Data-Driven Loop Bound Learning for Termination Analysis

Rongchen Xu†
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
xrc19@mails.tsinghua.edu.cn

Jianhui Chen†
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
chenjian16@mails.tsinghua.edu.cn

Fei He‡
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@tsinghua.edu.cn

ABSTRACT
Termination is a fundamental liveness property for program verification. A loop bound is an upper bound of the number of loop iterations for a given program. The existence of a loop bound evidences the termination of the program. This paper employs a reinforced black-box learning approach for termination proving, consisting of a loop bound learner and a validation checker. We present efficient data-driven algorithms for inferring various kinds of loop bounds, including simple loop bounds, conjunctive loop bounds, and lexicographic loop bounds. We also devise an efficient validation checker by integrating a quick bound checking algorithm and a two-way data sharing mechanism. We implemented a prototype tool called ddfTerm. Experiments on publicly accessible benchmarks show that ddfTerm outperforms state-of-the-art termination analysis tools by solving 13-48% more benchmarks and saving 40-77% solving time.

CCS CONCEPTS
• Software and its engineering → Formal software verification;
• Theory of computation → Logic and verification.

KEYWORDS
Termination analysis, loop bound, data-driven approach

1 INTRODUCTION
Termination is a fundamental liveness property for program verification. It plays a central role in proving the total correctness of programs. A loop bound is an upper bound of the number of loop iterations for a given program. A validated loop bound thus evidences the program’s termination. In this paper, we employ a black-box learning approach for termination proving, which consists of two components: (a) a learner that generates a loop bound candidate; and (b) a checker that either confirms the correctness of the candidate or produces a counterexample to refute the candidate.

There exist some prior works on data-driven termination proving [14, 31, 41]. The work [31] employs testing to generate data examples and then applies quadratic programming to learn the loop bound candidates. Unfortunately, this approach can only infer loop bounds in the form of simple affine functions (called simple loop bound in this paper). As a result, it is inapplicable to many realistic programs. The work [41] enriches the forms of the loop bounds. It learns a set of affine expressions by linear interpolation and then arranges them into piecewise, lexicographic, or multiphase forms. However, affine expressions obtained by simple combinations of data examples are imprecise in complex situations. The work [14] employs syntax-guided synthesis (SyGuS) [1] to generate a mass of expressions and then combines these expressions into the loop bound candidate. However, this approach takes into consideration only syntactical information of programs. Due to the lack of program semantics knowledge, their synthesizer can hardly infer proper loop bounds for complicated programs.

In this paper, we propose techniques to enhance both learner and checker of the data-driven loop bound learning approach. Firstly, we reinforce the learner by proposing a series of data-driven algorithms to learn various loop bounds, i.e., simple loop bounds, conjunctive loop bounds, and lexicographic loop bounds. Our data-driven algorithms consider program semantics. More specifically, the program states, together with their transition relations, are recorded and utilized during the algorithms. The expressibility and applicability of the loop bound approach are thus significantly enhanced. With a combination of these learning algorithms, our approach is able to prove the termination of complicated programs with non-linear loop bounds.

We propose a quick bound checking technique to enhance the checker. We observe that only in the last round does the checker validate the loop bound candidate, and in all other rounds, the checker is only responsible for providing counterexamples to refute the candidates. Given that falsification is always cheaper than verification, it is thus worthwhile to apply a quick falsification check before the complete validation check. To this end, we propose a bounded
model checking-based technique for quickly refuting the incorrect bound candidates.

The complete validation of a loop bound candidate can be reduced to a safety verification problem, which usually involves another learning process—the loop invariant learning. Obviously, loop bound learning and loop invariant learning are strongly related and should not be regarded as two independent processes. To this end, we propose a two-way data sharing mechanism between these two processes: on the one hand, the data examples in bound learning are reused in invariant learning; on the other hand, in case of safety verification failures, the generated counterexamples are reused by the bound learner to refine its bound candidates.

We implement a prototype tool called ddtTerm. We take the benchmarks from [14] to evaluate the efficiency of our approach. Compared with the state-of-the-art termination analysis tools, including APoVE [17, 18, 40], UAutorizer [9, 20], FreqTerm [14], and MuVal [24], ddtTerm solves 13-48% more benchmarks and reduces 40-77% analysis time.

To summarize, this paper makes the following contributions:

• We present a series of data-driven algorithms for inferring various loop bounds, including simple loop bounds, conjunctive loop bounds, and lexicographic loop bounds.
• We propose a quick bound checking algorithm for efficiently refuting incorrect loop bound candidates.
• We propose an efficient data sharing mechanism between bound and invariant learning.
• We implement a prototype tool and conduct experiments on publicly accessible benchmarks. Results show the outstanding performance of our approach over state-of-the-art tools.

The rest of the paper is organized as follows: Section 2 motivates our approach using a simple example. Section 3 introduces some background knowledge. Section 4 presents our data-driven loop bound learning algorithms. Section 5 employs the loop bound in termination proving. Section 6 reports evaluation results, Section 7 discusses related work and Section 8 concludes the paper.

2 OVERVIEW

We employ a simple program (in Figure 1) to show the basic idea of our approach. The original version of this program (i.e., the black codes) iteratively increases \( x \) until it equals 10 and then resets \( x \) to 0. The green codes are used for data generation, where the variable \( j \) records the current iteration number, and the \( \text{print}(\ldots) \) method outputs the current values of \( x \) and \( j \) at each iteration.

Figure 1: An example

Figure 2: Validation task

Along with a terminating execution of this program, its output is a sequence of value pairs

\[(x_0, 0) \land (x_1, 1) \land \cdots \land (x_k, k),\]

where \( x_i \) is the value of \( x \) at the \( i \)-th iteration, and \( k \) is the iteration at which the loop terminates. For each output value pair \( (x_i, i) \), let \( \text{idc}_i \triangleq k - i + 1 \) be the number of remaining iterations until the execution terminates, called iteration down counter. Each pair of \( x_i \) and \( \text{idc}_i \) is called a data example. We collect all data examples along this execution, and add them to a dataset \( \mathcal{H} \), i.e.,

\[\mathcal{H} = \{(x_i, \text{idc}_i) \mid 0 \leq i \leq k\}\]

We attempt to infer from \( \mathcal{H} \) a symbolic loop-bound expression \( m(x) \) that represents an upper bound candidate on the number of loop iterations (Section 4). Validation of the inferred loop-bound expression is reduced to the safety checking of the instrumented program in Figure 2 (Section 5). If the checking succeeds, the inferred loop bound is correct, and we prove the termination of the program. Otherwise, the safety checker returns a counterexample, with which we enlarge the dataset \( \mathcal{H} \) and refine the inferred loop bound expression.

The analysis procedure for our motivating example is shown in Table 1. Let us start with a trivial loop bound \( m_1(x) = 0 \). The validation task for \( m_1 \) is to verify the program in Figure 2 with the expression \( m(x) \) at line 1 being replaced by \( \theta \). This validation obviously fails, and a counterexample cex1 with \( x_0 = -2 \) is returned, where \( x_0 \) is the initial value of \( x \). We now have the information that a trace with \( x_0 = -2 \) can refute the loop bound \( m_1 \). We thus use the same initial value and other nearby values (by mutation test) to run the program in Figure 1 and collect the following data examples (marked as \( \Delta \) in Figure 3) from its outputs:

\[\mathcal{H}_a = \{(-4, 4), (-3, 3), (-2, 2), (-1, 1), (0, 0)\}\]

In the remainder of this paper, we simply say that these data examples are obtained from the counterexample cex1.

We learn from \( \mathcal{H}_a \) (by the approach in Section 4.1) a so-called simplest loop bound \( m_2(x) = -x \). Validation checking of \( m_2 \) gives a

Table 1: Analysis procedure of example

<table>
<thead>
<tr>
<th>#</th>
<th>Loop-Bound</th>
<th>Checking</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( m_1(x) = 0 )</td>
<td>cex1: ( x_0 = -2 )</td>
<td>( \mathcal{H}_a : {\Delta} )</td>
</tr>
<tr>
<td>2</td>
<td>( m_2(x) = x )</td>
<td>cex2: ( x_0 = 1 )</td>
<td>( \mathcal{H}_b : {\Delta, \emptyset} )</td>
</tr>
<tr>
<td>3</td>
<td>( m_3(x) = \text{max}(x - x + 11) )</td>
<td>cex3: ( x_0 = 11 )</td>
<td>( \mathcal{H}_c : {\Delta, \emptyset, \emptyset} )</td>
</tr>
<tr>
<td>4</td>
<td>( m_4(x) = \text{max}(x + 11, 1) )</td>
<td>pass</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3: Loop bound and dataset
new counterexample trace cex2 with $x_0 = 1$. Similar to the first round, more data examples (denoted as $\bigtriangleup$ in Figure 3) are obtained from cex2. The dataset becomes $H_0 = \{\bigtriangleup, \bigtriangledown\}$. Apparently, it is impossible to find a linear expression to cover all data examples tightly in $H_0$. However, we could use clustering to group $H_0$ into two subsets, e.g., $H_{01} = \{\bigtriangleup\}$ and $H_{02} = \{\bigtriangledown\}$, and learn two simple loop bounds, e.g., $m_{b_1} = -x$ and $m_{b_2} = -x + 11$, from them respectively. The maximal operation applied to these two expressions (consider no simplification), i.e., $m_1(x) = \max(-x, -x + 11)$, gives a new loop bound, called a conjunctive loop bound (see Section 4.2).

The loop bound $m_1(x)$ is still not valid. Its validation checking returns a counterexample trace cex3 with $x_0 = 11$, from which we obtain more data examples (denoted as $\bowtie$ in Figure 3). The dataset is now $H_1 = \{\bigtriangleup, \bigtriangledown, \bowtie\}$. Let $H_{11}, H_{12}, H_{13}$ be the set of $\bigtriangleup, \bigtriangledown, \bowtie$ in $H_1$, respectively. Note that $H_{13}$ contains an exceptional case (i.e., its leftmost example), which may affect the precision of the learning result. To address this problem, we propose a greedy tactic to learn a set of so-called close loop bounds, which characterize local features of the subsets. For example, the close loop bounds learned from $H_{13}$ could be $m_1(x) = 1$ and $m_2(x) = 2$. By combining all learned loop bounds together, we get $m_4(x) = \max(-x, -x + 11, 1, 2)$. Finally, we use a set covering-based technique to sift redundant loop bounds and get $m_4(x) = \max(-x + 11, 1)$. The expression $m_4(x)$ passes the validation checking and is a correct loop bound. The termination of the program is thus proved.

3 PRELIMINARIES

3.1 Notations

We use $A, B, \cdots$ to denote the general sets, and $x, y, \cdots$ to denote vectors. We denote $x[i]$ the $i$-th component of $x$, and $x \cdot y$ the scalar product of $x$ and $y$. Denote $\mathbb{N}$ the set of natural numbers.

Given a binary relation $\succ$ on a set $W$, we say $a \in W$ is a least element w.r.t. $\succ$, if $a \not\succ b$ for any $b \in W$. We say $\succ$ is a well-founded relation on $W$, if every non-empty subset of $W$ has a least element w.r.t. $\succ$. The set $W$, together with the well-founded relation $\succ$, is then called a well-ordered set. A chain is a sequence of elements $e_1, e_2, \cdots, e_n \in W$ such that $e_1 \succ e_{i+1}$ for $i = 2, \cdots, n - 1$. Given an element $e \in W$, we denote $|e|$ the length of the longest chain in $W$ starting at $e$. A well-ordered set has no infinite chain. Let $\succ^*$ be the reflexive closure of $\succ$, which is not well-founded. For example, $\succ$ is a well-ordered set w.r.t the greater-than relation $\succ^*$, its least element is 0, and there is no infinite chain from any positive number in $\mathbb{N}$. The reflexive closure of $\succ$ is $\succ^*$.

An affine function is composed of a linear function and a constant, e.g., $f(x, y) = ax + by + c$. A function $f(x) : X \to \mathbb{R}$ is said convex if $f(t \cdot x_1 + (1-t) \cdot x_2) \leq t \cdot f(x_1) + (1-t) \cdot f(x_2)$ for any $x_1, x_2 \in X$ and $t \in [0, 1]$.

3.2 Convex Optimization and Set Covering Problem

Given an objective function $f(x)$, the convex optimization problem [5] is to find an optimum $x^*$ that minimizes $f(x)$ and satisfies all constraints, i.e.,

$$\arg \min_x f(x) \quad \text{s.t.} \quad \forall i \leq m \quad g_i(x) \geq 0$$

where $f$ and $g_i(1 \leq i \leq m)$ are all convex functions.

Let $\mathcal{U}$ be a universe, and $S = \{S_1, S_2, \cdots, S_k\}$ be the set of its subsets. Define a selector variable $X_i$ (either 1 or 0) for each $S_i$ in $S$, representing if this subset is selected or not. Assume each $S_i$ is associated with a cost $C_i$, the set covering problem [8] is to

$$\arg \min_{X} \sum_{i=1}^{k} C_i \cdot X_i \quad \text{s.t.} \quad \mathcal{U} = \bigcup_{X_i=1} S_i$$

3.3 Clustering

The clustering problem is to divide a set of objects into several subsets, called clusters, such that similar objects are more likely to be assigned to the same cluster. There are many different clustering models, e.g., centroid-based [30, 39], density-based [2, 13], etc. This paper is mainly focused on the centroid-based clustering, where each cluster is represented as a central vector. The most widely-used centroid-based clustering algorithm is $k$-means [30]. Given a set of observations $(x_1, x_2, \cdots, x_n)$, where each observation is a vector, a $k$-means clustering aims to partition the $n$ observations into $k (\leq n)$ subsets $S = \{S_1, S_2, \cdots, S_k\}$ so as to minimize the within-cluster sum of squares

$$\arg \min_{S} \sum_{i=1}^{k} \sum_{x \in S_i} \|x - \mu_i\|^2$$

where $\mu_i$ is the mean of points in $S_i$.

4 DATA-DRIVEN LOOP BOUND LEARNING

This section discusses the learning of loop bounds from a given dataset. To simplify the discussions, we assume the input program contains a single loop.

A program state is a valuation of the program variables, usually represented as a vector of values. Let $X$ be the set of reachable program states at the loop header, and $W$ a well-ordered set w.r.t. a well-founded relation $\succ$.

**Definition 4.1.** A loop bound is a function $m$ that maps $X$ into $W$ such that for any state $x \in X$, the number of remaining iterations until termination is no more than $m(x)$.

The loop bound is originally unknown. However, we can extract some data examples from the program in the form of $(x, ids)$, where $x$ is a program state in $X$, and $ids \in W$ represents the iteration down counter at that state. After we collect a sufficient number of data examples, we are capable of inferring the loop bound function from these examples.

We use BoundLearn to represent any procedure of loop bound learning. It should satisfy the following definition:

**Definition 4.2.** Given a dataset $H$, the BoundLearn procedure outputs a loop bound candidate $m(x)$ such that

$$\forall (x, ids) \in H. \quad m(x) \succ ids$$

Programs containing multiple or nested loops can be handled by the techniques introduced in [16].
4.1 Simple Loop Bound Learning

Recall that a loop bound \( m(x) \) is a function over an \( n \)-dimensional vector \( x \). A simple loop bound uses \( \mathbb{N} \) as its well-ordered set and can be expressed in an affine function. In this case, the iteration down counter \( \text{idc} \) also belongs to \( \mathbb{N} \), denoted as \( \hat{\text{idc}} \). We employ affine templates, e.g., \( m(x) = a \cdot x + b \), to instantiate the simple loop bound candidates. More specifically, an affine function \( m(x) \) with undetermined coefficients can be expressed as a meta-function

\[
\hat{m}(\hat{a}, x) = \sum_{i=1}^{n} \hat{a}[i] \cdot x[i] + \hat{a}[n+1],
\]

where \( \hat{a} \) is an \( n+1 \)-dimensional vector representing the undetermined coefficients. Once \( \hat{a} \) is determined, the meta-function \( \hat{m}(\hat{a}, x) \) becomes an affine function \( m(x) \).

Given a dataset \( \mathcal{H} \), the simple loop bound learning task can be formalized as the following convex optimization problem over the vector space of the coefficients \( \hat{a} \):

\[
\min_{\hat{a}} \text{vCost}(\mathcal{H}, \hat{a}) + \text{mCost}(\hat{a}) \quad (1)
\]

\[\text{s.t.} \bigwedge_{(x, \text{idc}) \in \mathcal{H}} \hat{m}(\hat{a}, x) \geq \text{idc}\]

Many existing techniques for the convex optimization with constraints (e.g., COBYLA method [36] and SLSQP method [21]) can be used as SimpleBoundLearn.

Recall that convex optimization requires its objective function to be a convex function. The cost functions \( \text{vCost}(\mathcal{H}, \hat{a}) \) [31] and \( \text{mCost}(\hat{a}) \) should be convex. We design these two cost functions based on the following considerations. First, a good loop bound candidate should be close to the data examples. So, we define

\[
\text{vCost}(\mathcal{H}, \hat{a}) \triangleq a_0 \cdot \sum_{(x, \text{idc}) \in \mathcal{H}} (\hat{m}(\hat{a}, x) - \text{idc})^2,
\]

where \( a_0 \) is a hyperparameter. This \( \text{vCost} \) definition tends to make the loop bound as close as possible to the data examples. Differing from [31], we think that a good loop bound candidate should also have a natural form. For example, it is unwise to use a big constant \( \hat{a}[n+1] \) to cover all data examples. To this end, we define

\[
\text{mCost}(\hat{a}) \triangleq a_m \cdot \sum_{i=1}^{n+1} |\hat{a}[i]|^2,
\]

where \( a_m \) is also a hyperparameter. This \( \text{mCost} \) definition tends to make the coefficients as small as possible. A solution of the above optimization problem assigns values to the coefficients \( \hat{a} \) and thus produces a simple loop bound candidate \( m(x) \).

4.2 Conjunctive Loop Bound Learning

There are programs (e.g., the motivating example in Section 2) whose loop bound cannot be caught by a simple loop bound expression. For these programs, we cannot apply SimpleBoundLearn on the whole dataset. Instead, the so-called conjunctive loop bound should be applied, which can be viewed as a piecewise affine function over the program variables. Algorithm 1 depicts our algorithm for learning conjunctive loop bounds.

Dataset Clustering. A natural idea for learning a conjunctive loop bound is to divide the dataset \( \mathcal{H} \) into several subsets and learn a simple loop bound from each of these subsets. Then a combination of these simple loop bounds forms a conjunctive loop bound. For example, the dataset \( \mathcal{H}_0 \) in our motivating example can be partitioned into two subsets, i.e., \( \mathcal{H}_{01} : \{\triangle\} \) and \( \mathcal{H}_{02} : \{\Diamond\} \) (see Figure 3). We obtain two simple loop bound candidates, i.e., \( m_1 = -x, m_2 = -x + 11 \), by applying SimpleBoundLearn on these two subsets, respectively. A conjunctive loop bound candidate is \( m = \max(m_1, m_2) \), which is a valid bound on the domain of \( x \in (\infty, 10) \).

As presented on lines 2 to 3 in Algorithm 1, we first estimate a maximal number \( k \) of the clusters and then employ a centroid-based clustering (k-means) to partition the dataset \( \mathcal{H} \) into no more than \( k \) subsets. After clustering, we are able to learn local features from the subsets, which can hardly be learned from the whole dataset.

**Algorithm 1: ConjunctiveBoundLearn(\( \mathcal{H} \))**

```plaintext
input : A data set \( \mathcal{H} \)
output: A conjunctive loop bound candidate \( m \)

1. \( M \leftarrow \emptyset \)
2. \( k \leftarrow \text{GetMaxClusterNumber(} \mathcal{H} \text{)} \)
3. \( \mathcal{H}_1, \ldots, \mathcal{H}_k \leftarrow \text{Clustering(} \mathcal{H}, k \text{)} \)
4. for \( i = 1 \) to \( k \) do
5. \( m_i^1 \leftarrow \text{SimpleBoundLearn}(\mathcal{H}_i) \)
6. \( m_i^2, \ldots, m_i^n \leftarrow \text{CloseBoundLearn}(\mathcal{H}_i) \)
7. \( M \leftarrow M \cup \{m_i^1, m_i^2, \ldots, m_i^n\} \)
8. \( M_c \leftarrow \text{BoundCombine}(M, \mathcal{H}) \)
9. return \( \text{max}(M_c) \)

function CloseBoundLearn(\( \mathcal{H} \))

if \( \mathcal{H} \neq \emptyset \) then
10. \( m \leftarrow \text{SoftConvexOptimize}(\mathcal{H}) \)
11. \( C \leftarrow \text{Covered}(m, \mathcal{H}) \)
12. \( S \leftarrow \mathcal{H} - C \)
13. return \( \{m\} \cup \text{CloseBoundLearn}(S) \)
14. return \( 0 \)

function BoundCombine(\( M, \mathcal{H} \))

foreach \( m \in M \) do
15. \( C_m \leftarrow \text{GetCost}(M, \mathcal{H}) \)
16. foreach \( v \in \mathcal{H} \) do
17. \( a_{m,v} \leftarrow \text{GetCoverage}(m, p) \)
18. \( M_c \leftarrow \text{SetCoveringProblem}((C_m), \{a_{m,v}\}) \)
19. return \( M_c \)
```

Close Loop Bound. Although the centroid-based clustering helps to learn local features, there are still some exceptional cases. For example, the data examples around the discontinuities of \( m(x) \), e.g., \( x = 0 \) and \( x = 10 \) in Figure 3, might be grouped into the same subset. Moreover, the counterexamples returned by the checker are often near the discontinuities. In our motivation example, the
counterexamples in the second and the third rounds are very close to the discontinuities \(x = 0\) and \(x = 10\). As a result, data examples obtained from these counterexamples are very likely to get clustered together. If \(m(x)\) is continuous on the discontinuities, e.g., \(x = 10\) in Figure 3, we can hardly learn a suitable loop bound candidate from these subsets. For example, in the third round of the motivation example, if we directly apply SimpleBoundLearn to the cluster \(\mathcal{H}_3 : \{ O \}\), a loop bound candidate \(m_3 = -x/4 + 17/4\) could be learned. This loop bound is obviously inaccurate and can be refuted by any input \(x \geq 17\).

We propose close loop bounds to address the above problem. Basically, a close loop bound does not require covering all data examples in \(\mathcal{H}\). We introduce a slack set \(\mathcal{S} \subseteq \mathcal{H}\) and allow the data examples in \(\mathcal{S}\) to exceed the close loop bound.

A natural idea for learning a close loop bound is to drop the constraints relevant to \(\mathcal{S}\) from the convex optimization equation (1). However, it is very difficult to precisely determine \(\mathcal{S}\). Alternatively, we define a slack variable \(\delta_x\) for each data example \((x, idc) \in \mathcal{S}\), representing the distance of this example exceeding the loop bound candidate, i.e. \(\delta_x = idc - m(x)\). Apparently, the slack variables' values should be greater than or equal to 0. Then the close loop bound learning problem can be formalized as the following convex optimization with soft constraints (called SoftConvexOptimize):

\[
\begin{align*}
\text{min} & \quad vCost(\mathcal{H} - \mathcal{S}, a) + mCost(\hat{a}) + sCost(S) \\
\text{s.t.} & \quad \delta_x \geq 0 \\
& \quad \hat{m}(\hat{a}, x) - idc \geq 0 \\
& \quad \hat{m}(\hat{a}, x) + \delta_x - idc \geq 0 \\
& \quad \delta_x \geq 0 \\
\end{align*}
\]

(2)

where \(a_s\) is a hyperparameter. This cost function encodes our preference over fewer data examples that significantly exceed the loop bound, rather than a large number of data examples that slightly exceed the loop bound.

For each subset \(\mathcal{H}_i\), we learn a simple loop bound (at line 5) and a set of close loop bounds (at line 6). All of the learned simple and close loop bounds are added to \(\mathcal{M}\). The CloseBoundLearn procedure in Algorithm 1 depicts our learning algorithm for close loop bounds.

\[
\begin{aligned}
1 & \text{ while} (x \gg 0 \&\& y > 0) \quad 1 \text{ assume}(i1 \gg M1(x,y)); \\
2 & \{ \\
3 & \quad \text{print}(x, y); \\
4 & \quad \text{while}(x \gg 0 \&\& y > 0) \{ \\
5 & \quad \text{print}("B1"); \\
6 & \quad y = y - 1; \\
7 & \quad \} \\
8 & \quad \text{print}("B2"); \\
9 & \quad i1 = i1 - 1; \\
10 & \} \\
11 & \} \\
12 & \} \\
\end{aligned}
\]

Figure 4: Lexico. example Figure 5: Lexico. validation task

For each loop bound candidate \(m \in \mathcal{M}\), we introduce a variable \(C_m\) to represent its cost and a Boolean variable \(X_m \in \{0, 1\}\) to indicate if \(m\) is kept. The kept loop bounds should cover all data examples. We define an indicator variable \(a_{u,m}\) for each data example \(u \in \mathcal{H}\). The variable \(a_{u,m} = 1\) means that the data example \(u : (x, idc)\) is covered by \(m\), i.e., \(m(x) \geq idc\). The loop bounds combining task can then be formalized as the following set covering problem with minimum cost:

\[
\begin{align*}
\text{min} & \quad \sum_{m \in \mathcal{M}} C_m \cdot X_m \\
\text{s.t.} & \quad \sum_{v \in \mathcal{H}} a_{u,v} \cdot X_m \geq 1 \\
& \quad \bigwedge_{m \in \mathcal{M}} X_m \in \{0, 1\}
\end{align*}
\]

(3)

Note that both \(a_{u,m}\) and \(C_m\) can be calculated beforehand. After solving this optimization problem, we get a set of selected loop bound candidates \(\mathcal{M}_c = \{m | m \in \mathcal{M} \& X_m = 1\}\).

The BoundCombine procedure is presented in lines 19 to 25 of Algorithm 1. We first calculate the costs and the coverage indicators and then solve the corresponding set covering problem to get the reduced loop bound set \(\mathcal{M}_c\). In lines 8 to 9, we call this procedure and return the combination of the pruned bound candidates. Let us continue the verification of our motivation example on the third round, the set of loop bound candidates learned from \(\mathcal{H}_3\) is \(\mathcal{M} = \{-x, -x + 1, 1, 2\}\). After BoundCombine, a more concise set \(\mathcal{M}_c = \{-x + 1, 1\}\) is produced. Combining these candidates together, we get \(m = \max(-x + 1, 1)\), which is a correct loop bound, being able to prove the termination of the example program.

4.3 Lexicographic Loop Bound Learning

The conjunctive loop bound is still powerless in handling programs with rather complicated control-flows. This section proposes a new method to learn lexicographic loop bound (LexLB), composed of simple and conjunctive loop bounds but with more powerful expressibility. In fact, the simple and conjunctive loop bounds can be regarded as special cases \((n = 1)\) of \(n\)-dimensional lexicographic loop bound \((n\text{-LexLB})\).

Lexicographic Loop Bound. We employ a new example program (in Figure 4) to illustrate the lexicographic loop bound learning. The symbol \(*\) in the program denotes a non-deterministic value (bool or int). Note that the branches of the if-then-else statement may interleave arbitrarily across iterations, the maximal number of iterations of this loop can reach \(O(x_0 \cdot y_{max})\), where \(x_0\) is the initial
value of \( x \) and \( y_{\text{max}} \) is the maximum value that \( y \) may attain during the whole program execution. From this perspective, the loop bound of this program should be expressed in a non-linear expression. However, as we commonly know that learning a non-linear loop bound requires general non-linear optimization, and validating a non-linear loop bound requires non-linear constraint solving. In general, optimizing and validating non-linear loop bounds are both undecidable.

Instead, we adopt the 
lexicographic loop bound, which is an \( n \)-dimensional function over the program variables, i.e.,
\[
\mathbf{m}(x) \triangleq \langle m_1(x), \cdots, m_n(x) \rangle.
\]

For example, \( \mathbf{m}(x, y) = \langle y, x \rangle \) is a 2-LexLB of the program in Figure 4. The lexicographic loop bound function maps the program states into a well-ordered set formed by an \( n \)-dimensional vector space under a 
lexicographic order.

To learn a lexicographic loop bound, we need to address the following two problems: (1) how to decide the dimensionality of the lexicographic function? (2) how to decide the lexicographic order over these dimensions?

A lexicographic loop bound should cover all paths within the loop body. Let \( k \) be the number of paths in the loop body, and \( n \) be the assumed dimensionality of the lexicographic function. We consider all solutions of binding the \( k \) paths to the \( n \) dimensions, where each dimension is bound to at least one path. Each solution gives a lexicographic order. A solution is said feasible if from which we are able to learn a valid loop bound. If there are multiple feasible solutions, we evaluate their learned bounds in each dimension and choose the best one. The dimensionality of the lexicographic function is initialized to 2 and then progressively increased until either we find a feasible solution or a predefined dimensionality bound \( k_0 \) (\( k_0 \leq k \)) is reached.

Extract Data Examples. We employ the program in Figure 4 to illustrate the extraction of data examples. The method can be naturally extended to general cases.

Considering the instrumented codes (the green codes) in Figure 4, we use B1 and B2 to label the branch executed in each loop iteration. The program’s output is a sequence of tuples; each tuple corresponds to a loop iteration and is in the form of \( \langle x, y, lb \rangle \), where \( x, y \) are variables’ values, and \( lb \) is the label of the executed branch. For example, given an input of \( x = 2 \) and \( y = 1 \), a (possible) output of this program is:
\[
\rho : \langle 2, 1, B2 \rangle \rightarrow \langle 1, 1, B1 \rangle \rightarrow \langle 0, B2 \rangle \rightarrow \langle 0, B1 \rangle \rightarrow \langle 0, 1, B1 \rangle
\]

Denote \( \mathcal{H}_p \) the set of all outputted traces of this program.

The loop body of this program contains two paths (i.e., the two branches). Suppose we want to learn a 2-dimensional LexLB, e.g., \( \mathbf{m}(x, y) = \langle m_1, m_2 \rangle \). We first introduce a 2-dimensional iteration down counter \( \text{idc} = \langle \text{idc}_1, \text{idc}_2 \rangle \). Without loss of generality, suppose branch B1 is bound to \( \text{idc}_1 \) and branch B2 is bound to \( \text{idc}_2 \). For each trace in \( \mathcal{H}_p \), we calculate the values of \( \text{idc} \) backward along that trace. Let us consider the trace \( \rho \) for example, the value of \( \langle \text{idc}_1, \text{idc}_2 \rangle \) is \( (1, 0) \) at its last tuple \( (0, 1, B1) \) (the branch B1 is passed). We lexicographically increase \( \langle \text{idc}_1, \text{idc}_2 \rangle \) backward along \( \rho \): if the passed tuple contains the B1 label, we increase \( \text{idc}_1 \) by 1; otherwise, if it contains the B2 label, we increase \( \text{idc}_2 \) by 1 and reset \( \text{idc}_1 \) to 0. Finally, we get a sequence \( \sigma \) of 2-dimensional iteration down counters:
\[
\sigma : \langle 0, 2 \rangle \xrightarrow{B2} \langle 1, 1 \rangle \xrightarrow{B1} \langle 0, 1 \rangle \xrightarrow{B2} \langle 2, 0 \rangle \xrightarrow{B1} \langle 1, 0 \rangle
\]

Each counter of \( \sigma \) corresponds to a tuple of \( \rho \). Pairing the program state in each tuple of \( \rho \) and the corresponding iteration down counter of \( \sigma \) gives a set of data examples:
\[
\mathcal{H}((x, y), (\text{idc}_1, \text{idc}_2)) = \{(2, 1), (0, 2), (1, 1), (1, 1), (1, 0), (0, 1)\}
\]

Learn Lexicographic Loop Bound. Algorithm 2 presents our learning algorithm for \( n \)-dimensional lexicographic loop bounds. For each data example \( (x, \text{idc}) \) in \( \mathcal{H} \), we combine its program state \( x \) with each dimension \( \text{idc} \) of \( \text{idc} \). The result forms a set of data examples \( (x, \text{idc}) \) with \( 1 \)-dimensional counters. We add the data example \( (x, \text{idc}) \) to its corresponding dataset \( \mathcal{H} \) (lines 2 to 4 of Algorithm 2). Taking \( \mathcal{H}((x, y), (\text{idc}_1, \text{idc}_2)) \) as an example, after the above operations, we get two data sets:
\[
\mathcal{H}((x, y, \text{idc}_1)) = \{(2, 1, 0), (1, 1, 1), (1, 0, 0), (0, 2, 2), (0, 1, 1)\}
\]
\[
\mathcal{H}((x, y, \text{idc}_2)) = \{(2, 1, 2), (1, 1, 1), (1, 0, 1), (0, 2, 0), (0, 1, 0)\}
\]

Next, we apply SimpleBoundLearn (or ConjunctiveBoundLearn) on \( \mathcal{H}((x, y, \text{idc}_1)) \) and \( \mathcal{H}((x, y, \text{idc}_2)) \) respectively, and learn two expressions \( m_1 = y \) and \( m_2 = x \). Sequentially combining them together, we get a lexicographic loop bound candidate \( \mathbf{m}(x, y) = \langle y, x \rangle \).

4.4 Discussion

There exist other techniques for learning simple loop bounds, e.g., [31]. Compared to [31], our simple bound learning approach adds a new parameter \( m_{\text{Cost}} \) and requires the bound candidates to tend in a natural form.

The paper [41] learns a set of affine functions through a series of simple and random selections for data examples. After that, it arranges these affine functions into a bound template with conjunctive or lexicographic forms directly. Hence, it cannot support compound form bound learning (e.g., lexicographic bound with conjunctive bound in each dimension). In contrast, our conjunctive bound learning considers more global and local features among the dataset, including the distance information (cluster) and the outlier examples (slack set). Moreover, our lexicographic bound learning analyzes the executing trace information. With the help of

---

**Algorithm 2: LexicoBoundLearn(\( H, n \))**

<table>
<thead>
<tr>
<th>input</th>
<th>A dataset ( H ), a number of dimension ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>An ( n )-dimensional lexicographic loop bound candidate ( m )</td>
</tr>
<tr>
<td>( H_l, \cdots, H_n ) ← ( \emptyset ), ( \cdots ), ( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>( \text{foreach} \ (x, \text{idc}) \in H )</td>
<td></td>
</tr>
<tr>
<td>( \text{foreach} \ \text{idc}_i \in \text{idc} )</td>
<td></td>
</tr>
<tr>
<td>( H_i \leftarrow H_i \cup {(x, \text{idc}_i)} )</td>
<td></td>
</tr>
<tr>
<td>( \text{for} \ i = 1 \ \text{to} \ n )</td>
<td></td>
</tr>
<tr>
<td>( m_i \leftarrow \text{Simple/ConjunctiveBoundLearn}(H_i) )</td>
<td></td>
</tr>
<tr>
<td>( m \leftarrow \langle m_1, \cdots, m_n \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{return} \ m )</td>
<td></td>
</tr>
</tbody>
</table>
the additional information, our approach becomes more effective and efficient.

5 DATA-DRIVEN TERMINATION ANALYSIS
A validated loop bound proves the program’s termination. This section presents our loop bound-based termination analysis approach.

5.1 Overall Algorithm
Algorithm 3 presents our data-driven termination analysis algorithm. After initialization (line 1), the algorithm iteratively updates the loop bound \( m \) and the dataset \( \mathcal{H} \) (lines 2-9). Each iteration generates a validation task \( \Phi \) from the input program \( P \) and the current loop bound \( m \) and then sends the validation task to a checker. If \( \Phi \) passes the checking, the current loop bound is validated, the algorithm returns TERMINATES (line 6). Otherwise, the checker returns a counterexamples \( \text{cex} \), the algorithm then proceeds to enrich the dataset \( \mathcal{H} \) with the data examples obtained from \( \text{cex} \) (line 8) and learn a new loop bound \( m \) from this enriched dataset (line 9). The above procedure repeats until either a validated loop bound is found or TIMEOUT is reached; for the latter case, the algorithm returns UNKNOWN.

5.2 Validation Task
With a program \( P \) and a loop bound candidate \( m \), the validation task is a transformation [14, 41] to insert some statements related to the validation of \( m \) into the program \( P \). Suppose the program \( P \) is in the following form:

```
assume(Pre(x)); while(Guard(x)) { Trans(x, x’); } \tag{4}
```

where \( \text{Pre}(x) \) is the precondition representing the codes before the loop, \( \text{Guard}(x) \) is the loop guard, and \( \text{Trans}(x, x’) \) represents the loop body that updates the program variable from \( x \) to \( x’ \).

Then, the validation task transforms the program (4) with loop bound \( m \) into the following form:

```
assume(Pre(x)); assume(\( i \geq m(x) \)); \tag{5}
while(Guard(x)) { assert(i > \bot); Dec(i, i’); Trans(x, x’); }
```

where \( \text{Dec}(i, i’) \) is a decreasing function on the well-ordered set ‘\( \mathcal{W} \)’ which updates the validation counter \( i \) into \( i’ \).

Validate Simple & Conjunctive Bound Candidates. In this case, both the bound candidate \( m \) and the validation counter \( i \) are \( 1 \)-dimensional scalars and belong to \( \mathbb{N} \). Then, \( i \succ m(x) \) becomes \( i \geq m(x) \), \( i \succ \bot \) becomes \( i > 0 \), and the decreasing function \( \text{Dec}(i, i’) \) becomes \( i’ = i - 1 \). Figure 2 shows an example of the validation task transformed from the program in Figure 1, where \( m(x) \) can be instantiated using the learned loop bound function.

Validate Lexicographic Bound Candidates. For an \( n \)-dimensional lexicographic loop bound \( m \), the well-ordered set \( \mathcal{W} \) is a set of \( n \)-dimensional vectors, of whom each dimension belongs to \( \mathbb{N} \). The bottom element \( \bot \) is the zero-valued vector \( 0 \). For two vectors \( v, v’ \in \mathcal{W} \), the well-founded relation \( >_n \) over them is defined as

\[
v >_n v’ \triangleq \exists i > 0. v[i] > v’[i] \land \forall j \in (i,n]. v[j] = v’[j].\]

For example, we have \( (1, 2) >_2 (2, 1) \) and \( (2, 1) >_2 (1, 1) \) in the \( 2 \)-dimensional \( \mathcal{W} \).

In the transformation, we employ \( n \) variables from \( 1 \) to \( n \) to represent the \( n \)-dimensional validation counter vector \( i \). The decreasing function \( \text{Dec}(i) \) can be described as follows. We check the value \( i[j] \) for \( j = 1 \) to \( n \) until we find \( j^* \) such that \( i[j^*] \succ 0 \). Then, we decrease \( i[j^*] \) by 1. For all \( 1 \leq j < j^* \), we reset \( i[j] \) to a value greater than or equal to \( m[j^*] \). After executing these steps, the value of \( i \) becomes \( i’ \). We can easily prove that \( i’ > i \).

For example, the program in Figure 5 is the validation task transformed from the program in Figure 4 with a 2-dimensional \( \text{LexLB} \), where lines 5 to 9 encode the 2-dimensional decreasing function \( \text{Dec}(i) \).

5.3 Safety Checker
Figure 6 demonstrates our loop bound learning and validation framework. In line 4 of algorithm 3, after the bound candidate is learned, a validation task will be created and delivered to the safety checker for bound validation.

Finding an appropriate loop invariant is a basic way for the safety checker to prove the problem with loops safe. In our work, we use a data-driven loop invariant learning approach [16]. Similar to bound learning, most of the data-driven approaches for loop invariant synthesis [16, 22, 32, 42] also use program states at the loop header as their data examples. There are two sets of data examples for invariant learning \( \mathcal{S}^+ \) and \( \mathcal{S}^- \). Consider the program (5). \( \mathcal{S}^+ \) includes those program states which are reachable from the loop precondition \( \text{Pre}(x) \land i \succeq m(x) \), and \( \mathcal{S}^- \) includes those program states from which there exists a reachable trace leading to a failed assertion \( \neg(i > \bot) \).

The safety property requires no state in both \( \mathcal{S}^+ \) and \( \mathcal{S}^- \). Otherwise, let \( s \in \mathcal{S}^+ \cap \mathcal{S}^- \). According to the definition, we can easily construct an error trace from the loop precondition to the failed safety property assertion through the state \( s \) and report a bound validation error (the gray edge from bound validation to bound learning in Figure 6). When \( \mathcal{S}^+ \cap \mathcal{S}^- = \emptyset \), we use [16] with \( \mathcal{S}^+ \) and \( \mathcal{S}^- \) to learn a loop invariant hypothesis. If the safety checking passes with the hypothesis, the termination will be reported (the right-most outedge from bound validation in Figure 6). In the other case, \( \mathcal{S}^+ \) and \( \mathcal{S}^- \) will be extended to refine the invariant hypothesis in the next round (the gray cycle between invariant examples and the safety checker in bound validation in Figure 6).
The query (6) asks whether the loop body is still feasible after $k$ times of loop iterations. If this query is satisfiable, there exists a $k$-steps counterexample trace where the number of loop iterations exceeds the initial value of $m(x_0)$. Then, we return this counterexample trace. If no counterexample is found in the above quick checking procedure, we invoke a safety verifier to exhaustively check $\Phi$.

In our motivation example, the loop bound candidates $m_1$, $m_2$ and $m_3$ can be easily refuted by quick bound checking with $k^* \leq 1$.

### 5.5 Two-way Example Sharing

Section 5.3 shows that both bound learning examples and invariant learning examples are the program states at the loop header. The similarity inspires us to investigate their potential connections.

In our termination analysis framework, we design a two-way data sharing between bound and invariant examples, i.e., the gray edges in Figure 6. Firstly, the bound examples can be transformed into invariant examples in the bound validation. Secondly, the error traces provided by the bound validation to refute the bound candidate can be conveyed and mutated to generate more bound learning examples.

**Transform Bound Examples.** Recall that the examples in dataset $\mathcal{H}$ for the bound learning are in the form of a pair $(x, \text{idc})$ where $x$ is a program state at loop header, and $\text{idc}$ is the remaining iteration number for $x$ in an executing trace.

The main differences between bound learning examples and invariant learning examples mainly lie in two aspects:

1. Comparing to the original program (4), the program states in validation task (5) have a particular decreasing counter $i$.
2. Learning invariant needs two sets, i.e., $\mathcal{S}^+$ and $\mathcal{S}^-$. However, bound learning only uses one set, i.e., $\mathcal{H}$.

In our approach, attached with a concrete value of $i$, each bound learning example $(x, \text{idc})$ can be transformed into an invariant learning example $s^+ \in \mathcal{S}^+$ and an invariant learning example $s^- \in \mathcal{S}^-$. Let us explain our idea with a 1-dimensional bound $m$ for easy understanding. Suppose the current bound candidate is $m(x)$ and bound learning example $(x, \text{idc})$ is from an actual executing trace:

$$(x_0, \text{idc}) = (x_1, \text{idc}_1) \ldots (x_k, \text{idc}_k)$$

where the iteration down counter $\text{idc}_t = k - t + 1$ for all $t \in [0, k]$.

We can construct a program state example $s^+_t$ for program (5) with $x_0$ and a validation counter $m_0 = m(x_0)$. Obviously, because the program state $x_0$ is reachable from the precondition in program (4), the program state $s^+_0$ is reachable from the precondition in program (5).
We implement a prototype tool called \textit{ddlTerm} with three different backend solvers, i.e., \textit{FreqHorn}. We compare it with the state-of-the-art termination analysis tools: \textit{MuVal}, \textit{UAutomizer}, and \textit{AProVE}. We also calculated the average time spent on solving each benchmark. Our \textit{ddlTerm} needs 5.43s, while \textit{FreqTerm} needs 9.20s on average (41% improved), \textit{UAutomizer} needs 17.29s (69% improved), \textit{MuVal} needs 9.05s (40% improved), and \textit{AProVE} needs 23.56s (77% improved).

We also present more detailed numbers of solved cases in Table 2 for the baselines. \textit{ddlTerm} and baselines have their own advantages on different benchmarks because of the different technical routes. However, generally speaking, there are many cases that \textit{ddlTerm} can solve while the baseline cannot, and a small number of cases that the baseline can solve while \textit{ddlTerm} cannot.

Overall Comparison. We present the curve graph of cumulative running time for solved benchmarks of all tools in Figure 8. In this figure, it is easy to find that our \textit{ddlTerm} has both stronger solving ability and relatively higher efficiency compared with \textit{FreqTerm}, \textit{UAutomizer}, \textit{MuVal}, and \textit{AProVE}.

Separate Comparison. For each baseline, we present a scatter plot in Figure 7. Each point represents the running time for one specific benchmark of \textit{ddlTerm} (x-axis) and the baseline (y-axis). If the point reaches the top or right bound in the figure, it means the baseline or \textit{ddlTerm} fails in solving this benchmark. In these figures, the more the points are close to the top-left corner, the better our tool is. As we can see, \textit{ddlTerm} significantly outperforms \textit{FreqTerm}, \textit{UAutomizer}, \textit{MuVal}, and \textit{AProVE} on the number of solved benchmarks and the distributions of running time.

### 6.2 Effectiveness Evaluation

We also conduct several other experiments to evidence the effectiveness of our approaches.

Effectiveness of Bound Learning Algorithms. In this part, we evaluate the solving ability gain, which benefits from our bound learning methods. Figure 9 shows the number of additionally solved benchmarks when a new bound learning algorithm is applied. In this figure, the yellow part with the northwest pattern represents the cumulative number of solved benchmarks only using the simple loop bound learning (no lexicographic and no conjunctive). The purple part with the northeast pattern, the red part with the vertical pattern, and the green part with the horizontal pattern respectively represent the cumulative numbers of solved benchmarks when the conjunctive loop bound learning (no lexicographic but conjunctive), the lexicographic simple loop bound learning (lexicographic but no conjunctive) and the lexicographic conjunctive loop bound learning (lexicographic and conjunctive) are used. In total, the simple bound learning method solved 92 benchmarks, and the conjunctive bound learning method, the lexicographic simple bound learning method, and the lexicographic conjunctive bound learning method respectively solved 27%, 9%, and 6% more benchmarks than their former.

Effectiveness of Quick Bound Checking. Recall that both the quick bound checking and the safety verifier can provide a counterexample...
Table 2: General result for efficiency evaluation

<table>
<thead>
<tr>
<th>Tool</th>
<th>#Solved.</th>
<th>#Both Solved.</th>
<th>#ddlTerm Only.</th>
<th>#Baseline Only.</th>
<th>Time(s)</th>
<th>Time on Solved(s)</th>
<th>Avg. T. on Solved(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4818</td>
<td>738</td>
<td>5.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 7: Separate comparison on running time (x-axis represents #Solved and y-axis represents each baseline)

Figure 8: Cumulative time for solved benchmarks

Figure 9: Comparison on different bound learning strategies

to refine the incorrect loop bound candidate. We count the number of these refinement rounds during the termination analysis. We also track the counterexample provider of each refinement, i.e., either the quick bound checker or the safety checker.

The stacked histogram in Figure 10 shows the experimental results. To make the histogram clearer, we use two sub-figures with different scales for the number of refinements (y-axis). The red part in each bar represents the number of refinements advised by the quick bound checking, and the blue part represents the number of refinements advised by the safety verifier. The filled bars represent the successfully proved benchmarks, and the hollow bars represent benchmarks that are not proved. In this figure, it is obvious that our quick bound checking is generally working during the analysis. It also takes a significant proportion in the refinements for loop bound candidates.

Next, we investigate the solving ability enhancement and the time reduction benefiting from the quick bound checking. The quick bound checking is disabled in the baseline. The results are shown in Table 3. In this table, column "#Sol." represents the numbers of benchmarks solved by dd1Term with or without quick bound checking (Q.B.C.). With quick bound checking, dd1Term solves 29 more benchmarks. Notice that the other columns on the right represent the result on 107 benchmarks both strategies solved. Column "T." represents the total running time. Columns "T.I" and "T.Q" represent the time used by the safety verifier and the quick bound checking, respectively. Column "#R" represents the numbers of total numbers of the loop bound refinements. Columns "#R.I" and "#R.Q" represent the numbers of refinements advised by the safety verifier and the quick bound checking, respectively. With quick bound checking, dd1Term not only solves 29 more benchmarks but also reduces 186s spent by safety verifier (46%) and 64s in total (9%). Although it takes more iterations on loop bound refinements with quick bound checking, according to the improvement of the overall running time, we can still conclude that quick bound checking is necessary and makes our approach more efficient.

Effectiveness of Two-way Data Sharing. With our two-way data sharing, the safety checker obtains the transformed data examples
from the bound learning, and the bound learner also benefits from the counterexamples generated by the safety checker. In this experiment, we disable the data sharing in each direction and evaluate our approach.

Table 3: Comparative results of quick bound checking

<table>
<thead>
<tr>
<th>Strategy</th>
<th>#Sol.</th>
<th>On 107 benchmarks both solved.</th>
<th>T.s</th>
<th>T.I(s)</th>
<th>T.Q(s)</th>
<th>#R.</th>
<th>#R.Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Q.B.C.</td>
<td>136</td>
<td>631</td>
<td>219</td>
<td>79</td>
<td>355</td>
<td>112</td>
<td>243</td>
</tr>
<tr>
<td>Without Q.B.C.</td>
<td>107</td>
<td>695</td>
<td>405</td>
<td>-</td>
<td>210</td>
<td>210</td>
<td>-</td>
</tr>
</tbody>
</table>

Results are shown in Table 4. The first row, “Enable Both”, represents the strategy where both two directions of data sharing are both enabled. The second row, “Disable T.B.E.”, means we disable the transformation of bound examples to invariant examples. “Disable T.B.E.” is 8% slower than "Enable Both" in average time. The third row, “Disable M.E.T.”, means we disable the mutation test on error trace. Actually, to make the interactions between bound learning and validation go on, we cannot fully prevent the data sharing from the safety checker to the bound learner, i.e., the conveyance of error traces still exists. “Disable M.E.T.” solves one benchmark less and is 17.7% slower in average time than “Enable Both”. We can conclude that our two-way data sharing mechanism between bound learning and validation makes our approach more efficient.

Table 4: Comparative results of two-way data sharing

<table>
<thead>
<tr>
<th>Strategy</th>
<th>#Solved.</th>
<th>Time(s)</th>
<th>T. on Sol.(s)</th>
<th>Avg. T. on Sol(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable Both</td>
<td>136</td>
<td>4818</td>
<td>738</td>
<td>5.43</td>
</tr>
<tr>
<td>Disable T.B.E.</td>
<td>136 (0)</td>
<td>4882</td>
<td>802</td>
<td>5.90 (8.0%)</td>
</tr>
<tr>
<td>Disable M.E.T.</td>
<td>135 (-1)</td>
<td>5091</td>
<td>891</td>
<td>6.60 (17.7%)</td>
</tr>
</tbody>
</table>

8 CONCLUSION

We develop a data-driven loop bound learning approach for termination analysis. We propose a series of data-driven loop bound learning algorithms, i.e., simple loop bound learning, conjunctive loop bound learning, and lexicographic loop bound learning. With a combination of these learning algorithms, our approach is able to prove the termination of complicated programs with non-linear loop bounds. We also propose a quick bound checking method to efficiently refute the incorrect loop bound by a counterexample. We also design two-way data sharing to bridge the bound learning, and the bound learner also benefits from the counterexamples generated by the safety checker. In this experiment, we disable the data sharing in each direction and evaluate our approach.

7 RELATED WORK

Proving Termination. Termination analysis has always been a research hotspot in the field of program verification, and there are a lot of existing technologies. Most of them [3, 6, 19, 25, 28, 33, 35] rely on constraint solvers to synthesize the termination arguments, e.g., a ranking function together with its support invariant. With these technologies, lots of tools emerged, e.g., Terminator (together with its successor T2) [7, 11, 12], Armc[34], Tan[23], HipTNT+ [27], Ultimate Automizer [20] and so on.

Differing from the technologies above, there are some "guess-and-check"-style methods [14, 26, 31, 41]. They employ a lightweight approach to obtain the likely termination arguments and employ an off-the-shelf checker to validate them. DynamiTe [26] tries to get a ranking function candidate from dynamic execution traces and employs a program verifier to check it. Its method is different from ours, and it focuses on non-linear programs. FreqTerm [14] uses a syntax-guided approach to guess a loop bound and checks the candidate by a CHC solver. [41] learns different forms of loop bounds by combining several affine expressions inferred from the examples provided by the constraint solver. TpT [31] employs quadratic programming to infer a loop bound from testing data, which is closest to our approaches. However, TpT only learns a simple loop bound. Its applicability is thus limited.

Data-Driven Methods in Verification. Data-Driven methods have received more and more attention in recent years. In the field of verification, several applications have been appeared, particularly in loop invariant synthesis [16, 29, 37, 38, 42, 43] and termination proving [24, 26, 31, 41]. The ways these technologies deal with the data are quite different. Among these technologies, [16, 24, 42] use adapted decision trees to process the data. [29, 37] use SVM for data classification, [38, 43] use natural networks to generate logical expressions. [31] uses quadratic programming, [41] uses linear interpolation, and [26] uses constraint solving. In our approaches, we use the clustering algorithm to group the dataset and then employ convex optimization to generate the candidate loop bounds. We also employ the optimization algorithm to find a better combination of the bound candidates.
REFERENCES


