Problem. In *Proc. of the 12th International Static Analysis Symposium (SAS'05)*.

[50] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proc. of the 21st ACM Conference on Computer and Communications Security (CCS'14)*.

[51] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2017. Verifying Equivalence of Database-driven Applications. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 56:1–56:29.

[52] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic Summarization for Multipath String Loops. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*.

[53] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3Str2: An Efficient Solver for Strings, Regular Expressions, and Length Constraints. *Formal Methods in System Design (FMSD)* 50 (June 2017), 249–288.

[54] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*.