Loopster: Static Loop Termination Analysis

Xiaofei Xie[*]
Tianjin Key Laboratory of Advanced Networking
Tianjin University, China
xiexiaofei@tju.edu.cn

Bihuan Chen
Nanyang Technological University
Singapore
bhchen@ntu.edu.sg

Liang Zou
Nanyang Technological University
Singapore
zouliang@ntu.edu.sg

Shang-Wei Lin[†]
Nanyang Technological University
Singapore
shang-wei.lin@ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore
yangliu@ntu.edu.sg

Xiaohong Li[‡]
Tianjin Key Laboratory of Advanced Networking
Tianjin University, China
xiaohongli@tju.edu.cn

ABSTRACT

Loop termination is an important problem for proving the correctness of a system and ensuring that the system always reacts. Existing loop termination analysis techniques mainly depend on the synthesis of ranking functions, which is often expensive. In this paper, we present a novel approach, named Loopster, which performs an efficient static analysis to decide the termination for loops based on path termination analysis and path dependency reasoning. Loopster adopts a divide-and-conquer approach: (1) we extract individual paths from a target multi-path loop and analyze the termination of each path, (2) analyze the dependencies between each two paths, and then (3) determine the overall termination of the target loop based on the relations among paths. We evaluate Loopster by applying it on the loop termination competition benchmark and three real-world projects. The results show that Loopster is effective in a majority of loops with better accuracy and 20×+ performance improvement compared to the state-of-the-art tools.

CCS CONCEPTS

• **Theory of computation → Program analysis**; • **Software and its engineering → Automated static analysis**;

KEYWORDS

Loop Termination, Reachability, Path Dependency Automaton

ACM Reference format:
Xiaofei Xie, Bihuan Chen, Liang Zou, Shang-Wei Lin, Yang Liu, and Xiaohong Li. 2017. Loopster: Static Loop Termination Analysis. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 04-08, 2017 (ESEC/FSE'17),* 11 pages.
https://doi.org/10.1145/3106237.3106260

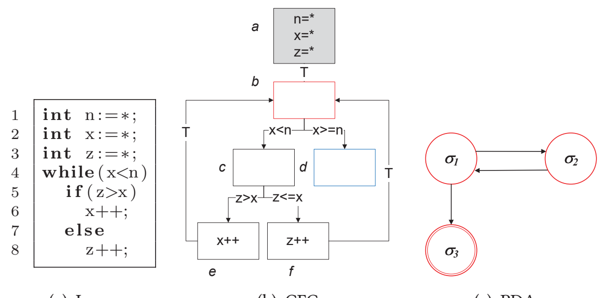
[*]Also with Nanyang Technological University, Singapore.
[†]Shang-Wei Lin and Yang Liu have equal contribution in this work.
[‡]Xiaohong Li is the corresponding author, School of Computer Science and Technology, Tianjin University.

1 INTRODUCTION

Program termination analysis is an important program analysis task to guarantee the correctness and reliability of systems. Nontermination bugs can cause performance problems or denial-of-service attacks [14]. Furthermore, termination proving techniques [17, 21] can be used to prove liveness properties. Loops, as the basic program structure, are the key challenge in program termination analysis.

Program termination analysis has received considerable attention, and a lot of advances [7, 9–11, 15, 20, 22, 28, 32, 40, 41, 43] have been made. The general approach is to synthesize termination arguments, which requires solving two problems [12]: the search for ranking functions and the validity of the ranking functions. However, termination proving has been proved to be undecidable [21], and it is not always possible to find suitable ranking functions. Thus, the existing techniques can only handle certain restricted types of programs. For example, the techniques in [9, 10, 43] are complete only for linear arithmetic loops. Besides, the search for ranking functions can be very expensive, especially for complex lexicographic ranking functions. For example, the techniques in [11, 28] may not terminate when proving the termination of some programs. In principle, enumeration can provide a complete method but is not practical [11]. Moreover, the validity of ranking functions usually depends on a safety checker to search for invariants on demand, which is known to be the bottleneck of ranking function-based techniques [32]. Instead of finding non-trivial lexicographic ranking functions, several techniques [20, 22, 32] attempt to find simple termination arguments based on Ramsey's theorem. However, these techniques still have to make trade-offs between the time overhead for the search and validity of ranking functions.

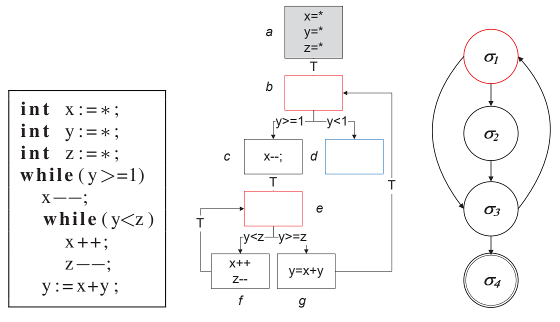
Therefore, the main limitation of ranking function-based techniques is the substantial time overhead for searching and validating ranking functions. From our study on loops, there are many loops whose termination can be quickly determined by analyzing the termination in each path and the dependencies between the paths. In this paper, we propose Loopster, a relatively lightweight static analysis-based approach to proving termination and non-termination for such loops, which does not depend on ranking



```
int  n:=*;
int  x:=*;
int  z:=*;
while(x<n)
  if(z>x)
     x++;
  else
     z++;
```

(a) Loop (b) CFG (c) PDA$_x$

Figure 1: Unnested Loop from [26] with CFG and PDA$_x$



```
int  x:=*;
int  y:=*;
int  z:=*;
while(y>=1)
  x--;
   while(y<z)
     x++;
     z--;
  y:=x+y;
```

(a) Loop (b) CFG (c) PDA$_x$

Figure 2: Nested Loop from [34] with CFG and PDA$_x$

functions. Our key idea is to use a divide-and-conquer approach at the path level to: 1) extract paths in a target loop and analyze the termination of each path, 2) analyze the dependencies between each pair of paths, and 3) determine the overall termination of the target loop based on the dependencies between paths.

Technically, to formally analyze multiple paths in a loop, we first extend the path dependency automaton (PDA$_x$) in our prior work [47] to capture the dependencies among the paths (Section 2.2). Based on the PDA$_x$, Loopster performs the termination analysis for a multi-path loop in three steps (Section 3). In the first step, it uses exit conditions as the slicing criteria to perform a program slicing on the target loop and obtains the control flow graph (CFG) of the loop. In the second step, it first extracts paths and analyzes the termination of each path by a static monotonicity analysis [39] or temporal-logic proving [18, 31]. Analyzing monotonicity of a certain function in each path is usually easier than inferring a ranking function in a multi-path loop. A path can be *terminating* or *nonterminating*. Then, it constructs the PDA$_x$ of the loop to capture the path dependencies. If there is any path that cannot be determined as terminating or nonterminating in the second step, a context-sensitive depth-first search on the PDA$_x$ is performed to refine the *unknown* paths in the last step. During the traversal, if one path marked as *nonterminating* is reachable, Loopster finds a nonterminating trace. If all the paths along the trace are marked as *terminating* and the traversal ends with an accepting state, Loopster concludes that the trace terminates. If there exists one path that is marked as *unknown* and cannot be refined, Loopster returns *unknown*.

We implemented Loopster and evaluated its effectiveness and performance (Section 4) by applying it on the loops from the termination analysis benchmark [1] and three open-source projects. For the 101 loop programs taken from the benchmark, Loopster can correctly handle 93 (92.08%) of the programs with only 7.76 seconds, while the best state-of-the-art tool UAutomizer [29] in our experiments can correctly handle 92 of them with 2246 seconds. For the total 6820 loops from real-world projects, Loopster can handle 2655 (39%) with 91 seconds. The results show that Loopster is scalable and effective, and the biggest advantage is that our static analysis can make a dramatic performance improvement (20×+) for most loops.

In summary, the contributions of our paper are threefold:

(1) We extend the path dependency automaton (PDA$_x$) to i) support nested loops and ii) compute transitions for the paths with non-induction variables.
(2) Based on the PDA$_x$, we propose an efficient static analysis to i) prove termination of multi-path loops and ii) detect nontermination through finding nonterminating state with reachability analysis on the PDA$_x$, and prove the soundness of our approach.
(3) We implement our approach in Loopster, and conducted an evaluation to demonstrate the effectiveness and scalability of Loopster on benchmarks and real-world projects.

2 LOOP MODELING

In this section, we define the scope of this work and extend the path dependency automaton (PDA$_x$).

2.1 Scope of the Work

We focus on multi-path loops in which the variables are over integers and the operations are standard integer operations (addition, subtraction, multiplication, and division). Let D be a finite integer domain and $X = \{x_1, x_2, \ldots, x_n\}$ be a finite set of variables ranging over D. An atomic predicate over X is of the form $f(x_1, x_2, \ldots, x_n) \sim b$, where $f : D^n \mapsto D$ is a function that represents the standard integer operations on X, $\sim \in \{=, <, \leq, >, \geq\}$, and $b \in D$. A predicate is a Boolean combination of atomic predicates over X. P_X is to denote the set of all possible predicates over X.

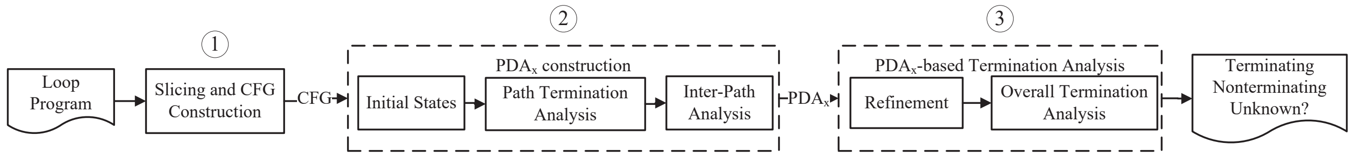
Loops containing data structures and function calls are not supported in this work. However, we can perform a program slicing if they do not affect the termination of the loop. The possible extensions for some unsupported loops is discussed in Section 4.4.

2.2 Path Dependency Automaton

The loop we considered can be modeled by a control flow graph (CFG), as formulated in Definition 2.1.

Definition 2.1. A control flow graph (CFG) of a loop is a tuple $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$, where (1) L is a set of basic blocks, each of which contains a sequence of assignment instructions. (2) $E \subseteq L \times P_X \times L$ is a set of directed edges connecting the basic blocks. (3) $l_{pre} \in L$ is the pre-header after which the entry block of the loop will execute. (4) $L_h \subset L$ is a set of header blocks. Given two blocks $l, l' \in L$, we say l dominates l' (l *dom* l'), if every path from l_{pre} to l' passes through l. $L_h = \{l \in L \mid \exists(l', p, l) \in E \wedge l \; dom \; l'\}$. (5) $L_e = \{l \in L \mid \forall l' \in L, p \in P_X, (l, p, l') \notin E\}$ is a set of exit blocks.

For simplicity, we use $l_1 \xrightarrow{p} l_2$ to denote an edge $(l_1, p, l_2) \in E$, which means that in the basic block l_1, if the condition p holds,



①
②
③
Loop Program
Slicing and CFG Construction
CFG
PDA$_x$ construction
Initial States
Path Termination Analysis
Inter-Path Analysis
PDA$_x$
PDA$_x$-based Termination Analysis
Refinement
Overall Termination Analysis
Terminating Nonterminating Unknown?

Figure 3: Flow Diagram of Loopster

then the (loop) program goes to the basic block l_2. In this work, we consider the weakest precondition operator *wp* and the strongest postcondition operator *sp*, based on Hoare Logic [30].

Definition 2.2. Given a CFG $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$, a path (denoted as σ) is a sequence $l_0 \xrightarrow{p_0} l_1 \xrightarrow{p_1} \dots \xrightarrow{p_{k-1}} l_k$, where $l_0 \in L_h$ is the head of σ, denoted by $head(\sigma)$; $l_k \in (L_h \cup L_e)$ is the tail of σ, denoted by $tail(\sigma)$; $\forall 1 \le i < k, l_i \notin (L_h \cup L_e)$. The weakest triggering condition of path σ (denoted as θ_σ) is computed as $wp(l_0, p_0) \wedge wp(l_0; l_1, p_1) \wedge \dots \wedge wp(l_0; \dots; l_{k-1}, p_{k-1})$, where the l_i in wp represents the instructions in the basic block.

For simplicity, we will use *path condition* to represent the weakest triggering condition of a path in the following sections. For example, the path condition for $while(y > 0)\{x{+}{+}; if(x < 0)z{+}{+};\}$ is $y > 0 \wedge x < -1$ rather than $y > 0 \wedge x < 0$.

The path σ is *feasible* only when θ_σ is satisfiable. Otherwise, it is an *infeasible* path. We use $\iota(\sigma)$ to denote a sequence of assignment instructions in σ, and σ^k to denote the consecutive repetition of path σ with k times. $\iota(\sigma^k)$ is k repetition of $\iota(\sigma)$ (i.e., $\iota(\sigma); \dots; \iota(\sigma)$). For example, if $\iota(\sigma)$ is $x{+}{+}$, then $\iota(\sigma^2)$ is $x{+}{+}; x{+}{+}$.

The precondition of the loop is a predicate $Pre(\mathcal{G})$ which constrains the possible valuations for the variables before executing the loop $\mathcal{G}$. We assume that the loop precondition is known. We use $Pre(\sigma)$ and $Pos(\sigma^k)$ to denote the precondition of the path σ and the postcondition after k executions of σ, respectively. $Pre(\sigma)$ and $Pos(\sigma^k)$ are predicates which constrain the possible valuations for the variables before executing the path σ and after executing the path σ for k times. Given a path with its precondition $Pre(\sigma)$, we can infer its postcondition $Pos(\sigma^k)$ as $sp(\iota(\sigma^k), Pre(\sigma))$, which gives the strongest condition after executing σ^k w.r.t. $Pre(\sigma)$.

Example 2.3. Fig. 1(b) gives the CFG of the unnested loop in Fig. 1(a), where there are six basic blocks named from a to f and seven edges. The basic block a is the pre-header from which the program enters the loop. b is the header block and d is the exit block. There is one predicate on each edge. For example, $b \xrightarrow{x>n} c$ is feasible if the predicate $x > n$ holds. T represents `true` which means the predicate always holds. There are three paths in the CFG: $\sigma_1 = b \xrightarrow{x<n} c \xrightarrow{z>x} e \xrightarrow{true} b, \sigma_2 = b \xrightarrow{x<n} c \xrightarrow{z<=x} f \xrightarrow{true} b, \sigma_3 = b \xrightarrow{x>=n} d$. For the path σ_1, its path condition is $\theta_{\sigma_1} = x < n \wedge z > x$. The loop precondition is `true` since x, n, z can be any value before the loop. Suppose $Pre(\sigma_1) = x > 0$, then we can infer $Pos(\sigma_1) = sp(\text{x++}, x > 0) = x > 1$ and $Pos({\sigma_1}^2) = sp(\text{x++;x++}, x > 0) = x > 2$.

Different from Fig. 1(b), there are two header blocks (b and e) in Fig. 2(b) since the loops are nested. In Fig. 2(b), there are four paths: $\sigma_1 = b \xrightarrow{y>=1} c \xrightarrow{true} e, \sigma_2 = e \xrightarrow{y<z} f \xrightarrow{true} e, \sigma_3 = e \xrightarrow{y>=z} g \xrightarrow{true} b, \sigma_4 = b \xrightarrow{y<1} d$.

Intuitively, one loop execution contains multiple iterations, which are the interleaving of the feasible paths in the CFG. To model the dependencies between different paths in a loop, we propose the path dependency automaton, as formulated in Definition 2.4.

Definition 2.4. Given a loop with the CFG $\mathcal{G} = (L, E, l_{pre}, L_h, L_e)$, a *path dependency automaton* (PDA$_x$) is $\mathcal{A} = (S, I, T, F)$ where:

- $S = \{\sigma \mid \sigma \text{ is a feasible path in } \mathcal{G}\}$ is a set of states.
- $I = \{\sigma \in S \mid ((l_{pre}, true, head(\sigma) \in E) \wedge Pre(\mathcal{G}) \wedge \theta_\sigma \text{ is satisfiable}\}$ is a set of initial states.
- $T = \{(\sigma, \sigma') \in S \times S \mid (tail(\sigma) = head(\sigma')) \wedge (\sigma \neq \sigma') \wedge (\exists i\colon \theta_\sigma \wedge (\bigwedge_{1 \le k \le i} sp(\iota(\sigma^k), \theta_\sigma)) \wedge \theta_{\sigma'} \text{ is satisfiable})\}$ is a set of transitions.
- $F = \{\sigma \in S \mid \forall \sigma' \in S\colon (\sigma, \sigma') \notin T\}$ is a set of accepting states.

Intuitively, a state in $\mathcal{A}$ corresponds to a path in $\mathcal{G}$. A state is an initial state if its corresponding path can be the first iteration under the loop precondition. The transition $(\sigma, \sigma') \in T$ represents that σ' can be executed (i.e., $\theta_{\sigma'}$ is satisfiable) after some (i) repetitions of σ. A state is an accepting state if it has no successors. A run of the PDA$_x$, denoted as τ, is a sequence of states $\tau = (\sigma_1, \sigma_2, \dots)$ where $\sigma_1 \in I$ and $\forall i \ge 1\colon (\sigma_i, \sigma_{i+1}) \in T$. The sequence can be infinite. The semantics of τ can be represented by the loop execution which is the interleaving of the paths. With different loop preconditions, a PDA$_x$ has different runs. We use $R_\mathcal{A}$ to represent the set of all runs of $\mathcal{A}$ under the loop precondition $Pre(\mathcal{G})$. The construction of T will be described in Section 3.2.

If $tail(\sigma)$ is an exit block, then path σ (called exit path) will end the loop. Hence, if each run of the loop can end with an exit path, the loop will terminate. If there is any run which is infinite, the loop is nonterminating (e.g., $while(x < 11) if(x < 10)x{+}{+}; else\ x{-}{-};$). Or if a run is finite but the last path is not an exit path, the loop enters one *stuck* state which has no successors and the loop does not terminate (e.g., $while(x > 1)x{+}{+};$).

Example 2.5. Fig. 1(c) shows the PDA$_x$ of the loop in Fig. 1(a), where $S = \{\sigma_1, \sigma_2, \sigma_3\}, I = \{\sigma_1, \sigma_2, \sigma_3\}, F = \{\sigma_3\}$ and $T = \{(\sigma_1, \sigma_2), (\sigma_2, \sigma_1), (\sigma_1, \sigma_3)\}$. The states σ_1, σ_2 and σ_3 represent the paths in the CFG. The precondition of the loop is *true*. σ_1, σ_2 and σ_3 can be initial states as they can be firstly executed under the precondition. For example, by Definition 2.4, we check whether σ_1 can be initial state by solving the condition $true \wedge x < n \wedge z > x$. The condition is satisfiable, hence σ_1 can be an initial state. There is a transition from σ_1 to σ_2 because the path condition θ_{σ_2} can be satisfiable after some execution of σ_1. The details about computing the transition in this example will be described in Example 3.2. There are four patterns which represent all possible runs of the PDA$_x$: $\tau_1 = (\sigma_3), \tau_2 = (\sigma_1, \sigma_3), \tau_3 = ((\sigma_1, \sigma_2)^+, \sigma_1, \sigma_3), \tau_4 = ((\sigma_2, \sigma_1)^+, \sigma_3)$. Fig. 2(c) is the PDA$_x$ of the nested loop in Fig.2(a).

3 LOOP TERMINATION ANALYSIS

We propose a static method for loop termination analysis. The key idea is to adopt the divide-and-conquer strategy: we analyze the termination of each path in the target loop, and determine the overall termination of the target loop based on the dependencies among the paths. Fig. 3 shows the overall flow of our approach. Given a loop as the input, our approach determines whether the loop terminates through the following three steps:

Step 1. We use the program slicing technique [38] to remove irrelevant statements from the loop program. Based on the sliced loop, we construct its CFG to extract paths in the loop. The details of slicing are omitted and can be found in [38].

Step 2. We construct the PDA_x w.r.t. the control flow graph generated in Step 2. We first identify the initial states, then perform the termination analysis of each path based on monotonicity analysis or prover (see Section 3.1). At last, the dependency analysis among paths (see Section 3.2) is performed.

Step 3. With the PDA_x constructed, we perform a reachability analysis on PDA_x to determine the overall termination of the target loop (c.f. Section 3.3). If the information is not sufficient to have a conclusive result, we perform a refinement step to obtain more information to determine the overall termination (c.f. Section 3.4).

3.1 Path Termination Analysis

A path is nonterminating if it can be executed infinitely before executing other paths; otherwise, it is terminating. To determine whether one path is terminating or not, we adopt a sufficient condition checking, i.e., we compute the sufficient conditions of nontermination and termination for each path, respectively. For a path σ, if the sufficient condition of nontermination is `true` (i.e., it does not terminate in any case), then we can conclude that it does not terminate. Similarly, if the sufficient condition of termination is `true`, then we can conclude that it terminates. Otherwise, we cannot conclude anything from the sufficient conditions. Based on the three cases above, we can identify each path as *terminating* (T), *nonterminating* (NT), or *unknown* (UN).

Obviously, a path is always terminating if its head and tail nodes are different. Thus, its sufficient terminating condition is `true` and nonterminating condition is `false`. In the following, we explain how to compute the nonterminating sufficient condition ϕ and the terminating sufficient condition ψ for the path whose head node and tail node are the same basic block. Consider a path σ with path condition $\theta_\sigma = p'_1 \wedge p'_2 \wedge \cdots \wedge p'_k$ where p'_i is a predicate of the form $f_i(x_1, x_2, \ldots, x_n) \sim b_i$ for $1 \leq i \leq k$. To simplify the notation, we use E_i to denote the function $f_i(x_1, x_2, \ldots, x_n)$. For each predicate p'_i, we define its sufficient conditions ϕ_i and ψ_i using the following monotonicity analysis:

(1) For a predicate of the form $E_i > b_i$ or $E_i \geq b_i$, its nonterminating sufficient condition ϕ_i should make E_i increase monotonically or not change in each iteration; the terminating sufficient condition ψ_i should make E_i decrease monotonically in each iteration.

(2) For the predicate of the form $E_i < b_i$ and $E_i \leq b_i$, its nonterminating sufficient condition ϕ_i should make E_i decrease monotonically or not change in each iteration; the terminating sufficient condition ψ_i should make E_i increase monotonically.

```
int x=*;
while(x>0)
  x++;
```

(a) NT

```
int x=*,y=*;
while(x<0 && y>0)
  x=x+y; y++;
```

(b) T

```
int x=*, a=*;
while(x>0)
  x=x+a;
```

(c) UN

Figure 4: Path Examples

(3) For the predicate of the form $E_i = b_i$, its nonterminating sufficient condition ϕ_i should make E_i keep the same value in each iteration; the terminating sufficient condition ψ_i should make E_i change the value in some iterations.

With the sufficient conditions for each predicate p_i, the overall sufficient conditions for the path σ are defined as:

$$\phi = \bigwedge_{1 \leq i \leq k} \phi_i \quad \text{and} \quad \psi = \bigvee_{1 \leq i \leq k} \psi_i$$

If ϕ is `true`, we conclude that σ does not terminate. If ψ is `true`, we conclude that σ terminates. Otherwise, we mark σ as unknown.

To compute each ϕ_i and ψ_i mentioned above, we need to determine the monotonicity of E_i. The monotonicity of one variable can be analyzed by the static technique in [39], which supports three types: basic monotonic statements, dependent monotonic statements, and cyclically monotonic statements. We extend it in two aspects: 1) check the monotonicity for a basic expression such as addition and subtraction on multiple variables. For example, when x is increasing and y is decreasing, then $x - y$ is increasing, $y - x$ is decreasing and $x + y$ is unknown. 2) a backward reasoning to compute the sufficient conditions which can make the variable or expression increase monotonically, decrease monotonically or be constant (i.e., does not change). For example, in the basic monotonic statement $x = x + a$, the sufficient condition, which makes x monotonic increasing, is $a > 0$, monotonic decreasing is $a < 0$ and constant is $a = 0$. In the dependent monotonic statements $x = x + y$; $y = y + a$, the sufficient condition, which makes x strictly monotonic increasing, is $y_0 > 0 \wedge a \geq 0$, where y_0 is the initial value for variable y before executing the path.

For those cases in which the monotonicity cannot be detected by [39] or the sufficient condition cannot be computed by our extension, we leverage temporal-logic proving techniques [18, 31]. We generate one dummy loop which only contains one path σ and prove the CTL property $AG[\theta_\sigma]$ in the loop. If the result is `true`, it means that the path condition θ_σ always holds during the execution of σ, i.e., the path σ does not terminate. If the result is `false`, we check the CTL property $AF[\neg\theta_\sigma]$. If the result of the second property is `true`, it means that the path condition θ_σ will not hold eventually, i.e., the path σ terminates. If the results of both properties are `false` or the CTL checking is unable to produce a conclusive result, we mark the path σ as unknown and set both ϕ and ψ to be `false` since we cannot compute the nonterminating and terminating sufficient conditions.

LEMMA 1. *If an accepting state $\sigma \in F$ is marked as terminating, then $tail(\sigma)$ must be an exit block.*

Example 3.1. In Fig. 4(a), there is a loop consisting of only one path, whose path condition is $x > 0$. We then compute the condition (as the nonterminating sufficient condition) which can make x increasing monotonically. Obviously, x is always increasing and thus ϕ is `true`. Hence the path is marked as NT. In Fig. 4(b), we know x and y are increasing monotonically, where y can be checked by *basic monotonic statements* and x can be checked by *dependent monotonic statements*. Thus, ϕ is `false` and ψ is `true`. Hence, the path is marked as T. For the loop in Fig. 4(c), our extended strategy can compute the sufficient nonterminating condition $a \geq 0$, under which x will increase monotonically or does not change, and the sufficient terminating condition $a < 0$, under which x will decrease monotonically. The termination of the path depends on the input of a. Thus we mark the path as UN.

3.2 Inter-Path Analysis

After analyzing whether each path is terminating, we need to further analyze the dependency between each two paths, i.e., whether a path is able to transit to another. Algorithm 1 shows how to construct the transitions in a PDA$_x$. The input is the CFG $\mathcal{G}$ of the loop and we use $\prod_{\mathcal{G}}$ to represent the set of all feasible paths in $\mathcal{G}$. Note that σ_i can only transit to the path σ_j whose head node is equal to the tail of σ_i (i.e., $tail(\sigma_i) = head(\sigma_j)$ at Line 2). A variable is an induction variable (IV) when it is changed by a constant value or assigned by constant values in each iteration of the path. A condition $e \sim 0$ is an IV condition [47] if the expression e (regarded as a variable) is an IV. If all variables in σ_i are IVs and all conditions in θ_{σ_i} and θ_{σ_j} are IV conditions, we introduce k_{ij} to represent the number of iterations of σ_i before transiting to σ_j. With the variable k_{ij}, we can calculate the effect after some executions of σ_i for all the IVs. For example, if $x = x + 1$ in each iteration of σ_i, then we can calculate $x = x + k$ after k executions of σ_i. We compute the value of IVs after $k_{ij} - 1$ and k_{ij} execution of σ_i, denoted by $X_{k_{ij}-1}$ and $X_{k_{ij}}$, respectively (Line 5–6). $\theta[X'/X]$ represents that the variables X in θ are substituted with X'. At Line 7, we compute the weakest precondition ω_{ij} for triggering the transition (σ_i, σ_j). $head(\sigma_i) = tail(\sigma_i)?k_{ij} \geq 1 : k_{ij} = 1$ means if the head node and tail node are the same node, then σ_i can be executed more than one time before executing σ_j. Otherwise, it can only execute one time. If ω_{ij} is satisfiable, then σ_j can be executed after k_{ij} iterations of σ_i (Line 8) and we add the transition to T (Line 9). Note that the introduced variable k_{ij} can be bounded in the predicate ω_{ij}.

If there are some non-IVs or non-IV conditions, we cannot compute the transition with k_{ij}. We use the temporal-logic prover to check whether the transition is feasible (Line 10–13). We create one dummy loop L which only contains the path σ_i and check the CTL property $EF[\theta_{\sigma_j}]$ during the loop execution (Line 12). If the property is satisfied, σ_i may transit to σ_j in some cases. If it is not satisfied, there is no transition from σ_i to σ_j.

Example 3.2. In Fig. 1, all variables are IVs and all the conditions are IV conditions. For example, $x < n$ is an IV condition because $x - n$ is changed by a constant. We can use the first step to compute the transition. To check whether σ_1 can transit to σ_2, we compute:

$$X_{k_{12}-1} : (x = x + k_{12} - 1) \wedge (n = n) \wedge (z = z)$$
$$X_{k_{12}} : (x = x + k_{12}) \wedge (n = n) \wedge (z = z)$$
$$\omega_{12} = (x < n) \wedge (z > x) \wedge (x + k_{12} - 1 < n) \wedge (z > x + k_{12} - 1) \wedge (x + k_{12} < n) \wedge (x + k_{12} \geq z) \wedge (k_{12} \geq 1)$$

Hence, there is a transition from σ_1 to σ_2 because ω_{12} is satisfiable. k_{12} is bounded by $k_{12} \geq 1 \wedge z > x + k_{12} - 1 \wedge x + k_{12} \geq z$.

Let us see how the second step works. In Fig. 4(b), to check whether there is a transition from the state σ_1 (θ_{σ_1} is $x < 0 \wedge y > 0$) to the accepting state σ_2 (θ_{σ_2} is $x \geq 0 \vee y \leq 0$), we cannot use the first step as the value change of x is not constant. In this case, we check $EF[\theta_{\sigma_2}]$, and it is true. Thus we know σ_1 can transit to σ_2.

Algorithm 1: `ComputeTran`($\mathcal{G}$)

```
input : $\mathcal{G}$: CFG
1  Let $\prod_{\mathcal{G}}$ be the set of all feasible paths in $\mathcal{G}$;
2  foreach $(\sigma_i, \sigma_j) \in \{(\sigma_m, \sigma_n) \mid \sigma_m \in \prod_{\mathcal{G}} \wedge \sigma_n \in \prod_{\mathcal{G}} \wedge tail(\sigma_m) = head(\sigma_n) \wedge m \neq n\}$ do
3      if All variables $X$ in $\sigma_i$ are IVs, and all conditions in $\theta_{\sigma_i}$ and $\theta_{\sigma_j}$ are IV conditions then
4          Let $k_{ij}$ represent the iteration count of $\sigma_i$ ;
5          $X_{k_{ij}-1} := F(X, k_{ij} - 1)$ computes the value after $k_{ij}$-1 iterations of $\sigma_i$;
6          $X_{k_{ij}} := F(X, k_{ij})$ computes the value after $k_{ij}$ iterations of $\sigma_i$;
7          $\omega_{ij} := \theta_{\sigma_i} \wedge \theta_{\sigma_i}[X_{k_{ij}-1}/X] \wedge \theta_{\sigma_j}[X_{k_{ij}}/X] \wedge (head(\sigma_i) = tail(\sigma_i)?k_{ij} \geq 1 : k_{ij} = 1)$;
8          if $\omega_{ij}$ is satisfiable then
9              $T := T \bigcup \{(\sigma_i, \sigma_j)\}$;
10     else
11         Create loop $L$ which only contains path $\sigma_i$;
12         if $EF[\theta_{\sigma_j}]$ holds for $L$ then
13             $T := T \bigcup \{(\sigma_i, \sigma_j)\}$;
```

3.3 Loop Termination Analysis

With the PDA$_x$ constructed, we can determine the overall termination of the target loop. Theorem 1 and Theorem 2 below give the sufficient conditions on a PDA$_x$ for its corresponding target loop being terminating or nonterminating, respectively. Based on the two theorems, a depth-first search can be used to check whether the target loop terminates. If there are some unknown states or cycles in the PDA$_x$, we will perform a refinement (Section 3.4) or reduce cycles from the PDA$_x$ (Section 3.5).

THEOREM 1. *If a PDA$_x$ satisfies the two conditions below, its corresponding loop terminates: 1) it is acyclic, and 2) its states (reachable from initial states) are all marked as terminating.*

THEOREM 2. *If a PDA$_x$ has a reachable state which is marked as nonterminating, its corresponding loop does not terminate.*

3.4 Refinement

So far, our approach is presented in a way such that each path is analyzed independently, i.e., the effects from other paths are not considered. For example, a path σ, if analyzed alone, may be marked as unknown. However, it could be actually terminating or nonterminating under the precondition $Pre(\sigma)$.

To be more accurate for determining the termination of each path, we need to refine the states that are marked as unknown according to the execution of the loop. Algorithm 2 gives a DFS-based algorithm to refine the states that are marked as unknown in an acyclic PDA$_x$. Given a state σ and its precondition $Pre(\sigma)$, Algorithm 2 refines the nonterminating or terminating sufficient conditions of the state σ and visits its successor states in a DFS-style. *issp* is a boolean value which denotes whether $Pre(\sigma)$ is a strongest postcondition of the previous state.

Algorithm 2: RefineState($\mathcal{A}, \sigma, Pre(\sigma), issp$)

```
input  : $\mathcal{A}$: PDA$_x$; $\sigma$: a state of $\mathcal{A}$;
         $Pre(\sigma)$: the precondition of $\sigma$;
         $issp$: boolean value;
output : A refined PDA$_x$
1  if $Pre(\sigma) \wedge \theta_\sigma$ is satisfiable then
2      if $\sigma$ is marked as UN then
3          if $issp = true \wedge Pre(\sigma) \implies \phi_\sigma$ then
4              mark $\sigma$ as NT;
5              return;
6          if $Pre(\sigma) \implies \psi_\sigma$ then
7              $\sigma$.list.append(T);
8  Let $\Sigma$ be the set of successors of $\sigma$ ;
9  foreach $\sigma' \in \Sigma$ do
10     if All variables X in $\sigma$ are IVs and all conditions in $\theta_\sigma$ and $\theta_{\sigma'}$ are
       IV conditions then
11         $Pre(\sigma') := sp(X := F(X, k_{ij}), Pre(\sigma) \wedge \omega_{(\sigma,\sigma')})$;
12     else
13         $Pre(\sigma') := \theta_{\sigma'}$;
14         $issp := false$;
15     RefineState($\mathcal{A}$, $\sigma'$, $Pre(\sigma')$, $issp$);
```

Given a state σ and its precondition $Pre(\sigma)$, we only refine the termination status of σ in the case where (1) σ is reachable under its precondition, i.e., $Pre(\sigma) \wedge \theta_\sigma$ is satisfiable (Line 1), and (2) σ is marked as unknown (Line 2). Note that if $Pre(\sigma)$ is the strongest postcondition of the traversed path and $Pre(\sigma) \wedge \theta_\sigma$ is satisfiable, then σ must be reachable. The refinement is performed for the following cases:

- If the condition $Pre(\sigma) \implies \phi_\sigma$ holds, it means that the non-terminating sufficient condition of σ is satisfiable under the precondition of σ. Then we can conclude that σ does not terminate (Line 3–5). In this case, we can conclude that the overall loop does not terminate by Theorem 3.4.
- If the condition $Pre(\sigma) \implies \psi_\sigma$ holds, the terminating sufficient condition of σ is satisfiable under the precondition of σ. In this case, we append T, which represents that the state σ is refined as terminating along the current trace, to the list σ.`list` (Line 6–7).

To continue the refinement on every successor of σ, we compute the postcondition after some execution of σ, which is also the precondition of σ'. At Line 11, we infer the strongest postcondition $Pre(\sigma')$, where $Pre(\sigma)$ is from input and $\omega_{(\sigma,\sigma')}$ has been computed at Line 7 of Algorithm 1. $X := F(X, k_{ij})$ is a sequence of assignment instructions and is computed at Line 6 of Algorithm 1. If the conjunction of $Pre(\sigma')$ and $\theta_{\sigma'}$ is satisfiable (i.e., the next traversal at Line 1), then there exists a variable k_{ij} that makes σ' reachable because $Pre(\sigma')$ is the strongest postcondition of the previous state σ. Otherwise, we use a conservative and sound condition, i.e., the path condition of σ', as the postcondition of σ, which also serves as the precondition of σ' (Line 13). $\theta_{\sigma'}$ is not a strongest postcondition and *issp* is assigned $false$ (Line 14).

Algorithm 3 refines the whole PDA$_x$ by invoking Algorithm 2 for all initial states, and returns a set of refined PDA$_x$ based on different initial states. We assume the precondition of the loop is a strongest postcondition and *issp* is assigned $true$. Algorithm 3 aims to check whether a state σ originally marked as unknown could

Algorithm 3: RefinePDA($\mathcal{A}, Pre(\mathcal{A})$)

```
input  : $\mathcal{A} = (S, I, F, T)$: a PDA$_x$;
         $Pre(\mathcal{A})$: the precondition of $\mathcal{A}$
output : A set of refined PDA$_x$
1  $\Omega \longleftarrow \emptyset$ ;
2  foreach $\sigma \in I$ do
3      $\mathcal{A}'$ = RefineState($\mathcal{A}$, $\sigma$, $Pre(\mathcal{A})$, $true$) ;
4      foreach state $\sigma$ in $\mathcal{A}'$ marked as UN do
5          if each element $e \in \sigma$.list is T then
6              mark $\sigma$ as T;
7      $\Omega \longleftarrow \Omega \cup \{\mathcal{A}'\}$;
```

be refined as terminating (Line 4–6). Notice that we can do so only when all the markings in the list σ.`list` are T, which means σ is terminating in all paths. Note that Algorithm 3 assumes that the input PDA$_x$ is acyclic such that the termination of Algorithm 2 is guaranteed, and the cyclic PDA$_x$ will be described in Section 3.5.

To sum up, given a target loop represented by a PDA$_x$ $\mathcal{A}$, if we cannot directly decide its termination by Theorem 1 and Theorem 2, we can apply Algorithm 3 on $\mathcal{A}$ to refine each state that is originally marked as unknown. Let Ω be the set of refined PDA$_x$ obtained by invoking RefinePDA($\mathcal{A}, Pre(\mathcal{A})$). We can further analyze the target loop based on Ω by Corollaries 3.3 and 3.4. If both of them do not hold, we mark the loop as *unknown*.

Corollary 3.3. *If $\mathcal{A}'$ satisfies Theorem 1 for all $\mathcal{A}' \in \Omega$, then the target loop terminates.*

Corollary 3.4. *If $\mathcal{A}'$ satisfies Theorem 2 for some $\mathcal{A}' \in \Omega$, then the target loop does not terminate.*

Notice that the refinement can only be used in acyclic PDA$_x$ currently. In cyclic PDA$_x$, the execution count of the cycle might be unknown. Thus, Algorithm 2 may not terminate for cyclic PDA$_x$; and we leave the refinment of cyclic PDA$_x$ in the future work.

3.5 Cyclic-PDA$_x$ Analysis

Theorem 1 requires the PDA$_x$ to be acyclic. Given a cyclic-PDA$_x$, even though all of its states are marked as terminating, we cannot conclude that the loop terminates since it may have an infinite execution between the states in the cycle. For example, in the loop *while(x<11){if(x<10) x++; else x- -;}*. All the three paths are marked as terminating, however the loop does not terminate since there is an infinite execution in the cycle. To determine the termination of a cyclic-PDA$_x$, we need further analysis on cycles.

We firstly detect the strongly connected components (SCCs) from the PDA$_x$. Notice that we only consider the SCC with more than one state (i.e., the cycle in the PDA$_x$) here. Our main idea of termination analysis on cycles is as follows:

(1) Try to prove the termination by reducing the cyclic-PDA$_x$ to acyclic-PDA$_x$ (c.f. Section 3.5.1).
(2) Try to prove the nontermination by finding one reachable nonterminating state in the SCC (c.f. Section 3.5.2).

3.5.1 *Proving Termination.*

A key observation is that some states will not be executed after some iterations of the SCC, then the SCC can be reduced to a simple structure. The main idea of proving termination is to reduce the cyclic-PDA$_x$ to an acyclic-PDA$_x$ by finding the pivot states which will not be executed eventually. For example, in the cyclic-PDA$_x$ Fig. 1(c), the SCC is $\{\sigma_1, \sigma_2\}$. To determine whether the SCC can terminate, we find the pivot state σ_1 which will not be executed after x increases to be n. Hence, the SCC is reduced (after σ_1 terminates) and becomes acyclic. We formulate the concept as follows.

Inspired by ranking functions, we check whether each state can be a pivot state as formulated in Definition 3.5.

Definition 3.5. Let $\sigma_m \in S$ be a state in $\mathcal{A} = (S, I, F, T)$. The state σ_m is a *pivot* state if we can find a function $f(X)$ such that:

- $\theta\sigma_m \implies f(X) \geq c$, where c is a constant value.
- $f(X)$ will be monotonic decreasing after some execution of σ_m.
- $f(X)$ is decreasing or not changed in any other state $\sigma \in S$.

Intuitively, a pivot state can be executed for a finite number of iterations. Thus, we can break the SCC by removing the transitions which end with a pivot state, as formulated in Definition 3.6.

Definition 3.6. Let $\mathcal{A} = (S, I, F, T)$ be a PDA$_x$ with all states marked as terminating. If $\mathcal{A}$ has an SCC, denoted by C, we say $\mathcal{A}$ is *safe* to be reduced as $\mathcal{A}_{\sigma_m} = (S, I, F, T')$ if we can find a *pivot* state $\sigma_m \in C$, where the transition T' is defined as $T' = T \setminus \{(\sigma_1, \sigma_2) \in T \mid (\sigma_2 = \sigma_m \land \sigma_1 \in C)\}$.

Lemma 2. *Let $\mathcal{A}$ be a cyclic PDA$_x$ such that $\mathcal{A}$ has one SCC and all states of $\mathcal{A}$ are marked as terminating. If $\mathcal{A}$ can be safely reduced by a pivot state σ_m and $\mathcal{A}_{\sigma_m}$ is acyclic, then $\mathcal{A}$ terminates.*

Notice that if the reduced PDA$_x$ is acyclic, then we can conclude that the loop terminates. If the reduced PDA$_x$ is still cyclic, we recursively apply Lemma 2 (if possible) on the reduced PDA$_x$ until the reduced PDA$_x$ is acyclic. Otherwise, we mark the termination of the loop as *unknown*.

Example 3.7. In Fig. 1(c), we can infer that $n - x$ is monotonically decreasing in σ_1, $n - x > 0$, and $n - x$ does not change in σ_2. Hence σ_1 is a pivot state, and it is terminating after removing the transition (σ_2, σ_1). In Fig. 2(c), from $z > y \geq 1$ and z decreases in σ_2, we can first find one pivot state σ_2. After removing (σ_1, σ_2) from the SCC, it is a cyclic PDA$_x$ and the cycle in the new PDA$_x$ is (σ_1, σ_3). In the cycle, from $x - - \land y = y + x$, we know y will decrease eventually. Then we find another pivot state σ_3. Finally, the PDA$_x$ becomes acyclic and we prove it can terminate.

3.5.2 Proving Nontermination.

To prove the nontermination of a cyclic-PDA$_x$, which contains the nonterminating or unknown states, we unroll the SCC for several times to refine the unknown state and check whether any nonterminating state can be reached. To make the checking efficient, we only traverse each cycle once (the number of unrolling can be adjusted). If we cannot find the reachable nonterminating state after a certain number of iterations of the cycle, then we conclude the loop is *unknown*.

4 IMPLEMENTATION AND EVALUATION

We implemented Loopster based on LLVM 3.7 [35] and the SMT solver Z3 [24]. The input is a loop program in C language, which is compiled into LLVM IR. We use the slicing technique [42] to reduce irrelevant paths in the constructed control flow graph.

The goal of our evaluation is to demonstrate (1) the effectiveness and performance improvement of our static approach by comparing with the state-of-the-art tools on the loops in a benchmark (Section 4.1 and 4.2) and (2) the effectiveness and scalability of our static approach on the loops in real-world projects (Section 4.3). Note that we use wall time in the experimental results.

4.1 Experimental Results on Loop Benchmarks

To evaluate the performance of Loopster, we selected 69 difficult loop programs from the *termination-crafted* benchmark and 32 programs with numeric nested loops from the termination category in Competition on Software Verification 2016 (SV-COMP 16) [1]. The *termination-crafted* benchmark consists of 85 non-trivial programs, and 70 of them contain loops. We used the 70 programs in the experiments, where 38 programs are known to be termination and 31 programs are known to be non-termination. One program *Collatz_unknown-termination.c* is known to be *unknown* and we omit this case for avoiding ambiguity. For the 32 nested loops, 27 of them are terminating, and 5 of them are nonterminating. All these programs are designed to be simplified programs which do not need slicing. Hence, slicing does not affect the fairness of the comparison.

We compared Loopster against three termination analysis tools. AProVE and UAutomizer respectively won the first and second prize for the termination category in SV-COMP 16 [1]. T2 is an advanced verification tool, which has much better performance and effectiveness than other tools or configurations (e.g., ARMC [44], Size-change/MCNP [16], and KITTeL [25]), as empirically demonstrated in [12]. Thus, the selected three tools represent the state of the art. Specifically, AProve and UAutomizer we used are the provided versions for SV-COMP 16 and T2 is the version 2016. We ran all the tools with the timeout being 900 seconds.

Table 1 reports the results of these tools on the benchmarks. The second column reports the numbers of terminating (*T*) and nonterminating (*NT*) loop programs in unnested and nested benchmarks; *CR* represents the number of programs that can be correctly analyzed, *CT* represents the time overhead for the correctly analyzed programs, and *TT* represents the time overhead for all the programs (including those that time out). For Loopster, *CTL* represents the number of programs which used the temporal-logic prover.

From Table 1, we can see that Loopster can correctly handle 93 (92.08%) programs (including 61 (92.85%) terminating programs and 32 (88.89%) nonterminating programs) in 7.76 seconds. For the other tools, AProVE can correctly handle 84 (83.17%) programs in 11964.95 seconds; UAutomizer can correctly handle 92 (90.10%) programs in 2246.62 seconds; and T2 can correctly handle 80 (79.21%) programs in 1620.19 seconds. This indicates that the time overhead of these three tools is very expensive (some programs may time out); and the performance improvement of Loopster over these tools is dramatic. Note that even only considering the time overhead of the correctly analyzed programs (ignoring the programs that time out), Loopster is still much more efficient: it took 7.36 seconds for 93 programs; but AProVE took 174.30 seconds for 84 programs, UAutomizer took 315.18 seconds for 92 programs, and T2 took 249.09 seconds for 80 programs.

For the monotonicity analysis in Loopster, 89 (88.12%) programs can be handled by our extended static technique; and 12 programs

Table 1: Experimental Results on the Benchmark

Program		Loopster				AProVE			UAutomizer			T2		
		CR	CT(s)	TT (s)	CTL	CR	CT (s)	TT (s)	CR	CT (s)	TT (s)	CR	CT (s)	TT (s)
Unnested	T (38)	35	3.74	4.03	4	32	94.06	3739.13	34	137.09	1104.93	25	80.16	1439.76
	NT (31)	28	1.48	1.51	4	21	30.20	7274.47	27	56.40	120.00	23	150.28	161.78
Nested	T (27)	26	1.76	1.79	4	26	41.42	942.73	26	105.17	1005.17	27	14.93	14.93
	NT (5)	4	0.38	0.43	0	5	8.62	8.62	5	16.52	16.52	5	3.72	3.72
Total (101)		93	7.36	7.76	12	84	174.30	11964.95	92	315.18	2246.62	80	249.09	1620.19

Table 2: Detailed Results of Comparison

Program	Loopster	AP.	UA.	T2
93 programs	√	78	85	75
McCarthy91_Iteration_true-termination.c	Cycle	√	√	√
NO_03_false-termination.c	Cycle	√	√	√
TelAviv-Amir-Minimum_true-termination.c	Cycle	×	√	×
Piecewise_true-termination.c	Cycle	√	√	×
LogAG_true-termination.c	Mon	√	√	√
NonTermination2_false-termination.c	Mon	√	√	×
Division_false-termination.c	Mon	√	√	√
LexIndexValue-Pointer_false-valid-deref.c	MEM-N	×	×	√

cannot be handled, and thus the prover is used for their path termination analysis. It indicates that the monotonicity can be successfully detected in most programs, and it is a useful property that can be used for loop termination analysis. The average time used with prover and monotonicity are similar (<0.1s) in the benchmarks.

Further, we provide a detailed analysis of how often different analysis methods in our approach are needed to prove (non)termination for the benchmark. For the 93 handled programs, 36 (39%) programs can be proved with Theorem 1 and 2 after we construct the PDA$_x$. 27 (29%) programs contain the unknown states in the constructed PDA$_x$, and we first perform refinement analysis and then conclude the result with Theorem 1 and 2. 30 (32%) programs contain cycles in the constructed PDA$_x$, and thus we perform the PDA$_x$ reduction analysis. The results show that the refinement and cycle reduction are very important and effective to prove loop termination.

In summary, Loopster can handle more programs correctly with much less time overhead (20×+ performance improvement even if only considering those correctly analyzed programs), which shows the effectiveness and performance improvement of Loopster over the state of the art. The biggest advantage of Loopster owes to our static analysis, while other tools need expensive time overhead to infer and validate ranking functions on demand.

4.2 Detailed Comparison with State-of-the-Art

To discuss the advantages and disadvantages of Loopster over the ranking function-based approaches, we give the detailed results of the programs that can/cannot be handled by Loopster but cannot/can be handled by AProVE, UAutomizer and T2 in Table 2.

The first row indicates that, among the 93 programs that Loopster can handled, AProVE, UAutomizer and T2 can respectively handle 78, 85 and 75. After analyzing the programs that cannot be handle by other tools, we summarize our advantages as follows.

- *Using Monotonicity.* Monotonicity [39] is a common and useful property in the loop iteration. By using the monotonicity, we can often conclude termination or nontermination quickly. For example, all the other three tools failed to handle the program *Hanoi_plus_false-termination.c*, but Loopster proved its nontermination in about 0.04 seconds. Specifically, Loopster can compute a nonterminating sufficient condition after analyzing the monotonicity of x. In this program, the monotonicity of x, y and z belong to cyclic monotonicity, and Loopster computes the nonterminating sufficient condition as $x_0 > 0 \wedge y_0 > 0 \wedge z_0 > 0$.
- *Considering the Precondition.* We can handle programs whose termination depends on the precondition of the loop while the other tools may fail. For example, all the other three tools failed to handle *Gothenburg_v2_true-termination.c* as its result depends on the precondition $a == b + 1 \wedge x < 0$. However, our refinement analysis refines the termination status of each state by considering the precondition.
- *Reducing Complexity.* With our divide-and-conquer strategy (i.e., the termination analysis in each path, and then the overall termination of the loop), we can reduce the complexity when proving the termination of some programs that need complex ranking functions. For example, proving the termination of *Pure3Phase_true-termination.c* needs a 3-phase ranking function, and T2 spent 45 seconds and UAutomizer timed out. Proving the termination of *aaron3_true-termination.c* needs a multi-dimensional ranking function, and UAutomizer spent 17 seconds and AProVE timed out. Instead, Loopster correctly handled them in less than one second without computing complex ranking function.

On the other hand, there are eight programs that Loopster failed to handle correctly, as shown in Table 2. The second column reports the failure reasons, including 1) the cycle cannot be reduced (marked as *Cycle*), 2) the monotonicity cannot be detected (marked as *Mon*), and 3) the nontermination of the program is caused by a memory vulnerability which interferes the monotonicity (marked as *MEM-N*). Hence, the main disadvantages of our approach are as follows.

- *Monotonicity cannot be detected.* In such cases, we cannot compute the (non)terminating sufficient condition, and thus the refinement may fail. For example, for *NonTermination2_false-termination.c*, we cannot compute the nonterminating sufficient condition as the monotonicity of $x/oldx$ cannot be detected.
- *The cycle in a cyclic-PDA$_x$ may not be reduced.* In such cases, we cannot 1) prove the termination, or 2) detect the nontermination that is caused by the cycle. For example, all the paths in *NO_03_false-termination.c* are terminating, but Loopster returns unknown as we cannot find the pivot state to reduce the cyclic-PDA$_x$ for proving its termination.

In summary, Loopster and the ranking function-based approaches are complementary to each other. For the loops that AProVE, UAutomizer and T2 cannot handle, Loopster can be used by them to reduce the complexity and improve the efficiency for inferring complex ranking functions. For the loops we cannot handle, we can extend Loopster to provide advanced monotonicity analysis based on ranking functions.

Table 3: Experimental Results on Real Projects

Project	Loops	Sliced	Handled	Time (s)	NT
gmp	1545	376 (24%)	1188 (77%)	13.07	13
libxml	3847	2218 (58%)	1081 (28%)	42.73	8
httpd	1428	803 (56%)	386 (27%)	35.57	2
Total	6820	3397 (50%)	2655 (39%)	91.37	23

4.3 Experimental Results on Real Projects

To evaluate the effectiveness and scalability of our static approach, we ran Loopster on the loops in three open source real-world projects, including the arithmetic library gmp-6.0.0, the XML C parser and toolkit libxml2-2.9.3, and the Apache HTTP Server project httpd-2.4.18. We extracted the loops in these projects by *llvm loop pass* [4]. The precondition of each loop is determined by the initialization of the variables before the loop. If the initial value depends on the computation before the loop, we assume that it is nondeterministic, which may cause inaccurate result but does not affect the soundness of our approach.

Table 3 shows the results of our analysis on these projects. The column *Loops* shows the number of extracted loops in each project. The column *Sliced* lists the number of loops that can be sliced by our slicing technique. The columns *Handled* and *Time* give the number of loops we can handle correctly and the time required for the analysis. The column *NT* reports the number of nonterminating loops we identified.

Among the total 6820 loops, 3397 (50%) loops can be sliced to reduce irrelevant paths and improve the effectiveness and performance of our static approach. For example, one program from httpd-2.4.18 contains more than 10 paths. After slicing, the loop becomes *for(c=0;c<256;++c)* and has only one path, making it easy to analyze its termination. Therefore, slicing is necessary to termination analysis, especially for complex loops in real-world projects.

On one hand, 2655 (39%) loops can be correctly handled in 91.37 seconds. There are 23 nonterminating loops since we set their precondition to be nondeterministic for simplicity and cause an over-approximation to their real precondition. For example, in Fig. 5, the value of *chunk_nbits* depends on the computation before this loop, but we set it to be nondeterministic. Thus, the loop is terminating when *chunk_nbits* is greater than zero, and nonterminating when *chunk_nbits* is less than or equal to zero.

On the other hand, Loopster cannot handle 4165 (61%) loops due to two main reasons: 1) the loops contain complex data structures (e.g., arrays, heaps) and function calls, 2) the variables are updated by complex computation (e.g., bitwise calculator), which makes the monotonicity calculation difficult. Actually, the two challenges are orthogonal to our research and the possible solutions will be discussed in Section 4.4.

In particular, gmp is an arithmetic library, which has many integer or simple pointer calculation related loops. Thus, Loopster can handle 77% of the loops. libxml and httpd contain many complex data structures (e.g., XML parser in libxml and HTTP protocol handling in httpd), and thus we can only handle 27% of the loops. Hence, except for the challenging loops described above, which are orthogonal to our approach and also non-trivial to be handled by the state-of-the-art tools, a large part of the loops in real-world projects can be analyzed by the static approach efficiently.

```
int rbitpos=__VERIFIER_nondet_int();
int chunk_nbits=__VERIFIER_nondet_int();
int nbits=__VERIFIER_nondet_int()
while (rbitpos + chunk_nbits <= nbits)
rbitpos += chunk_nbits;
```

Figure 5: A Simplified Loop from gmp

We also attempted to compare Loopster with the state-of-the-art tools on the loops from these real-world projects. However, the state-of-the-art tools cannot directly work on these real loops without modification. Therefore, we manually modified a set of loop programs based on the loops in these real-world projects by the following three steps: 1) remove the simple loops that are trivial to check as well as the loops that are not supported by Loopster and the state-of-the-art tools (e.g., loops that contain data structures and function calls that affect the loop termination), 2) randomly select 10 loops for each project from the remaining loops, and 3) adapt the selected loops by putting the loops in a main function, adding initialization for the variables in the loops and simplifying complex statements and function dependencies that are not relevant to loop termination to make them executable. These modified programs can be found in [5], as a benchmark for future research on loop termination analysis. Notice that, we compared Loopster with AProVE and UAutomizer but not T2 because llvm2KITTeL [3] failed to convert most of the real-world loops into T2's input format. Instead, we ran the tool 2LS [2] (verion 0.3.0) since it has been empirically demonstrated that 2LS can check larger programs with thousands of lines of code [14]. Here we set the timeout to 300 seconds.

Table 4 shows the comparison results on the modified loop programs. The column *CR* lists the number of correctly handled programs, and the column *T* reports the time overhead. Among the total 30 modified loop programs (under the column *Modified*), Loopster can check all of them correctly in 1.39 seconds, while AProVE can only check 3 programs in 356 seconds, UAutomizer can check 16 programs in 423 seconds, and 2LS can check 13 programs in 332 seconds. The results show that the three tools can only handle a small part of the loops but need much time since the loops in real projects contain some complex statements (e.g., array comparison and data structure) that are irrelevant to the termination. Loopster can handle such loops well since we perform a program slicing to reduce the irrelevant statements. To ensure a fair comparison across these tools, we sliced the loop programs manually and also ran the state-of-the-art tools on the sliced version (under the column *Sliced*). With the sliced programs, all the three tools can check more programs with much less time. Still, Loopster can handle much more programs with much less time than other tools.

In summary, Loopster can handle some loops (39%) in real-world projects efficiently, which indicates the effectiveness and scalability of our tool on real-world loops. In addition, the state-of-the-art tools often have limitations to analyze real-world loops, and slicing is a useful preprocessing step for termination analysis.

4.4 Discussion on the Unsupported Loops

Advanced monotonicity analysis can be developed to support the loops with data structures. For example, by modeling the contents of heaps or abstracting pointer operators as arithmetic programs. Slicing can also help to remove the structures that do not affect the

Table 4: Experimental Results on the Real-world Loop Programs

Project	Loopster		AProVE				UAutomizer				2LS			
			Modified		Sliced		Modified		Sliced		Modified		Sliced	
	CR	T (s)	CR	T (s)	CR	T (s)	CR	T (s)	CR	T (s)	CR	T (s)	CR	T (s)
gmp (10)	10	0.51	1	19.19	1	13.42	5	257.47	7	125.39	6	9.32	6	4.88
libxml (10)	10	0.50	0	11.99	2	18.09	6	49.11	6	45.37	4	2.74	4	2.21
httpd (10)	10	0.40	3	320.31	6	18.75	5	114.01	7	89.42	3	319.83	6	5.60
Total (30)	30	1.39	3	356.90	9	118.36	16	423.49	20	270.34	13	332.05	16	13.05

termination. In addition, we also plan to investigate the possibility to integrate the existing techniques for handling heaps [8] and bitvectors [14, 19, 23]. For multi-thread loop programs, we can try to translate such programs into sequential programs [6] and then apply our approach. On the other hand, if we cannot handle the cycle in a PDA$_x$, the (non)termination of the loop cannot be decided. In this case, we can perform more specific analysis on the whole effect of the cycle, e.g., unrolling the cycle for some times to get more information or computing ranking functions.

5 RELATED WORK

In this section, we discuss the related work on termination proving and non-termination proving.

5.1 Termination Proving

Existing research on termination proving mainly focuses on the synthesis of termination arguments. Michael and Henny [40, 41] first introduce the automatic synthesis of one-dimensional linear ranking functions over linear loops. Then Podelski and Rybalchenko [43] propose a complete method to synthesize linear ranking functions. However, it is only used for single-path linear loops.

For complex programs, multi-dimensional and lexicographic ranking functions are needed [7, 9, 10, 15]. Christophe et al. [15] propose a complete approach to compute multi-dimensional ranking functions. Bradley et al. [9] generalize the method in [43] to multi-path loops and synthesize lexicographic linear ranking functions based on inductive linear invariants. Then, this method is extended with *template trees* [10]. Ben-Amram and Genaim [7] study the complexity of the search for ranking functions and prove it to be a coNP-complete problem. Rybalchenko [45] proves program termination and safety by ranking functions, interpolants, invariants, resource bounds, and recurrence sets with a series of illustrating examples. Leike and Heizmann [37] introduce some linear ranking templates, such as multiphase, nested, piecewise, parallel and lexicographical ranking templates.

In general, the main difficulties of termination proving based on ranking functions are two-fold. First, the search for efficient ranking functions is non-trivial. Lexicographic ranking functions are needed for complex programs, while linear ranking functions can be only used in certain types of loops (e.g., [40, 43]). Unfortunately, it is difficult and expensive to find lexicographic ranking functions [22]. For some programs such as non-termination loops, the search may even not terminate. Second, the validity of the ranking functions may be expensive [22]. The validity of constraint-based ranking functions often depends on invariants that must be strong enough [12], or an iterative approach [20]. Thus, such techniques are often expensive, especially for complex multi-path loops.

To mitigate the difficulty in constraint-based synthesis for lexicographic ranking functions, several techniques [20, 22, 32, 36] are proposed based on Ramsey's theorem. The basic idea is to find a set of simple linear ranking functions rather than finding non-trivial lexicographic ranking functions [22]. However, the validity of the termination argument is still difficult.

Compared with the above techniques, our static approach avoids searching ranking functions. Hence, Loopster is much faster than ranking function-based techniques, as shown in the evaluation. The limitation is that Loopster fails when monotonicity cannot be detected or some SCC cannot be reduced by the static analysis.

Besides, several advances have been made to support division and modulo arithmetic [11], bitvectors [14, 19, 23] and heaps [8], which are not yet supported in Loopster.

5.2 Nontermination Proving

The approaches based on ranking function usually cannot decting nontermination. Several techniques [6, 13, 27, 28, 33, 46] have been proposed to prove non-termination. Gupta et al. [27] propose to generate all possible lassos by executing the program until some control location is re-visited, then check whether each lasso is feasible by template-based constraint solving. Invel [46] proves non-termination by the combination of theorem proving and invariant generation. TREX [28] and T2 [13] check non-termination by searching and refining an under-approximation of the loop that can make the loop never terminate. CppInv [33] proves non-termination with a MAX-SMT-based invariant generation. Atig et al. [6] propose to detect non-termination in multi-threaded programs by a systematic translation to sequential programs.

Compared with the above techniques, Loopster reduces the non-terminating proving to the reachability problem. The disadvantage is that we currently consider the nonterminating path, the nonterminating SCC is not considered yet, which is our future work.

6 CONCLUSION

In this paper, we have proposed a novel approach for loop termination analysis based on path termination analysis and path dependency reasoning. We implemented the approach in Loopster and demonstrated the capability of Loopster on a competition benchmark and three real-world projects. The results have demonstrated the effectiveness, performance and scalability of Loopster over the state-of-the-art tools. In the future, we plan to extend Loopster by supporting recursive programs and complex structures, and handling more loops with unsupported SCC.

ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-30), the project M4081588.020.500000, and the National Science Foundation of China (No. 61572349 and 61272106).

REFERENCES

[1] 2016. Competition on Software Verification 2016. http://sv-comp.sosy-lab.org/2016. (2016).

[2] 2017. 2LS for Program Analysis. http://www.cprover.org/wiki/doku.php?id=2ls_for_program_analysis. (2017).

[3] 2017. llvm2kittel. (2017). https://github.com/s-falke/llvm2kittel.

[4] 2017. LLVM's Analysis and Transform Passes. http://llvm.org/docs/Passes.html. (2017).

[5] 2017. Loopster. https://sites.google.com/site/looptermination. (2017).

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, Michael Emmi, and Akash Lal. 2012. Detecting Fair Non-termination in Multithreaded Programs. In *CAV*. 210–226.

[7] Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4 (2014), 26:1–26:55.

[8] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. 2006. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *CAV*. 386–400.

[9] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Linear ranking with reachability. In *CAV*. 491–504.

[10] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. The polyranking principle. In *ICALP*. 1349–1361.

[11] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005. Termination Analysis of Integer Linear Loops. In *CONCUR 2005 - Concurrency Theory*. 488–502.

[12] Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better termination proving through cooperation. In *CAV*. 413–429.

[13] Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving nontermination via safety. In *TACAS*. 156–171.

[14] Hong-Yi Chen, Daniel Kroening, Peter Schrammel, and Bjoern Wachter. 2015. Synthesising Interprocedural Bit-Precise Termination Proofs. In *ASE*. 53–64.

[15] Alias Christophe, Darte Alain, Feautrier Paul, and Gonnord Laure. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *SAS*. 117–133.

[16] Michael Codish, Igor Gonopolskiy, Amir M. Ben-Amram, Carsten Fuhs, and Jürgen Giesl. 2011. SAT-Based Termination Analysis Using Monotonicity Constraints over the Integers. *CoRR* (2011).

[17] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. 2007. Proving That Programs Eventually Do Something Good. In *POPL*. 265–276.

[18] Byron Cook, Eric Koskinen, and Moshe Vardi. 2011. Temporal property verification as a program analysis task. In *CAV*. 333–348.

[19] Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2010. Ranking Function Synthesis for Bit-vector Relations. In *TACAS*. 236–250.

[20] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Termination Proofs for Systems Code. In *PLDI*. 415–426.

[21] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2011. Proving program termination (Review article). In *Communications of the ACM*. 88–98.

[22] Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. lexicographic termination proving. In *TACAS*. 47–61.

[23] Cristina David, Daniel Kroening, and Matt Lewis. 2015. Unrestricted Termination and Non-termination Arguments for Bit-Vector Programs. In *ESOP*. 183–204.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS*. 337–340.

[25] Stephan Falke, Deepak Kapur, and Carsten Sinz. 2011. *Termination analysis of C programs using compiler intermediate languages*. Technical Report. 41–50 pages.

[26] Sumit Gulwani and Florian Zuleger. 2010. The Reachability-Bound Problem. In *PLDI*. 292–304.

[27] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-termination. In *POPL*. 147–158.

[28] William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *SAS*. 304–319.

[29] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schatzle, and Andreas Podelski. 2016. Ultimate Automizer with Two-track Proofs (Competition Contribution). In *TACAS*.

[30] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* (1969), 576–580.

[31] Eric Koskinen. 2012. *Temporal verification of programs*. Ph.D. Dissertation. University of Cambridge.

[32] Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination analysis with compositional transition invariants. In *CAV*. 89–103.

[33] Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2014. Proving Non-termination Using Max-SMT. In *CAV*. 779–796.

[34] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2013. Proving termination of imperative programs using Max-SMT. In *FMCAD*. 218–225.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*. 75–88.

[36] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The Size-change Principle for Program Termination. In *POPL*. 81–92.

[37] Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *TACAS*. 172–186.

[38] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. 2009. A Fast and Precise Static Loop Analysis based on Abstract. In *CGO*. 136–146.

[39] Spezialetti Madalene and Gupta Rajiv. 1995. Loop Monotonic Statements. *IEEE Trans. Softw. Eng.* 21, 6 (1995), 497–505.

[40] Colón Michael and Sipma Henny. 2001. Synthesis of Linear Ranking Functions. In *TACAS*. 67–81.

[41] Colón Michael and Sipma Henny. 2002. Practical Methods for Proving Program Termination. In *CAV*. 442–454.

[42] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. In *SDE*. 177–184.

[43] Andreas Podelski and Andrey Rybalchenko. 2004. A complete method for the synthesis of linear ranking functions. In *VMCAI*. 239–251.

[44] Andreas Podelski and Andrey Rybalchenko. 2007. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*. 245–259.

[45] Andrey Rybalchenko. 2010. Constraint Solving for Program Verification: Theory and Practice by Example. In *CAV*. 57–71.

[46] Helga Velroyen and Philipp Rümmer. 2008. Non-termination Checking for Imperative Programs. In *TAP*. 154–170.

[47] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *FSE*. 61–72.