S-Looper: Automatic Summarization for Multipath String Loops

Xiaofei Xie[1*] Yang Liu[2] Wei Le[3] Xiaohong Li[1†] Hongxu Chen[2]

[1]Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin University, China
{xiexiaofei, xiaohongli}@tju.edu.cn
[2]Nanyang Technological University, Singapore {yangliu, hxchen}@ntu.edu.sg
[3]Iowa State University, USA weile@iastate.edu

ABSTRACT

Loops are important yet most challenging program constructs to analyze for various program analysis tasks. Existing loop analysis techniques mainly handle well loops that contain only integer variables with a single path in the loop body. The key challenge in summarizing a multiple-path loop is that a loop traversal can yield a large number of possibilities due to the different execution orders of these paths located in the loop; when a loop contains a conditional branch related to string content, we potentially need to track every character in the string for loop summarization, which is expensive. In this paper, we propose an approach, named $S\text{-}Looper$, to automatically summarize a type of loops related to a string traversal. This type of loops can contain multiple paths, and the branch conditions in the loop can be related to string content. Our approach is to identify patterns of the string based on the branch conditions along each path in the loop. Based on such patterns, we then generate a loop summary that describes the path conditions of a loop traversal as well as the symbolic values of each variable at the exit of a loop. Combined with vulnerability conditions, we are thus able to generate test inputs that traverse a loop in a specific way and lead to exploitation. Our experiments show that handling such string loops can largely improve the buffer overflow detection capabilities of the existing symbolic analysis tool. We also compared our techniques with KLEE and PEX, and show that we can generate test inputs more effectively and efficiently.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

[*]The work of this author was done in NTU.

[†]Xiaohong Li is the corresponding author, School of Computer Science and Technology, Tianjin University.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771815

General Terms

Algorithms, Reliability, Verification

Keywords

Symbolic execution, Loop summarization, String constraints

1. INTRODUCTION

With the recent advance of constraint solvers [8, 2, 9], symbolic execution [7, 15] has demonstrated an initial success in bug finding and test input generation [23, 11, 5, 20, 13, 3]. Symbolic execution substitutes inputs with symbolic values and symbolically performs program operations on values. When symbolic execution reaches a branch, it performs a fork and creates two different symbolic states. On each path, it maintains a set of path conditions which is a formula over the symbolic values that must hold in order to exercise the designated path. A symbolic execution tree is created to represent all execution paths during the symbolic execution process.

A key challenge of symbolic execution is to handle loops in the programs. On every iteration of a loop, the symbolic execution tree creates one or more new branches, and the size of the path conditions can become too large with even a simple loop. Even worse, some loops are dependent on the input of the program, and the number of iterations for such loops can be unbounded [28]. A recent study [28] shows that symbolic execution may keep unfolding the loop without covering any new branches and fail to detect bugs and achieve high coverage testing.

There have been techniques proposed to summarize the loop and compute the symbolic values at the exit of the loop [25, 14, 10, 19, 22]. However, these tools do not handle well when there are multiple paths in the loop and when the variables in the loop are not limited to integers. For example, even for the simple loop shown in Fig. 1, none of the existing techniques (to the best of our knowledge) can report the loop effect on variables i, f and fb at line 11. First, the conditions at lines 5 and 6 in the loop are related to the content of an input string, which is difficult to model using integer constraints. Second, there exists a branch condition in the loop body at line 6, and dependent on whether the branch is taken on each iteration of a loop, there can be different interleavings between paths in the loop. Existing string solvers [30, 27] can generate string constraints from a program statement but not from a multipath loop. On the other hand, being able to handle such string loops is necessary to determine many vulnerabilities. For example, we would fail to detect the buffer overflow at line 11 in Fig. 1, because we do not know the bound of fb at the exit of the loop.

We did a survey on a set of representative loop summarization tools, including APC, LESE and APLS [25, 14, 22]. Shown in Ta-

Table 1: Loop summarization tools comparison

Challenges	APC [25]	LESE [22]	APLS [14]	S-Looper
Multipath loops	Yes	No	No	Yes
String content	No	Partial	No	Yes

ble 1, we found that only APC can analyze multiple-path loops but APC mainly handles induction variables and integer array. LESE handles branch conditions related to string. But it only works when the condition is related to properties of a string such as the length of the string but not the content of a string in general.

In this paper, we present *S-Looper*, a novel static technique for summarizing multipath, string loops. We specifically target a type of loops that perform a string traversal and the conditional branches in the loop compare string properties such as string content or length to define patterns of the string for each path (Fig. 1 shows an example). In our empirical studies (shown in Section 4), we found such loops can commonly exist and are important for determining vulnerabilities and generating test inputs.

The key ideas of *S-Looper* are as follows. For handling multi-paths in the loop, we introduce *path counters* [25] to symbolically represent how many times a particular path in the loop is executed during a loop traversal. These path counters are then used to compute symbolic values for variables updated along the path. For handling string comparisons and operations in the loop, we introduce string operators that can be understood by the string constraint solvers [30, 27]. For resolving branch conditions related to string content, we identify the patterns of strings implied by these conditional branches and construct the corresponding string constraints.

S-Looper takes three steps to summarize the loop. In the first step, we construct a *flow graph* [25] which specifies the control flow of the loop. In the second step, we perform symbolic analysis on the flow graph to infer the range of the string index for each path. In the third step, we analyze path conditions for each path in the loop and generate string and integer constraints. The result of the analysis is the constraints, satisfying which a loop with designated path counters can be executed.

The loops we can handle satisfy the following conditions: 1) the variables used in the loops are the types of integer, Char pointer, string and/or an array of characters, 2) the variables used in conditional branches in the loop are loop induction variables (we also summarize loops with non-induction variables, but the results can be imprecise), 3) the number of times a path in the loop executes can have an impact on the loop summary; however, the order of the path interleavings in the loop does not have an impact, and 4) the loop does not contain nested loops.

We have implemented *S-Looper* and integrated it into the Marple framework [20] for detecting buffer overflows and generating test inputs. Our evaluation is conducted on the buffer overflow benchmarks designed by Ku et al. [17]. We show that using *S-Looper*, we are able to handle 75.7% of the loops previously reported as unknown. We are able to report 21 out of 37 known bugs. The analysis with loop unrolling for 3 times does not find any such vulnerabilities, and using loop unrolling for 8 times, we are only able to detect 13 of such bugs. Our experiments on test input generation show that compared to PEX [26] and KLEE [4], we are able to generate test inputs more efficiently and generate test inputs that both of the tools fail to generate.

In summary, the paper presents a loop summarization technique to handle loops with multiple paths and string manipulations. Although in this paper, we are not yet able to handle the general cases of all such loops, our techniques serve the first step for exploring a promising direction of a very hard but important program analysis problem. The contributions of the paper are three-fold.

```
1   #define MAXLINE 50
2   void request(char *input){
3      char fbuf[MAXLINE+1];
4      int i=0,fb=0,f=0;
5      while(input[i]!=‘\0’){
6        if(input[i]!=‘\n’ && f<=MAXLINE)
7          fb++;
8        f++;
9        i++;
10     }
11     fbuf[fb]=‘\0’;   //buffer overflow
12  }
```

Figure 1: A Simplified Version from Sendmail

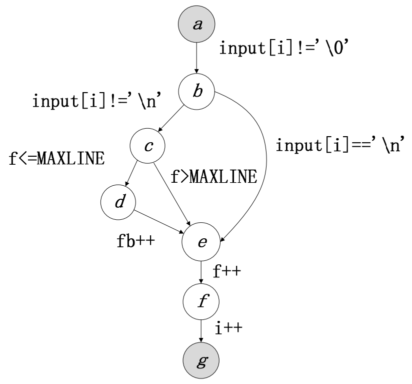


Figure 2: The flow graph for the loop program in Figure 1

- We propose an automatic approach to summarize the multipath string loops based on the reasoning about variable value changes along different paths and pattern matching.
- We provide analytical understandings on which loops we can and cannot handle.
- We implement *S-Looper* and integrate it with existing symbolic analysis tool. Experimental results show that our techniques are effective and can improve the state of the art symbolic execution for finding bugs and generating test inputs.

The rest of the paper is organized as follows. In Section 2, we use an example to present an overview of the work. In Section 3, we explain the detailed algorithm. In Section 4, we present the experimental results, followed by the related work in Section 5 and conclusions in Section 6.

2. MOTIVATING EXAMPLE

Here, we use a simple example to provide an intuitive description of our approach. In Fig. 1, we show a C code snippet from *sendmail*. Suppose we aim to determine whether the code contains a buffer overflow at line 11. Without loss of generality, we use a demand-driven buffer overflow detection algorithm [20] to explain how we detect this buffer overflow.

At line 11, we raise a query Q: [$fb > \mathsf{MAXLINE}$] to inquiry whether the vulnerability condition—-the value of fb is larger than MAXLINE—-can occur along any path that traverses the loop at lines 5–10 and reaches line 11. To determine the resolution of this query, we propagate the query backwards. At lines 5–10, the analysis encounters a loop.

To perform loop analysis and resolve the path conditions related to the loop traversal, we first construct a *flow graph* that specifies program paths in a loop, shown in Fig. 2. Node a is the entry of the loop at line 5 in Fig. 1, and node g is the exit of the loop at line 10. Each edge in the flow graph represents an assignment or a conditional test in the loop.

The flow graph shows that on each loop iteration, the execution can potentially exercise one of the following three paths $\langle abcdefg\rangle$, $\langle abcefg\rangle$ or $\langle abefg\rangle$ in the loop. We introduce *path counters* to symbolically denote how many times the three paths in a loop are executed during a loop traversal: κ_1 for path $\langle abcdefg\rangle$, κ_2 for $\langle abcefg\rangle$, and κ_3 for $\langle abefg\rangle$. We aim at using these path counters to summarize the effect of each path on the loop summary.

From the flow graph, we identify loop induction variables i, f and fb—-their values are increased in a fixed amount on every loop iteration. Suppose the initial values of i, f and fb before entering the loop are i_0, f_0 and fb_0. The values of the three variables after traversing the three paths κ_1, κ_2, and κ_3 times respectively thus can be represented as:

$$\begin{array}{l} \kappa_1 \geq 0, \kappa_2 \geq 0, \kappa_3 \geq 0 \\ i = i_0 + 1 \times \kappa_1 + 1 \times \kappa_2 + 1 \times \kappa_3 \\ f = f_0 + 1 \times \kappa_1 + 1 \times \kappa_2 + 1 \times \kappa_3 \\ fb = fb_0 + 1 \times \kappa_1 + 0 \times \kappa_2 + 0 \times \kappa_3 \end{array} \tag{1}$$

The above formulas indicate that i and f are updated in every iteration independent of which path in the loop taken; in each iteration, the values are increased by one. fb is only updated along path $\langle abcdefg\rangle$. Thus, fb is only related to κ_1.

In the second step, we perform symbolic analysis on the flow graph to identify the range of the string index i for each path. Along paths $\langle abcdefg\rangle$ and $\langle abcefg\rangle$, we determine $f == i$ holds for every iteration of the loop and $fb_0 == i_0$. Thus, $i \leq \mathsf{MAXLINE}$ hold for $\langle abcdefg\rangle$, and $i > \mathsf{MAXLINE}$ hold for $\langle abcefg\rangle$. Along path $\langle abefg\rangle$, there is no additional constraint for i, and thus i can be any value between i_0 and the length of the string $input$.

In the third step, we specify the path conditions in the loop using string and integer constraints. Specifically, to handle string content in the path conditions, we identify patterns of the strings from the conditions. For example, along path $\langle abcdefg\rangle$, we have conditions $input[i]! =$ '\0', $input[i]$!='$\backslash n$' and $i \leq \mathsf{MAXLINE}$. These conditions imply that there are at least κ_1 characters that are neither '\0' nor '$\backslash n$' in $input$ between its i_0^{th} and $\mathsf{MAXLINE}^{th}$ characters. We express such strings using the following constraints:

$$\begin{array}{l} str = input[i_0, \mathsf{MAXLINE}] \\ str_1 = str(\text{“}\backslash 0\text{”}/\text{“”}) \\ str_2 = str_1(\text{“}\backslash n\text{”}/\text{“”}) \\ |str_2| \geq \kappa_1 \end{array} \tag{2}$$

In this paper, we write $STR[a, b]$ to denote a substring of string STR from index a to index b. $|STR|$ returns the length of string STR. $STR(s_1/s_2)$ denotes a new string by replacing all occurrences of string s_1 by string s_2 in STR. $contains(STR, s)$ denotes the string STR contains the substring s. Applied to the above formulas, the first constraint selects a sequence of characters from $input$ in the range of i_0 and $\mathsf{MAXLINE}$. The second and third constraints suggest that after removing '\0' and '$\backslash n$' from the substring generated above, the length of str_2 is still greater or equal to κ_1.

Similarly, for path conditions, $input[i]! =$ '\0', $input[i]$!='$\backslash n$' and $i > \mathsf{MAXLINE}$ along $\langle abcefg\rangle$, we construct the following constraints, which say that the substring of $input$ in the range between $\mathsf{MAXLINE}+1$ and $i-1$ at least has κ_2 characters that do not contain '\0' and '$\backslash n$'.

$$\begin{array}{l} str_3 = input[\mathsf{MAXLINE} + 1, i - 1] \\ str_4 = str_3(\text{“}\backslash 0\text{”}/\text{“”}) \\ str_5 = str_4(\text{“}\backslash n\text{”}/\text{“”}) \\ |str_5| \geq \kappa_2 \end{array} \tag{3}$$

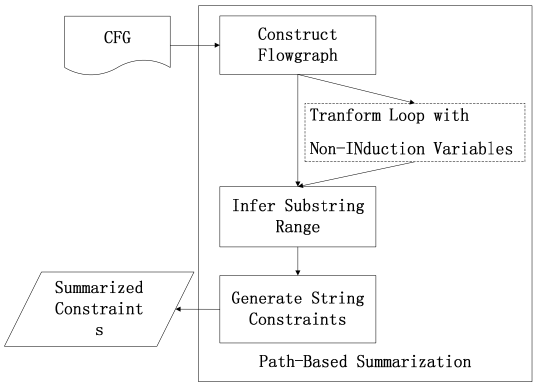


Figure 3: Overview of the S-Looper

Analyzing the path conditions for $\langle abefg\rangle$ including $input[i]! =$ '\0' and $input[i] ==$ '$\backslash n$', we determine that a string whose i^{th} character is '$\backslash n$' satisfies the condition. The following constraints can describe the pattern:

$$\begin{array}{l} str_6 = input[i_0, i - 1] \\ str_7 = str_6(\text{“}\backslash n\text{”}/\text{“”}) \\ |str_6| - |str_7| \geq \kappa_3 \end{array} \tag{4}$$

The formula means that for a substring that gets the characters of $input$ from the range of i_0 to $i-1$, the number of '$\backslash n$', represented as $|str_6| - |str_7|$, should be at least κ_3.

Finally, the path condition at the entry of the loop $input[i]! =$ '\0' suggests that during any iteration of the loop, we should not encounter a '\0' character. The constraint thus is

$$\neg contains(str_6, \text{“}\backslash 0\text{”}) \tag{5}$$

After processing all the path conditions in the loop, we conjunct constraints (1) to (5) and generate the loop summary. To determine whether exists $input$ that can lead to a feasible execution to line 11 and exploit the vulnerability, we conjunct constraints (1) to (5) together with the vulnerability condition $fb > \mathsf{MAXLINE}$. Given the fact that $f_0 = 0$, $fb_0 = 0$ and $i_0 = 0$ obtained at line 4, the constraint solver reports an instance $input =$ “$a...a\backslash 0$” (there are 51 'a'), κ_1=51, κ_2=κ_3=0 that can trigger the vulnerability.

3. PATH-BASED LOOP SUMMARIZATION

This section is devoted to the detailed explanation of our approach. Fig. 3 shows the work flow of our approach. *S-Looper* takes the the control flow graph (CFG) of the loop as input, and summarizes the loop constraint in three steps: 1) Firstly, we construct the corresponding flow graph from the CFG, and identifies induction variables of the loop (Section 3.3). If there are non-induction variables in the loop, we will try to make the variables inducible by deleting the non-induction paths (Section 3.6). 2) For each path in the flow graph, the range inference component infers the range of the substring that the path covers (Section 3.4). 3) The string constraints generation component then generates the string constraints based on extracting pattern for each path (Section 3.5). In the end, *S-Looper* returns the conjunction of the string constraints for all paths as the loop summary.

3.1 Preliminaries

In this paper, our target is the string loop which traverses a string variable STR with index variable $\imath$ changing monotonically (initially having value $\imath_0$). For example, if STR is a char array, $\imath$ is the array index. For the ease of presentation, we write the string loops in the while programming language[1], extended with *string conditions* in the form of $STR[\imath] \bowtie C$, where $\bowtie \in \{=, \neq\}$ is the comparison operator, and C is a char. For a variable v used in the loop, we use v_0 to denote its value before executing the loop.

We adopt the notion of flow graph [25] to perform the loop summarization, which is formally defined below.

DEFINITION 1. *A flow graph is a tuple $G = (V, E, l_s, l_t, \iota)$, where V is a set of vertices, $E : V \times V$ is a set of edges linking the vertices, $l_s, l_t \in V$ are entry and end nodes in the loop, ι is a function assigning every edge $e \in E$ an instruction $\iota(e)$.*

Given a flow graph, we define a path π as a finite sequence $\langle v_1 v_2 \ldots v_k \rangle$ of nodes such that $(v_i, v_{i+1}) \in E$, where $k \geq 1, v_1 = l_s$ and $v_k = l_t$. In control flow diagram, one path in the loop is a circle. Flow graph removes the edge (v_i, l_s) which goes back to the entry node l_s from some node v_i of the loop and adds one edge (v_i, l_t) pointing to the end node l_t. Clearly, a flow graph can be decomposed to a finite number of paths. For example, the flow graph in Fig. 2 has three paths $\langle abcdefg \rangle$, $\langle abcefg \rangle$ and $\langle abdfg \rangle$.

To understand the effect of path execution quantitatively, we define a variable in the loop as an *induction variable* (IV) if it is changed by a constant amount during each iteration of every path.

DEFINITION 2. *A condition is IV-dependent [14] if it is in the form $L \sim R$ where $\sim \in \{<, \leq, >, \geq, =, \neq\}$, L and R are expressions, and a dummy variable assigned with the value $L - R$ is an IV in the loop.*

In Fig. 1, f, i and fb are all induction variable. Note that fb has different changes in the three paths, but for each path, the change is constant. $f \leq \mathsf{MAXLINE}$ and $f > \mathsf{MAXLINE}$ are two IV-dependent conditions because $dummy = f - \mathsf{MAXLINE}$ is also an induction variable in this loop.

3.2 S-Looper Algorithm

Algorithm 1 illustrates the detailed computation of *S-Looper*. The algorithm takes the CFG of the target string loop as the input, and returns the summarized constraints ϕ for the loop. At line 1, it constructs the flow graph of the loop fl, identifies induction variables and non-induction variables in V_I and V_{NI} respectively. Lines 3 to 3 handle the non induction variables, if any. For each non-induction variable v_i, we invoke ReduceNonIV function to make an attempt to reduce it to an induction variable. If the attempt succeeds, we add v_i into V_I, i.e., v_i will be treated same as other IV afterwards. ReduceNonIV function returns the condition of the reduction, which is stored in predicate ϕ_{niv}. The details will be described in Section 3.6.

For the flow graph fl, we introduce a path counter κ_i for every path π_i to record the number of executions of π_i during a loop traversal (line 5). To generate the loop summary, there are three different kinds of string constraints to be considered.

Firstly, we find the post conditions of all IV after the loop and store them in ϕ_p. In order to find the value of the IVs after the loop, we track the paths where IVs are modified and summarize the changes. For each $v \in V_I$, let c_i be the amount that v changed in path π_i, the post value of v after loop is calculated as $v_p = v_0 + c_1 \times \kappa_1 + ... + c_n \times \kappa_n$. We store the constraint $v == v_p$ in ϕ_p at line 6. Similarly, the constraint for the index variable $\imath$ is also computed and saved in ϕ_0 since $\imath$ is also an induction variable.

Secondly, we calculate the constraints on each path as shown from lines 7-7. From the IV-dependent conditions in each path, we calculate the range of the traversed substring in each path at line 8.

[1]Other loops like for loop can be converted to the while loop.

Algorithm 1: *S-Looper* Algorithm

```
input  : $CFG$: the CFG of the input string loop
output: $\phi$: the summarized constraints for the input loop
1  $(fl, V_I, V_{NI}) \leftarrow$ Construct flow graph from $CFG$ ;
2  $\phi_{niv} \leftarrow true$;
3  foreach $v_i \in V_{NI}$ do
4      $\phi_{niv} \leftarrow \phi_{niv} \wedge$ ReduceNonIV$(fl, v_i, V_I)$;
5  Let $\kappa_1, \ldots, \kappa_n$ be the path counters for paths $\pi_1, \ldots, \pi_n$ in
   the flow graph $fl$;
6  $\phi_p \leftarrow$ Compute post loop constraint of $V_I$;
7  for $\pi_i$ from $\pi_1$ to $\pi_n$ do
8      $\gamma_i \leftarrow$ InferRange$(\pi_i, V_I)$;
9      $\phi_i \leftarrow$ GenStrCon$(\pi_i, \gamma_i)$;
10 $\phi_0 \leftarrow$ Generate loop entry constraint;
11 $\phi \leftarrow \phi_{niv} \wedge \phi_0 \wedge \cdots \wedge \phi_n \wedge \phi_p$ ;
12 return $\phi$;
```

For instance, in the example of Fig. 2, we can get the ranges [i_0, MAXLINE] and (MAXLINE, i) of the traversed substrings from the IV-dependent conditions $f \leq \mathsf{MAXLINE}$ and $f > \mathsf{MAXLINE}$. At line 9, the path constraints is returned by extracting the pattern from the string conditions of the path. For example, in the Fig. 2, we extract the pattern from conditions $input[i]! =$ '\0' $\wedge$ $input[i]$!='$\backslash n$' of path $\langle abcdefg \rangle$: there are at least κ_1 chars which are neither '\0' nor '\n' in the range [i_0, MAXLINE] of the input. The generated constraints can be seen in the Section 2.

Thirdly, we generate the constraints by analyzing the string conditions in the loop entry condition at line 10. Obviously, each path of the loop will execute the entry condition, which should be satisfied during the loop iterations. If the entry condition is $STR[\imath]! = C$, the generated constraint is $-contain(STR[\imath_0, \imath - 1], C)$. If the entry condition is $STR[\imath] = C$, the generated constraint is $STR[\imath_0, \imath - 1](C\backslash$"") ="".

Lastly, the summarized loop constraint is the conjunction of the generated constraints above as shown in line 11.

3.3 Flow Graph Construction

Given a CFG of a string loop program, we can construct its flow graph according to Definition 1. The process of flow graph construction and analysis is described below.

Firstly, *S-Looper* finds the paths and integer variables in the CFG by analyzing the basic blocks. We add the start node and end node for the graph, the back edge is changed and make them point to the end node in each path. Secondly, we perform a program slicing in every path of the CFG, then remove the irrelevant statements which are not related to STR and do not change the value of variables, e.g., I/O library calls like $print$. Thirdly, we track the statements which modified the value of the variables in each path, and get the value change in each iteration of the path. If the value change in each path is constant, the variable is an induction variable. Otherwise, we record the path in which the value change is not constant and mark the variable as a non-induction variable.

3.4 Substring Range Inference

To generate the string constraints with length abstraction, we compute the range of the substring traversed by each path. Given a path π with path counter κ, the traversed string content in the κ iterations of π should be included in the substring. In this section, we explain how to calculate the range of the substring covered by a path by analyzing the relationship between the string index variable $\imath$ and IV in the IV-dependent conditions of the path.

Firstly, we show the range inference idea using examples in Fig. 4, where we assume the initial value of variables i, fb are 0. Fig. 4(a) contains two IV-dependent conditions $i < 10$ and $i >= 10$ in the two paths (represented as π_1 and π_2 respectively). From the condition $i < 10$, we know the range of the traversed string in the path π_1 is [0,10), which decides the number of the iterations in π_1. As the condition is directly related to the string index i, we can get the range of the substring easily. In Fig. 4(b), the IV-dependent condition is $fb < 10$, which has no direct relationship with i. In order to calculate the range of the substring generally, we need to analyze the relationships between fb and i in the paths, which is the key purpose of the inference algorithm.

3.4.1 Inference Algorithm Preliminaries

For the ease of presenting and explaining the algorithm, we first introduce several notations as follows.

DEFINITION 3. *Given a flow graph $G = (V, E, l_s, l_t, \iota)$ and a branching node $\beta \in V$ with an outgoing edge e, let $c = \iota\{e\}$ be the branch condition, we define $G_\beta^c = (V_c, E_c, \beta, l_t, \iota)$ as a subgraph starting from β and e, where $V_c \subseteq V$ contains all nodes reachable from β via e and $E_c \subseteq E$ contains all the edges formed by V_c.*

For a branching node β which has two outgoing edges, we have two subgraphs G_β^c and $G_\beta^{\neg c}$ generated by the branch condition c and its negation $\neg c$. In order to express it briefly, we also call the branch as c if the condition of the branch is c. We simply write G^c and $G^{\neg c}$ if β is clear in the context.

Given a flow graph G_β^c and IV v in G, we use δ_v^G to denote the change of v in the graph G if the change of the induction variable v is same for all paths in the graph. We use δ_v^π to denote the change of v in a path π. Our target is to infer the range of the traversed string on the condition c.

Given a flow graph G with n paths $(\pi_1, \pi_2, \dots, \pi_n)$, we define $\rho_G = \kappa_1 + \dots + \kappa_n$ as the sum of all path counters in G. For an IV-dependent condition $L \sim R$ where $\sim \in \{<, \leq, >, \geq, =, \neq\}$, we can convert it to a reduced form $(Vl \sim Cr)$ such that Vl is an induction variable and Cr is an expression consisting of constant values and other variables which do not depend on the loop.

In the example of Fig. 4(b), the branch node has two outgoing edges whose conditions are $c = fb < 10$ and $\neg c = fb >= 10$. G^c is the subgraph starting from the branch c and it is the sub-path of π_1 after the edge $fb < 10$ here. The change value of fb in the subgraph is $\delta_{fb}^{G^c}$ and its value is 1. Particularly, $\delta_{fb}^{G^c}$ is equal to $\delta_{fb}^{\pi_1}$ since graph G^c only contains path π_1.

3.4.2 Inference Algorithm

Algorithm 2 takes one path π and the IV set V_I as the input, and returns the computed range γ of the substring (initially is $[\imath_0, \imath)$) for the path. Overall, Algorithm 2 calculates the number of executions of a branch with IV-dependent condition $b = (v \sim C_b)$ firstly, which is decided by the value change δ_v^G in the subgraph G^b and the value changes in other paths. Then the maximum range of the substring covered by the path can be calculated by analyzing the symbolic relationship between string index $\imath$ and the induction variable v. If we cannot compute the range of the substring, it will use the default range $[\imath_0, \imath)$. In the algorithm, we only consider the operators $\{<, \leq, >, \geq\}$ of the condition b only. The operators $=$ and $\neq$ in the condition b are special cases and we can convert them to $\{<, \leq, >, \geq\}$ according to the value change of the variable v.

Algorithm 2 starts by collecting all IV-dependent conditions B of path π at line 1. For every IV-dependent condition $b \in B$, we

Algorithm 2: InferRange

```
   input : $\pi$: a path in the flow graph
   input : $V_I$: a set of IV variables
   output: $\gamma \leftarrow [\imath_0, \imath)$: the output range of substring covered by
           the path
1  Let $B$ be set of IV-dependent conditions in $\pi$ ;
2  foreach $b = (v \sim C_b) \in B$ do
3      if $\sim \in \{<, \leq\}$ then
4          Let $G^b$ be the subgraph started from $b$;
5          Let $\Pi = \{\pi_1 \dots \pi_n\}$ be the paths that do not contain
           the branch edge whose condition is $b$, and every
           change value $\delta_v^{\pi_i}$ is negative;
6          if $\delta_v^{G^b} > 0 \wedge \Pi \neq \emptyset$ then
7              $C_b \leftarrow C_b + (-\delta_v^{\pi_1} \times \kappa_1 + \dots + -\delta_v^{\pi_n} \times \kappa_n)/\delta_v^{G^b}$;
8          if $\delta_\imath^{G^b} \neq \delta_v^{G^b}$ then
9              $C_b \leftarrow C_b + (\delta_\imath^{G^b} - \delta_v^{G^b}) \times \rho_{G^b}$;
10         $C_b \leftarrow C_b + \imath_0 - v_0$;
11         $\gamma \leftarrow \gamma \bigcap [\imath_0, C_b)$ ;
12     else if $\sim \in \{>, \geq\}$ then
13         $\dots\ \gamma \leftarrow \gamma \bigcap [C_b, \imath)$ ;
14 return $\gamma$;
```

compute the substring range covered by b from lines 2 to 2. Here, we assume b is in its reduced form $(v \sim C_b)$, where $v \in V_I$ is an induction variable and C_b is an expression. $\delta_\imath^{G^b}$ is the change value of the index $\imath$ in the graph G^b. *Note that we assume that the change of v ($\imath$) is same for all paths in the graph.* If the value changes are different in the paths of the graph, the inference may depend on the execution order of the paths, which is very challenging to summarize as described in example 4(c) later. $\delta_v^{G^b}$ and $\delta_v^{\pi_i}$ denote the change of v in the subgraph G^b and the path π_i respectively. To get the maximum range of the traversed substring, we will calculate the upper bound C_b for the condition b if $\sim \in \{<, \leq\}$ (lines from 3 to 3), or the lower bound if $\sim \in \{>, \geq\}$ (lines from 12 to 12).

Different change values $\delta_v^{G^b}$ and $\delta_v^{\pi_i}$ will cause different execution orders among the subgraph G^b and path π_i. Let us consider $\sim \in \{<, \leq\}$ first. If $\delta_v^{G^b} \leq 0$ or $\delta_v^{\pi_i} \geq 0$, the iterations in the G^b and path π_i will not be interleaved, e.g., $\langle G^b, \pi_i, G^b \rangle$ will never happen. If $\delta_v^{G^b} > 0$ and $\delta_v^{\pi_i} < 0$, it will cause the interleaving execution, such as $\langle G^b, \dots \pi_i \dots G^b \dots \pi_i \rangle$. The decrement of v in path π_i will lead to the execution of subgraph G^b. We consider this kind of interleaving execution from lines 6 to 6.

At line 5, it gets the paths which do not execute the branch b, and their change values of v are negative. The additional executions are computed if $\delta_v^{G^b} > 0$ and the paths set Π is not empty (line 7). For each path $\pi_i \in \Pi$ with path counter κ_i, the total decreased value is $\delta_v^{\pi_i} \times \kappa_i$. We can compute the total decrement in the iterations of all paths (the change value in the paths are negative), and the additional number of the executions is $((-\delta_v^{\pi_1} \times \kappa_1) + \dots + (-\delta_v^{\pi_n} \times \kappa_n))/\delta_v^{G^b}$, where $\delta_v^{G^b}$ is the increment of v in the subgraph G^b.

For example, consider the left path π_1 in Fig. 4(b), $fb < 10$ is the condition b. The change value $\delta_{fb}^{G^b}$ is 1 and the $\delta_{fb}^{\pi_2}$ is -1. It leads to the interleaving execution between π_1 and π_2. With our algorithm, we compute the additional number of executions is $(-(-1) \times \kappa_2)/1$ (its value is κ_2). To illustrate the interleaving clearly, we analyze its iterations in the two paths. Table 2 shows the path iterations of the

Table 2: The iteration of the loop

$\imath$ value	0-9	10	11	12	13	14	15	...
Executed path	π_1	π_2	π_1	π_2	π_1	π_2	π_1	...

loop in Fig. 4(b). The first row is the traversed string (its index) in the loop and the second row is the path executed at different indices. We see that it first executes path π_1 in the substring whose range is [0,10), then it interleaves between the left path and the right path in the range [10,i). It will execute one more time in the left path when it executes the right path every time (κ_2 times totally).

Then we infer the range of the substring by analyzing the relationship between the induction variable v and string index $\imath$ in the branch b. If $\delta_\imath^{G^b}$ is not equal to $\delta_v^{G^b}$ (line 8), the increment of v and $\imath$ are different in each execution of b. In each iteration, the $\imath$ increase $(\delta_\imath^{G^b} - \delta_v^{G^b})$ times more. Line 9 computes the total different value for $\imath$ as $(\delta_\imath^{G^b} - \delta_v^{G^b}) \times \rho_{G^b}$, where ρ_{G^b} is the iterations of branch b. In Fig. 4(b), both of $\delta_\imath^{G^b}$ and $\delta_v^{G^b}$ are 1, so the value of $(\delta_\imath^{G^b} - \delta_v^{G^b})$ is 0.

At last, considering the different initial value for the index $\imath$ and induction variable v, we add the value $(\imath_0 - v_0)$ at line 10. Now the upper bound C_b is calculated, and the range of the traversed string in the branch b is $[\imath_0, C_b)$. The range of the string in the path π is the intersection of the ranges at line 11. For example, the intersection of the two ranges [0,3) and [0, 5) is [0,3). If the intersection cannot be computed, we have to ignore the new computed range, which weakens the constraints and may cause imprecision in the result.

Similarly, if $\sim \in \{>, \geq\}$ in the condition b, we can compute the lower bound C_b and get the range $[C_b, \imath)$ (line 13). We omit the details for the interests of space.

From the condition $fb < 10$ in Fig. 4(b), we infer the range [0,10+κ_2) in the end. It means the traverse content of input in the path π_1 is in the range [0,10+κ_2) of the substring. We can see the substring is not consecutive chars of input, but it does not affect the precision of the summarization on fb. Our approach summarizes the loop with length abstraction which cares about the number of the iterations in each path more than the real order of the input.

Limitation In Fig. 4(c), there are two paths in the subgraph G^b from branch $fb < 10$. The change of fb in the two paths are different, our algorithm cannot infer the ranges precisely in this case. Due to the different change values (1 and 2), it cannot decide the number of interleaving execution between the subgraph $G^{fb<10}$ and the right path which contain $fb >= 10$. The reason is that it needs to remember the interleaving order in the two sub-paths of the subgraph $G^{fb<10}$. In fact, this is a challenge for all loop summarization techniques [14, 25, 22]. In this case, we use the default range $[\imath_0, \imath)$ as the range of the b as an over-approximation.

3.5 String Constraints Generation

After getting the ranges of the substrings for a path, we generate the path constraints by summarizing string conditions in the path. The summarization is based on the pattern extraction on the string conditions as described below.

We present $\textsf{GenStrCon}(\pi, \gamma)$ method in Algorithm 1. Given a path π with path counter κ, let $(STR[g_1] \bowtie_1 c_1), \ldots, (STR[g_k] \bowtie_k c_k)$ be the string conditions in π, where for $1 \leq i \leq k$, $\bowtie_i \in \{\neq, =\}$, $g_i = f_i(\imath)$ and f_i is a linear function about string index $\imath$. Suppose the range of the substring covered by π is $\gamma = [r_1, r_2]$, GenStrCon outputs the constraints of π by the following rules:

RULE 1. *If a subset of $\{g_1, \ldots, g_k\}$ can form a continuous sequence $\langle g_{j_1}, g_{j_2}, \ldots, g_{j_m}\rangle$ where $g_{j_{i+1}} = g_{j_i} + 1$ for all $1 < i < m$, and their relation operators are all =, then we can consider them together and return the following constraints:*

$$\begin{aligned} &str_1 = STR[r_1, r_2] \\ &str_2 = str_1(\text{“}c_{j_1}c_{j_2}\ldots c_{j_m}\text{”}/\text{“”}) \\ &|str_1| - |str_2| \geq \kappa \times |\text{“}c_i \ldots c_j\text{”}| \end{aligned}$$

During the κ iterations of the path, there are at least κ substring “$c_{j_1}c_{j_2}\ldots c_{j_m}$” in the range $[r_1, r_2]$. We remove all occurrences of “$c_{j_1}c_{j_2}\ldots c_{j_m}$” in str_1, then the length of all “$c_{j_1}c_{j_2}\ldots c_{j_m}$” in the string is the whole length minus the length of non “$c_{j_1}c_{j_2}\ldots c_{j_m}$” string $str2$. This length should be greater than or equal to $\kappa \times |\text{“}c_{j_1}c_{j_2}\ldots c_{j_m}\text{”}|$.

RULE 2. *If a subset of $\{g_1, \ldots, g_k\}$, say $\{g_{j_1}, g_{j_2}, \ldots, g_{j_m}\}$, are the same and their relation operators are $\neq$, we also consider them together and return the following constraints:*

$$\begin{aligned} &str_0 = STR[r_1, r_2] \\ &str_1 = str_0(\text{“}c_{j_1}\text{”}/\text{“”}) \\ &\ldots \\ &str_m = str_{m-1}(\text{“}c_{j_m}\text{”}/\text{“”}) \\ &|str_k| \geq \kappa \end{aligned}$$

We extract the pattern from the conditions: there are at least κ characters which are not equal to all the characters in {‘c_{j_1}’, ’c_{j_2}’, ..., ‘c_{j_m}’}. Similarly, we substitute the characters {‘c_{j_1}’, ’c_{j_2}’, ..., ‘c_{j_m}’} as empty character and the remaining string str_k are non-{‘c_{j_1}’, ‘c_{j_2}’, ..., ‘c_{j_m}’} characters. Its length is greater than or equal to κ.

RULE 3. *For other cases, we can generate the constraints by rule 1 and 2 for every string condition in the path independently. For every condition $STR[g_i] \bowtie c_i$, we adopt rule 1 if $\bowtie_i$ is =, or rule 2 if $\bowtie_i$ is $\neq$.*

Due to the limited operations of the string solvers, we can only support patterns above for string conditions. As shown in our experiment section, these patterns are sufficient for most of the string loops because the programs often care about the string content only. We believe our approach can extend easily with the improvement of the string solver in the future.

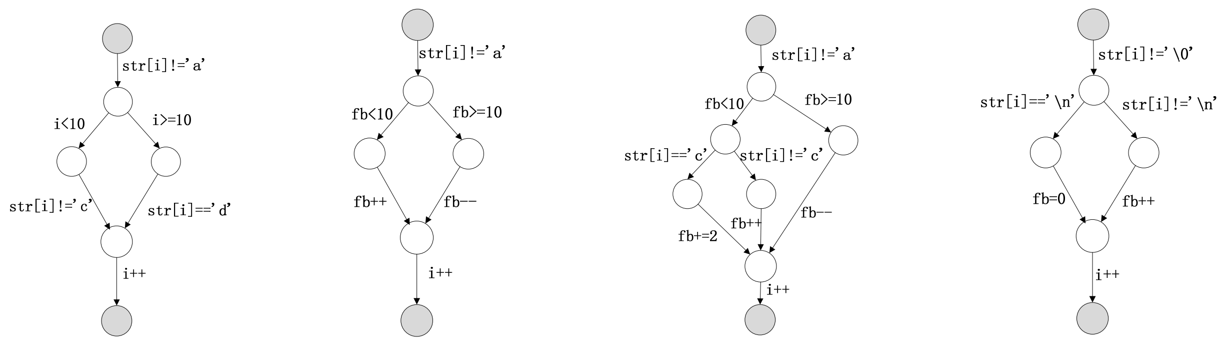
3.6 Summarizing Non-induction Variables

In our empirical studies, we found that not all non-induction variables are relevant to the properties under tracking. If a path in the loop contains such non-induction variables, we can remove them and summarize the loop partially only for relevant variables. Algorithm 3 gives the procedure to handle non IV based on this observation. The input is the flow graph fl, a non IV v and the IV set V_I. The output ϕ is the constraints to make the non IV related paths infeasible.

At line 1, it first finds the paths which make the variable v non inducible. We then assign 0 to these path counters of the non induction paths (lines 2 to 2), which indicates the paths will not execute in the loop such that variable v becomes IV.

At line 4, we use $Conds$ to denote the set of conditions from which we can make the paths $\{\pi_1, \pi_2, \ldots, \pi_k\}$ infeasible and other paths in fl feasible. For example, in Fig. 4(d), $str[i]! = $ ‘ $\backslash$ n’ is the condition in $Conds$, the left path is infeasible and the right path is feasible when it is false.

Lines 5 to 5 generate the output constraints. If b is an IV-dependent condition, b can be added to the output constraints directly (line 7). If b is a string condition, and the operator is =, let the context of each index in the string STR be char C, we use the constraint



(a) Example 1 (b) Example 2 (c) Example 3 (d) Example 4

Figure 4: Flow Graphs Examples

```
Algorithm 3: ReduceNonIV
  input : $fl$: a flow graph
  input : $v$: a non induction variable
  input : $V_I$: a set of IV variables
  output: $\phi \leftarrow true$: the output constraints
1  Let $\{\pi_1, \pi_2, \dots, \pi_k\}$ be the paths which make the $v$ non
   induction variable;
2  foreach $\pi_i \in \{\pi_1, \pi_2, \dots, \pi_k\}$ do
3      $\phi \leftarrow \phi \wedge (\kappa_i = 0)$
4  Let $Conds$ be the conditions that rule out the execution of
   $\{\pi_1, \pi_2, \dots, \pi_k\}$ in $fl$ ;
5  foreach $b_i \in Conds$ do
6      if $b_i$ is IV-dependent condition then
7          $\phi \leftarrow \phi \wedge b_i$;
8      else if $b$ is a string condition then
9          if $b$ is the form $STR[i] = C$ then
10             $\phi \leftarrow \phi \wedge STR(C/$“”$)$ ;
11         else if $b$ is the form $STR[i] \neq C$ then
12             $\phi \leftarrow \phi \wedge \neg$contains$(STR, C)$ ;
13 $V_I \leftarrow V_I \bigcup \{v\}$ ;
14 return $\phi$;
```

$STR(C,$“”$)=$“” to describe the constraint (line 10). If the operator is $\neq$ (i.e., there should be no char C in the string STR), we use the constraint $\neg contains(STR, C)$ to describe the constraint (line 12). After the for loop, the variable v becomes IV and it is added into the IV set (line 13).

Consider the example in Fig. 4(d). The variable fb is a non induction variable. It finds the left path (suppose the path counter is κ_1) that makes the variable non inducible. The feasible condition of the path is $str[i]$ == ‘ $\backslash$ n’ (its negation can make the left path infeasible and the right path feasible). We can make it inducible by deleting the path (assign zero to its path counter). To make the left path infeasible, it must satisfy that there is no character ‘ $\backslash$ n’ in str. We generate the constraint ($\neg contains(str,$ “ $\backslash$ n”)) for the special case. Now fb is an induction variable and $fb = fb_0 + \kappa_2$, it can summarize the variable using the Algorithm 1 partially.

3.7 Discussions

In this paper, we mainly target loops that traverses a string. The precision of *S-Looper* is dependent on the following assumptions:

- The string STR is read-only in the loop. Since we generate constraints by extracting patterns from the string conditions, it will generate imprecise results if STR's content or length is changed during the loop.
- The string index $\imath$ is changing monotonically in all paths. If the string index both increases and decreases in the paths, the loop may traverse the same content of STR multiple times, which causes imprecise results for the length abstraction. In this paper, we assume the string index $\imath$ increases from $\imath_0$. It is similar when the index decreases from $\imath$.
- The branch conditions in the loop are only related to string STR and/or induction-variables. We currently do not handle branch conditions beyond these two types. For example, if there is a library call in the condition, the constraints generated for the loop can be imprecise.
- The string conditions do not contain $\{<, \leq, >, \geq\}$ operators. This assumption is due to the limitation of the existing string solvers. If a string condition contains $\{<, \leq, >, \geq\}$ operators, we convert them to $\{=, \neq\}$ by choosing one special char. For example, $str[i] >$ ‘a’ is converted to $str[i] =$ ‘b’. It may cause imprecise summarization in some situations.
- We perform symbolic analysis to infer the range of the string index for each path. The imprecision can occur if there are complex pointers or library calls that we do not handle. In this case, we use the default range (the maximum one).
- The order of the interleaving of the paths in the loop does not impact the loop summary. For the cases where one path in the loop uses the results computed from another path in the loop, this assumption does not hold, and thus the analysis can generate imprecise results.

In our approach, we have not tackled the nested loops. Since the constraint solver we use only can handle integer and string constraints, we are not able to summarize the loops that contain floating point types or complex data structures.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

This section presents the implementation details and evaluation results of *S-Looper*. The goals of our experiments are to show: 1) the capabilities of our approach on summarizing multipath string loops, and 2) the usefulness of the techniques in improving precision and scalability of symbolic execution. Specifically, we applied symbolic execution on buffer overflow detection and test input generation, and compared our techniques to the existing tools such as Marple [20], KLEE [4] and PEX [26].

Table 3: Loop Summarization for Buffer Overflow Detection

Benchmark	Versions	Total	Summarized		Total Vul	S-Looper			Loop Unrolling	
	Faulty+Patched	Loops	IV-Only	Non-IV		Detected	Miss_1	Miss_2	Unroll-3	Unroll-8
Edbrowse	2+2	4	2	2	2	1	0	1	0	0
Bind	1+1	4	0	0	1	0	1	0	0	0
Wu-Ftpd	2+2	4	2	0	2	1	1	0	0	0
SpamAssassin	1+1	2	2	0	1	0	1	0	0	1
Libgd	2+2	4	0	4	2	2(2)	0	0	0	1
MADWiFi	2+2	4	4	0	2	2	0	0	0	2
NETBSD	6+6	16	8	0	6	3	3	0	0	2
OpenSER	4+4	20	16	0	4	2	2	0	0	2
Apache	4+4	20	20	0	4	4	0	0	0	2
Sendmail	13+13	33	14	10	13	6(3)	5	2	0	3
Total	**37+37**	**111**	**68**	**16**	**37**	**21**	**13**	**3**	**0**	**13**

4.1 Implementation and Experimental Setup

Our loop summarization techniques are integrated into the new implementation[2] of the *Marple* framework [20], using LLVM version 3.4 [18] and the string constraint solver S3 [27]. The targeted languages are C and C++ programs, which are converted into LLVM-IR [18] by LLVM frontend as our input.

We conducted two experiments to evaluate $S\text{-}Looper$. In the first experiment, we implemented the demand-driven, path-sensitive buffer overflow detection algorithm [20] and integrated $S\text{-}Looper$ in this approach. The analysis first scans the program to find the buffer access statement if any. It then raises queries at these statements to check whether there are paths that can reach the buffer access and lead to buffer overflow. When traversing the loop, we use the loop summarization techniques described in Section 3 to generate constraints. We compare this technique with a version of Marple where we unroll loops for a fixed amount of times.

In the second experiment, we implemented a weakest precondition propagation routine in Marple to automatically generate test input that can hit an assertion after a loop. We compare the performance of our work with KLEE and PEX.

For the two experiments, we used the buffer overflow benchmark developed by Ku et al. [17], which contains 298 programs from 12 projects. We removed programs that do not contain loops; some of the programs in the benchmark are very similar to each other, and we only kept one copy. As a result, we collected a total of 74 programs for our experiments, among which 37 programs contain vulnerabilities reported by Common Vulnerabilities and Exposures (CVE) [1] and another 37 programs are the corresponding patched versions. All of our experiments were conducted on a 3.2GHZ Intel Core 4 duo with 4GB of RAM.

4.2 Experimental Results

By analyzing 450 loops in 268 code fragments of benchmark [17], 277 loops of them (61.5%) totally satisfy the assumptions and can be handled by our approach. In the following, we first report the results from buffer overflow detection and then the ones from test input generation.

4.2.1 Evaluation on Buffer Overflow Detection

In this experiment, we ran buffer overflow detection on both faulty and patched versions to evaluate the false negatives and false positives of our techniques. In Table 3, under *Benchmark*, we list 10 projects in the buffer overflow benchmarks that contain loops. Under *Versions*, we list the number of faulty and patched versions we used for each project. Under *Total Loops*, we show the total number of loops for all the versions considered. Under *Summarized*, we report the number of loops we summarized, among which under *IV-Only*, we show the number of loops whose branch conditions do not contain non-induction variables, and under *Non-IV*, we show the number of loops we summarized with approximation due to the presence of non-induction variables.

The data in Table 3 show that among the total 111 loops in the 74 programs, we have summarized a total of 84 (75.7%) loops; 68 of them (81%) are summarized precisely and 16 of them (19%) are summarized partially. All of them are multipath string loops which cannot be summarized precisely by other techniques [25, 14, 22]. We manually analyzed the loops we cannot handle, and we found the following categories:

- Loops that are related to complex types (e.g., the list traversal in big_bad.c in the *bind* project)
- Loops whose branch condition depends on a function call (e.g., prefix_bad.c in wu-ftpd)
- Nested loops

Under *Total Vul*, we list the total number of vulnerabilities the benchmarks report. Under *S-Looper*, we show that Marple with the loop summarization techniques reports a total of 21 (56.8%) known bugs with 0 false positives. Encouragingly, although some loops contain non-IVs and are summarized partially, we are still able to detect vulnerabilities precisely, as the approximation does not affect the vulnerability conditions. The two buffer overflows in *Libgd* and three in *Sendmail*, shown in parenthesis under *S-Looper*, belong to such cases. The data also suggest that there is still s-pace for further improving the loop summarization techniques, as we still missed 16 vulnerabilities. We inspected these vulnerabilities and found that 13 of them are related to the loops we cannot summarize (see column Miss_1 and 3 are related to the imprecision introduced when we handle non-induction variables in the loop, shown in column Miss_2.

Fig. 5 shows a bug we missed from mime_fromqp_arr_bad.c. In the `while` loop, we identified that along the `if` branch from lines 4 to 7, variable $nchar$ is set to 0, and thus $nchar$ is a non-IV. To handle such non-induction variable, we introduce an assumption that the input str does not contain “\n" before entering the loop. With this assumption, the `if` branch (lines 4 to 7) can n-ever be taken during loop iteration, and $nchar$ therefore becomes an IV. We then can summarize the simplified loop. To denote this assumption, we add constraint $\neg contains(str, \text{“\textbackslash n"})$ to the loop summary. Unfortunately, the buffer overflow vulnerability cannot be detected with the approximation, since it depends on precisely tracking variable $nchar$ in each path. We thus missed the bug.

As a comparison, we also ran Marple with the configurations of unrolling loops a fixed number of times. In the experiment, we se-

[2]The original Marple implementation is based on Microsoft Phoenix for analysing .NET programs.

```
char  outfile[BASE_SZ];
int  out=0,nchar=0,i=0;
while(str[i]!='\0'){
   if(str[i] ==  '\n'){
      nchar=0;
      continue;
   }else{
      nchar++;
      if(nchar>BASE_SZ)
         break;
      outfile[out]=str[i];
      out++;
   }
   i++;
}
```

Figure 5: Simplified Code from mime_fromqp_arr_bad.c

lect a small bound 3 and another bigger 8 randomly. The results are given under *Unroll-3* and *Unroll-8* respectively. Our results show that by unrolling loops 3 times, we cannot detect any of the vulnerabilities in the benchmark, and by unrolling loops 8 times, we are able to detect 13 out of 37 vulnerabilities, still not comparable to our loop summarization technique. There are 2 vulnerabilities that are detected by unrolling the loops 8 times but not by our technique. The vulnerabilities are related to non-induction variables in the loop, where we lose precision. The results suggest that without loop summarization techniques, it is very hard to decide how many times we should unroll the loop. Improper unrolling can increase the performance of the analysis but still miss the vulnerabilities.

For the performance evaluation, we found that S-*Looper* is able to summarize the loops and analyze the vulnerabilities within one second, which is dominated by the constraining solving time by the string solver. For Marple using unrolling loops, the running time grows quickly as the number of unrolling loops increases, which is ranging from several seconds to tens of minutes. Therefore, S-*Looper* is more scalable compared with the default loops unrolling approach in the symbolic analysis. The further development in string constraint solver could make our approach practical for real programs with complicated string conditions. We omit the actual execution time for the interest of space and the performance evaluation will be discussed in the second experiment later.

To conclude, the experimental data shown in Table 3 indicate that combined with symbolic analysis, our loop summarization technique is effective for bug finding, and it performs better than traditional loop unrolling techniques.

4.2.2 Evaluation on Improving Test Generation

In the second experiment, we compared the performance and capabilities of *PEX*, *KLEE* and *Marple* with S-*Looper* on generating test inputs for paths that contain loops. Similar to the experimental approaches used in Strejček et al's paper [25], we inserted a conditional `assert` after a loop, shown as follows:

```
  //loop related to buffersize
  if(i > buffersize)
     assert(false);
```

During the testing, our goal is to generate test inputs that can trigger the assert statement. In the above code list, $buffersize$ stores the size of the string traversed in the loop. By changing the value of $buffersize$ in different runs, we simulate the scenarios on generating test inputs for different sizes of the loop.

In the experiment, we observed that the value of $buffersize$ decides the test input generation time for *KLEE* and *PEX*. When $buffersize$ is small, the symbolic execution just needs to traverse the loop a few times, and test generation is thus fast. But when $buffersize$ becomes bigger, the symbolic execution needs to iterate through the loop more times and explore much more paths; as a result, test input generation becomes expensive. In the experiment, we compare the test case generating time using different $buffersize$. As *PEX* works on .NET, we implemented the corresponding C# programs for all the C benchmark.

Table 4 shows the result for running test generation in *KLEE*, *PEX* and *Marple* with S-*Looper*. Under *Benchmark*, we show 11 programs used in this study. Column *Paths* shows the number of paths in the loop for each program. Under *KLEE*, *PEX* and *Marple with* S-*Looper*, we list the time (in seconds) used to generate test inputs under different $buffersize$. Specifically, the time reported under *Marple with* S-*Looper* includes the time used for both loop summarization and constraint solving phases. In the experiments, we choose different sizes for $buffersize$ as 10, 20, 300, 1000. In the table, we use *T/O* to represent time out, when the tool spent more than 30 minutes but are still not able to generate test input. We use *F* to represent that the tool failed to generate any test input before it terminates.

Our results show that when $buffersize$ increases, *KLEE* and *PEX* significantly increase the time needed for test input generation, except for $interproc\_bad.c$. As the loop in this program is only related to the size of the string, but not the content of the string. Even when $buffersize$ is 20, *KLEE* report time out for three programs. When $buffersize$ increases to 1000, 7 programs time out in *KLEE*, and for *PEX*, there are 3 programs that cannot generate any test input and 1 program time out (we set pex with maxconditions, maxtime and maxruns).

Using S-*Looper* to summarize the loops, *Marple* can generate test inputs for all the programs in less than a second when $buffersize$ is set to less than or equal to 300. We observed that the majority of the time is used for constraint solving, and the loop summarization takes less than 0.05 seconds for all the programs. When $buffersize$ increases to 1000, *Marple* needs tens of seconds (at most 25) to finish the test case generation, and the time is mainly spent in the string solver.

In the experiments, we also see that the time used by *PEX* and *KLEE* depends on the types of statements in the loop. For example, *KLEE* needs more time when the loop contains the statement $continue$ in one branch. In the first program, there is a branch $if(c1 == `=')$ $continue$ located at the beginning of the loop. *KLEE* takes 36 seconds to generate a test input when $buffersize$ is set to 10. *KLEE* reported time out when $buffersize$ is 20. On the other hand, *PEX* is more expensive when handling the equal comparisons of string content. For example, even for the simple loop in program $full\_bad.c$, *PEX* reported time out for the following loop when $buffersize$ is 300. The equal comparison at line 2 cause a big impact on the spent time in PEX.

```
1  while (uri[cp] != EOS && c < TOKEN_SZ){
2     if (uri[cp] == '?')
3        ++c;
4     ++cp;
5  }
```

In summary, the above experimental results show that

- The state-of-the-art symbolic execution tools *KLEE* and *PEX* can take much time or be blocked when it needs to traverse a loop many times. The statements such as *continue* or comparisons of string content can have a significant impact on the performance of test input generation;

Table 4: **Comparison result of test generation with KLEE, PEX and Marple with** S-*Looper*

Benchmark	Paths	KLEE (seconds)				PEX (seconds)				Marple with S-Looper (seconds)			
		10	20	300	1000	10	20	300	1000	10	20	300	1000
mime7to8arronechar_heavy_bad.c	5	36	T/O	T/O	T/O	1.02	1.13	23	43	0.32	0.42	0.82	13.51
mime_fromqp_arr_bad.c)	8	43	T/O	T/O	T/O	1.21	2.33	54	782	0.54	0.51	1.16	23.1
close_angle_ptr_one_test_bad.c	3	0.03	0.03	13	172	0.14	0.31	13	412	0.42	0.73	0.72	13.58
outer_bad.c	6	0.02	0.02	1.10	T/O	1.02	1.15	48	63	0.37	0.45	0.73	13.56
iter2_prefixShort_arr_bad.c	5	0.3	25	T/O	T/O	1.16	1.19	39	F	0.38	0.46	1.21	20.3
full_bad.c	4	21	29	T/O	T/O	1.06	2.21	T/O	T/O	0.41	0.52	1.41	25.1
gd_simple_bad.c	3	20	T/O	T/O	T/O	2.01	3.34	26	182	0.35	0.36	0.82	13.39
interproc_bad.c	2	0.11	0.13	0.2	0.6	0.31	0.35	1.4	7.21	0.03	0.03	0.03	0.04
strchr_bad.c	3	0.02	0.02	28	563	0.30	1.16	F	F	0.23	0.31	0.37	0.72
noAnyMeta_int_bad.c	4	0.1	0.3	52	1376	0.3	4.1	392	F	0.33	0.53	0.77	3.2
loops_bad.c	8	0.4	1.06	170	T/O	1.07	257	653	F	0.13	0.37	0.7	13.32

- With the loop summarization technique we developed, *Marple* can generate test input quickly even for the loops *PEX* and *K-LEE* cannot handle. Our approach can be used by the existing tools to improve the coverage of testing.

5. RELATED WORK

This section summarizes the related literature on loop summarization. Xiao et al. provided characteristic studies on loop problems and existing techniques to address the problems for dynamic symbolic execution (DSE) [28]. The main techniques to address the loop problems include bounded iteration [21, 6, 12], search-guiding heuristics [5, 29, 26] and loop summarization [22, 14, 25]. In bounded model checking [21, 6], researchers often provided a bounded iteration count to unroll the loop. Search-guiding heuristics attempts to guide symbolic execution to explore paths that are likely to achieve higher coverage, and prevent dynamic symbolic execution from being stuck in loops. Bounded iteration and search-guiding heuristics are the mostly used techniques in the existing tools, since it is easy to implement and it is effective in some cases. But they may lose precision when determining the program properties requires iterating through the loops beyond a fixed bound.

Loop summarization aims to summarize the variables and invariants to a set of constraints. As loop summarization can describe the relationship between variables at the entry and the exit of a loop, it is more useful and precise than bounded iteration and search-guiding heuristics. However, currently the main loop summarization techniques [22, 25, 14] aim at integer variables or integer arrays, it is difficult to handle the string related loops. Our approach focuses on string loops and summarizes the relation between string variable and integer variables with the string constraint solver [27].

Several approaches are proposed for summarizing loops recently. The LESE [22] introduces the symbolic variable $trip\ count$ for the number of times each loop executes. They then infer the relationship between inputs and variables with $trip\ count$. LESE determines the relationships between variables and $trip\ count$ and links $trip\ count$ with an input grammar. The input grammar can capture some repetitive features for the string content. Our approach handles the string content comparison conditions in a more general way, and we can extract more patterns for the input. The approach [14] discovers the loops and detect induction variables on the fly. It then infers the simple partial loop invariants about the IVs and generates pre and post conditions as the summarization of the loop. Compared with the work in [14, 22], our approach provides more precise summarization for multi-paths by introducing $path\ counter$ to represent the path iteration count. The closest work related to ours is APC [25]. The paper also introduces $path\ counter$ for every path and represents overall effect of all changes by $path\ counter$. It summarizes the loop by computing the necessary condition on the loop conditions. Inspired by [25], our work is also based on the flow graph. The biggest difference between our work and [25] is that our approach can handle the string content comparison and aims to represent the relationship between the variables and input while APC and other approaches have difficulties to generate such relationships.

The loop invariant generation are studied in the program verification [10, 16, 24]. The techniques in [10, 16] synthesize invariants by learning methods. The paper [24] uses the randomized search to discover candidate invariants and validate them by constraint solvers. These techniques are mainly used in program verification while symbolic execution requires to know the relationships among variables. In [19], the author proposed the segmented symbolic analysis which can infer the transfer functions for loops. It performs a unit test for code segments and infer the liner relationships via regression analysis. It generates the transform function by dynamic analysis which may cause imprecision. Although all of the above techniques can handle loops at different degrees, we are not aware of any other work focused on how to handle multipath string loops.

6. CONCLUSIONS

In this work, we presented S-*Looper* that summarizes a type of multipath, string loop. S-*Looper* introduces path counters to represent symbolic values for each path in the loop. It handles branch conditions related to string content by extracting patterns of the string along each path and representing them using string constraints. We implemented S-*Looper* and integrated it to the *Marple* framework. Our experimental results show that it can summarize 75.7% of the loops previously reported as unknown. Compared to existing tools, the technique largely improves the capabilities of buffer overflow detection and test input generation.

One direction of future work is to extend our approach to support more complicated loop structures (e.g., nested loops) and string conditions. Another important topic is to enhance the string constraint solvers to provide better support for the string loop analysis.

7. ACKNOWLEDGMENTS

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30), by the National Science Foundation of China (No. 91118003, 61272106, 61340039) and 985 funds of Tianjin University and Tianjin Key Laboratory of Cognitive Computing and Application, and by the US National Science Foundation (NSF) under Award 1542117.

8. REFERENCES

[1] Cve-common vulnerabilities and exposures. `http://cve.mitre.org/`.

[2] C. Barrett and C. Tinelli. Cvc3. In *CAV*, volume 4590, pages 298–302, 2007.

[3] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. *IEEE Symposium on Security and Privacy*, pages 2 – 16, 2006.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322 – 335, 2006.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *TACAS*, volume 2988, pages 168–176, 2004.

[7] L. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[8] L. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, volume 4963, pages 337–340, 2008.

[9] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, volume 4590, pages 519–531, 2007.

[10] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A robust framework for learning invariants. In *CAV*, volume 8559, pages 69–87, November 2014.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of ACM SIGPLAN*, volume 40(6), pages 213–223, June 2005.

[12] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *NDSS*, volume 2988, 2008.

[13] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. In *Queue*, volume 10(1), page 20, January 2012.

[14] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *ISSTA*, volume 8559, pages 23–33, 2011.

[15] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[16] S. Kong, Y. Jung, C. David, B.-Y. Wang, and K. Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS*, volume 6461, pages 328–343, November 2010.

[17] K. Ku, T. E. Hart, M. Chechik, and D. Lie. A buffer overflow benchmark for software model checkers. In *ASE*, pages 389–392, 2007.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

[19] W. Le. Segmented symbolic analysis. In *ICSE*, pages 212–221, 2013.

[20] W. Le and M. L. Soffa. Marple: a demand-driven path-sensitive buffer overflow detector. In *FSE*, pages 272–282, November 2008.

[21] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In *VSTTE*, volume 7152, pages 146–161, 2012.

[22] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *ISSTA*, pages 225–236, 2009.

[23] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, September 2005.

[24] R. Sharma and A. Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, volume 8559, pages 88 – 105, November 2014.

[25] J. Strejček and M. Trtík. Abstracting path conditions. In *ISSTA*, pages 155–165, 2012.

[26] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *NDSS*, volume 4966, pages 134–153, 2008.

[27] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *CCS*, volume 8559, pages 1232–1243, 2014.

[28] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *ASE*, pages 246 – 256, November 2013.

[29] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *DSN*, pages 359–368, 2009.

[30] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *FSE*, pages 114–124, 2013.