S-Looper: Automatic Summarization for Multipath String Loops

Xiaofei Xie1†, Yang Liu2†, Wei Le1, Xiaohong Li‡, Hongxu Chen2

†Tianjin Key Laboratory of Cognitive Computing and Application, Tianjin University, China
‡Nanyang Technological University, Singapore

{xiexiaofei, xiaohongli}@tju.edu.cn
{yangliu, hxchen}@ntu.edu.sg
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-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 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.

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 ACM 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.

ISBN: 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multipath loops</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>String content</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In Table 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 multipaths 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.

![Figure 2: The flow graph for the loop program in Figure 1](image)

1. We propose an automatic approach to summarize the multi-path string loops based on the reasoning about variable value changes along different paths and pattern matching.
2. We provide analytical understandings on which loops we can and cannot handle.
3. 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: \[fb > \text{MAXLINE}\] to inquire whether the vulnerability condition—the value of \(fb\) is larger than \(\text{MAXLINE}\)—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 execute one of the following three paths \((abcedf)\), \((abcefg)\) or \((abefg)\) 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: \(k_1\) for path \((abcedf)\), \(k_2\) for \((abcefg)\), and \(k_3\) for \((abefg)\). 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 \(k_1, k_2,\) and \(k_3\) respectively thus can be represented as:

\[
\begin{align*}
k_1 &\geq 0, k_2 \geq 0, k_3 \geq 0 \\
i & = i_0 + 1 \times k_1 + 1 \times k_2 + 1 \times k_3 \\
f & = f_0 + 1 \times k_1 + 1 \times k_2 + 1 \times k_3 \\
fb & = fb_0 + 1 \times k_1 + 0 \times k_2 + 0 \times k_3
\end{align*}
\]

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 \((abcedf)\). Thus, \(fb\) is only related to \(k_1\).

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 \((abcedf)\) and \((abcefg)\), we determine \(f \equiv f\) holds for every iteration of the loop and \(fb_0 \equiv i_0\). Thus, \(i \leq \text{MAXLINE}\) holds for \((abcedf)\), and \(i > \text{MAXLINE}\) hold for \((abcefg)\). Along path \((abefg)\), there is no additional constraint for \(i\), and thus \(i\) can be any value between \(i_0\) and the length of the string \(\text{input}\).

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 \((abcedf)\), we have conditions \(\text{input}[i]! = \backslash{}0'\) and \(\text{input}[i]! = \backslash{}n'\) and \(i \leq \text{MAXLINE}\). These conditions imply that there are at least \(k_1\) characters that are neither \(\backslash{}0'\) nor \(\backslash{}n'\) in \(\text{input}\) between its \(i_0\)th and \(\text{MAXLINE}\)th characters. We express such strings using the following constraints:

\[
\begin{align*}
\text{str} & = \text{input}[i_0, \text{MAXLINE}] \\
\text{str}_1 & = \text{str}(\backslash{}0'/i) \\
\text{str}_2 & = \text{str}(\backslash{}n'/i) \\
|\text{str}_2| & \geq k_1
\end{align*}
\]

In this paper, we write \(\text{STR}[a, b]\) to denote a substring of string \(\text{STR}\) from index \(a\) to index \(b\). \(\text{STR}|\) returns the length of string \(\text{STR}\). \(\text{STR}(s_1/s_2)\) denotes a new string by replacing all occurrences of string \(s_1\) by string \(s_2\) in \(\text{STR}\). \(\text{contains(STR, s)}\) denotes the string \(\text{STR}\) contains the substring \(s\). Applied to the above formulas, the first constraint selects a sequence of characters from \(\text{input}\) in the range of \(i_0\) and \(\text{MAXLINE}\). The second and third constraints suggest that after removing \(\backslash{}0'\) and \(\backslash{}n'\) from the substring generated above, the length of \(\text{str}_2\) is still greater or equal to \(k_1\).

Similarly, for path conditions, \(\text{input}[i]! = \backslash{}0'\), \(\text{input}[i]! = \backslash{}n'\) and \(i > \text{MAXLINE}\) along \((abcefg)\), we construct the following constraints, which say that the substring of \(\text{input}\) in the range between \(\text{MAXLINE} + 1\) and \(i - 1\) at least has \(k_2\) characters that do not contain \(\backslash{}0'\) and \(\backslash{}n'\).

\[
\begin{align*}
\text{str}_3 & = \text{input}[\text{MAXLINE} + 1, i - 1] \\
\text{str}_4 & = \text{str}(\backslash{}0'/i) \\
\text{str}_5 & = \text{str}(\backslash{}n'/i) \\
|\text{str}_5| & \geq k_2
\end{align*}
\]

![Figure 3: Overview of the S-Looper](image)

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

\[
\begin{align*}
\text{str}_6 & = \text{input}[i_0, i - 1] \\
\text{str}_7 & = \text{str}(\backslash{}n'/i) \\
|\text{str}_6| - |\text{str}_7| & \geq k_3
\end{align*}
\]

The formula means that for a substring that gets the characters of \(\text{input}\) from the range of \(i_0\) to \(i - 1\), the number of \(\backslash{}n'\), represented as \(|\text{str}_6| - |\text{str}_7|\), should be at least \(k_3\).

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

\[
\neg(\text{contains(\text{str}_6, \backslash{}0')}\]

After processing all the path conditions in the loop, we conjunct constraints (1) to (5) and generate the loop summary. To determine whether exists \(\text{input}\) 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 \(fb > \text{MAXLINE}\). Given the fact that \(f_0 = 0, fb_0 = 0\) and \(i_0 = 0\) obtained at line 4, the constraint solver reports an instance \(\text{input} = "a\ldots\alpha\backslash{}0"\) (there are 51 \('a')\), \(k_1 = 51, k_2 = k_3 = 0\) 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 \(\text{STR}\) with index variable \(i\) changing monotonically (initially having value \(i_0\)). For example, if \(\text{STR}\) is a char array, \(i\) 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 \(STR[i] \geq C\), where \(\geq \{=, \neq\}\) 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 \(G = (V, E, l_v, l_t, e)\), where \(V\) is a set of vertices, \(E : V \times V\) is a set of edges linking the vertices, \(l_v, l_t \in V\) are entry and end nodes in the loop, \(e\) is a function assigning every edge \(e \in E\) an instruction \(i(e)\).

Given a flow graph, we define a path \(\pi\) as a finite sequence \((v_1, v_2, ..., v_k)\) of nodes such that \((v_i, v_{i+1}) \in E\), where \(k \geq 1, v_1 = l_v\) and \(v_k = l_t\). In control flow diagram, one path in the loop is a circle. Flow graph removes the edge \((v_i, l_t)\) which goes back to the entry node \(l_v\) 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 \((abcdefg)\), \((abcefg)\) and \((abdfg)\).

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. \(f \leq MAXLINE\) and \(f > MAXLINE\) are two IV-dependent conditions because \(dum\) \(=\) \(f - MAXLINE\) 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 \(\phi\) for the loop. At line 1, it constructs the flow graph of the loop \(fl\), identifies induction variables and non-induction variables in \(V_l\) 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 \(V_l\), 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 \(\phi_{niv}\). The details will be described in Section 3.6.

For the flow graph \(fl\), we introduce a path counter \(\kappa\) for every path \(\pi\), to record the number of executions of \(\pi\) 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 \(\phi_p\). 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 \(v \in V_{l}\), let \(c_i\) be the amount that \(v\) changed in path \(\pi_i\), 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 \(\phi_p\) at line 6. Similarly, the constraint for the index variable \(i\) is also computed and saved in \(\phi_i\) since \(i\) 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.

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 \(f \leq MAXLINE\) and \(f > MAXLINE\). 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 \(input[i]\) \(=\) \("0"\) \&\& \(input[i]\) \(=\) \("\"\) of path \((abcdefg)\); there are at least \(\kappa_1\) chars which are neither "0" nor "\"" 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 \(STR[i] = C\), the generated constraint is \(-contain(STR[i_0, i-1], C)\). If the entry condition is \(STR[i] = C\), the generated constraint is \(STR[i_0, i-1](\"\") = \"\"\).

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 \(\pi\) with path counter \(\kappa\), the traversed string content in the \(\kappa\) iterations of \(\pi\) 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 $i$ 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 $\pi_1$ and $\pi_2$ respectively). From the condition $i < 10$, we know the range of the traversed string in the path $\pi_1$ is $[0,10)$, which decides the number of the iterations in $\pi_1$. 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 $G = (V,E,I,l_0,l_1,i)$ and a branching node $\beta \in V$ with an outgoing edge $e$, let $c = \{v\}$ be the branch condition, we define $G_{\beta} = (V_{\beta},E_{\beta},I_{\beta},l_{0\beta},i_{\beta})$ as a subgraph starting from $\beta$ and $e$, where $V_{\beta} \subseteq V$ contains all nodes reachable from $\beta$ via $e$ and $E_{\beta} \subseteq E$ contains all the edges formed by $V_{\beta}$.

For a branching node $\beta$ which has two outgoing edges, we have two subgraphs $G_{\beta}^1$ and $G_{\beta}^2$ 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 $G_{\beta}^c$ and $G_{\beta}^{\neg c}$ if $\beta$ is clear in the context.

Given a flow graph $G^v$ and IV $v$ in $G$, we use $\delta_{G^v}^v$ 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 $\delta_{G^v}^v$ to denote the change of $v$ in a path $\pi$. Our target is to infer the range of the traversed string on the condition $c$.

Given a flow graph $G$ with $n$ paths $(\pi_1, \pi_2, \ldots, \pi_n)$, we define $\rho_G = \kappa_1 + \cdots + \kappa_n$ as the sum of all path counters in $G$. For an IV-dependent condition $I \sim R$ where $\sim \in \{<,\leq,>,\geq,=,\neq\}$, we can convert it to a reduced form $(V/I \sim R)$ such that $V/I$ is an induction variable and $R$ 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 $\neg c = fb >= 10$. $G_{\beta}^c$ is the subgraph starting from the branch $c$ and it is the sub-path of $\pi_1$ after the edge $fb < 10$ here. The change value of $fb$ in the subgraph is $\delta_{G_{\beta}^c}^{fb}$ and its value is 1. Particularly, $\delta_{G_{\beta}^c}^{fb}$ is equal to $\delta_{fb}^1$ since graph $G_{\beta}^c$ only contains path $\pi_1$.

3.4.2 Inference Algorithm

Algorithm 2 takes one path $\pi$ and the IV set $V_l$ as the input, and returns the computed range $\gamma$ of the substring (initially is $[l_0,l_1]$) 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 $\delta_{G^b}^v$ 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 $i$ and the induction variable $v$. If we cannot compute the range of the substring, it will use the default range $[l_0,l_1]$. 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 $\pi$ at line 1. For every IV-dependent condition $b \in B$, we 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 $v \in V_l$ is an induction variable and $C_b$ is an expression. $\delta_{C_b}^v$ is the value change of the index $i$ in the graph $G_b$. Note that we assume that the change of $v$ 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. $\delta_{G^c}^v$ and $\delta_{G^{\neg c}}^v$ denote the change of $v$ in the subgraph $G^c$ and the path $\pi$, respectively.

At line 5, it gets the paths which do not execute the branch $b$ if $\sim \in \{<,\leq\}$ (lines from 3 to 3), or the lower bound if $\sim \in \{>,\geq\}$ (lines from 12 to 12).

Different changes $\delta_{c}^v$ and $\delta_{\neg c}^v$ will cause different execution orders among the subgraph $G^c$ and path $\pi$. We let $\sim \in \{<,\leq\}$ first. If $\delta_{G^c}^v \leq 0$ or $\delta_{G^{\neg c}}^v \geq 0$, the iterations in the $G^b$ and path $\pi$ will not be interleaved, e.g., $(G^b, \pi, G^{\neg c})$ will never happen. If $\delta_{G^c}^v > 0$ and $\delta_{G^{\neg c}}^v < 0$, it will cause the interleaving execution, such as $(G^c, \pi, G^{\neg c})$. The decrement of $v$ in path $\pi$ will lead to the execution of subgraph $G^c$. 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 $\delta_{G^c}^v > 0$ and the paths set $\Pi$ is not empty (line 7). For each path $\pi_i \in \Pi$ with path counter $\kappa_i$, the total decreased value is $\delta_{G^c}^v \times \kappa_i$. 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 $((-\delta_{G^{\neg c}}^v \times \kappa_i) + \cdots + (-\delta_{G^{\neg c}}^v \times \kappa_n))/\delta_{G^c}^v$, where $\delta_{G^c}^v$ is the increment of $v$ in the subgraph $G^c$.

For example, consider the left path $\pi_1$ in Fig. 4(b), $fb < 10$ is the condition $b$. The change value $\delta_{G^b}^{fb}$ is 1 and the $\delta_{G^{\neg b}}^{fb}$ is -1. It leads to the interleaving execution between $\pi_1$ and $\pi_2$. With our algorithm, we compute the additional number of executions is $((-1) \times \kappa_2)/1$ (its value is $\kappa_2$). To illustrate the interleaving clearly, we analyze its iterations in the two paths. Table 2 shows the path iterations of the
<table>
<thead>
<tr>
<th>$i$ value</th>
<th>0-9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executed path</td>
<td>$\pi_1$</td>
<td>$\pi_2$</td>
<td>$\pi_1$</td>
<td>$\pi_2$</td>
<td>$\pi_1$</td>
<td>$\pi_2$</td>
<td>$\pi_1$</td>
<td>$\pi_2$</td>
</tr>
</tbody>
</table>

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 $\pi_1$ in the substring whose range is $[0,10)$, then it interleaves between the left path and the right path in the range $[10,11)$. It will execute one more time in the left path when it executes the right path every time ($\kappa \times 2$ times totally).

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

At last, considering the different initial value for the index $i$ and induction variable $v$, we add the value $(v_0 - v_0)$ at line 10. Now the upper bound $C_0$ is calculated, and the range of the traversed string in the branch $b$ is $[v_0, C_0]$. The range of the string in the path $\pi$ 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 $\varepsilon \subseteq \{>, \geq\}$ in the condition $b$, we compute the lower bound $C_0$ and get the range $[C_0, i)$ (line 13). We omit the details for the interests of space.

From the condition $fb < 10$ in Fig. 4(b), we infer the range $[0,10+\kappa \times 2]$ in the end. It means the traverse content of input in the path $\pi_1$ is in the range $[0,10+\kappa \times 2]$ 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 $fb \geq 10$. 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 $[v_0, i)$ 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 GenStrCon($\pi, \gamma$) method in Algorithm 1. Given a path $\pi$ with path counter $\kappa$, let (STR[$g_1$] $\times c_1$), …, (STR[$g_k$] $\times c_k$) be the string conditions in $\pi$, where for $1 \leq i \leq k$, $g_i \in \{\geq, =\}$. $g_i = f_i(s)$ and $f_i$ is a linear function about string index $i$. Suppose the range of the substring covered by $\pi$ is $\gamma = [r_1, r_2]$. GenStrCon outputs the constraints of $\pi$ 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 $\neq$, then we can consider them together and return the following constraints:

$$str_1 = STR[r_1, r_2]$$

$$str_2 = str_1(\langle c_1, c_2, \ldots, c_m \rangle / \omega)$$

$$|str_1| - |str_2| \geq \kappa \times |\langle c_1, c_2 \rangle|$$

During the $\kappa$ iterations of the path, there are at least $\kappa$ substring $\langle c_1, c_2, \ldots, c_m \rangle$ in the range $[r_1, r_2]$. We remove all occurrences of $\langle c_1, c_2, \ldots, c_m \rangle$ in $str_1$, then the length of all $\langle c_1, c_2, \ldots, c_m \rangle$ in the string is the whole length minus the length of non $\langle c_1, c_2, \ldots, c_m \rangle$ string $str_2$. This length should be greater than or equal to $\kappa \times |\langle c_1, c_2, \ldots, c_m \rangle|$.

**Rule 2.** If a subset of $\{g_1, g_2, \ldots, g_m\}$, say $\{g_1, g_2, \ldots, g_m\}$, are the same and their relation operators are $\neq$, we also consider them together and return the following constraints:

$$str_0 = STR[r_1, r_2]$$

$$str_1 = str_0(\langle c_1 \rangle / \omega)$$

$$\ldots$$

$$str_m = str_{m-1}(\langle c_m \rangle / \omega)$$

$$|str_k| \geq \kappa$$

We extract the pattern from the conditions: there are at least $\kappa$ characters which are not equal to all the characters in $\langle c_1, c_2, \ldots, c_m \rangle$. Similarly, we substitute the characters $\langle c_1, c_2, \ldots, c_j \rangle$ as empty character and the remaining string $str_k$ are non-$\langle c_1, c_2, \ldots, c_j \rangle$ characters. Its length is greater than or equal to $\kappa$.

**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 $STR[g_i] \neq c_i$, we adopt rule 1 if $\varepsilon_0 = \neq$, or rule 2 if $\varepsilon_0 = \neq$.

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 $V_j$. The output $\phi$ is the constraints to make the non IV related paths infeasible.

At line 1, if it finds the paths which make the variable $v$ non infeasible. We then assign 0 to these path counters of the non inducible. We then assign 0 to these path counters of the non inducible. We then assign 0 to these path counters of the non inducible.
Algorithm 3: ReduceNonIV

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

\(STR(C, \wedge) = \wedge\) to describe the constraint (line 10). If the operator is \(\neq\) (i.e., there should be no character \(C\) in the string \(STR\)), we use the constraint \(\neg\text{contains}(STR, C)\) 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 \(\kappa_1\)) that makes the variable non inducible. The feasible condition of the path is \(str[i] == \text{'}\backslash n\text{'}\) (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 \(\text{'}\backslash n\text{'}\) in \(str\). We generate the constraint \(\neg\text{contains}(str, \text{'}\backslash n\text{'})\) 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 \(i\) 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 \(i\) increases from \(t_0\). It is similar when the index decreases from \(i\).
- 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 \{\text{', '<, '<, '>'\} operators. This assumption is due to the limitation of the existing string solvers. If a string condition contains \{\text{', '<, '<, '>'\} operators, we convert them to \{\text{', '=\} by choosing one special char. For example, \text{str[i]} > \text{'a'} is converted to \text{str[i]} = \text{'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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Versions</th>
<th>Total Loops</th>
<th>Summarized</th>
<th>Total Vul</th>
<th>S-Looper</th>
<th>Loop Unrolling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Faulty+Patched</td>
<td></td>
<td>IV-Only</td>
<td>Non-IV</td>
<td>Detected</td>
<td>Miss1</td>
</tr>
<tr>
<td>Edbrowse</td>
<td>2+2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bind</td>
<td>1+1</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Wu-Ftpd</td>
<td>2+2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SpanAssassin</td>
<td>1+1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Libgd</td>
<td>2+2</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2(2)</td>
</tr>
<tr>
<td>MADWiFi</td>
<td>2+2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>NETBSD</td>
<td>6+6</td>
<td>16</td>
<td>8</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>OpenSER</td>
<td>4+4</td>
<td>20</td>
<td>16</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Apache</td>
<td>4+4</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Sendmail</td>
<td>13+13</td>
<td>33</td>
<td>14</td>
<td>10</td>
<td>13</td>
<td>6(3)</td>
</tr>
<tr>
<td>Total</td>
<td>37+37</td>
<td>111</td>
<td>68</td>
<td>16</td>
<td>37</td>
<td>21</td>
</tr>
</tbody>
</table>

4.1 Implementation and Experimental Setup

Our loop summarization techniques are integrated into the new implementation 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-Looper. In the first experiment, we implemented the demand-driven, path-sensitive buffer overflow detection algorithm [20] and integrated S-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 space 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 Miss1 and 3 are related to the imprecision introduced when we handle non-induction variables in the loop, shown in column Miss2).

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 never 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 ¬contains(str, ‘\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-
assert falso);
}
Figure 5: Simplified Code from mime_fromqp_arr_bad.c

select 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 vul-
nearabilities 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 tech-
nique. 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. There-
fore, S-Looper is more scalable compared with the default loops
unrolling approach in the symbolic analysis. The further develop-
ment 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 tech-
nique is effective for bug finding, and it performs better than tradi-
tional loop unrolling techniques.

4.2.2 Evaluation on Improving Test Generation

In the second experiment, we compared the performance and ca-
pabilities 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 condi-
tional assert after a loop, shown as follows:

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 it-
erate through the loop more times and explore much more paths;
as a result, test input generation becomes expensive. In the ex-
periment, we compare the test case generating time using different
buffersize. As PEX works on .NET, we implemented the corre-
sponding 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 gener-
ation, 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 pro-
grams. 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 maxcon-
ditions, maxtime and maxruns).

Using S-Looper to summarize the loops, Marple can gener-
ate 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 sec-
onds (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 exam-
ple, KLEE needs more time when the loop contains the state-
ment continue in one branch. In the first program, there is a branch
if(cl == "=") continue located at the beginning of the loop. K-
LEE 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 com-
parisons 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.

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 com-
parisons of string content can have a significant impact on
the performance of test input generation;
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. However, 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 \( \text{trip count} \) for the number of times each loop executes. They then infer the relationship between inputs and variables with \( \text{trip count} \). LESE determines the relationships between variables and \( \text{trip count} \) and links \( \text{trip count} \) 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.

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Paths</th>
<th>KLEE (seconds)</th>
<th>PEX (seconds)</th>
<th>Marple with S-Looper (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>10 20 300 1000</td>
<td>10 20 300 1000</td>
<td>10 20 300 1000</td>
</tr>
<tr>
<td>mime/0toSarronechar_heavy_bad.c</td>
<td>5</td>
<td>36 1/T/O 1/T/O</td>
<td>1.02 1.13 23 43</td>
<td>0.32 0.42 0.82 13.51</td>
</tr>
<tr>
<td>mime/fromnp_arr_bad.c</td>
<td>8</td>
<td>43 1/T/O 1/T/O</td>
<td>1.21 2.33 54 782</td>
<td>0.54 0.51 1.16 23.1</td>
</tr>
<tr>
<td>close_angle_ptr_one_test_bad.c</td>
<td>3</td>
<td>0.03 0.03 13 172</td>
<td>0.14 0.31 13 412</td>
<td>0.42 0.73 0.72 13.58</td>
</tr>
<tr>
<td>outer_bad.c</td>
<td>6</td>
<td>0.02 0.02 1.10</td>
<td>1.02 1.15 48 63</td>
<td>0.37 0.45 0.73 13.56</td>
</tr>
<tr>
<td>iter2_prelxShort_arr_bad.c</td>
<td>5</td>
<td>0.3 25 1/T/O 1/T/O</td>
<td>1.16 1.19 39</td>
<td>0.38 0.46 1.21 20.3</td>
</tr>
<tr>
<td>full_bad.c</td>
<td>4</td>
<td>21 29 1/T/O 1/T/O</td>
<td>1.06 2.21 1/T/O 1/T/O</td>
<td>0.41 0.52 1.41 25.1</td>
</tr>
<tr>
<td>gd_simple_bad.c</td>
<td>3</td>
<td>20 1/T/O 1/T/O 1/T/O</td>
<td>2.01 3.34 26 182</td>
<td>0.35 0.36 0.82 13.39</td>
</tr>
<tr>
<td>interproc_bad.c</td>
<td>2</td>
<td>0.11 0.13 0.2 0.6</td>
<td>0.31 0.35 1.4 7.21</td>
<td>0.03 0.03 0.03 0.04</td>
</tr>
<tr>
<td>strchr_bad.c</td>
<td>3</td>
<td>0.02 0.02 28 563</td>
<td>0.30 1.16</td>
<td>F</td>
</tr>
<tr>
<td>noAnyMeta_int_bad.c</td>
<td>4</td>
<td>0.1 0.3 52 13/6</td>
<td>0.3 4.1 392</td>
<td>F</td>
</tr>
<tr>
<td>loops_bad.c</td>
<td>8</td>
<td>0.4 1.06 170 1/T/O</td>
<td>1.07 257 653</td>
<td>F</td>
</tr>
</tbody>
</table>

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


