IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 30, 2021, accepted November 21, 2021, date of publication November 25, 2021,
date of current version December 6, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3130569

Directed Model Checking for Fast Abstract Reachability Analysis

NAKWON LEE[1], YUNHO KIM[2], MOONZOO KIM[1], DUKSAN RYU[3], AND JONGMOON BAIK[1], (Member, IEEE)

[1]School of Computing, Korea Advanced Institute of Science and Technology, Yuseong-gu, Daejeon 34141, Republic of Korea
[2]Department of Computer Science, Hanyang University, Seongdong-gu, Seoul 04763, Republic of Korea
[3]Department of Software Engineering, Jeonbuk National University, Jeonju-si, Jeollabuk-do 54896, Republic of Korea

Corresponding author: Duksan Ryu (duksan.ryu@jbnu.ac.kr)

This work was supported in part by the Ministry of Science and ICT (MSIT), Korea, under the Information Technology Research Center (ITRC) Support Program supervised by the Institute of Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2021-2020-0-01795; in part by IITP through MSIT under Grant 2021-0-00905; in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) by MSIT under Grant NRF-2017M3C4A7068177; and in part by the NRF Grant through MSIT under Grant NRF-2020R1F1A1071888, Grant NRF-2019R1G1A1005047, Grant NRF-2019R1A2B5B01069865, Grant NRF-2021R1A2C2009384, Grant NRF-2020R1C1C1013996, and Grant 2021R1A5A1021944.

ABSTRACT We propose a novel technique (TOUR) to improve both bug detection ability and verification speed of ARMC by detecting a target path quickly. The key idea of TOUR is an *error location directed search* that utilizes *the distance to an error location* and *function call context* at runtime. TOUR applies four different distance metrics and a distance metric selection heuristic using static features of a target program. We have extensively evaluated TOUR on 3,042 real-world C programs in a software verification competition benchmark. The experiment results show that TOUR, due to its error location directed search, finds bugs in 20% more programs in 11% less model checking time than the state-of-the-art ARMC technique (i.e., block-abstraction memoization) for 354 buggy programs. Also, TOUR verifies 15% more programs within 15% less model checking time than the block-abstraction memoization for 652 *complex* clean programs.

INDEX TERMS Software verification, software testing, symbolic model checking, abstract reachability, interprocedural analysis, directed search.

I. INTRODUCTION

Abstract Reachability-based Model Checking (ARMC) techniques verify whether a program can reach an error location (i.e., reachability) in an abstraction of the program to mitigate the state-explosion problem [1]–[3]. Although many studies reduce the ARMC model checking time [2]–[5], time-outs are the reason for 75% (772 out of 1,031) of cases where the state-of-the-art ARMC technique [6], combining their benefits, fails to verify in our preliminary experiment on real-world C programs.

Existing ARMC techniques are slow to detect a path reaching an error location (i.e., *a target path*) due to an inefficient search method. A target path is either an actual bug path (i.e., a counterexample of the reachability property) or a false alarm triggering the refinement of the abstraction [7]. However, existing ARMC techniques explore an abstract state space with an inefficient search method, e.g., depth-first search, for detecting a target path.

In this paper, we propose a novel ARMC technique TOUR (effecTive error lOcation directed search via an interprocedUral Runtime distance calculation), which is *the first approach* to apply a directed search for fast target path detection in ARMC for interprocedural programs. TOUR explores an abstract search space by first selecting a state with *the shortest distance* to an error location. Note that TOUR calculates the distance by considering the *function call context at runtime* because a function in an interprocedural program has various function call contexts. TOUR calculates distances of two different types – one is dependent on the function call context (**rel-dist**) and the other is independent from the context (**abs-dist**): ① **rel-dist** is the sum of the distance from the function call abstract state to the error location and the distance from the abstract state to the function exit location; and ② **abs-dist** is the distance from the abstract state to the error location without considering the function call context.

The associate editor coordinating the review of this manuscript and approving it for publication was Antonio Piccinno.

TOUR labels each program location with **abs-dist** and **exit-dist** where **exit-dist** is the distance from a location to the function exit location.

To further improve the direct search strategy of TOUR, we propose and compare the four distance metrics. Then, we developed a *program-specific* distance metric selection method by learning from historical results of TOUR using distance metrics.

The contributions of this paper are as follows:

1) *TOUR is the first error location directed search that speed up ARMC*: TOUR is the first approach that applies an error location directed search to ARMC for fast target path exploration on an interprocedural program, which improves bug detection ability and verification speed. We successfully apply TOUR to two existing ARMC techniques Block-Abstraction Memoization for interprocedural program analysis (BAM) [6] and Lazy Abstraction (LA) [8].
2) *TOUR improves ARMC's bug detection ability*: BAM using TOUR finds bugs in 20% more programs within 11% less total CPU-time for 354 real-world buggy C programs.
3) *TOUR improves ARMC's verification speed*: For 652 complex real-world clean C programs classified by the Cyclomatic Complexity, BAM using TOUR verifies 15% more programs within 15% less total CPU-time.
4) *An effective distance metric selection method for target programs*: BAM using TOUR with the distance metric selection solves 10% more programs within 22% less total CPU-time for 3,042 real-world C programs compared to the case using the worst single metric.

The rest of the paper is organized as follows. Section III introduces background knowledge for understanding TOUR. Section IV describes TOUR by dividing it into two main parts. Section V describes the experimental setting including research questions. Section VI demonstrates the experimental results for the research questions. Section II presents related work and Section VII concludes the paper with future work.

II. RELATED WORK

A. DIRECTED SEARCH

Although several program analysis techniques (e.g., model checking with distance-preserving abstractions or symbolic executions) utilize directed search strategies, TOUR is the *first ARMC technique* that applies directed search for fast target path detection on interprocedural programs.

Directed model checking with distance-preserving abstractions utilizes directed search, but it targets a state transition system, not a high-level C program [9]. It abstracts a state transition system by grouping states with the same distance to an error state to mitigate the state-explosion problem. TOUR is naturally distance-preserving because TOUR does not abstract the program counter variable of a target C program (i.e., the program location) for which TOUR defines the distance. The distance-preserving abstraction technique uses only a single distance metric, while TOUR uses and compares four different distance metrics.

Shortest-Distance Symbolic Execution (SDSE) applies directed search to symbolic execution, while TOUR applies a directed search to ARMC [10]. SDSE computes the distance to an error location in an interprocedural C program and conducts the shortest error distance first search. SDSE applies a directed search for fast bug path detection, but TOUR improves both bug detection ability and verification speed for ARMC. SDSE also uses a single distance metric (i.e., the number of edges), while TOUR uses four distance metrics.

B. SPEED UP ARMC

Although there are several techniques to speed up ARMC (e.g., lazy abstraction, block-encoding, and block-abstraction memoization), TOUR improves the ARMC's speed further as a *complementary* technique.

TOUR improves the speed of Block-Abstraction Memoization for interprocedural program analysis (BAM), even though BAM is the state-of-the-art ARMC technique combining the benefits of existing optimization techniques [6]. BAM combines three existing speed-up techniques for ARMC, i.e., Lazy Abstraction (LA), Block-Encoding (BE), and block-abstraction memoization. LA reuses the abstract states that are not relevant to a false alarm after removing the false alarm to save time for state exploration [8], [11]. BE takes advantage of the efficiency of constraint solving techniques: it is more efficient to compute an abstract state for a large block of multiple statements at once than to compute abstract states of the multiple statements separately with multiple computations [2], [3]. Block-abstraction memoization stores constructed substate graphs to a cache and reuses them later to save state exploration time [5]. TOUR complements the three speed-up techniques because it tries to search as few states as possible instead of reusing or efficiently constructing them.

C. ABSTRACT DOMAIN

TOUR is *independent* of abstract domains and thus can be applied to ARMC techniques with various abstract domains.

We successfully apply TOUR to BAM, which uses two different abstract domains (i.e., predicate domain and explicit-value domain) [6] because TOUR is independent of abstract domains. The predicate domain is a widely used abstract domain for ARMC of C programs [3], [11]–[17]. The predicate domain presents an abstract data state formula as a set of predicates over program variables. It is also effective for unbounded loop analysis because it easily checks whether a loop fixpoint is reached [13], [18]. The explicit-value domain uses an explicit-value assignment of some program variables (not all the program variables) as an abstract data state formula [19]–[21]. It is effective to verify a program in which the reachability of the program is primarily related to some specific variables. TOUR is independent of the domains because the domains affect only the abstract data state formula and maintain program locations as nonabstracted. We also apply TOUR to LA which uses the predicate abstract domain [8].

Although TOUR can be applied to LAI, it may not be effective to apply TOUR to Lazy Abstraction with Interpolants (LAI). LAI is an ARMC technique that uses the empty abstract domain for the entire model checking procedure [22]–[24]. LAI cannot compute abstract data states during state exploration because of the empty abstract domain. It computes abstract data states during the removal of false alarms. TOUR saves time during state exploration, which is not effective for LAI because LAI does not have much time for exploration due to the empty abstract domain.

Applying TOUR to a combination-based model checking technique (e.g., CPA-seq and PeSCo) is not effective because the technique may assign a short time to an ARMC component constituting the technique. Combination-based techniques assume that a suitable abstract domain for model checking of a program may differ program by program. CPA-seq sequentially invokes model checking techniques, including ARMC components and other model checking techniques, to improve the chance of solving a program [25], [26]. PeSCo [27], [28] extends CPA-seq by dynamically changing the order of constituting techniques by predicting the probability of solving. Both CPA-seq and PeSCo assign a short time limit (only 11% (=100s/900s) of the total time limit) to an ARMC component because they think that a technique is not suitable for a program if it cannot solve the program quickly. However, TOUR helps ARMC solve a complex program with reasonable time, while ARMC fails to verify the program due to time-out.

III. BACKGROUND

This section explains background knowledge for understanding TOUR, including programs, abstract states, and function call context.

A. PROGRAMS

A program consists of *locations* that model program counter variables and *operations* executed when a control moves from a location to another location. Figure 1 shows an example C program consisting of main and f, which is labeled with program location numbers. We assume simple interprocedural C programs with limited operations: assignment operations (e.g., L5 → L7), branch condition operations (e.g., L2 → L3, L2 → L8), function call operations (e.g., L3 → L21), function return operations (e.g., L22 → L4), and dummy operations (e.g., L7 → L10) that move program counters without changing program states. A function has one *function entry location* (e.g., L1, L21) and one *function exit location* (e.g., L12, 22). The location L11 is the *error location* which unwillingly terminates the program by invoking the `abort` function.

B. ABSTRACT STATES

An *abstract state* consists of a corresponding location and an abstract data state describing reachable program states under an abstraction [3]. Figure 2 shows an example abstract state graph for the program in Figure 1, abstracting all operations as nondeterministic (i.e., any program state is reachable after executing an operation, represented as the formula *True*). A graph node denotes an abstract state with the corresponding location (left) and the abstract data state formula (right). A graph edge denotes a *successor/predecessor* relationship between two abstract states. It also represents the operation between the location of the successor abstract state and that of the predecessor abstract state. The *initial abstract state* s_1 corresponds to the function entry location of the main function (i.e., L1) and always has *True* as the abstract data state formula. A *target path* is the sequence of abstract states from the initial abstract state to an abstract state corresponding to an error location with a non*False* abstract data state formula (e.g., $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \rightarrow s_7 \rightarrow s_8 \rightarrow s_9 \rightarrow s_{10}$).

```
    void main(int *x, int k){
L1:   int i;
L2:   if(x[1]>x[2]){
L3:     f(x[1]);
L4:     if(k==0){
L5:       x[2]=x[1];
        }else{
L6:       x[1]=x[2];
L7:     }
      }else{
L8:     f(x[2]);
L9:   }
L10:  if(x[1]>x[2]){
L11:    abort();
      }
L12:}
    void f(int i){
L21:  int a=i;
L22:}
```

FIGURE 1. An example C program.

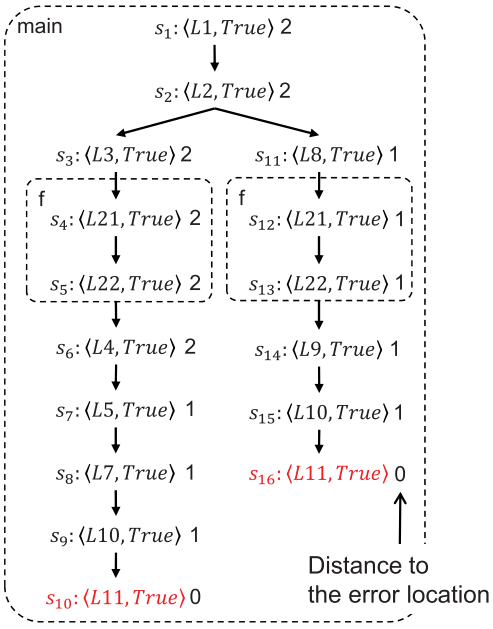


FIGURE 2. An example abstract state graph for the example program in Figure 1.

C. FUNCTION CALL CONTEXT

A *function call context* of an abstract state s is the function call abstract state of s denoted as `call`(s). For example in Figure 2, the function call context of s_4 and s_5 is s_3 because they correspond to the function f invoked at s_3.

Abstract states corresponding to the same location can have different distance values on different function call contexts [10]. For example in Figure 2, we annotate the distance from each abstract state to the error location in terms of the number of basic blocks (i.e., the number of branch condition operations). The abstract states s_4 and s_{12} have different distance values due to the different function call contexts, although they correspond to the same location L21.

An error location-directed search considering the function call context can find a target path faster than other search methods. For example, in Figure 2, the shortest distance first directed search finds the target path from s_1 to s_{16} by calculating the distance on function call contexts. It is shorter than the target path from s_1 to s_{10} detected by the depth-first search with a true branch first manner.

IV. EFFECTIVE ERROR LOCATION DIRECTED SEARCH VIA AN INTERPROCEDURAL RUNTIME DISTANCE CALCULATION

Figure 3 shows the graphical overview of TOUR. TOUR consists of two main parts: *(1) interprocedural runtime distance calculation* (Section IV-A) and *(2) distance metric selection using a model for program-specific selection* (Section IV-B).

- (1) TOUR calculates the two types of distance values, i.e., rel-dist and abs-dist, at runtime. TOUR annotates information necessary for the calculation to a target program before runtime. The annotated information is the abs-dist of a program location to an error location and the distance from a location to the function exit location (exit-dist). TOUR annotates the information according to the selected distance metric. TOUR tracks the function call context, i.e., which abstract state is a function call abstract state of which abstract states, during the run-time for the rel-dist calculation.
- (2) TOUR selects a distance metric among four distance metrics using a model for program-specific distance metric selection. We propose four distance metrics for TOUR regarding the characteristics of existing ARMC techniques. TOUR generates a model for the program-specific distance metric selection by learning from historical results with static program features. TOUR can select the best distance metric among the four distance metrics for an ARMC technique and a given target program.

A. INTERPROCEDURAL RUNTIME DISTANCE CALCULATION

TOUR calculates the two distance values (rel-dist and abs-dist) that complement each other. An abstract state has no abs-dist if its distance is only valid under a function call context. The rel-dist value of an abstract state is not the shortest distance if the abstract state has a valid abs-dist value smaller than the rel-dist value. TOUR selects the smaller distance of rel-dist and abs-dist as the abstract state distance.

The abs-dist value of an abstract state is the abs-dist value of the corresponding program location. The rel-dist value of an abstract state is the sum of the exit-dist and the distance of the corresponding function call of the abstract state. Each abstract state tracks its corresponding function call to the abstract state at runtime for rel-dist calculation. TOUR annotates the information (i.e., exit-dist and abs-dist) necessary for calculating the distance to a target program for runtime calculation efficiency.

In the remainder of this section, we assume the number of basic blocks (i.e., the number of branch edges) is the selected distance metric for TOUR.

1) THE exit-dist AND abs-dist ANNOTATION

The exit-dist value of a program location is the shortest distance from the location to the function exit location according to the selected distance metric. TOUR calculates and annotates exit-dist for each function of a target program by using a single-source shortest distance calculation algorithm in the backward direction. The function exit location is the single source of the algorithm.

Figure 4 shows an example exit-dist annotation for the example program in Figure 1. It describes the control-flow graph of the example program labeled with exit-dist, where a node represents a program location. TOUR applies the exit-dist annotation algorithm for each function f and main in order because f has no callee function and is the callee function of main. Since we use the number of branch edges as the distance metric, the distance weight of an edge is one for the edges, L2 $\rightarrow$ L3, L2 $\rightarrow$ L8, L4 $\rightarrow$ L5, L4 $\rightarrow$ L6, L10 $\rightarrow$ L11, and L10 $\rightarrow$ L12, and zero for the other edges. TOUR initializes the exit-dist value of the function exit location as zero and that of other locations as ∞. The single source of the algorithm for the function f is L22 and that for the function main is L12. The exit-dist value of a location l is the sum of the exit-dist of l's successor location and the edge weight between them. The algorithm ignores function return edges (e.g., L22 $\rightarrow$ L4 and L22 $\rightarrow$ L9). TOUR calculates the exit-dist value of a function call location (e.g., L3 and L8) as the sum of the exit-dist value of the corresponding function return location (e.g., L4 and L9 respectively) and the function's shortest distance (e.g., the function f). A function's shortest distance means the shortest distance from the function entry to the function exit location (i.e., the exit-dist of the function entry location, e.g., L21). Thus, TOUR calculates the exit-dist for a callee function first.

The abs-dist value of a program location is the shortest distance from the location to an error location according to the selected distance metric. TOUR calculates and annotates abs-dist to a target program using a single source shortest

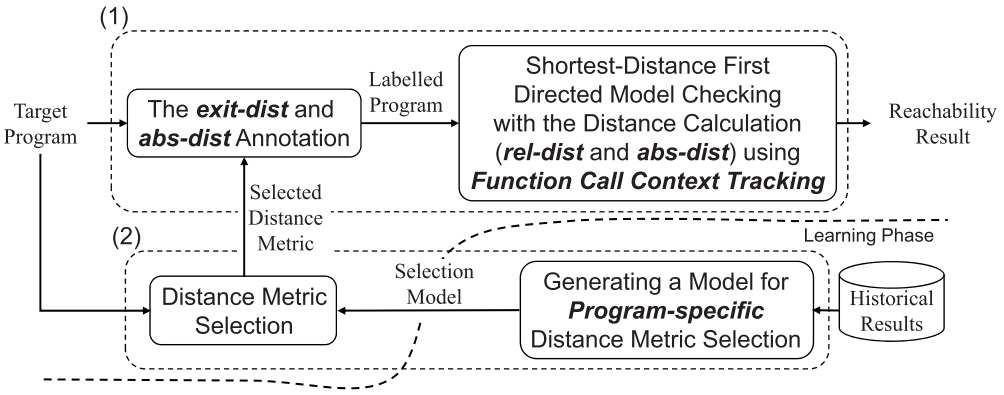


FIGURE 3. Overview of model checking using TOUR (effectTive error lOcation directed search via an interprocedUral Runtime distance calculation).

distance algorithm in the backward direction. An error location is the single source of the algorithm.

Figure 4 shows an example **abs-dist** annotation for the example program in Figure 1. It describes the control-flow graph of the example program labeled with **abs-dist**. TOUR initializes the **abs-dist** value of the error location (**L11**) as zero (because it is the single source of the algorithm) and that of other locations as ∞. The edge weights are the same as the **exit-dist** annotation algorithm uses. The **abs-dist** value of a location l is the sum of the **abs-dist** of l's successor location and the edge weight between them. The algorithm ignores the function return edges, **L22** $\rightarrow$ **L4** and **L22** $\rightarrow$ **L9**. TOUR calculates the **abs-dist** value of a function call location (e.g., **L3** and **L8**) as the sum of the **abs-dist** value of the corresponding function return location (e.g., **L4** and **L9**, respectively) and the function's shortest distance (e.g., the function **f**). Since the **abs-dist** calculation algorithm uses the **exit-dist** values (as the function's shortest distance), TOUR calculates **exit-dist** for all functions first and then calculates **abs-dist**.

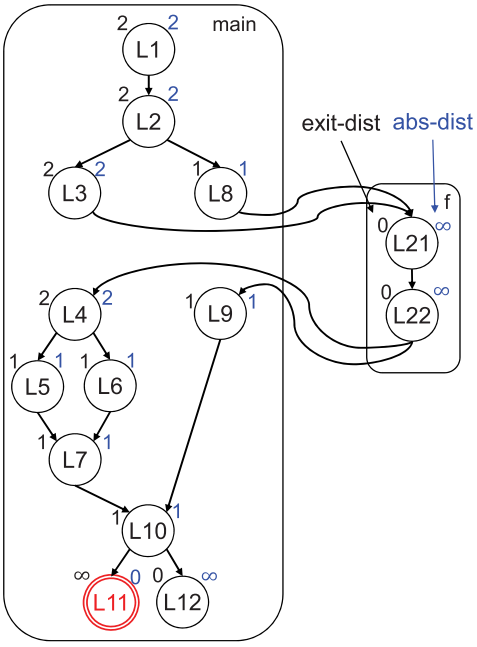


FIGURE 4. The control-flow graph of the example program in Figure 1 labeled with exit-dist and abs-dist.

2) DISTANCE CALCULATION USING FUNCTION CALL CONTEXT TRACKING

TOUR calculates **abs-dist** and **rel-dist** of an abstract state using the annotated information (i.e., **abs-dist** and **exit-dist**) of the program locations and a function call context at runtime. The **abs-dist** of an abstract state is the same as the **abs-dist** of the corresponding program location. The **rel-dist** value of an abstract state is the sum of the **exit-dist** of the corresponding location and the distance of the function call of the abstract state. The distance of an abstract state is the smaller distance of **abs-dist** and **rel-dist**.

TOUR tracks the function call context (i.e., the corresponding function call abstract state) in each abstract state. TOUR assigns the function call abstract state of an abstract state s' ($\texttt{call}(s')$) as follows using the predecessor abstract state s and the program operation op between s and s':

- If s' is the initial abstract state (no successor abstract state s and no op)
$$\texttt{call}(s') = nil$$
- If op is a function call operation
$$\texttt{call}(s') = s$$
- If op is a function return operation
$$\texttt{call}(s') = \texttt{call}(\texttt{call}(s))$$
- Otherwise
$$\texttt{call}(s') = \texttt{call}(s)$$

Finally, the distance of an abstract state s (i.e., $\texttt{dist}(s)$) is calculated as follows if $s \neq nil$ using $\texttt{loc}$, the mapping

function from an abstract state to the corresponding program location:

$$\texttt{dist}(s) = \min(\texttt{loc}(s).\textsf{exit-dist} + \texttt{dist}(\texttt{call}(s)), \texttt{loc}(s).\textsf{abs-dist}), \quad (1)$$

If $s == nil$:

$$\texttt{dist}(s) = \infty \quad (2)$$

B. DISTANCE METRICS AND PROGRAM-SPECIFIC DISTANCE METRIC SELECTION

We propose four different distance metrics for TOUR because the best distance metric is probably different for each ARMC technique. We propose the distance metrics considering which program statements mainly affect the performance of each ARMC technique.

We also think that the best distance metric is different for each program, even if a distance metric is generally good for an ARMC technique. Thus, we propose a method to generate a model for program-specific distance metric selection to select the best distance metric among the four distance metrics for a target program. TOUR generates such a model by learning from historical directed ARMC results with static program features.

1) DISTANCE METRICS

The four distance metrics proposed for TOUR are a number of statements (st), a number of basic blocks (bb), a number of loop heads (lh), and a number of loop heads and function entry/exit points (lf).

- **The st metric** considers each statement equally and is widely used in model checking or symbolic execution [9], [10], [29].
- **The bb metric** weighs branch condition statements to focus on the number of basic blocks instead of the number of statements. It is effective for programs including many statements that hardly affect ARMC technique performance, e.g., an assignment statement with no operation.
- **The lh metric** weighs loop branch condition statements. It is effective for ARMC techniques that compute an abstract state for a large number of acyclic statements at once, e.g., block encoding [2], [3].
- **The lf metric** weighs loop branch condition statements and the function call and return statements. It is effective for ARMC techniques that explore each function's search space separately, e.g., block-abstraction memoization [5], [6].

We define each metric as a control-flow edge (i.e., statement) weight according to their operation. The edge weight function W of an edge e for each metric is defined as follows:

- For a number of statements (st)

$$W(e) = 1 \quad (3)$$

- For a number of basic blocks (bb)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation} \\ 0, & \text{otherwise} \end{cases} \quad (4)$$

- For a number of loop heads (lh)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation corresponding to a loop condition} \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

- For a number of loop heads and function entry/exit points (lf)

$$W(e) = \begin{cases} 1, & \text{if } e \text{ is a condition operation corresponding to a loop condition} \\ 1, & \text{if the predecessor of } e \text{ is a function entry location} \\ 1, & \text{if } e \text{ is a function return operation} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

2) GENERATING A MODEL FOR PROGRAM-SPECIFIC DISTANCE METRIC SELECTION

TOUR generates a model for program-specific distance metric selection by learning from historical results consisting of determining the best distance metric and static feature value of each program. A generated model selects the best distance metric among the four distance metrics by regarding the static program feature values.

Figure 5 shows the procedure for generating a program-specific distance metric selection model from historical results. The historical directed model checking results are composed of 25 static feature values and the best distance metric of each program. The 25 features represent size (Locations and Edges categories), modularity (Loops and Functions categories), data flow (Variables category), and complexity (Cyclomatic Complexity [30] category), as shown in Table 1. For features defined for a single function, TOUR uses the maximum, average, and standard deviation values of functions in a program as the representative values. We label each program in the historical results with its best distance metric in terms of the CPU time [1] among the four classes, st, bb, lh, and lf.

TOUR generates a decision tree as a selection model by learning from the historical results. Figure 6 shows an example decision tree to select a distance metric. Each internal node represents a static feature name and the decision condition for the feature to select a child node (i.e., select the left

[1]The metric with the shortest CPU time among the metrics that success the model checking without exceeding resource limits.

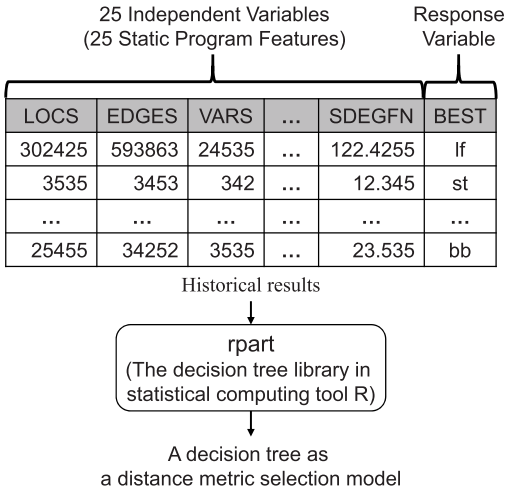


FIGURE 5. The procedure for generating a model for program-specific distance metric selection from historical results.

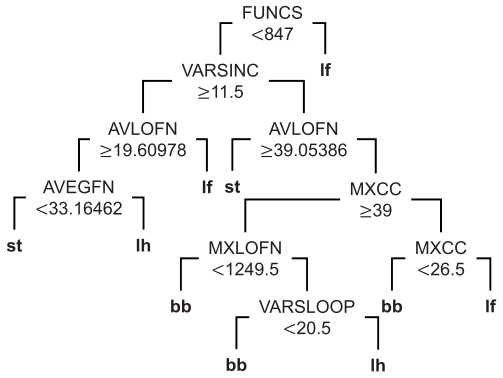


FIGURE 6. An example distance metric selection model represented as a decision tree.

child node if the decision condition is satisfied). Each terminal node represents the selected distance metric. For example, according to the decision tree in Figure 6, if a program has static feature values as FUNCS = 840, VARSINC = 11, AVLOFN = 40, and MXCC = 30, TOUR selects bb as the distance metric for the program.

TOUR uses the recursive partitioning algorithm (the rpart library in open source statistical computing tool R [32]) to build a decision tree. We use the recursive partitioning algorithm because it is widely used for multiclass classification over the past three decades and is known as accurate. As a result of optimization, we filter out historical results of less than 40 seconds of CPU time difference caused by the distance metric change.

V. EXPERIMENTAL SETUP

We designed the following four research questions to evaluate TOUR.

A. RESEARCH QUESTIONS

- **RQ1. Effectiveness of TOUR on bug detection ability:** Does TOUR improve the bug detection ability of ARMC?
  We investigate whether TOUR finds bugs in more real-world buggy C programs than existing ARMC techniques.
- **RQ2. Effectiveness of TOUR on verification speed:** Does TOUR save the verification time of ARMC?
  We investigate whether TOUR saves CPU time with two different existing ARMC techniques for real-world clean C programs.
- **RQ3. Effectiveness of the proposed distance metrics:** Which metric among the four distance metrics is the best distance metric for each ARMC technique?
  We investigate which distance metric is the best in terms of the number of solved programs and CPU time among the four distance metrics for each ARMC technique.
- **RQ4. Effectiveness of the distance metric selection:** Does TOUR select the best distance metric for each target program?

TABLE 1. Category and description of the 25 program features used for distance metric selection rule generation; Cyclomatic Complexity: the well-known McCabe's cyclomatic complexity [31].

Category	Feature name	Description
Locations	LOCS	Number of program locations
	MXLOFN	Maximum number of locations per function
	AVLOFN	Average number of locations per function
	SDLOFN	Standard deviation of the number of locations per function
Edges	EDGES	Number of control-flow edges
	MXEGFN	Maximum number of control-flow edges per function
	AVEGFN	Average number of control-flow edges per function
	SDEGFN	Standard deviation of the number of control-flow edges per function
Loops	LOOPS	Number of loops in a program
	MXLPFN	Maximum number of loops per function
	AVLPFN	Average number of loops per function
	SDLPFN	Standard deviation of the number of loops per function
Functions	FUNCS	Number of functions in a program
	MXCALLS	Maximum number of function calls per function
	AVCALLS	Average number of function calls per function
	SDCALLS	Standard deviation of the number of function calls per function
Variables	VARS	Number of relevant variables (used in conditions and their dependent variables)
	VARSASM	Number of variables used in conditions
	VARSLOOP	Number of variables used in loop exit conditions
	VARSINC	Number of variables used as increasing/decreasing count variables for for statements
	FIELDS	Number of relevant bit-field variables
Cyclomatic Complexity	MXCC	Maximum cyclomatic complexity per function
	SMCC	Sum of cyclomatic complexity per function
	AVCC	Average cyclomatic complexity per function
	SDCC	Standard deviation of cyclomatic complexity per function

We investigate whether TOUR using the program-specific metric selection model takes less CPU time than TOUR using a single metric for target programs.

B. TECHNIQUES TO COMPARE

We implemented TOUR in a form integrated into two state-of-the-art ARMC techniques, i.e., lazy abstraction and block-abstraction memoization. By applying TOUR on *the most widely used optimization method*, i.e., lazy abstraction [3], [6], [17], [19]–[22], [29], we investigate whether TOUR is generally effective in speeding up ARMC. We also investigate whether TOUR improves the practical performance of ARMC for practitioners. Thus, we investigate how much TOUR improves the bug detection ability and saves the model checking time of *a current state-of-the-art ARMC technique*, i.e., block-abstraction memoization [6].

- **LA**: Lazy Abstraction (LA) is an ARMC technique that applies different abstractions for different substate graphs [8]. Thus, LA can eliminate a false alarm by refining only the substate graph relevant to the false alarm. LA uses the depth-first search. We use LA implemented in the open-source model checking tool CPAchecker [33] (CPAchecker 2.0.1-svn). LA uses MathSAT5 [34] (MathSAT5 version 5.6.5 (63ef7602814c)) as the SMT-solver for constraint solving. We used LA for RQ1 and RQ2 as a baseline ARMC technique.
- **LA.T**: LA.T is the technique that applies TOUR to LA. LA.T uses the default parameter values of the rpart library except two parameters `maxcompete` $=$ 0 and `maxsurrogate` $=$ 0 to generate a selection model [32]. LA.T generates a distance metric selection model by excluding the target program from the historical result to avoid over-fitting. We used LA.T for RQ1, RQ2, and RQ4.
- **BAM**: Block-Abstraction Memoization for interprocedural program analysis (BAM) is a current state-of-the-art ARMC technique [6]. It extends block-abstraction memoization [5] to handle recursive programs. It combines lazy abstraction [8], predicate abstraction, explicit-value abstraction [19], large-block encoding [3], and block-abstraction memoization. We use BAM implemented in the open-source model checking tool CPAchecker [33] (CPAchecker 2.0.1-svn). While adopting the depth-first search, BAM uses SMTInterpol [35] (SMTInterpol 2.5-732-gd208e931) as the SMT solver for constraint solving. We used BAM for RQ1 and RQ2 as a baseline ARMC technique.
- **BAM.T**: BAM.T is the technique that applies TOUR to BAM. BAM.T uses the same parameter setting of LA.T for rpart to generate a selection model. BAM.T generates a distance metric selection heuristic by excluding the target program from the historical result to avoid overfitting. We used BAM.T for RQ1, RQ2, and RQ4.
- **LA.T.{st,bb,lh,lf}**: LA.T.{st,bb,lh,lf} are the techniques that apply TOUR using a single distance metric to LA without distance metric selection. We compare LA.T.{st,bb,lh,lf} in RQ3 and RQ4.
- **BAM.T.{st,bb,lh,lf}**: BAM.T.{st,bb,lh,lf} are the techniques that apply TOUR using a single distance metric to BAM without distance metric selection. We compare BAM.T.{st,bb,lh,lf} in RQ3 and RQ4.

C. TARGET PROGRAMS

- **RealWorld**: RealWorld is the set of C programs obtained from the SoftwareSystems category of SV-COMP '21 [36]. The category contains 3,184 programs from *real-world sources*, and we exclude programs with no predefined error location [37]. As a result, RealWorld includes 3,042 programs consisting of 2,688 clean programs and 354 buggy programs. We used RealWorld for all experiments. Additionally, we divided RealWorld into four subsets according to the complexity of programs.
- **RealWorld.{1,2,3,4}**: RealWorld.{1,2,3,4} are the subsets of RealWorld divided by the program complexity. We use *the SuM of the Cyclomatic Complexity per function* (SMCC) of a program to represent the complexity of the program. Cyclomatic complexity is a well-known complexity metric for programs written in an imperative programming language such as C [31]. We divided RealWorld into four subsets considering quantiles. Table 2 presents the boundary values and the number of programs in each subset. We used RealWorld.{1,2,3,4} in the experiment for RQ2 to show the increasing effectiveness of TOUR on verification speed as the complexity of programs increases.

These RealWorld programs are used for experiments because many recent model checking studies use them as verification targets [20], [26], [38]–[40]. In particular, BAM participated in SV-COMP 21' and *showed the best verification performance* (i.e., solving the largest number of verification tasks) among other participating techniques for the SoftwareSystems category. Note that the limitation of using target programs with predefined error locations is acceptable because various studies in the verification and testing fields evaluate their technique on programs with assertion statements, which are a type of predefined error location [18], [41]–[44].

D. MEASUREMENT

We measure *CPU time* and *the number of solved programs* for each technique to compare. Since we limit the model checking time for each target program, not only the CPU

TABLE 2. The statistics of RealWorld.{1,2,3,4}; SMCC: the sum of Cyclomatic Complexity per function.

Index	SMCC	Number of Programs
1	$[24, 87)$	748
2	$[87, 448.5)$	773
3	$[448.5, 1412.5)$	760
4	$[1412.5, 132302]$	761

time but also the number of solved programs shows the time efficiency of a technique.

- **CPU time**: We measure the CPU time of a model checking run, which includes the CPU time of all subprocesses that the main process invokes. The CPU time of TOUR includes the time for extracting program features. We exclude the time for generating a distance metric selection heuristic from the CPU time because a generated heuristic is used for multiple target programs. The CPU time is measured by the benchmark execution tool BENCHEXEC (see Section V-E).
- **The number of solved programs** We count the number of correctly solved programs of a technique. A technique correctly solves a buggy program if it finds a bug for the program within a resource limit. A technique correctly solves a clean program if it finishes model checking without a bug or a false alarm within a resource limit. A technique fails to solve a target program if it incorrectly solves or fails to finish due to the time-out, memory-out, or internal error of the technique.

E. MODEL CHECKING ENVIRONMENTAL SETUP

We conducted all the model checking runs by using BENCHEXEC 3.6 [45] (i.e., a benchmark execution platform used in SV-COMP '21) to ensure reliable experimental results. We used five machines with a 3.4GHz CPU and 16 GB memory for all model checking runs. All machines used Ubuntu 20.04 and OpenJDK 1.13. The resource limit for a model checking run was four CPU cores, 15 GB RAM, and up to 900 seconds of CPU time. The 15 GB RAM and up to 900 seconds of CPU time are the same resource limits of SV-COMP '21 which is considered standard [36]. We provide the implementation of TOUR, replication guide for experiments, and the raw data of the experiment results publicly in a GIT repository (see `ACCESS.README.md` file in the repository). [2]

F. THREATS TO VALIDITY

- **Internal validity:** A threat to internal validity is possible bugs in TOUR implementation and the other techniques we studied. We meticulously verified our implementations to address this threat.
  Additionally, we controlled the model checking execution environment using machines with the same specification and operating environment. Furthermore, we conducted the experiments using the benchmarking tool BENCHEXEC used in SV-COMP '21. BENCHEXEC isolates the model checking executions from the interruption of other processes executed in the same machine to obtain reliable results. Thus, we believe that the threat to internal validity is limited.
- **External validity:** A threat to external validity is the representativeness of our target programs. We expect that this threat is limited since the target programs are widely used benchmark programs and tested by many other researchers.
  We used target programs that represent general real-world C programs. RealWorld includes programs obtained from real-world software projects (e.g., Linux-device driver programs), and they cover most of the C features. Thus, we believe that the threat to external validity is also limited.
- **Construct validity:** We used two measures, i.e., the number of solved programs and CPU time. These measures have been widely and generally used performance criteria for model checking in recent studies [21], [25], [28], [29], [40], [46]. Thus, we believe that the threat to construct validity is limited.

[2] https://github.com/Nakwon-Lee/pacc_cpachecker/tree/DMCforAR

TABLE 3. The number of solved buggy RealWorld programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU time: the total model checking time for 354 buggy RealWorld programs.

	BAM	BAM.T	
Count.Solved	100	120	(↑ **20**%)
Count.Time-out	152	131	(↓ **14**%)
Count.Others	102	103	
Total CPU-time	154, 555s	136, 856s	(↓ **11**%)

VI. EXPERIMENTAL RESULT

A. RQ1: EFFECTIVENESS OF TOUR ON BUG DETECTION ABILITY

The results show that TOUR *meaningfully improves the bug detection ability* of an ARMC technique. BAM.T finds bugs in 20% (=(120-100)/100) more programs than BAM for the 354 buggy programs within 11% (=(154,555-136,856)/154,555) less total CPU time, as shown in Table 3. LA.T finds bugs in 118% (=(98-45)/45) more programs than LA for the 354 buggy programs within 3% (=(126,043-121,921)/121,921) more total CPU time as shown in Table 4.

The reason for the increased number of solved buggy programs caused by TOUR in BAM is the reduced number of failed results caused by time-out. Table 3 shows the number of buggy RealWorld programs that BAM.T and BAM fail due to time-out. BAM.T timed out for 131 (↓ 14%) programs, while BAM timed out for 152.

The seemingly slower speed (the 3% (=(126,043-121,921)/121,921) more total CPU time) of LA.T is due to the weakness of LA that does *not* support recursive function calls. In contrast to LA, which terminates as soon as possible when it observes a recursive execution, LA.T spends more time avoiding the recursive execution to detect a target path that does not include recursive executions. As a result, LA.T reduces the number of failed results caused by the recursion (i.e., reduced 65% (=(78-27)/78) compared to LA), as shown in Table 4.

TABLE 4. The number of solved buggy RealWorld programs and the total CPU time of LA and LA.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Error(recursion): the number of programs that a technique fails due to the recursive function calls; Count.Others: the number of programs that a technique fails due to reasons other than time-out and recursion; Total CPU time: the total model checking time for 354 buggy RealWorld programs.

	LA	LA.T	
Count.Solved	45	98	(↑ **118%**)
Count.Time-out	99	94	
Count.Error(recursion)	78	27	(↓ **65%**)
Count.Others	132	135	
Total CPU-time	121,921s	126,043s	(↑ 3%)

B. RQ2: EFFECTIVENESS OF TOUR ON VERIFICATION SPEED

The results show that TOUR *meaningfully improves the verification speed* of an ARMC technique. BAM.T verifies 15% (=(378-330)/330) more programs in 15% (=(336,940-287,911)/336,940) less total CPU-time than BAM for the 652 clean programs in RealWorld.4, i.e., the most complex program group, as shown in Table 5. We focus on complex target programs (i.e., RealWorld.4) because it is more meaningful to solve challenging programs quickly than to solve easy programs quickly.

The reason for the increased number of solved clean programs caused by TOUR in BAM is the reduced number of failed results caused by time-out. Table 5 shows the number of clean programs that BAM and BAM.T fail due to time-out. BAM.T reduces the number of failed results caused by timeout by 15% (=(316-268)/316) compared to BAM for the complex programs.

1) IMPROVED VERIFICATION SPEED ON THE ENTIRE SET OF THE 2688 CLEAN RealWorld PROGRAMS

The results show that TOUR improves the verification speed of an ARMC technique on the 2,688 clean RealWorld programs. BAM.T verifies 4% (=(1,996-1,911)/1,911) more programs in 12% (= (657,265-581,045)/657,265) less total CPU time than BAM for the 2,688 clean RealWorld programs as shown in Table 6.

LA.T solves more programs than LA, but it takes more total CPU time than LA. LA.T verifies 1% (=(1,420-1,404)/1,404) more programs in 4% (= (673,734-650,580)/650,580) more total CPU time than LA for the 2,688 clean RealWorld programs as shown in Table 6.[3]

2) INCREASING EFFECTIVENESS AS COMPLEXITY INCREASES

The results show that the effectiveness of TOUR increases with the increasing complexity of clean programs. Table 7 shows the increasing effectiveness of TOUR (in parentheses) for clean RealWorld.{1,2,3,4} programs with the number of solved clean programs and the total model checking time of BAM.T. BAM.T verifies the same number of programs by taking less than 1% more total CPU-time than BAM for RealWorld.1, i.e., the set of the least complex programs. BAM.T solves 15% more programs in 15% less total CPU-time than BAM for RealWorld.4, i.e., the set of the most complex programs.

[3]LA.T cannot verify a clean program by bypassing recursions that LA.T does for buggy programs.

TABLE 5. The number of solved clean RealWorld.4 programs and the total CPU time of BAM and BAM.T; Count.Solved: the number of correctly solved programs; Count.Time-out: the number of programs that a technique fails due to time-out; Count.Others: the number of programs that a technique fails due to reasons other than time-out; Total CPU-time: the total model checking time for the 652 clean RealWorld.4 programs.

	BAM	BAM.T	
Count.Solved	330	378	(↑ **15%**)
Count.Time-out	316	268	(↓ **15%**)
Count.Others	6	6	
Total CPU time	336,940s	287,911s	(↓ **15%**)

TABLE 6. The number of solved programs and total CPU time of each technique for 2,688 clean RealWorld programs; C.SOLV: the number of programs solved by each technique; T.TIME: the total CPU-time of each technique.

Technique	C.SOLV		T.TIME	
BAM	1,911		657,265s	
BAM.T	1,996	(↑ **4%**)	581,045s	(↓ **12%**)
LA	1,404		650,580s	
LA.T	1,420	(↑ **1%**)	673,734s	(↑ 4%)

TABLE 7. The number of solved programs and total CPU time of BAM.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by BAM.T for each PROG.SET (difference ratio compared to that of BAM); T.TIME: the total CPU-time of BAM.T for each PROG.SET (difference ratio compared to that of BAM).

PROG.SET	C.SOLV		T.TIME	
RealWorld.1	691	(0%)	32,222s	(0%)
RealWorld.2	542	(↑ **1%**)	89,683s	(↓ **6%**)
RealWorld.3	385	(↑ **8%**)	171,229s	(↓ **11%**)
RealWorld.4	378	(↑ **15%**)	287,911s	(↓ **15%**)

TABLE 8. The number of solved programs and total CPU-time of LA.T for clean RealWorld.{1,2,3,4} programs; C.SOLV: the number of programs solved by LA.T for each PROG.SET (difference ratio compared to that of LA); T.TIME: the total CPU-time of LA.T for each PROG.SET (difference ratio compared to that of LA).

PROG.SET	C.SOLV		T.TIME	
RealWorld.1	685	(0%)	46,794s	(↑ 1%)
RealWorld.2	499	(↑ **2%**)	97,608s	(↓ **11%**)
RealWorld.3	175	(↑ **4%**)	230,153s	(↑ 3%)
RealWorld.4	61	(0%)	299,179s	(↑ 13%)

Increasing effectiveness is not shown in LA for clean RealWorld.{1,2,3,4} programs because both LA and LA.T hardly verify complex programs, as shown in Table 8. LA.T verifies 2% more programs in 11% less total CPU-time than LA for clean RealWorld.2 programs. LA.T verifies 4% more programs within 3% more total CPU-time than LA for clean RealWorld.3 programs. For RealWorld.1, LA.T does not outperform LA because both LA.T and LA solve most of the target programs. For RealWorld.4, LA.T does not outperform

LA because both LA.T and LA hardly solve the complex programs.

C. RQ3: EFFECTIVENESS OF THE PROPOSED DISTANCE METRICS

The results show that **lf** *is the best distance metric* in terms of both the number of solved programs and the total CPU time for BAM. For LA, **bb** *is the best distance metric* in terms of the number of solved programs. BAM.T.lf (i.e., the best distance metric in terms of both the number of solved programs and the total CPU time for BAM) solves 9% (=(2,096-1,923)/1,923) more programs in 21% (=(924,237-732,531)/924,237) less total CPU-time than BAM.T.st (i.e., the worst distance metric for BAM) as shown in Table 9. LA.T.bb (i.e., the best distance metric in terms of the number of solved programs for LA) solves 2% (=(1,500-1,471)/1,471) more programs in 12% (=(817,056-730,531)/730,531) more total CPU-time compared to LA.T.lh (i.e., the worst distance metric for LA) as shown in Table 10.

The results also show that appropriate distance metric selection is critical for improving the effectiveness of ARMC. BAM.T.st, i.e., the worst distance metric selection (solves 1,923 programs in 924,237 seconds of total CPU time), does not outperform BAM (solves 2,011 programs in 811,820 seconds of total CPU time) despite the error location directed search.

D. RQ4: EFFECTIVENESS OF THE DISTANCE METRIC SELECTION

The results show that TOUR with *program-specific distance metric selection outperforms* TOUR with a single-distance metric. BAM.T solves from 20 (=2,116-2,096) to 193 (=2,116-1,923) more programs by taking from 2% (=(732,531-717,901)/732,531) to 12% (=(924,237-717,901)/924,237) less total CPU time than BAM.T.{st,bb,lh,lf} techniques as shown in Table 11. LA.T solves from 47 (=1,518-1,471) to 18 (=1,518-1,500) more programs than LA.T.{st,bb,lh,lf} techniques as shown in Table 12.

TABLE 9. The number of solved programs and total CPU time of BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs.

	C.SOLV		T.TIME	
BAM.T.st	1,923		924,237s	
BAM.T.bb	2,044		791,910s	
BAM.T.lh	2,030		789,454s	
BAM.T.lf	2,096	(↑ **9**%)	732,531s	(↓ **21**%)

TABLE 10. The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs.

	C.SOLV		T.TIME	
LA.T.st	1,500		822,982s	
LA.T.bb	1,500	(↑ **2**%)	817,056s	(↑ 12%)
LA.T.lh	1,471		730,531s	
LA.T.lf	1,499		827,401s	

TABLE 11. The number of solved programs and total CPU time of BAM.T and BAM.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs.

	C.SOLV		T.TIME	
BAM.T	2,116		717,901s	
BAM.T.st	1,923	(↓ **10**%)	924,237s	(↑ **12**%)
BAM.T.bb	2,044	(↓ **4**%)	791,910s	(↑ **9**%)
BAM.T.lh	2,030	(↓ **4**%)	789,454s	(↑ **9**%)
BAM.T.lf	2,096	(↓ **1**%)	732,531s	(↑ **2**%)

TABLE 12. The number of solved programs and total CPU time of LA.T.{st,bb,lh,lf}; C.SOLV: the number of programs solved; T.TIME: The total CPU time for 3,042 (both buggy and clean) RealWorld programs.

	C.SOLV		T.TIME	
LA.T	1,518		799,777s	
LA.T.st	1,500	(↓ **1**%)	822,982s	(↑ **3**%)
LA.T.bb	1,500	(↓ **1**%)	817,056s	(↑ **2**%)
LA.T.lh	1,471	(↓ **3**%)	730,531s	(↓ 10%)
LA.T.lf	1,499	(↓ **1**%)	827,401s	(↑ **3**%)

LA.T takes from 3% (=(827,401-799,777)/827,401) to 2% (=(817,056-799,777)/817,056) less total CPU-time than LA.T.{st,bb,lf} while LA.T.lh takes less total CPU-time than LA.T.

VII. CONCLUSION AND FUTURE WORK

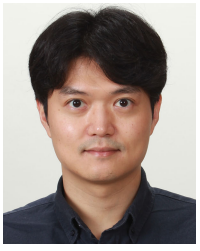
In this paper, we introduced TOUR which improves both the bug detection ability and the verification speed of ARMC. TOUR is the first ARMC technique that applies directed search (based on the distance to an error location) for fast target path detection in interprocedural programs. Additionally, as shown in Section VI-D, the program-specific distance metric selection heuristic of TOUR contributes to improving the performance of ARMC. The experimental results on the 3,042 real-world C program benchmark confirm the improved model checking performance of TOUR. For example, LA using TOUR finds bugs in 118% more programs than LA for 354 real-world buggy C programs, and BAM using TOUR finds bugs in 20% more programs than BAM in 11% less total CPU time than BAM. BAM using TOUR verifies 15% more programs in 15% less total CPU time for 652 real-world complex clean C programs.

In future work, we plan to develop new distance metrics to detect not only short but also ''good'' target paths. Additionally, we plan to extend the directed search with dynamically adjusting search methods such as those in the dynamic symbolic execution area [47].

REFERENCES

[1] D. Beyer, T. A. Henzinger, and G. Théoduloz, ''Configurable software verification: Concretizing the convergence of model checking and program analysis,'' in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2007, pp. 504–518.
[2] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, S. F. Univers, and R. Sebastiani, ''Software model checking via large-block encoding,'' in *Proc. Formal Methods Comput.-Aided Design*, Nov. 2009, pp. 25–32.
[3] D. Beyer, M. E. Keremoglu, and P. Wendler, ''Predicate abstraction with adjustable-block encoding,'' in *Proc. Formal Methods Comput.-Aided Design (FMCAD)*, 2010, pp. 189–197.

[4] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds. Berlin, Germany: Springer, 2003, pp. 235–239.
[5] D. Wonisch and H. Wehrheim, "Predicate analysis with block-abstraction memoization," in *Proc. Int. Conf. Formal Eng. Methods*. Berlin, Germany: Springer, 2012, pp. 332–347.
[6] D. Beyer and K. Friedberger, "Domain-independent interprocedural program analysis using block-abstraction memoization," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Nov. 2020, pp. 50–62.
[7] E. Clarke, "Counterexample-guided abstraction refinement," in *Proc. 10th Int. Symp. Temporal Represent. Reasoning, 4th Int. Conf. Temporal Logic.* Berlin, Germany: Springer, 2000, pp. 154–169.
[8] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *Proc. 29th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*, New York, NY, USA, 2002, pp. 58–70, doi: 10.1145/503272.503279.
[9] K. Dräger, B. Finkbeiner, and A. Podelski, "Directed model checking with distance-preserving abstractions," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 1, pp. 27–37, Feb. 2009.
[10] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2011, pp. 95–111.
[11] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, nos. 5–6, pp. 505–525, Oct. 2007.
[12] S. Das, D. L. Dill, and S. Park, "Experience with predicate abstraction," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 1999, pp. 160–171.
[13] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *ACM SIGPLAN Notices*, vol. 36, 2001, pp. 203–213.
[14] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2005, pp. 570–574.
[15] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, no. 7, pp. 68–76, Jul. 2011.
[16] D. Beyer and A. Stahlbauer, "BDD-based software verification," *Int. J. Softw. Tools Technol. Transf.*, vol. 16, no. 5, pp. 507–518, Oct. 2014.
[17] P. Andrianov, V. Mutilin, M. Mandrykin, and A. Vasilyev, "CPA-BAM-slicing: Block-abstraction memoization and slicing with region-based dependency analysis," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Cham, Switzerland: Springer, 2018, pp. 427–431.
[18] J. Jaffar, J. A. Navas, and A. E. Santosa, "Unbounded symbolic execution for program verification," in *Proc. Int. Conf. Runtime Verification*. Berlin, Germany: Springer, 2011, pp. 396–411.
[19] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," in *Fundamental Approaches to Software Engineering*, V. Cortellessa and D. Varró, Eds. Berlin, Germany: Springer, 2013, pp. 146–162.
[20] D. Beyer, S. Lowe, and P. Wendler, "Refinement selection," in *Model Checking Software*, B. Fischer and J. Geldenhuys, Eds. Cham, Switzerland: Springer, 2015, pp. 20–38.
[21] K. Friedberger, "CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2016, pp. 912–915.
[22] K. L. McMillan, "Lazy abstraction with interpolants," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2006, pp. 123–136.
[23] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with impact," in *Proc. Formal Methods Comput.-Aided Design*, Oct. 2013, pp. 210–217.
[24] A. Albarghouthi, A. Gurfinkel, and M. Chechik, "From under-approximations to over-approximations and back," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2012, pp. 157–172.
[25] M. Dangl, S. Löwe, and P. Wendler, "CPAchecker with support for recursive programs and floating-point arithmetic," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2015, pp. 423–425.
[26] D. Beyer and M. Dangl, "Strategy selection for software verification based on Boolean features," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*. Berlin, Germany: Springer, 2018, pp. 144–159.
[27] C. Richter and H. Wehrheim, "PeSCo: Predicting sequential combinations of verifiers," in *Pro. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Cham, Switzerland: Springer, 2019, pp. 229–233.
[28] C. Richter, E. Hüllermeier, M.-C. Jakobs, and H. Wehrheim, "Algorithm selection for software validation based on graph kernels," *Automated Softw. Eng.*, vol. 27, nos. 1–2, pp. 153–186, Jun. 2020.
[29] A. Hajdu and Z. Micskei, "Efficient strategies for CEGAR-based model checking," *J. Automated Reasoning*, vol. 64, no. 6, pp. 1–41, 2019.
[30] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. Boca Raton, FL, USA: CRC Press, 2014.
[31] T. McCabe, "Cyclomatic complexity and the year 2000," *IEEE Softw.*, vol. 13, no. 3, pp. 115–117, May 1996.
[32] (2019). *rpart*. Accessed: Apr. 14, 2021. [Online]. Available: https://cran.r-project.org/package=rpart
[33] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2011, pp. 184–190.
[34] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The MathSAT 4 SMT solver," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2013, pp. 93–107.
[35] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An interpolating SMT solver," in *Proc. Int. SPIN Workshop Model Checking Softw.* Berlin, Germany: Springer, 2012, pp. 248–254.
[36] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 12652. Cham, Switzerland: Springer, 2021, pp. 401–422.
[37] D. Beyer and A. K. Petrenko, "Linux driver verification," in *Proc. Int. Symp. Leveraging Appl. Formal Methods, Verification Validation*. Berlin, Germany: Springer, 2012, pp. 1–6.
[38] D. Beyer, S. Löwe, and P. Wendler, "Sliced path prefixes: An effective method to enable refinement selection," in *Formal Techniques for Distributed Objects, Components, and Systems*, S. Graf and M. Viswanathan, Eds. Cham, Switzerland: Springer, 2015, pp. 228–243.
[39] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Proc. Haifa Verification Conf.* Cham, Switzerland: Springer, 2017, pp. 99–114.
[40] D. Beyer, M. Dangl, and P. Wendler, "A unifying view on SMT-based software verification," *J. Automated Reasoning*, vol. 60, no. 3, pp. 299–335, Mar. 2018.
[41] D. Beyer and T. Lemberger, "Symbolic execution with CEGAR," in *Proc. Int. Symp. Leveraging Appl. Formal Methods*. Cham, Switzerland: Springer, 2016, pp. 195–211.
[42] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, Washington, DC, USA, Sep. 2008, pp. 443–446, doi: 10.1109/ASE.2008.69.
[43] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *Proc. Int. Conf. Comput. Aided Verification*. Berlin, Germany: Springer, 2012, pp. 758–766.
[44] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, New York, NY, USA, Oct. 2013, pp. 19–32, doi: 10.1145/2509136.2509553.
[45] D. Beyer, "Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016)," in *Proc. Int. Conf. Tools Algorithms Construct. Anal. Syst.* Berlin, Germany: Springer, 2016, pp. 887–904.
[46] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A framework for abstraction-and interpolation-based software verification," in *Proc. Int. Conf. Comput. Aided Verification*. Springer, 2012, pp. 672–678.
[47] S. Cha and H. Oh, "Concolic testing with adaptively changing search heuristics," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, Aug. 2019, pp. 235–245.



NAKWON LEE received the B.S. degree in computer engineering from Konkuk University, Seoul, South Korea, in 2013, and the M.S. degree in computer science from the Korea Advanced Institute of Science and Technology, Deajeon, South Korea, in 2015, where he is currently pursuing the Ph.D. degree in computer science. His current research interests include software model checking, abstract interpretation, and software reliability engineering.

YUNHO KIM received the Ph.D. degree in computer science from KAIST, in 2017. After working as a Postdoctoral Researcher and a Research Assistant Professor with the School of Computing, KAIST, he has been with the faculty of Hanyang University, since 2020. He currently focuses on developing automated testing and debugging techniques to make an impact to industries. His research interests include data-driven software testing, AI-based software debugging, and security vulnerability detection.

MOONZOO KIM received the Ph.D. degree in computer and information science from the University of Pennsylvania. He is currently an Associate Professor with the School of Computing, Korea Advanced Institute of Science and Technology (KAIST). Also, he is a CEO of V+Lab Inc., which provides automated software testing tools and services to industries (https://vpluslab.kr). His research interests include automated software testing and automated debugging, such as fault localization and bug repair.

DUKSAN RYU received the bachelor's degree in computer science from Hanyang University, in 1999, the dual master's degree in software engineering from KAIST and Carnegie Mellon University, in 2012, and the Ph.D. degree from the School of Computing, KAIST, in 2016. He is currently an Assistant Professor with the Software Engineering Department, Jeonbuk National University. His research interests include software analytics based on AI, software defect prediction, mining software repositories, and software reliability engineering.

JONGMOON BAIK (Member, IEEE) received the B.S. degree in computer science and statistics from Chosun University, in 1993, and the M.S. and Ph.D. degrees in computer science from the University of Southern California, in 1996 and 2000, respectively. He worked as a Principal Research Scientist with the Software and Systems Engineering Research Laboratory, Motorola Labs, where he was responsible for leading many software quality improvement initiatives. He is currently an Associate Professor with the School of Computing, Korea Advanced Institute of Science and Technology (KAIST). His research activities and interests include software six sigma, software reliability and safety, and software process improvement. He is a member of ACM.