LoopSCC: Summarizing Complex Multi-branch Nested Loops via Periodic Oscillation Interval

Kai Zhu[*†‡] ZhuKai__@outlook.com
Haofeng Li[§‡] lihaofeng@ict.ac.cn
Kuihao Yan[*†] yankuihao@iie.ac.cn
Rongqing Wang[*†] wangrongqing@iie.ac.cn
Jiaming Guo[*†] guojiaming@iie.ac.cn
Haoran Yang[*†] yanghaoran@iie.ac.cn
Jie Lu[§] lujie@ict.ac.cn
Lei Yu[†] yulei@iie.ac.cn
Xiaoqi Jia[*†¶] jiaxiaoqi@iie.ac.cn
Chenkai Guo[‖¶] guochenkai@nankai.edu.cn
Haichao Du[†] duhaichao@iie.ac.cn
Qingjia Huang[†] huangqingjia@iie.ac.cn
Yamin Xie[†] xieyamin@iie.ac.cn
Jing Tang[†] tangjing@iie.ac.cn

ABSTRACT

Analyzing programs with loops is a challenging task, suffering from potential issues such as the indeterminate number of iterations and exponential growth of control flow complexity. Loop summarization based on static symbolic analysis receives increasing focus in the field of loop program analysis. However, current loop summarization methods are only suitable for single-branch loops or multi-branch loops with simple cycles and fail to handle complex multi-branch nested loops with intricate non-functional variables and irregular jumps between multiple branches.

In this paper, we propose a novel loop summarization approach based on symbolic analysis to address the complex loops mentioned above and develop a tool named LoopSCC. For a non-nested loop, LoopSCC constructs an *SPath Graph*, where each node represents a path from entry to exit of the control flow graph of the loop and contracts it to an acyclic graph by collapsing each strongly connected component (SCC) into a single node. The summarization of a loop is the disjunction of the summaries of each path in the acyclic graph. For non-functional variables and irregular jumps, we iteratively model a valid value and refine the conditions until reaching the minimum. We introduce the oscillation interval, within which SPaths in the SCC can jump between one another. Outside this interval, no such jumps occur. This allows us to transform loop iteration transitions into a piecewise function, then summarize the interest variables according to the piecewise function. For nested loops, we construct summaries using the approaches above and apply them sequentially from the innermost loop to the outermost.

Comparing state-of-the-art loop summarization tools, LoopSCC can handle more types of loops with 100% precision. We integrated LoopSCC with the symbolic execution tool, Angr, to enhance its efficiency, which enabled the detection of **18** previously unknown vulnerabilities across 11 IoT projects, **7** of which were assigned CVEs. The LoopSCC is publicly available at https://github.com/hahaleyile/LoopSCC.

KEYWORDS

Loop Summarization, Data-flow Analysis, Multi-branch Loop, Recurrence Analysis

ACM Reference Format:
Kai Zhu, Haofeng Li, Kuihao Yan, Rongqing Wang, Jiaming Guo, Haoran Yang, Jie Lu, Lei Yu, Xiaoqi Jia, Chenkai Guo, Haichao Du, Qingjia Huang, Yamin Xie, and Jing Tang. 2026. LoopSCC: Summarizing Complex Multi-branch Nested Loops via Periodic Oscillation Interval. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773169

1 INTRODUCTION

Static program analysis enables vulnerability detection by simulating runtime properties without code execution [24]. However, this approach faces significant challenges when analyzing complex program structures, particularly *loops* [17, 63]. For instance, techniques such as symbolic execution and model checking suffer from path explosion, leading to computational inefficiency and incomplete analysis results [6, 23].

Figure 1a presents the simplified code related to CVE-2023-35178, an out-of-bounds write vulnerability (CWE-787), which has consistently ranked among the top 2 in the CWE list for 5 consecutive years [25]. In the loop (lines 1-10), there is an out-of-bounds write bug (line 6) where `b+c` will exceed the length of the array `a`. If we use a symbolic execution tool, such as Angr [63], to detect this vulnerability, the worst case will result in 2^{198} paths, due to the loop containing two branches and the loop executing a maximum of 198 times ($n = 198$). To address this challenge, many works [11, 12, 19, 36, 53] adopt the solution of *iteration limit*, which limits the number of loop iterations, simulating the loop as a few finite paths and executing it sequentially. However, the semantics of a loop exceeding the limited iterations will be lost causing unsound results. In addition, there are still 2^5 paths even though limiting the iterations to 5, a small enough limit. Fortunately, this loop can be summarized into a non-loop form as shown in Figure 1b, where symbolic execution would only generate two paths and handle the loop precisely.

[*]University of Chinese Academy of Sciences, Beijing, China
[†]Institute of Information Engineering, CAS, Beijing, China
[‡]Both authors contributed equally to this research.
[§]SKLP, Institute of Computing Technology, CAS, Beijing, China
[¶]Corresponding authors
[‖]Nankai University, Tianjing, China

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773169

```
1  do {
2      if (b > 0x20) {
3          c = c + 0x20;
4          b = 0;
5      } else {
6          a[b + c] = ...;
7          b = b + 1;
8      }
9      n -= 1;
10 } while (n > 0);
```

(a) Original code

```
1  if (b > 0x20){
2      b = (n-1) % (0x20+2);
3      c = [1 + (n-1) / (0x20+2)] * 0x20 + c0;
4      a[c0+0x20] = ...; ... ; a[b + c] = ...;
5  } else {
6      b = (b0+n) % (0x20+2);
7      c = (b0+n) / (0x20+2) * 0x20 + c0;
8      a[b0+c0] = ...; ... ; a[b + c] = ...;
9  }
10 assert(b + c < size(a));
```

(b) After summarizing

Figure 1: The code snippet of an out-of-bounds write (CWE-787) vulnerability (CVE-2023-35178).

Building on the above finding, Saxena et al. [58] introduced loop summarization, a technique that efficiently captures the input-output relationships of loops through symbolic constraints without requiring execution. Loop summarization techniques can be broadly categorized into two approaches based on their handling of loop behavior approximation:❶ *Abstract interpretation* employs carefully designed abstract domains to represent a superset of the loop's actual semantics, ensuring comprehensive coverage of all reachable states in the original loop [32].❷ *Symbolic analysis [41]* derives precise loop semantics by extracting operational constraints on loop variables and expressing them as mathematical formulas. These constraints are then solved to generate closed-form expressions for variables of interest [64].

While abstract interpretation provides comprehensive state coverage, its over-approximation leads to potential false positives—a limitation shared with iteration-bound approaches. Symbolic analysis, in contrast, generates precise loop models through mathematical formulas. However, as established by Rice's theorem [57] and the halting problem [27], computing concrete semantics is fundamentally undecidable, limiting current research to specific decidable loop structures.

Early symbolic analysis approaches [35, 58] primarily addressed single-branch loops—the simplest loop structures without conditional branching. However, multi-branch loops, characterized by complex inter-branch transitions, present significant challenges in capturing the intricate interleaving effects among branches.

To address these challenges, several formal methods-based approaches [13, 64, 65] have emerged. These methods introduce specialized constructs, such as path dependency automaton (PDA), to model execution dependencies between paths and transform regular loop patterns into parameterized periodic iterations. By analyzing parameter expressions (e.g., iteration counters), these approaches can interpret periodic executions and generate semantic summaries for multi-branch loops.

However, significant limitations remain in handling:

- ***Non-functional variables***: The variables related to loop iteration cannot be explicitly expressed, and the interest variables are not solvable in closed form automatically.
- ***Irregular jump***: Cases between multiple branches where the subsequent branch selection cannot be statically determined upon the completion of the current branch execution.
- ***Nested loops***: A crucial limitation given that nested structures constitute 48% of loops in real-world applications [64].

In this work, instead of the traditional parameterized periodic summarization, we propose a novel loop summarization approach to handle nested multi-branch loops that contain non-functional variables and irregular jumps. Firstly, we construct *single-iteration paths (SPaths)*, defined in Definition 1, which are the paths from entry to exit of the control flow graph (CFG) of a loop. Then, based on SPaths, we construct a *SPath Graph* (Definition 4) to describe SPath jumps between different loop iterations. By simplifying the nodes in the SPath graph at the granularity of strongly connected components (SCC), we can derive a directed acyclic graph, referred to as *contracted SPath graph* (*CSG* for short), to avoid infinite paths. Finally, we collect and calculate the conditions and induce feasible expressions of interest variables, particularly those that are critical in triggering potential security vulnerabilities according to CSG. To address non-functional variables and irregular jumps, one key point is to determine the number of iterations. To solve it, we model a valid value and iteratively refine the conditions until reaching the minimum. Subsequently, we introduce *oscillation interval* to transform loop iteration transitions into a piecewise function. This enables us to summarize the interest variables based on the piecewise function. For nested loops, we generate summaries using the approaches described above and apply them sequentially from the innermost loop to the outermost.

To validate our approach, we implemented a novel loop summarization tool, named LoopSCC, using Python with integration of the Z3 SMT solver [28] and SymPy symbolic mathematics library [49]. We evaluated the effectiveness of LoopSCC from different perspectives through extensive experiments. Firstly, we evaluated the summarization precision of LoopSCC compared to state-of-the-art baselines on public datasets, where LoopSCC can handle more types of loops with 100% precision. Secondly, we performed program verification using the benchmark SV-COMP2024 [9]. The results indicate that LoopSCC correctly verifies 86.0% of the loop programs, outperforming the best-competing tool VeriAbsL [26] by 8.8%. Thirdly, LoopSCC was integrated into Angr, a representative symbolic execution tool to evaluate its support for program analysis. The results show that LoopSCC significantly enhances the efficiency of Angr. Finally, we ran LoopSCC, integrated with Angr, on 11 IoT projects, resulting in the detection of **18** previously unknown vulnerabilities, **7** among which were assigned CVEs.

In summary, this work makes the following contributions:

- We proposed a novel loop summarization approach that effectively handles complex multi-branch nested loops with non-functional variables and irregular jumps.
- We implemented our approach as LoopSCC. Extensive experiments on public datasets and real-world programs demonstrated LoopSCC's superior performance, achieving 100% precision across diverse loop types and outperforming state-of-the-art methods.
- We extended LoopSCC for assertion verification, achieving an 8.8% improvement in loop validation compared to the best baseline across benchmark datasets.
- We enhanced Angr with LoopSCC, pioneering vulnerability detection through loop summarization in symbolic execution. This integration discovered **18** previously unknown vulnerabilities, including **7** CVEs.

The rest of the paper is organized as follows. Section 2 motivates our approach. Section 3 illustrates the framework of LoopSCC.

We evaluate the effectiveness and efficiency of our approach in Section 4. Section 5 reviews related work and Section 6 concludes this paper.

2 MOTIVATION

Figure 2 lists four typical loops that motivate our work. Figure 2a presents the simplest multi-branch loop which contains three branches (A, B, and C). The branch conditions of all three branches are determined by the variable x which exactly increases by one in each branch. Since there are no cyclic jumps among the three branches, such loops are essentially similar to single-branch loops and can be accurately summarized using traditional methods such as LESE [58], APLS [35], APC [60], Proteus [64] and WSummarizer [13]. Here, let the initial states of variables x and i be x_0 and i_0. The summarization of branches A, B and C are $x = x_0 + N_1 \wedge i = i_0 + 3 * N_1$, $x = x_0 + N_2 \wedge i = i_0 + 5 * N_2$, and $x = x_0 + N_3 \wedge i = i_0 + 7 * N_3$ where N_1, N_2, and N_3 are the number of iterations of the three branches respectively. However, in Figure 2b, branches A and B form a simple cyclic jump, which cannot be handled by traditional single-branch methods [35, 58]. Let the initial variable x_0 be greater than 8. The number of iterations N_1 for branch A must satisfy the conditions $(x_0 - (N_1 - 1) \geq 8) \wedge (x_0 - N_1 < 8)$, such that the value is $x_0 - 7$, and the states of x and i become 7 and $i_0 + 3 * (x_0 - 7)$ respectively after $x_0 - 7$ iterations of branch A. Subsequently, branch B is executed once, after which the states of x and i change to 8 and $i_0 + 3x_0 - 20$ and then branch A is executed in the next iteration. In the example, the number of iterations for $A \leftrightarrow B$ cycle can be represented as a parameterized expression of the initial variables x_0 and i_0. Existing efforts (Proteus [64] and WSummarizer [13]) can handle such simple cyclic loops by transforming them into a single-branch loop for summarization.

However, such parameterized expression transformation does not always work in the loop summarization. For instance, in Figure 2c, there exist two situations where existing efforts fail to handle. ❶ For single branch C, the number of iterations N_3 must satisfy the conditions $((x_0 + 3N_3)^3 \geq x_0 + 3N_3) \wedge ((x_0 + 3(N_3 - 1))^3 < x_0 + 3(N_3 - 1))$. However, we cannot get an explicit range for N_3 since it is an implicit expression. Therefore, such loops are hard to be summarized by previous works. ❷ Supposing the initial variable x_0 is equal to or greater than 8, the iteration number of branch A is $\lfloor \frac{x_0-3}{5} \rfloor$, such that the states of x become $x_0 - 5 * \lfloor \frac{x_0-3}{5} \rfloor$, denoted as x_1. Subsequently, the iteration number of branch B is $\lfloor \frac{9-x_1}{2} \rfloor$, such that the states of x become $x_1 + 2 * \lfloor \frac{9-x_1}{2} \rfloor$. For the $A \leftrightarrow B$ cycle, we can get a parameterized expression of variable x, $x_0 - 5 * \lfloor \frac{x_0-3}{5} \rfloor + 2 * \lfloor \frac{9-x_0+5*\lfloor \frac{x_0-3}{5} \rfloor}{2} \rfloor$, which is hard to solve via state-of-the-art recurrence relation solvers: RSolve [56] and PURRS [5]. If we simply modify "x += 2" to "x *= 2" in branch B, the constraints of iteration numbers N_2, $\lfloor log_2^{\frac{7}{x_0}} \rfloor + 1$, will become more complex. So that the closed form of variable x will become impossible to solve.

In our approach, based on the control flow graph of the loop in Figure 2c shown in Figure 3a, we can derive the SPath of the loop. As illustrated in Figure 3b, $SPath_A$ corresponds to the path 0-1-2-3, with its condition and operation being $x \geq 8$ and $x = x - 5 \wedge i = i + 3$. So we can easily get the $SPath_A$ summary as $x = x - 5 * N_1 \wedge i = i_0 + 3 * N_1$, by calculating the closed-form of operation expressions. Since there exist satisfiable values that allow $SPath_B$ to be executed immediately after $SPath_A$ completes, a jump relationship from $SPath_A$ to $SPath_B$ exists. By treating SPath as nodes and SPath jump relationships as edges, we construct a directed graph, referred to as the SPath Graph, as shown in Figure 3c. In this graph, a path from the start node to the end node represents a possible execution sequence of the loop, meaning the loop execution process can be viewed as a SPath sequence. Due to that the SPath graph may contain infinite paths, we contract the graph based on strongly connected components (SCCs), transforming it into a directed acyclic graph, as depicted in Figure 3d. The summary of the loop is the combination of all the SCCs' summaries.

The summarization of SCC_C is $x = x_0 + 3 * N_3 \wedge i = i_0 + 7 * N_3$. As we state before, the number of iterations N_3 cannot get an explicit range. To address this, we propose to model a valid value and iteratively refine the conditions until reaching the minimum. Initially, we model a satisfiable value N_3 for $(x_0 + 3N_3)^3 >= (x_0 + 3N_3)$ using Z3. Our objective is to determine the minimum number of iterations, N_3, that causes x to fail the condition and exit the SCC iteration loop. Therefore, each time we compute a satisfiable value n for N_3, we add the constraint $N_3 < n$ and continue solving until the solver returns UNSAT. This process ensures that we identify the precise value of N_3 that triggers the exit condition.

In SCC_{AB}, there exists an oscillation interval (Definition 6) $[3, 10)$, any value of the condition variable, x, within an interval remains within the interval after one iteration. The summary of SCC_{AB} is the disjunction of summaries outside and inside the oscillation interval. For values outside the oscillation interval, specifically in $[0, 3)$ and $[10, +\infty)$, the summaries are $x = x_0 + 2N_1$ and $x = x_0 - 5N_2$ where $x_0 + 2N_1 >= 3$ and $x_0 - 5N_2 < 10$ respectively. For values within the oscillation interval, there are two summarization approaches: ❶ Compute a closed-form expression of the piecewise iterative function $f(x) = \begin{cases} x - 5 & 8 \leq x < 10 \\ x + 2 & 3 \leq x < 8 \end{cases}$. For example, using PURRS, the closed-form expression can be derived as $x_n = (x_0 + 2 * n)\%7 + 3$. However, such a closed-form expression may not always exist. ❷ When the oscillation interval is finite, enumerate all possible values within the interval to determine their iteration behavior. As Figure 4 shows, within the oscillation interval $[3, 10)$, the value of the condition variable x returns to 3 after 7 iterations if its initial value was 3. And the operation sequence is $\langle +2, +2, +2, -5, +2, +2, -5 \rangle$. Therefore, when the initial value of x is 3 and the number of iterations is n, the operation sequence will be executed $\lfloor \frac{n}{7} \rfloor$ times. The remaining $n\%7$ times operations can be determined easily.

Additionally, multiple branches within the loop may evolve in irregular jumps, an branch may jump to multiple other branches depending on the condition variable. As Figure 2d shows, when branch A completes its iterations, it is hard to determine the destination branch (B or C) for jumping. Therefore, previous works fail to summarize such loops with irregular jumps due to the iteration numbers are difficult to determine. Our approach mentioned above can also handle such irregular jumps. Consider Figure 2d as an example, the oscillation interval of SCC_{ABC} is $[-29, 11)$. We can summarize the oscillation interval using the same method.

```python
while i < ...:
    # Branch A
    if x >= 8:
        x += 1
        i += 3
    # Branch B
    elif 0 <= x < 8:
        x += 1
        i += 5
    # Branch C
    else:
        x += 1
        i += 7
```

(a) Acyclic jump

```python
while i < ...:
    # Branch A
    if x >= 8:
        x -= 1
        i += 3
    # Branch B
    elif 0 <= x < 8:
        x += 1
        i += 5
    # Branch C
    else:
        x += 1
        i += 7
```

(b) Simple cyclic jump

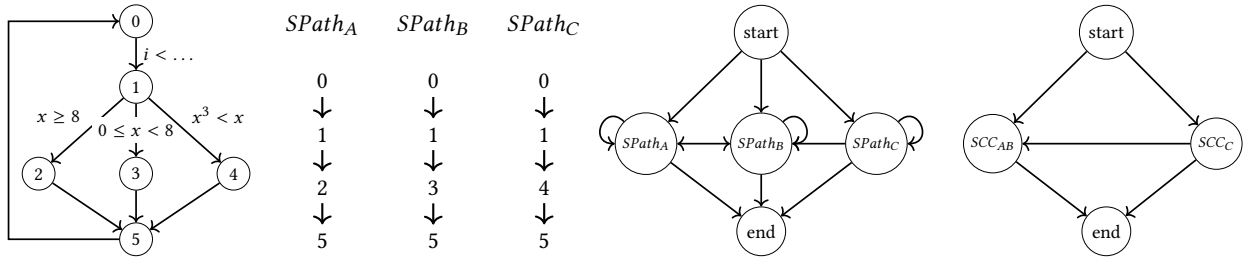
```python
while i < ...:
    # Branch A
    if x >= 8:
        x -= 5
        i += 3
    # Branch B
    elif 0 <= x < 8:
        x += 2
        i += 5
    # Branch C
    elif x^3 < x:
        x += 3
        i += 7
```

(c) Non-functional variables

```python
while i < ...:
    # Branch A
    if x >= 8:
        x -= 37
        i += 3
    # Branch B
    elif 0 <= x < 8:
        x += 2
        i += 5
    # Branch C
    else:
        x += 11
        i += 7
```

(d) Irregular jump

Figure 2: Motivating Examples. The differences between figures (b), (c), (d), and (a) have been highlighted in light green.



(a) CFG (b) Spaths (c) SPath Graph (d) Contracted SPath Graph

Figure 3: Our approach for the example in Figure 2c. To save space, the conditions on the edges and the operations in the nodes of SPaths are omitted.

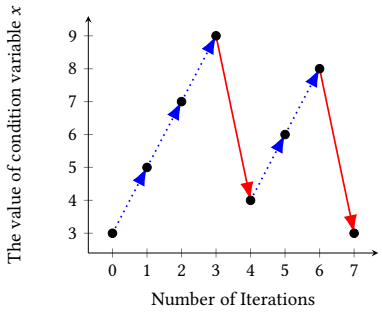


Figure 4: For the example in Figure 2c, the variation of x within the oscillation interval $[3, 10)$ with the number of iterations. The dotted line arrows indicate iteration on $SPath_B$, while the solid line arrows represent iteration on $SPath_A$.

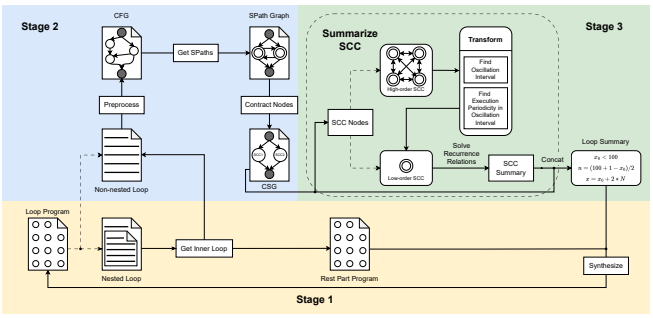


Figure 5: The Workflow of LoopSCC

3 METHODOLOGY

This section formally illustrates our efforts in summarizing complex multi-branch nested loops. Figure 5 presents the workflow of LoopSCC. Since the nested loops are prone to complicating the construction of transitions between different iterations, LoopSCC adopts the inside-out approach (details in Section 3.2.3) which always summarizes the innermost loop, applies the summarization and unrolls the innermost loop, and then handles the outer loop iteratively (stage ❶). In stage ❷, for the non-nested loop, LoopSCC extracts SPaths which represent the paths from entry to exit of the control flow graph of the loop. Subsequently, LoopSCC constructs the SPath graph, where each node corresponds to a SPath, and contracts it according to SCC (details in Section 3.1). Thereafter, the SCCs are comprehensively computed to generate the final loop summarization (stage ❸), which can be divided into three parts according to the order of SCCs: 1) For 0-order SCCs, the loop summarization can be directly determined and obtained by corresponding SPath operations. 2) For 1-order SCCs, the loop summarization requires solving the closed forms of branch condition variables which may be non-functional. 3) For high-order SCCs, we introduce the oscillation interval and iterate over each discrete value in the interval to determine the number of executions for each SPath.

3.1 SPath Graph

Definition 1. ***Single-iteration Path (SPath).*** *SPath* $sp = \{n_1 \xrightarrow{c_1} n_2 \xrightarrow{c_2} \dots \xrightarrow{c_{k-1}} n_k\}$ *is a path from the entry to the exit of the control flow graph (CFG) of a loop where a node* $n_i \in N$ *represents a basic block of the CFG and* $n_i \xrightarrow{c_i} n_{i+1} \in E$ *is an edge with the jump condition* c_i.

We utilize SPath to denote a path from the entry point to the exit point of a loop's control flow graph. In each basic block, we maintain only the variables in the loop and branch conditions and those relevant to the properties to be verified. It should be noted that a loop containing multiple entries or exits (e.g., impacted by `break` or `goto` statement) can be transformed to the canonical form containing the unique entry and exit according to *Böhm-Jacopini theorems* [16, 42]. To achieve this, we have implemented a generalized transformation module that adopts traditional program normalization algorithms [3, 4].

Definition 2. ***SPath Operation and Conditions.*** *Given a SPath sp, the sp.Op is a map of the interest variables and their values at the current node in the SPath. The sp.Pre and sp.Post maintain the maps before and after the execution of the SPath. The sp.Cond contains all condition expressions after the execution of the SPath.*

Algorithm 1 provides a method for calculating the loop conditions and operations of the SPath through a forward traversal. We iterate each statement s in the nodes of the SPath sequentially and solve the symbolic expression of the left variable ($s.lvalue$) of s (lines 2-3, 7-8). The function *Substitute* is used to substitute variables in the expression (the first parameter) with corresponding values in the map (the second parameter). We iterate each condition expression in the edges of the SPath and construct all valid conditions along the path (lines 5-6). After handling all nodes and edges, we put the current maps of $sp.Op$ into $sp.Post$. It is easy to generate a summary for a SPath, sp, based on its associated $sp.Post$.

Definition 3. ***SPath Jump.*** *Given two SPaths* sp_1 *and* sp_2*, if the result conditions in* $sp_2.Cond$ *hold when each variable* v *in* $sp_2.Cond$ *is replaced with* $sp_1.Post[v]$ *, there is a jump between* sp_1 *and* sp_2*, denoted as* $\mathcal{J}(sp_1, sp_2)$.

A jump from path sp_1 to sp_2 signifies that sp_2 may be executed immediately after sp_1 completes. To determine if such a jump exists, we can formulate it as a satisfiability problem. Specifically, the existence of a jump implies that the following two constraints are met. ❶The initial states ($sp_1.Pre$) must satisfy the conditions of sp_1 ($sp_1.Cond$). ❷Then, after applying the operations of sp_1 ($sp_1.Op$) to transform the states, the resulting states ($sp_1.Post$) must satisfy the conditions of sp_2 ($sp_2.Cond$). We then use an SMT solver like Z3 to determine if these constraints are satisfiable. If a satisfying model is found, the jump relationship is established.

Definition 4. ***SPath Graph.*** *A SPath graph* SG_l *for a loop* l *is a quadruple,* $\langle sp_s, sp_e, SP, \mathcal{J} \rangle$*, where* sp_s *and* sp_e *are two empty SPaths representing the start node and end node; SP is the set of SPaths in the* l*;* $\mathcal{J}$ *denotes the set of SPath jumps in the* l.

In the SPath graph, the SPaths and jumps in the loop are abstracted as nodes and edges, respectively. Then, the execution of the loop can be abstracted as the sequence of SPaths from the sp_s to sp_e, i.e., $sp_s, sp_1, sp_2, ..., sp_e$. However, due to the potential presence of cycles in the SPath graph, it is difficult to summarize the infinite execution sequence. To alleviate it, LoopSCC transforms the original SPath graph into a directed acyclic graph, named *Contracted SPath graph (CSG)*, based on the SCC-based contraction according to the definition of SCC in Definition 5.

Algorithm 1: SPath Operation and Conditions.

```
Input  : sp = ⟨N, E⟩
Output : sp.Cond, sp.Post
1 sp.Op ← sp.Pre;
2 foreach statement s ∈ n_1 do
3     sp.Op[s.lvalue] = Substitute(s.rvalue, sp.Op);
4 foreach n_i --c_i--> n_{i+1} ∈ E do
5     foreach expresion e ∈ c_i do
6         sp.Cond.insert(Substitute(s, sp.Op))
7     foreach statement s ∈ n_{i+1} do
8         sp.Op[s.lvalue] = Substitute(s.rvalue, sp.Op);
9 sp.Post ← sp.Op;
```

Definition 5. ***Strongly Connected Component (SCC) in SPath Graph.*** *In a SPath graph* $SG_l = \langle sp_s, sp_e, SP, \mathcal{J} \rangle$*, SCC* $scc = \langle n^{scc}, e^{scc} \rangle$ *where* $n^{scc} \in SP$ *and* $e^{scc} \in \mathcal{J}$ *is a cycle, and there is not any additional node can be added to maintain the cycle, where* n^{scc} *and* e^{scc} *are the set of nodes and edges of scc, respectively.*

According to the number of n^{scc} and e^{scc}, we can classify SCC into three types: 1) 0-order SCC contains a single node without any edge ($|n^{scc}|$=1, $|e^{scc}|$=0), 2) 1-order SCC is called the *self-loop* ($|n^{scc}|$=1, $|e^{scc}|$=1), and 3) high-order SCC contains more than one node ($|n^{scc}|$>1).

To construct the CSG, we generate the entire control flow graph (CFG) for the target program by existing construction algorithm [2]. Using the reverse post-order depth-first search algorithm [62], we identified all dominator nodes in the CFG, which allowed us to generate all SPaths of the CFG through dominator connections. Meanwhile, the jump satisfiability between SPaths can be determined by the SMT solver Z3. Subsequently, the SPath graph can be constructed for the generated SPaths and jump relations. After that, all the SCCs within the SPath graph are explored with Tarjan's algorithm [61]. Then we construct CSG based on the SPath graph by collapsing the SCC into a single node.

Example. Let us apply our approach to construct CSG on the example in Figure 2c. Based on the control flow graph (in Figure 3a), we can get three SPaths $SPath_A$, $SPath_B$, and $SPath_C$ (Figure 3b). For $SPath_A$, the condition $SPath_A.cond$ is $i < \dots \wedge x \geq 8$, and the initial values are $SPath_A.Pre[x] = x_0$ and $SPath_A.Pre[i] = i_0$. The final values are $SPath_A.Post[x] = x_0 - 5$ and $SPath_A.Post[i] = i_0 + 3$. For $SPath_B$, the constraints are $SPath_B.cond = i < \dots \wedge 0 \leq x \leq 8$, $SPath_B.Pre[x] = x_0$, $SPath_B.Pre[i] = i_0$, $SPath_B.Post[x] = x_0 + 2$, $SPath_B.Post[i] = i_0 + 5$. For $SPath_C$, the constraints are $SPath_C.cond = i < \dots \wedge x^3 \leq x, SPath_C.Pre[x] = x_0, SPath_C.Pre[i] = i_0, SPath_C.Post[x] = x_0 + 3, SPath_C.Post[i] = i_0 + 7$. When $x \geq 13$, after $SPath_A$ is executed, it jumps back to $SPath_A$. When $8 \leq x \leq 12$, $SPath_A$ is executed and jumps to $SPath_B$. Therefore, there are edges $SPath_A \rightarrow SPath_A$ and $SPath_A \rightarrow SPath_B$. Similarly, there

are edges $SPath_B \rightarrow SPath_B$ when $6 \leq x < 8$ and $SPath_B \rightarrow SPath_A$ when $0 \leq x < 6$. In addition, there are also $SPath_C \rightarrow SPath_C$ and $SPath_C \rightarrow SPath_B$. Figure 3c shows the SPath graph. After collapsing the SCC $SPath_A \leftrightarrow SPath_B$, we can get the contracted SPath graph (Figure 3d).

3.2 Loop Summarization

To construct the summarization of a loop, we can collect and concatenate the summarization of the nodes in CSG. For 0-order SCC nodes, we can simply summarize them using the conditions and operations of corresponding SPaths ($sp.Cond$ and $sp.Post$). Therefore, the challenge becomes how to summarize the 1-order and high-order SCC nodes.

3.2.1 *Summarization for 1-order SCC.*

The 1-order SCC can be treated as n iterations of a SPath sp. The problem then transforms into how to compute the $sp_n.Post$ given the $sp_1.Pre$, where sp_i denotes the SPath of the i-th iteration. For single iteration, $sp_i.Post$ can be derived according to $sp_i.Op$ given $sp_i.Pre$. Due to the SPath in 1-order SCC always jumping to itself, the input states of the $i+1$ iteration ($sp_{i+1}.Pre$) are the same as the output states of the i iteration ($sp_i.Post$). Furthermore, the number of iterations, represented by the variable n, is constrained by the conditions of the SPath ($sp.Cond$). It can be observed that in i-th iteration ($i < n$), the $sp_i.Pre$ satisfies the $sp_i.Cond$ but $sp_n.Post$ does not satisfy the $sp_n.Cond$. LoopSCC translates such existential quantification statements into constraints and feeds them into an SMT solver to compute the satisfiable values of n. However, there may be many satisfiable values for n that satisfy constraints, whereas only the smallest positive integer is the correct number of iterations. This "smallest value" relationship cannot be directly expressed as a simple constraint. Previous methods attempted to circumvent this issue by deriving an explicit expression for the iteration number. However, this is not always feasible, as the conditions of n may be constrained by some complex non-linear functions that are difficult to solve. Therefore, we design an improved dynamic programming algorithm to find such value as shown in the Algorithm 2.

In lines 1-2 of Algorithm 2, we first model a valid value of n according to its constraints using the SMT solver (here, we use Z3). Once a satisfiable value n_{val} is found, an extra constraint $n < n_{val}$ is added to explore a smaller n_{val}. The process continues until no smaller n_{val} can be found, where the actual number of iterations in 1-order SCC is determined (lines 3-8). In our experience, this approach quickly finds the optimal solution (only 3.3 attempts on average), benefiting from Z3 's efficient polynomial integer arithmetic solving. It should be noted that we accelerate the entire solving process by leveraging the incremental constraint solving approach [45], which avoiding redundant computations by reusing intermediate assertion stack frames created during previous satisfiability checks.

Comparing with previous works. Advanced loop summarization efforts [13, 64] proposed to extract the explicit symbolic representation for the number of iterations n, i.e., compute the n from $sp_n.Cond$. However, practical conditions often contain *implicit expressions* that can not be directly computed, making existing efforts invalid. For instance, for a simple loop, `while x^7 < x^3 + 2: x = x + 2`, whose $sp.Op$ and $sp.Cond$ are $x = x+2$ and $x^7 < x^3+2$, respectively, the final condition can be referred as $(x_0+2*n)^7 < (x_0+2*n)^3+2$.

Algorithm 2: Determination of the Number of Iterations

Input : $sp.Cond$, $solver$
Output: the number of iterations in 1-order SCC

1 $solver \leftarrow \{n > 0, sp_{n-1}.Cond, \neg sp_n.Cond)\}$;
2 $n_{val} \leftarrow solver.model()$;
3 **while** *True* **do**
4 $solver \leftarrow n < n_{val}$;
5 **if** $solver.solve()$ *is UNSAT* **then**
6 break ;
7 **else**
8 $n_{val} \leftarrow solver.model()$;

This is a typical *implicit expression*, from which an explicit expression regarding n cannot be extracted. On the contrary, LoopSCC employs an existential quantification strategy to identify an appropriate n that satisfies the $sp_n.Cond$.

3.2.2 *Summarization for High-order SCC.*

The key to summarizing a high-order SCC lies in identifying the conditions under which multiple SPaths jump to each other. In other words, given a variable x in SPath conditions of the SCC, we need to determine the minimal value range of x within which the SPaths in the SCC can jump to one another, outside this range, no such jumps occur. We refer to such a range as *oscillation interval* (Definition 6). In this way, the summarization of the high-order SCC problem will be divided into two parts: the summary within the interval and the summary outside the interval.

Definition 6. ***Oscillation Interval.*** *For a SCC, scc, the oscillation interval $\mathbb{I} = \bigcup_{i=1}^{k}[a_i, b_i]$ is a minimal interval set, that satisfies the following constraints. Given a condition variable (variable in SPath conditions) x, for all $v \in \mathbb{I}$, if v satisfies $sp_i.Cond$ where $sp_i \in scc$, then after applying v to $sp_i.Post[x]$ to obtain the value v', v' is still in $\mathbb{I}$. Meantime, for each $sp_i \in scc$, there exists $v'' \in \mathbb{I}$ that satisfies $sp_i.Cond$.*

According to the definition of the oscillation interval, the oscillation interval is fundamentally a closed interval. This means that any condition variable value within this interval is guaranteed to remain within the interval after executing any SPath operation of the SCC, whether it is a self-iteration or a jump to another SPath. Outside the oscillation interval, a SPath will continuously iterating itself, until jump inside the oscillation interval or jump to the next SCC node in CSG. These cases can be viewed as a summarization of 0-order and 1-order SCCs. Therefore, we can focus on the oscillation interval to summarize high-order SCCs.

The oscillation interval can be computed by following these steps. ❶Initialization: determining the domain of an SPath, which is the value ranges of the condition variable for which the SPath's conditions ($sp_i.Cond$) are satisfied. These ranges can be computed from the SPath conditions using a symbolic math library like Sympy. For instance, if an SPath's condition is `x*x < 1`, then its domain can be calculated as the interval $(-1, 1)$. Applying this domain to the operations in the SPath, we can compute the resulting interval. The intersection between the resulting interval and the domains of other SPaths belongs to the oscillation interval. This implies that after any transition between different SPaths, the condition variable

Algorithm 3: Identification of oscillation Interval

```
Input  : sp_i: each node of the target SCC scc;
         x: the condition variable;
         [a_i, b_i]: domain of x for sp_i in scc;
Output : 𝕀: oscillation interval;
1  foreach sp_i ∈ scc do
2      [a'_i, b'_i] = calculate(sp_i, [a_i, b_i], x);
3      foreach sp_j ∈ scc; i ≠ j do
4          [s, t] = [a'_i, b'_i] ∩ [a_j, b_j];
5          if [s, t] ≠ ∅ then
6              𝕀 ∪ [s, t];
7  while 𝕀 is changed do
8      foreach sp_i ∈ scc do
9          foreach [m, n] ∈ 𝕀 do
10             [m', n'] = calculate(sp_i, [m, n], x);
11             if [m', n'] ≠ ∅ then
12                 𝕀 ∪ [m', n'];
13 Function calculate(sp, [a, b], x):
14     [a', b'] = apply [a, b] to sp.Post[x];
15     return [a', b'];
```

value is inevitably contained within this interval. ❷Iteration: We iteratively treat the newly added interval as the domain of each SPath, and check whether it allows entry into the SPath and, after applying this domain to its operations, enables transitions to other SPaths. New intervals are continuously computed until a fixed point is reached. This ensures that any value within the interval remains within the interval after passing through any SPaths in the SCC.

Algorithm 3 demonstrates an algorithmic implementation of the above recursive definition. Each SPath can be regarded as a function of the condition variable x. In lines 13-15 of Algorithm 3, the function `calculate` is used to determine the range of a SPath for a given domain $[a, b]$. If the range of sp_i and the domain of sp_j intersect at the interval $[s, t]$, sp_i can transition to sp_j when $[s, t]$ is non-empty. We iterate each SPath to calculate the transition interval and merge them (lines 1-6). By using the interval obtained above as the domain, we apply this domain to each SPath to compute its range. The function `calculate` will return empty if the interval is outside the processing of a SPath. We then merge the domains and ranges and then repeat this process until we obtain a fixed closure interval, which is the oscillation interval (lines 7-12).

Given an oscillation interval, the condition variable, x, always changes within this interval as the SCC iterates, and the change sequence will be periodic which is called *Execution Periodicity in Oscillation Interval* (Definition 7). The SPath to be executed for each value of x is determined within this periodic interval. In other words, the operation sequence of an interest variable, x, is determined within this periodic interval, denoted as $\langle sp_i.Post[x], sp_j.Post[x], \dots \rangle$. Suppose that the number of iterations of a loop is N and the length of the periodic interval is M. The operation sequence above will be executed $\lfloor \frac{N}{M} \rfloor$ times. The remaining $N\%M$ operations after $\lfloor \frac{N}{M} \rfloor \times M$ iterations can be determined easily. The final summarization is a piecewise function depending on the initial condition variable.

Definition 7. ***Execution Periodicity in Oscillation Interval.*** *In a high-order SCC, its oscillation interval is $[m, n]$ for condition variable x. Let the initial value of x be x_0. After multiple iterations, we can obtain a sequence of x values, $[x_0, x_1, ..., x_k, x_0, ...]$. Since the values of x always change within the oscillation interval (Definition 6), x_0 always appears periodically. The sequence $[x_0, x_1, ..., x_k]$ is called execution periodicity in oscillation interval according to condition variable x.*

Note that LoopSCC can only handle finite oscillation intervals, as discussed in Section 3.3. According to Definition 6, if variable state is within an oscillation interval, then after any number of iterations, it will remain within that interval. Since the interval is finite, two identical variable states will eventually arise for condition variable according to the pigeonhole principle [38]. The program's execution behavior depends solely on the current code location and variable state. Therefore, subsequent SPath executions will be periodic.

Let us apply our approach to identify the oscillation interval and generate the summary of the example in Figure 2d. The domains of condition variable x for $SPath_A$, $SPath_B$, and $SPath_C$ are $[8, +\infty)$, $[0, 8)$, and $(-\infty, 0)$ respectively. The ranges over the aforementioned domains are $[-29, +\infty)$, $[2, 10)$, and $[-\infty, 11)$ respectively. After executing lines 1-6 in Algorithm 3, the $\mathbb{I}$ becomes $[-29, 11)$. Then we apply lines 7-12 in Algorithm 3 until reaching a fixed point, obtaining the final oscillation interval, $[-29, 11)$. We iteratively traverse each value within the oscillation interval. For instance, starting with $x = -29$, we obtain the sequence of values of x through each iteration: -29,-18,-7,4,6,8,-29. When two identical numbers appear in the iteration sequence, it indicates that an execution periodicity has been identified. In the execution periodicity, SPaths C, C, C, B, B, A, are executed in sequence. Subsequently, we determine the periodic value sequences for all values. We record the periodic value sequences corresponding to each value and its offset within the sequence using an array $Attr$. For example, $Attr$["-29"][0] = [-29,-18,-7,4,6,8] represents the periodic value sequence for -29, and $Attr$["-29"][1] denotes the position of -29 within that sequence. Based on this, the summary of the variable x can be expressed as follows: $summary = \begin{cases} Arr = Attr['x'_0][0] \\ offset = Attr['x'_0][1] \\ x = Arr[(N + offset) \bmod size(Arr)] \end{cases}$.

3.2.3 Summarization for Nested Loop.

To address nested loops, We adopt an inside-out approach. This method first generates and applies summarizations of the innermost loop, then transforms it into a linear program and reintegrates it into the original nested loop program iteratively, thereby eliminating the nesting. A typical loop summarization (details in Section 3.2.1 and Section 3.2.2) contains two parts: 1) Value range of involved variables; 2) Mappings between pre-variables and post-variables, which are transformed by LoopSCC separately. For the variable values in the summarization, LoopSCC transforms them into conditions for different branches based on the condition states. For variable mapping relationships, LoopSCC converts them into assignment statements within corresponding branches according to the operation types.

For instance, if the innermost loop of a target nested loop is summarized as "$(x_0 >= 100 \wedge x = x_0) \vee (x_0 < 100 \wedge N = (100 + 1 -$

$x_0)/2 \wedge x = x_0 + 2 * N)$", LoopSCC can transform it into an equivalent non-nested linear program in two steps. ❶ For $(x_0 >= 100 \wedge x = x_0)$, the condition $(x_0 >= 100)$ is transformed to condition sentence `if(x >= 100)`, and the assignment $x = x_0$ is transformed to operation `x = x` for the first branch. ❷ For $(x_0 < 100 \wedge N = (100 + 1 - x_0)/2 \wedge x = x_0 + 2 * N)$, it can be transformed into statements `N = (100 + 1 - x) / 2; x = x + 2 * N;` with condition `if(x < 100)` for another branch.

3.3 Limitations

In prior work, Proteus [64, 65] categorized loops into four types based on two criteria: the presence of non-induction variables and the patterns of path interleaving. While Proteus is capable of handling Type 1 loops, LoopSCC makes a significant advancement by handling irregular jumps, thereby extending loop summarizing capability to Type 2 loops as well. Handling Type 3 and 4 loops, which involve non-induction variables, remains a challenging mathematical problem that we also do not address (as will be detailed in the second point below). Even within these tractable classes (Type 1 and 2), however, there are still some certain boundary cases hard to handle:

First, our method is currently restricted to loops only involving numerical operations. For loops with memory-related operations (e.g., array operations with variable indices or pointer operation), LoopSCC is not applicable. This is a common limitation shared by previous symbolic analysis approaches [13, 64, 65]. Memory operations can be addressed by combining abstract interpretation techniques with sacrificing some precision. Combining symbolic analysis and abstract interpretation is a topic worthy of separate investigation.

Second, for loops with numerical operations, our approach requires that closed-form expressions can be derived for all SPath operations ($sp.Op$) in the CSG. This is a fundamental prerequisite for summarizing both 1-order SCCs (Section 3.2.1) and high-order SCCs (Section 3.2.2). Previous loop summarization techniques have similar requirements [13, 60, 64, 65], analogous to the conception of induction variables.

Subsequently, among the remaining loop types, our method failed if the loop is both input-dependent [35] (some type of infinite loops) and the number of iterations cannot be derived to an explicit expression (Figure 2c in Motivation). However, an extension of our method could still yield an unsound summary for use in over-approximate program analysis. This potential for extension is unique to our approach and not feasible for prior methods [13, 64, 65].

Finally, LoopSCC cannot handle cases where the oscillation interval is infinite, which did not occur in our experiments.

3.4 Preserving Concrete Semantic

Our approach is under-approximation, which means it may exclude certain loop behaviors, as discussed in Section 3.3. And it ensures completeness in the sense that its loop summaries contain no spurious behaviors (zero false positives).

Proof Sketch. A CSG is composed of 0-order, 1-order, and high-order SCCs. For 0-order SCC, the summary is derived directly from its SPath operations and conditions, which preserve concrete semantics. For 1-order SCC, the repetitive iteration of the SPath is reduced to a closed-form expression for the iteration count, n, again without any loss of semantics. Since the correct value of n can be determined either by deriving its explicit expression or by applying Algorithm 2. For high-order SCC, we search the oscillation interval according to Definition 6. Out-of-interval behavior is handled in the same way as for 0-order or 1-order SCC. Inside the interval, the sequence of SPath executions always occurs periodically. We can precisely model the concrete semantics of the sequence SPaths in the same way as for 1-order SCC. Therefore, we can preserve concrete semantics for all kinds of SCC. However, LoopSCC may exhibit false negatives (under-approximation) because it fails to handle certain cases – such as memory-related operations, input-dependent loops, and others – as discussed in Section 3.3.

Table 1: Precision results for tools on our benchmarks. The #S represents the number of loop programs that can be successfully executed (precision is not 0%). The #A represents the average precision in successfully executed cases. CPA represents CPAchecker and WS represents WSummarizer.

Loop ID	CBMC	CPA	ICRA	VeriAbsL	WS	Proteus	LoopSCC
fig1	75.3	52.3	100	52.3	81.6	100	100
fig4-4	100	100	100	100	38.2	100	100
fig4-5	55.3	100	100	100	0	100	100
ex3	74.2	100	100	100	0	100	100
ex4	72.1	100	100	49.1	0	100	100
fig2-1	69.0	100	100	73.6	80.4	100	100
fig2-2	64.5	78.0	50.3	78.0	94.7	100	100
nested-multiple	68.3	100	100	87.4	0	100	100
nested-single	77.0	100	100	100	0	100	100
sequential-single	95.3	100	100	100	100	100	100
simple-multiple	67.5	100	100	100	100	100	100
simple-single	74.5	100	100	100	100	100	100
simple-single-2	46.1	25.4	100	25.4	86.7	100	100
t07	58.1	47.2	100	100	0	100	100
t08	82.8	12.1	100	100	100	100	100
t10	75.9	100	100	100	100	100	100
t11	73.1	100	100	77.9	100	100	100
t13	100	100	100	100	0	0	100
t15	97.9	95.0	100	100	0	0	100
t16	1.9	2.8	4.4	0	0	0	100
t19	83.8	100	100	100	100	100	100
t20	66.2	51.1	100	100	100	100	100
t27	55.5	51.9	51.9	51.9	0	0	0
t28	3.1	90.0	100	100	0	100	100
t30	44.9	27.8	0	27.8	0	0	0
t47	96.0	100	100	52.1	52.7	100	100
bobble	51.5	50.6	50.6	51.9	100	100	100
inchworm	49.9	48.4	48.4	50.5	100	100	100
leapdiff2	49.6	48.3	48.3	50.1	0	100	100
leapfrog	52.5	51.1	51.1	51.1	71.2	0	100
leapfrog-asymmetric	51.6	50.0	72.9	50	0	0	100
leapspin	51.7	48.5	48.5	48.5	100	100	100
leapthree	51.6	47.7	47.7	47.2	100	100	100
microbobble	55.9	52.6	52.6	53	100	100	100
simple-bl	55.5	50.6	50.6	52.4	0	0	0
#S	35	35	34	34	20	27	32
#A	64.2	70.9	81.7	74.4	90.3	100	100

4 EVALUATION

LoopSCC is implemented in Python 3.12.0 (1,887 LOC), equipped with the SMT solver Z3 (4.13.0) for condition solving and Sympy (1.21.1) for interval computation. To showcase the usefulness and precision of LoopSCC, we have undertaken the following efforts. We compared LoopSCC with six loop analysis tools on C4B [18] to show that we can handle more types of loops with 100% precision. We applied LoopSCC for software verification and symbolic execution and compared it with related tools. Finally, we use LoopSCC to detect vulnerabilities in real-world applications. We achieve

the above evaluation goals by answering the following research questions (RQs):

- RQ1: How does LoopSCC perform on loop summarization?
- RQ2: What is the effectiveness of LoopSCC in supporting practical software verification?
- RQ3: Can LoopSCC enhance the performance of symbolic execution?
- RQ4: Can LoopSCC detect vulnerabilities in real-world programs?

The experimental setup and results are discussed in the following sections. All experiments were conducted on an Intel i9-13900 machine with 32 GB of RAM, running Fedora 40 OS.

4.1 RQ1: Performance of LoopSCC.

4.1.1 Benchmark. We evaluated the types of loops that LoopSCC can summarize, as well as the precision, on the C4B [18], a public benchmark of loop programs that is commonly used in previous related works [41, 52], and the Branching-Loop developed by ICRA [41]. The C4B collected 36 challenging and representative loop programs from open-source software and literature. After removing memory-related loops and those containing nested function calls, as discussed in Section 3.3, we collected a set of 26 loop programs, which are listed in the top 26 rows of Table 1. Out of the 39 loop programs in Branching-Loop, 9 have been retained after excluding duplicate loop programs. For instance, the differences between "`bobble.c`" and "`bobble_unbound.c`" are the lines `int N = 10` and `int N = _VERRIFIER_nondet_int()` where the variable `N` is within the loop conditions. Both programs are considered duplicates because we treat N as a random input as described in the experimental settings (Section 4.1.3).

4.1.2 Baselines. We select six state-of-the-art loop analysis tools as baselines for the comparison experiments, including four abstract interpretation methods (CBMC [21], CPAchecker [10], VeriAbsL [26], and ICRA [41]), which under-approximate or over-approximate loop behaviors, and two advanced symbolic analysis methods (Proteus [64] and WSummarizer [13]), which preserve the original program semantics. We reimplemented Proteus based on its paper since its source and executable code are unavailable publicly.

4.1.3 Experimental Settings. We randomly generated 1k inputs for each test case and executed the original loop to obtain 1k actual outputs as ground truth. In our benchmarks, the inputs are closely related to the number of loop iterations. We execute each tool to interpret the loop output for each given input and calculate its precision by comparing the output to the corresponding ground truth. Specifically, precision is the ratio of the tool's outputs that exactly match the ground truth (the actual execution results). For instance, if LoopSCC produced 1,000 outputs for the 1,000 random inputs and 700 of them were identical to the ground truth, its precision is reported as 70.0%. For all tools, a time limit of 5 minutes is imposed for each input. Exceeding this limit is regarded as an interpretation failure.

4.1.4 Results. Table 1 gives the precision results for tools on our benchmarks. Compared with abstract interpretation tools (CBMC, CPAchecker, ICRA, and VeriAbsL), LoopSCC provides a more precise interpretation of loops. For successfully executed cases, LoopSCC achieves 100% precision for all loop programs. In comparison, CBMC, CPAchecker, ICRA, and VeriAbsL obtain average precisions of 64.2%, 70.9%, 81.7%, and 74.4%, respectively. Compared with advanced symbolic analysis tools (WSummarizer and Proteus), LoopSCC can handle more types of loops. Out of 35 loop programs, LoopSCC handles 32 loop programs, 5 and 12 more than Proteus and WSummarizer respectively. This is because LoopSCC can correctly summarize multi-path nested loops with complex non-functional variables and irregular jumps. The loop programs `t13`, `t15`, and `t16` belong to nested loops, and the two loops in Branching-Loop (`leapfrog` and `leapfrog-asymmetric`) contain non-functional variables.

LoopSCC only fails to summarize three loop programs, which also cannot be analyzed by Proteus and WSummarizer. This is due to the inability to derive closed-form expressions. For example, `t30` involves a multivariate interdependence, $x_n = y_{n-1}$, $y_n = x_{n-1} - 1$. Deriving a closed-form expression for such multivariate recurrences has not yet been implemented in LoopSCC, as discussed in Section 3.3.

For each tool, the average execution time per input is as follows: CBMC 22.5 seconds, CPAchecker 75.5 seconds, ICRA 0.6 seconds, VeriAbsL 47.4 seconds, WSummarizer 6.9 seconds, Proteus 1.4 seconds, and LoopSCC 1.6 seconds. As the results indicate, LoopSCC is highly efficient, with performance comparable to that of ICRA and Proteus.

4.2 RQ2: Application on Software Verification

4.2.1 Benchmark. We collect loop programs from SV-COMP2024[1], one of the most well-known competitions in the field of software verification. Based on the presence of nested loops, we organized the data into the following categories: `Loop-new`, which primarily consists of non-nested loops, and `Loop-Simple`, which only encompasses nested loops. In addition to the above programs, SV-COMP2024 also includes several benchmark branches contributed by well-known loop analysis tools, such as `Loop-Acceleration` [21], `Loop-Crafted` [1, 26], `Loop-Invariants` [10], `Loop-Zilu` [44], and `Loops` [48]. Ultimately, after excluding programs that were out of our scope, specifically, those involving memory operations or nested function calls (as discussed in Section 3.3), we collected a total of 114 loop programs for our experiments.

4.2.2 Baselines. In the six baselines figured in RQ1, only CBMC, CPAchecker, VeriAbsL, and Proteus can be successfully applied in the software verification and achieve promising results. Especially, VeriAbsL won the Top 1 in the competition of *SV-COMP 2024 ReachSafety*. Therefore, the four tools are selected as comparative baselines in this experiment.

4.2.3 Experimental Settings. Software verification aims to ensure that software meets specified requirements. In practice, the software verification employs specific `assert` statements to determine whether the target property within a certain program path holds. Similar to the verification approach in Proteus, we have extended LoopSCC to enable it to check the assertions within the loops and outside the loops: ❶ For assertions within the loops, LoopSCC

[1]SV-COMP 2024: https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/blob/svcomp24-final

identifies each SPath, sp_i, that contains properties to be verified and converts the negation of the assert conditions into expressions in $sp_i.Cond$ through variable substitution. After that, LoopSCC computes the loop summarization. If there exists a valid loop summarization that includes the summarization of sp_i, then LoopSCC deems this assertion to be incorrect. ❷ For assertions outside the loops, we directly append the negation of the assert condition to the loop summarization and check whether the new summarization can be satisfied. LoopSCC considers the assertion to be correct if the new summarization is valid.

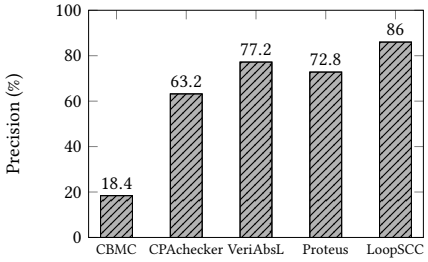


Figure 6: Precision Comparison in Software Verification.

4.2.4 Results. Figure 6 gives the precisions of five tools on our benchmarks. LoopSCC successfully verified 86.0% of the programs, outperforming the baselines, VeriAbsL, Proteus, CPAchecker, and CBMC by 8.8%, 13.2%, 22.8%, and 67.6% respectively. From further analysis, we found that the verification errors in LoopSCC are primarily attributed to the incomplete implementation of complex operations, resulting in the closed-form expressions being hard to represent. For instance, LoopSCC fails to verify the programs `phases_2-1`, `phases_2-2` and `underapprox`, which involve operations like *square* or *division*.

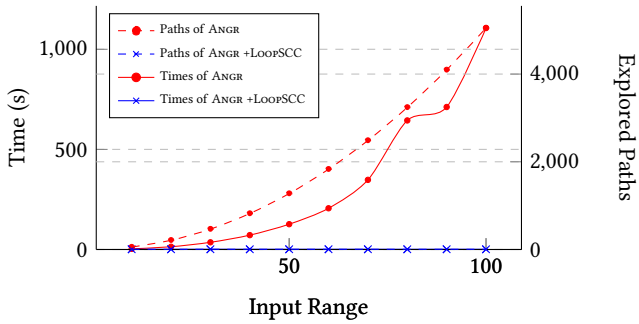
4.3 RQ3: Application on Symbolic Execution

In this experiment, we evaluate the performance of our summarization approach in solving the path explosion problem caused by loops in symbol execution by comparing the performance of Angr with and without integrated LoopSCC on 5 programs used in Section 4.1 that can be solved by the LoopSCC but cannot be addressed by other symbolic analysis tools [13, 64].

4.3.1 Experimental Settings. To integrate LoopSCC and Angr, we leverage Angr to derive the constraints of basic blocks outside loops and use LoopSCC to capture the constraints within loops. Specifically, we employ the `LoopFinder` module provided by Angr to identify all loops within the specified function. We record the starting addresses of all loops and configure Angr 's symbolic execution to explore up to the positions immediately preceding the loops. Inside the loop, we determine the loop conditions, path conditions, and path operations by inspecting the types of `VEX IR` instructions. For instance, conditional jumps correspond to the `IRStmt.Exit` instruction type, from which we extract the `guard` attribute to identify the conditional variable being evaluated. By tracing upward to locate the `compare` instruction, we can retrieve the variable involved in the branching condition and its relation. Using this approach and the control flow structure, we reconstruct the loop structure, which is then used as input for LoopSCC to compute the constraints for the loop variables. In accordance with Angr's requirements, we constrain the input range to ≤ 100 by default. For all benchmarks, the inputs are closely related to the number of loop iterations. Real-world programs often involve significantly more loop iterations. For example, as shown in Figure 1, the maximum number of loop iterations reaches 198.

Programs	Angr	Angr + LoopSCC
`t13`	1,730.9	1.2
`t15`	1,238.3	1.3
`t16`	>5h	1.2
`leapfrog`	>5h	1.3
`leapfrog-asymmetric`	1,106.2	1.1

(a) Time costs (s) with/without LoopSCC.



(b) The relation between performance and input ranges on `leapfrog-asymmetric`.

Figure 7: Symbolic execution supported by LoopSCC.

4.3.2 Results. Figure 7a gives the time of Angr integrated with and without LoopSCC. Without summarization provided by LoopSCC, Angr takes more than 5 hours at worst (`t16` and `leapfrog`) and 1,106.2 seconds at best (`leapfrog-asymmetric`), while Angr +LoopSCC executes in less than 1.4 seconds for each program.

To further demonstrate the advantages of our approach, we compare the performance of Angr and Angr +LoopSCC in solving `leapfrog-asymmetric` across various input limits, ranging from 10 to 100. As Figure 7b shows, the times and explored paths of Angr grow exponentially as the input range increases, whereas those of Angr +LoopSCC remain nearly unchanged. This demonstrates that our approach is insensitive to the number of iterations.

4.4 RQ4: Scalability on Real-World Loops

We have integrated LoopSCC into Angr to detect out-of-bounds write (CWE-787) vulnerability, which has consistently ranked among the top 2 in the CWE list for 5 consecutive years [25]. In our method, we treat the computation of offset addresses as potential risk points. If an offset address is computed within a complex loop and subsequently used in an array write operation, we analyze whether the offset might exceed the array's size. Compared with Angr which struggles to handle loops effectively and leads to path explosion, LoopSCC can summarize complex loops to significantly improve the performance of Angr, as demonstrated in Section 4.3.

We analyzed 11 IoT devices using Angr equipped with LoopSCC and uncovered **4** known vulnerabilities (CVE-2023-35178, CVE-2018-5924, CVE-2018-5925, CVE-2023-27971) as well as **18** previously undiscovered ones. Among the latter, **7** have been assigned CVE identifiers, while **11** are still under review by the vendors.

Table 2 lists the information of the **7** CVEs. All CVEs have received high CVSS scores. Due to the significant impact of the zero-day vulnerabilities we uncovered, which extend across multiple product lines and pose significant risks to vendor security, we withhold the details about these vulnerabilities at the vendors' request. Instead, we will demonstrate our method using examples of one-day vulnerabilities we identified.

Table 2: Unknown CVEs detected by LoopSCC.

No.	Vendors	Model	CVE ID	CVSS v3 score
#1	TOTOLINK	A3600R	CVE-2024-7173	8.8
#2		A7000R	CVE-2024-7212	8.8
#3			CVE-2024-7213	8.8
#4		N35ORT	CVE-2024-7462	9.8
#5	Tenda	AC10 v4.0	CVE-2024-32317	7.5
#6		AC15	CVE-2024-2808	9.8
#7	TP-LINK	TL-WR841ND	CVE-2024-9284	6.5

Case Study: Figure 8 shows the core snippet of CVE-2023-35178. Line 8 is an array write operation with `b + c` as the array offset. We use Angr to get the constraints of b and c before the loop (lines 3-13) and use LoopSCC to summarize the constraints within the loop. The property we needed to verify was: $b \leq 0x20 \wedge b + c < 6 * 0x20$. Using our algorithm described in Section 3, we discovered a satisfiable user input that violates the above constraints, indicating the potential existence of the vulnerability.

```
1  void func(){
2    char str[6][0x20];
3    do {
4      if (b > 0x20) {
5        c = c + 0x20;
6        b = 0;
7      } else {
8        str[b + c] = ch;
9        b = b + 1;
10     }
11     arg = arg + 1;
12     ch = *arg;
13   } while ((ch & 0xdf) != 0);
14 }
```

Figure 8: The code snippet of CVE-2023-35178

5 RELATED WORK

5.1 Loop Analysis

Loop analysis is a core focus in program analysis, with key techniques like loop summarization and invariant inference.

Loop summarization focuses on generating constraint expressions to describe the relations between loop inputs and outputs, enabling loops to be replaced by these expressions during program analysis. Existing techniques can be categorized into symbolic analysis [13, 35, 35, 40, 58, 64, 65] and abstract interpretation methods [43, 50, 51]. Symbolic analysis derives precise loop semantics by extracting operational constraints on loop variables and expressing them as mathematical formulas. For instance, APLS [35] and APC [60] detect linear relationships among induction variables [35] to derive loop summaries. These methods often face challenges when analyzing multi-branch loops due to difficulties handling complex loop control flow. To address such challenges, data-flow analysis methods like Proteus [64, 65] and WSummarizer [13] have been introduced. Proteus models jump relations between paths using PDA automaton, propagating expressions along simple cycles to derive summaries. WSummarizer combines loop unwinding with Proteus' method, enabling it to summarize specific loops that contain irregular jumps. Additionally, S-Looper [66] and Kapus et al.'s approach [40] focus on specific string loop operations. Abstract interpretation, on the other hand, derives a superset of the loop's semantics, offering approximations rather than precise summaries. Compositional recurrence analysis (CRA) [32] adopts a divide-and-conquer approach to synthesize abstractions for subprograms. There are different variations of CRA methods, such as program composition based on predicate abstraction [43] and approximation based on numerical abstraction for program behavior [50, 51]. ICRA [41], an extension of CRA, introduces context-sensitive interprocedural reasoning and integrates Newton iteration methods [31] to process context free structures.

Loop invariant generation derives constraint relations that hold throughout loop execution. Analysis-based techniques explicitly analyze program structures to derive invariants such as counterexample-guided abstraction refinement (CEGAR) [20], predicate abstraction [7, 8, 33, 37], Craig interpolation [46, 47] and IC3/PDR-based methods [15]. Learning-based approaches leverage machine learning to infer invariants such as ICE learning [34] refines invariant candidates using counterexamples, while deep learning-based methods like Code2Inv [59] and Pei et al.'s approach [54] extract features from loop structures to predict invariants effectively.

5.2 Applications of Loop Analysis

Loop analysis has been employed in a wide range of analysis techniques, including model checking [22, 55] and symbolic execution [14], to address path explosion [6, 23, 30]. Bounded Model Checking (BMC) [12, 21] reduces the number of iterations in a loop using loop unwinding, which mitigates state explosion but can lead to unsound verification if the unwinding bound is insufficient. The k-induction [29, 39] approach has been proposed to identify more loop invariants by considering k consecutive iterations. In the domain of symbolic execution, Loop-Extended Symbolic Execution (LESE) [58] introduces the concept of symbolizing loop iteration counts as variables and accelerates loop execution by eliminating redundant paths. Loop summarization has shown significant success by abstracting loop behaviors into concise summaries to accelerate both software verification and symbolic execution [35, 41, 60, 64, 65]. LoopSCC can precisely summarize more types of loops, including those with non-functional variables, irregular jumps, and nested loops, to further accelerate software verification and symbolic execution.

6 CONCLUSION

This paper proposes an SCC-based summarization approach for complex multi-branch nested loops with non-functional variables and irregular jumps. Compared to state-of-the-art methods, the proposed approach covers a wider range of loop types and achieves higher summarization precision. Additionally, we integrated LoopSCC with the symbolic execution tool, Angr, to detect vulnerabilities. We found **18** previously unknown vulnerabilities, **7** of which were assigned CVEs.

ACKNOWLEDGMENTS

This work was partially supported by Project GJ020302 and the NSFC 62402474. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any finding agencies.

REFERENCES

[1] Mohammad Afzal, A Asia, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Advaita Datar, Shrawan Kumar, and R Venkatesh. 2019. VeriAbs: Verification by abstraction and test generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1138–1141.

[2] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[3] Z. Ammarguellat. 1992. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering* 18, 3 (March 1992), 237–251. doi:10.1109/32.126773

[4] E. Ashcroft and Z. Manna. 1979. *The translation of "go to" programs to "while" programs*. Yourdon Press, USA, 49–61.

[5] Roberto Bagnara. 2025. PURRS. https://www.cs.unipr.it/purrs/. [Accessed: 16-January-2025].

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[7] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. 203–213.

[8] Thomas Ball and Sriram K Rajamani. 2002. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1–3.

[9] Dirk Beyer. 2024. State of the art in software verification and witness validation: SV-COMP 2024. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 299–329.

[10] Dirk Beyer and M Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 184–190.

[11] Armin Biere. 2021. Bounded model checking. In *Handbook of satisfiability*. IOS press, 739–764.

[12] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems: 5th International Conference, TACAS'99 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99 Amsterdam, The Netherlands, March 22–28, 1999 Proceedings 5*. Springer, 193–207.

[13] Martin Blicha, Jan Kofroň, and William Tatarko. 2022. Summarization of branching loops. In *Proceedings of the 37th ACM/SIGAPP symposium on applied computing*. 1808–1816.

[14] Robert S Boyer, Bernard Elspas, and Karl N Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices* 10, 6 (1975), 234–245.

[15] Aaron R Bradley. 2011. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 70–87.

[16] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (May 1966), 366–371. doi:10.1145/355592.365646

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[18] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. 2015. Compositional certified resource bounds. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 467–478.

[19] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Formal methods in system design* 19 (2001), 7–34.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*. Springer, 154–169.

[21] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004) (Lecture Notes in Computer Science, Vol. 2988)*, Kurt Jensen and Andreas Podelski (Eds.). Springer, 168–176.

[22] Edmund M Clarke and E Allen Emerson. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on logic of programs*. Springer, 52–71.

[23] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. 2018. *Handbook of model checking*. Vol. 10. Springer.

[24] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

[25] CWE. 2025. CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/. [Accessed: 16-January-2025].

[26] Priyanka Darke, Bharti Chimdyalwar, Sakshi Agrawal, Shrawan Kumar, R Venkatesh, and Supratik Chakraborty. 2023. VeriAbsL: Scalable verification by abstraction and strategy prediction (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 588–593.

[27] Martin Davis. 2013. *Computability and unsolvability*. Courier Corporation.

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[29] Leonardo De Moura, Harald Rueß, and Maria Sorea. 2003. Bounded Model Checking and Induction: From Refutation to Verification: (Extended Abstract, Category A). In *International Conference on Computer Aided Verification*. Springer, 14–26.

[30] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178.

[31] Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2010. Newtonian program analysis. *Journal of the ACM (JACM)* 57, 6 (2010), 1–47.

[32] Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, Austin, TX, USA, 57–64. doi:10.1109/FMCAD.2015.7542253

[33] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 191–202.

[34] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. 2014. ICE: A robust framework for learning invariants. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 69–87.

[35] Patrice Godefroid and Daniel Luchaup. 2011. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 23–33.

[36] Christopher Healy, Mikael Sjodin, Viresh Rustagi, and David Whalley. 1998. Bounding loop iterations for timing analysis. In *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No. 98TB100245)*. IEEE, 12–21.

[37] Thomas A Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 58–70.

[38] Israel N Herstein. 1991. *Topics in algebra*. John Wiley & Sons.

[39] Temesghen Kahsai and Cesare Tinelli. 2011. PKind: A parallel k-induction based model checker. *arXiv preprint arXiv:1111.0372* (2011).

[40] Timotej Kapus, Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Cristian Cadar. 2019. Computing summaries of string loops in C for better testing and refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 874–888.

[41] Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.

[42] Dexter Kozen and Wei-Lung Dustin Tseng. 2008. *The Böhm–Jacopini Theorem Is False, Propositionally*. Lecture Notes in Computer Science, Vol. 5133. Springer Berlin Heidelberg, Berlin, Heidelberg, 177–192. doi:10.1007/978-3-540-70594-9_11

[43] Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M Wintersteiger. 2008. Loop summarization using abstract transformers. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 111–125.

[44] Jiaying Li, Jun Sun, Li Li, Quang Loc Le, and Shang-Wei Lin. 2017. Automatic loop-invariant generation anc refinement through selective sampling. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 782–792.

[45] Tianhai Liu, Mateus Araújo, Marcelo d'Amorim, and Mana Taghdiri. 2014. A comparative study of incremental constraint solving approaches in symbolic execution. In *Haifa Verification Conference*. Springer, 284–299.

[46] Kenneth L McMillan. 2003. Interpolation and SAT-based model checking. In *Computer Aided Verification: 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003. Proceedings 15*. Springer, 1–13.

[47] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*. Springer, 123–136.

[48] Rafael Menezes, Mohannad Aldughaim, Bruno Farias, Xianzhiyu Li, Edoardo Manino, Fedor Shmarov, Kunjian Song, Franz Brauße, Mikhail R. Gadelha, Norbert Tihanyi, Konstantin Korovin, and Lucas C. Cordeiro. 2024. ESBMC 7.4: Harnessing the Power of Intervals. In *30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24) (Lecture Notes in Computer Science, Vol. 14572)*. Springer, 376–380. doi:10.1007/978-3-031-57256-2_24

[49] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj

Singh, et al. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (2017), e103.

[50] David P Monniaux. 2009. Automatic modular abstractions for linear constraints. *ACM SIGPLAN Notices* 44, 1 (2009), 140–151.

[51] Markus Müller-Olm and Helmut Seidl. 2004. Precise interprocedural analysis through linear algebra. *ACM SIGPLAN Notices* 39, 1 (2004), 330–341.

[52] Van Chan Ngo, Quentin Carbonneaux, and Jan Hoffmann. 2018. Bounded expectations: resource analysis for probabilistic programs. *ACM SIGPLAN Notices* 53, 4 (2018), 496–512.

[53] Alexandru Nicolau. 1985. *Loop quantization: unwinding for fine-grain parallelism exploitation.* Technical Report. Cornell University.

[54] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can Large Language Models Reason about Program Invariants?. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 27496–27520. https://proceedings.mlr.press/v202/pei23a.html

[55] Jean-Pierre Queille and Joseph Sifakis. 1982. Specification and verification of concurrent systems in CESAR. In *International Symposium on programming*. Springer, 337–351.

[56] Wolfram Research. 2025. RSolve. https://reference.wolfram.com/language/ref/RSolve.html. [Accessed: 16-January-2025].

[57] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366.

[58] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. 2009. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 225–236.

[59] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2inv: A deep learning framework for program verification. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 151–164.

[60] Jan Strejček and Marek Trtík. 2012. Abstracting path conditions. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 155–165.

[61] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[62] Robert Tarjan. 1974. Finding dominators in directed graphs. *SIAM J. Comput.* 3, 1 (1974), 62–89.

[63] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.

[64] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 61–72.

[65] Xiaofei Xie, Bihuan Chen, Liang Zou, Yang Liu, Wei Le, and Xiaohong Li. 2017. Automatic loop summarization via path dependency analysis. *IEEE Transactions on Software Engineering* 45, 6 (2017), 537–557.

[66] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic summarization for multipath string loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 188–198.