Evaluation of Program Slicing in Software Verification*

Marek Chalupa and Jan Strejček[0000−0001−5873−403X]

Masaryk University, Brno, Czech Republic
{chalupa,strejcek}@fi.muni.cz

Abstract. There are publications that consider the use of program slicing in software verification, but we are aware of no publication that thoroughly evaluates the impact of program slicing on the verification process. This paper aims to fill in this gap by providing a comparison of the effect of program slicing on the performance of the reachability analysis in several state-of-the-art software verification tools, namely CPAchecker, DIVINE, KLEE, SeaHorn, and SMACK. The effect of slicing is evaluated on the number of solved benchmarks and running times of the tools. Experiments show that the effect of program slicing is mostly positive and can significantly improve the performance of some tools.

1 Introduction

Program slicing [36] is a method that takes a program and extracts a subprogram called *sliced program* or simply *slice* that contains only statements relevant for a given *slicing criterion*. In this paper, we consider static backward slicing with slicing criterion given as some statement of the program. The slice comprises the program statements that have some influence on the reachability or the arguments of the slicing criterion.

Program slicing has applications in many areas of computer science including (but not limited to) program debugging, code comprehension, code maintenance and re-engineering, regression testing, and software verification. Many applications are mentioned in several program slicing surveys [31, 6, 27, 15, 37, 28].

This paper is concerned with the last mentioned use case. In software verification, program slicing has been usually used simply as a preprocessing step. That is, the program is sliced with respect to the verified property before the actual verification process starts. Nevertheless, the impact of applying program slicing in such settings was usually certified by only few examples, if any. This paper aims to evaluate the impact of applying program slicing before verification on a large number of benchmarks.

We took the Symbiotic framework [9], which has the capability of slicing llvm [24] bitcode, and integrated several state-of-the-art software verification tools processing llvm bitcode into this framework. The tools, namely

* This work has been supported by the Czech Science Foundation grant GA18-02177S.

CPAchecker [4], DIVINE [25], KLEE [7], SeaHorn [14], and SMACK [8], were selected such that each uses a different verification approach.

We conducted experiments on benchmarks from Software Verification Competition (SV-COMP) [3]. More precisely, we took more than 6500 benchmarks, which are sequential C programs concerned with the reachability of a specified error location. Reachability of an error location represents probably the most common verification task, which may be used for verification of assertion validity, absence of division by zero, etc.

Each tool was run in three configurations. The first configuration does not use any slicing. The second configuration uses a slicing originally designed for terminating programs. This slicing can remove potentially non-terminating loops. As a result, a sliced program may contain a reachable error location that is not reachable in the original program. Therefore, if the sliced program contains no reachable error location, then the original program has no reachable error location as well, but the opposite implication does not hold. The third configuration uses a less aggressive slicing that preserves termination properties of program loops. The experiments show that the application of program slicing has a positive effect as it increases the number of decided benchmarks and often speeds up the whole verification process.

The rest of the paper is organized as follows. In the next subsection, we summarize the related literature. Details on program slicing and its use in software verification is the content of Section 2. Section 3 deals with the relevant aspects of tools we used to evaluate the effect of slicing on the reachability analysis. In particular, we describe the slicing functionality of Symbiotic and how we integrated the tools CPAchecker, DIVINE, KLEE, SeaHorn, and SMACK into the Symbiotic framework. Section 4 presents the experiments and discusses their results. The last section concludes the paper.

1.1 Related Work

There are many papers that present some use of program slicing in software verification but without any analysis of the contribution of program slicing to the efficiency of the considered verification approach [16, 20, 19, 26, 21, 2, 23]. In the following, we mention papers that provide some evaluation of the effect of program slicing.

Vasudevan et al. [33, 34] use program slicing to speed-up LTL bounded model checking of Verilog models. The authors leverage the information from the LTL specification to improve the effectiveness of slicing over standard static backward slicing. Improvements over plain bounded model checking as well as over bounded model checking preceded by standard static backward slicing are reported. The authors evaluated the contribution of program slicing during bounded model checking (with bound 24) of 9 different LTL properties on a Verilog implementation of USB 2.0 Function Core.

Dwyer et al. [12] study the effect of applying program slicing in model checking of concurrent object-oriented programs. The work shows that slicing brings an additional reduction to partial order reduction, but no significant gain was

achieved for simple assertion checking. The authors suggest that it may have been caused by the structure of the 10 benchmarks used in the study.

Wang et al. [35] use slicing to reduce a given program before model checking it for buffer overrun. The authors report significant performance gain due to slicing, but they witness it only by verification of 5 assertions in *minicom* program.

Sabouri and Sirjani [30] use program slicing to slice Rebeca models of concurrent programs before model checking. The evaluation is provided for 9 benchmarks, each parametrized with several properties. One property was always the presence of a deadlock and the other properties were unspecified. The authors conclude that slicing reduces state space of the models and can significantly help reducing the time of model checking.

Chebaro et al. [10, 11] combine a fast static analysis with program slicing and concolic testing. The fast static analysis finds possible bugs in the program, program slicing then reduces the program with respect to these bugs (either to all of them or to a selected subset), and concolic testing tries to confirm whether the bug is real. In the two publications, the authors show the positive effect of slicing on 5 and 9 programs, respectively.

Trabish et al. [32] evaluate the use of program slicing during *chopped symbolic execution*. Chopped symbolic execution executes some (pre-determined) functions only on-demand when needed. Program slicing is used to further lower the cost of the execution of these functions, so it is also invoked on-demand during the analysis. The evaluation was done on 6 security vulnerabilities, each parametrised by 3 different search heuristics. The authors report that program slicing can significantly help their technique in some cases, but report also a slowdown in some other cases. They planned to avoid this problem by an automatic analysis which decides when to use program slicing.

2 Program Slicing

Program slicing was introduced by Mark Weiser in 1980's as a code decomposition technique for debugging [36]. The Weiser's algorithm is based on a backward data-flow analysis [36]. Ferrante et al. advocated using *program dependence graph* (PDG) for program slicing [13]. Their algorithm was extended by Horwitz et al. to programs with function calls [18] using so-called *system dependence graph* (SDG). The algorithms based on PDG/SDG gained on popularity as they are more intuitive and flexible. Research in the area of program slicing therefore focuses mainly on algorithms based on dependence graphs.

Now we describe slicing algorithms based on dependence graphs and discuss obstacles connected with program slicing in the context of program verification.

2.1 Slicing Programs Using Dependence Graphs

Slicing algorithms based on dependence graphs first build a dependence graph and then compute a slice from the graph. Nodes of the dependence graph correspond to program statements and edges capture all dependencies among these statements. Two basic kinds of dependence are *data* and *control dependence*.

```
1  n = input();
2  i = 0;
3  while (i < n) {
4    c = input();
5    if (i == 0) {
6      min = c;
7      max = c;
8    }
9    if (c < min)
10     min = c;
11   if (c > max)
12     max = c;
13   i = i + 1;
14 }
15 print(min);
16 print(n);
```

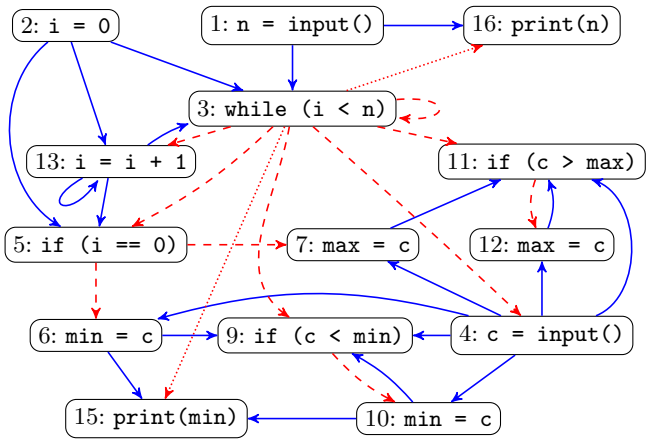


Fig. 1. A program (left) and its program dependence graph (right). Solid blue edges are data dependencies. Dashed red edges are control dependencies. The dotted red edges are extra control dependence edges added when using non-termination sensitive control dependence.

Data dependence may arise between two statements of a program that work with the same memory. More precisely, a statement s_r is dependent on a statement s_w if there exists a program execution where the statement s_r reads a value that has been written by the statement s_w. For example, consider the program in Figure 1 (left). The statement on line 3 is data dependent on the statement on line 2 because the value of variable `i` written on line 2 is read on line 3. Because the value of `i` read on line 3 may have been written also by the statement on line 13, the statement on line 3 is data dependent also on the statement on line 13. All data dependencies of the program are shown by solid blue edges in the graph of Figure 1 (right).

The notion of control dependence is more complicated since there are several definitions that can be used, each of them leading to slices with different properties. We introduce two of them. The first one is *standard control dependence* as defined by Ferrante et al. [13] and the other is *non-termination sensitive control dependence* introduced by Ranganath et al. [29].

Before defining the mentioned forms of control dependence, we must recall some of the standard concepts from program analysis. A *path* in a directed graph $G = (V, E)$ is a nonempty finite or infinite sequence $n_1, n_2, \ldots$ of nodes from V such that for each pair n_i, n_{i+1} of consecutive nodes it holds $(n_i, n_{i+1}) \in E$. A path is *non-trivial* if it has at least two nodes. *Maximal path* is an infinite path or a path whose last node has no successors (i.e., a path that cannot be prolonged). If there exists a path starting in a node n that contains a node m, we say that m is *reachable* from n. *Control flow graph* (CFG) is a directed graph where nodes

represent statements of a program and edges represent possible flow of control between statements in the program. We assume that a CFG has a distinguished *entry* node with no predecessors, from which all nodes are reachable. If a CFG contains also a unique *exit* node that has no successors and is reachable from any other node, we say that the CFG has the *unique exit property.* In such control flow graphs, we can define the post-dominance relation: we say that a node n *post-dominates* a node m if n appears on every path from m to the exit node. If, additionally, $n \neq m$, we say that n strictly post-dominates m.

Definition 1 (Standard control dependence). *Consider a CFG with the unique exit property. We say that a node n is* control dependent *on a node b if*

1. *there exists a non-trivial path π from b to n with any node on π (excluding b) post-dominated by n, and*
2. *b is not strictly post-dominated by n.*

In other words, a statement n is control dependent on a statement b if b is the "closest" point where the program may go some way that misses n. Figure 1 (right) depicts standard control dependencies as dashed red edges.

The standard control dependence is sufficient for the most of use cases, but it is problematic in several ways. First, it is not applicable to CFG without the unique exit property. Second, this definition of control dependence does not take into account the fact that loops may not terminate (as post-dominance considers only the paths that actually reach the exit node). That may result in incorrect slices where a non-terminating loop is sliced away making a previously unreachable code reachable. Both these problems can be solved by using *non-termination sensitive control dependence* [29]:

Definition 2 (Non-termination sensitive control dependence). *Given a CFG, a node n is* non-termination sensitive control dependent *on a node b if b has successors s_1 and s_2 such that*

1. *n occurs on all maximal paths form s_1, and*
2. *there exists a maximal path from s_2 on which n does not occur.*

For any CFG with the unique exit property, the definition actually extends the standard control dependence in the sense that if a node n is control dependent on a node b, then n is also non-termination sensitive control dependent b. The opposite implication does not hold. Figure 1 (right) depicts the additional non-termination sensitive control dependencies as dotted red edges.

Using non-termination sensitive control dependence improves the applicability of program slicing to a greater class of programs, but may be sometimes too strict – it results in keeping all loops of the program from which the slicing criterion is reachable in the slice. We elaborate on this matter later in Section 2.2.

Having a PDG and a slicing criterion, slicing the PDG is as easy as collecting all the nodes that are backward reachable through dependence edges from the

```
 1  n = input();
 2  i = 0;
 3  while (i < n) {
 4    c = input();
 5    if (i == 0) {
 6      min = c;
 8    }
 9    if (c > min)
10      min = c;
13    i = i + 1;
14  }
15  print(min);
```

```
 1  n = input();
16  print(n);
```

```
 1  n = input();
 2  i = 0;
 3  while (i < n) {
13    i = i + 1;
14  }
16  print(n);
```

Fig. 2. Slices of the program from Figure 1 for slicing criterion `print(min)` using standard control dependence (left), for `print(n)` using standard control dependence (middle), and for `print(n)` using non-termination sensitive control dependence (right).

node of the slicing criterion statement. For example, Figure 2 shows slices of the program in Figure 1 for slicing criteria `print(min)` (on the left) and `print(n)` (in the middle) using standard control dependence. The slice on the right considers also slicing criterion `print(n)`, but non-termination sensitive control dependence is used. In contrast to the slice in the middle, the slice on the right obtained using non-termination sensitive control dependence keeps also the header and the counter of the loop, as the slicing criterion is never reached if the loop does not terminate. Note that a program can be sliced with respect to more slicing criteria at once. Such slice consists of all nodes of the dependence graph that are backward reachable from at least one of the slicing criteria nodes.

If a program is structured into procedures and contains call statements, one can naturally build a single PDG for the program by working with *interprocedural control flow graph (ICFG)* instead of isolated control flow graphs for each procedure. An ICFG is a graph containing a CFG for each procedure and edges that go from call sites to entry points of procedures and from exits from procedures to return sites. This approach is comfortable as it does not require any changes in the PDG construction or in the slicing algorithm. Nevertheless, there are better algorithms (i.e., producing smaller slices) that work on system dependence graphs [18, 6] instead of on PDG. We do not go into details of these algorithms as we do not use them.

2.2 Program Slicing in Verification

In this section, we describe several obstacles that must be considered when applying program slicing before reachability analysis in software verification. These are mainly the use of user-defined assumptions and possibly non-terminating loops.

```
1  int x = nondet();
2  assume(x > 0);
3  assert(x > 0); // slicing crit.
```

```
1  int x = 0;
2  assume(x == 1);
3  assert(0); // slicing crit.
```

Fig. 3. The code on the left contains an assumption that behaves like a modifier to the data and the code on the right shows an assumption that changes the control flow. In both cases, the assertion is neither control nor data dependent on the assumption and thus the assumption would be sliced out if the assertion is taken as the slicing criterion. However, slicing the assumption away would introduce an error in the program.

Assumptions If the analyzed code contains user-defined assumptions (further represented as calls to function `assume()`), a special care must be taken to slice the program correctly. The reason is that slicing criteria are not dependent on assumptions as an assumption only reads values of variables. However, assumptions can influence a program execution in two ways:

- Assumptions can restrict the value of non-deterministic variables and thus act in a sense as a write to the variable.
- Assumptions can change the control flow of the program (e.g., execution is terminated if an assumption is not satisfied).

An example of such effects can be seen in Figure 3. On the left, the call to `assert` is dependent only on the statement on line 1 as the call to `assume` on line 2 only reads the variable `x` and does not modify it. However, if we would slice away the call to `assume`, we introduce an error to the program as the assertion on line 3 could be violated. Similar problem is shown in the code on the right, where the assertion on line 3 is independent of the rest of statements, but because of the unsatisfied assumption it is unreachable. Slicing away the assumption again introduces an error.

A simple solution, that we utilized also in our experiments, is to set the assumption statements as additional slicing criteria. This solution is imprecise – many of the assumptions that are left in the code along with their dependencies may be in fact irrelevant to the verified property. However, because the assumptions are usually localized to the beginning of the program where they constrain possible inputs, the increase of the slice size is mostly small.

The Choice of Control Dependence Here we discuss advantages and disadvantages of the two definitions of control dependence provided in Section 2.1.

Using the standard control dependence (Definition 1) may result in slicing away non-terminating loops and thus making previously unreachable code reachable. In particular, it may transform an unreachable error location into a reachable one. For example, consider the program in Figure 3 (right) with the error location `assert(0)` and replace the call `assume(x == 1)` by `while(1)`. The error location is unreachable due to the infinite loop, but if we slice the program with respect to the assertion as the slicing criterion, we get just the last line.

Hence, the whole process of slicing and program verification can report spurious errors. On the positive side, when using error locations as slicing criteria, slicing with standard control dependence cannot transform a reachable error location into unreachable one. Hence, if a slice is correct (i.e., it does not contain any reachable error location), then the original program is correct as well. Moreover, slices obtained with standard control dependence are smaller than these obtained with non-termination sensitive control dependence.

The non-termination sensitive control dependence (Definition 2) does not allow to slice away a loop (more precisely, its header and counters) if the loop may cause the unreachability of a slicing criterion by cycling forever. From the correctness point of view, one must therefore choose the non-termination sensitive control dependence. However, the price for this correctness is relatively high as only the loops that do not lie on a path in CFG from the entry node to any slicing criterion can be sliced away from the program.

To sum up, the main disadvantage of standard control dependence over non-termination sensitive control dependence is potential introduction of spurious errors by slicing away some non-terminating loops. Because these spurious errors can be ruled out by trying to reproduce each discovered error trace in the original program, we believe that using standard control dependence is also meaningful. In our experiments, we therefore consider both standard and non-termination sensitive control dependence and we report the numbers of correct as well as incorrect results.

3 Considered Tools

To evaluate the effect of program slicing on efficiency of reachability analysis, we integrated five state-of-the-art verification tools into the Symbiotic framework that has the capability of slicing programs. This section briefly describes Symbiotic, the implementation of the slicing procedure, and the five verification tools including important details about integration of these tools.

3.1 Symbiotic

Symbiotic [9] is a verification framework that applies program slicing to reduce the analyzed program before passing it to a verification backend. Slicing in Symbiotic works on programs in llvm [24], which is an assembly-like language extended with types. An llvm bitcode file is divided into functions. Instructions in a function are composed into basic blocks that form the control flow graph of the function. Memory manipulations (reads and writes) are done via pointers.

The workflow of Symbiotic when deciding reachability of an error location is straightforward. Symbiotic takes as input a list of C sources and compiles them into a single llvm bitcode file (if the input is not already an llvm bitcode). As the next step, the bitcode is optimized using the llvm infrastructure, then it is sliced with error locations as slicing criteria, and optimized again. Finally, the sliced and optimized bitcode is passed to a selected verification tool. In

this paper, we do not use the two optimization steps as we want to observe the pure effect of slicing. Note that the optimization steps can substantially improve efficiency of the verification process, but some care must be taken not to introduce unsoundness in the case the program contains undefined behavior.

3.2 Slicing Algorithm

The slicing procedure implemented in SYMBIOTIC is based on dependence graphs capturing dependencies between LLVM instructions. To compute dependencies for LLVM bitcode, one must

- perform *pointer analysis*,
- compute data dependencies by computing *reaching definitions*, and
- compute control dependencies from the bitcode structure.

Pointer analysis is needed to identify what memory is accessed by memory-manipulating instructions. SYMBIOTIC provides several pointer analyses with various precision and computation cost. Here we use interprocedural flow-insensitive field-sensitive inclusion-based pointer analysis [17] to compute information about pointers.

The information about pointers and the knowledge about the control flow of the analyzed program are then used in the (interprocedural) *reaching definitions analysis*. For each instruction that reads from memory, the analysis computes which instructions may have written the values read by the reading instruction. SYMBIOTIC applies the classic data-flow approach to reaching definitions computation [1]. With the results of reaching definition analysis, data dependencies can be computed easily.

Control dependencies (either standard or non-termination sensitive) are computed on the basic block level. Since computing control dependencies on ICFG can be impractical for big programs and programs that contain multiple calls of a function (which create a loop in ICFG), SYMBIOTIC computes control dependencies only intraprocedurally. Interprocedural control dependencies that arise from the possibility of not returning from a called function (e.g., when the function calls `exit` or `abort`, or loops indefinitely) are then filled in by post-processing.

When the algorithm computing standard control dependencies is used and the analyzed CFG does not have the unique exit property because it has multiple exit nodes, we establish the property by adding an artificial exit node that is the immediate successor of all original exit nodes. If the original CFG does not have the property because it has no exit node, then we conservatively make all instructions of each basic block control dependent on instructions immediately preceding the basic block.

SYMBIOTIC builds a single PDG for the whole bitcode. The slice is then computed as all instructions that are backward reachable from the slicing criteria nodes. The current slicing procedure in SYMBIOTIC cannot handle calls to `setjmp` and `longjmp` functions. However, these appear only rarely and do not appear in our benchmarks.

3.3 Verification Tools and Their Integration

In order to evaluate the effect of program slicing on reachability analysis, we integrated several state-of-the-art verification tools that can work with LLVM into SYMBIOTIC framework. Besides the symbolic executor KLEE, which has been used as SYMBIOTIC's verification back end for many years, the framework now supports also CPACHECKER, DIVINE, SEAHORN, and SMACK. The tools were integrated using *BenchExec* [5] tool-info modules that take care of assembling command line for a given tool setup.

Some of the tools have particular requirements on the input bitcode, e.g., that the bitcode does not contain any switch instructions. These requirements had to be addressed during the integration. We briefly describe the integrated tools along with the extra steps where the integration of the tool differs from the default configuration.

CPAchecker [4] is a configurable program analysis framework implementing many modern program analysis algorithms. For experiments, we used our fork of CPACHECKER that contains several fixes for the LLVM backend[1] and SV-COMP 2019 configuration (`-svcomp19` option). This configuration runs several analyses sequentially chained one after each other (each with a given time budget). The actual sequence of the analyses depends on the structure of the program and include, for instance, bounded model checking, explicit value analysis, predicate abstraction, and k-induction.

Note that the support for LLVM in CPACHECKER is still experimental. The required version of LLVM is 3.9.1.

DIVINE [25] is an explicit model checker that have recently added a support for verifying programs with inputs via instrumenting the symbolic computation directly into the analyzed program. Symbolic computations in DIVINE do not support 32-bit bitcode, therefore all the experiments with DIVINE assumed that the programs are written for 64-bit architectures. This assumption is void for most of the benchmarks that we used, but there are several cases where it led to an incorrect result.

We used DIVINE 4.3.1 (the static binary downloaded from DIVINE's web page) in experiments. The required LLVM version is 6.

KLEE [7] is a highly optimized symbolic executor. Before passing a bitcode to KLEE, SYMBIOTIC makes external globals internal, and replaces undefined functions with functions that return non-deterministic values (which should have no effect in our experiments). Also, SYMBIOTIC transforms standard SV-COMP functions that model non-determinism (named `__VERIFIER_*`) to ones that call KLEE's internal functions with equivalent semantics.

[1] `https://github.com/mchalupa/cpachecker/tree/llvm-fixes2`

Symbiotic has its own fork of KLEE based on KLEE 2.0. The fork differs from the mainstream version mainly by the ability of handling memory allocations of symbolic size. We used this fork of KLEE built for llvm in version 8.0.0 in our experiments.

SeaHorn [14] is a modular verification framework that uses constrained Horn clauses as the intermediate verification language. The verification condition is model checked using PDR/IC3.

We used the nightly build in version `0.1.0-rc3-61ace48` obtained from the docker image. The llvm version is 5.0.2.

SMACK [8] is a bounded model checker that internally compiles the program into Boogie and then uses Corral [22] to perform the analysis.

We used the version of SMACK that competed in SV-COMP 2019. The required llvm version is 3.9.1.

4 Experiments and Evaluation

We conducted a set of experiments on 6571 benchmarks from Software Verification Competition (SV-COMP) 2019 [3], namely all benchmarks from the category *ReachSafety* and from the subcategory *LinuxDeviceDrivers64* of the category *Systems*. Each of these benchmarks is a sequential C program that contains some marked error location. Moreover, each benchmark comes with the information whether at least one of its error locations is reachable or not.

The *ReachSafety* category contains several thematically focused subcategories like *Arrays*, *BitVectors*, and *Floats* with rather small programs. Then there is the subcategory *ProductLines* of generated models for e-mail communication, elevator, and mine pump, the subcategory *Sequentialized* of sequentialized parallel programs which often contian non-terminating loops, and the subcategory *ECA* of huge, synthetic benchmarks with extensive branching. Finally, *LinuxDeviceDrivers64* is a category of benchmarks generated from real Linux kernel device drivers.

The experiments were run on machines with *Intel Core i7-8700* CPU running at 3.20 GHz and 32 GB RAM. Each run was restricted to a single core, 8 GB of memory, and 15 minutes of CPU time. The presented times are running times of the whole process including compilation to llvm and program slicing (if applied).

Each tool was run on each benchmark in three configurations:

- without any slicing (referred as *No slicing*),
- with slicing using standard control dependencies (*Standard CD*), and
- with slicing using non-termination sensitive control dependencies (*NTS CD*).

Table 1 shows the numbers of decided benchmarks by each tool configuration summarized over all considered benchmarks.[2] It can be clearly seen that the

[2] Detailed numbers for each configuration and subcategory can be found at: `https://github.com/staticafi/symbiotic/releases/tag/ifm2019`

Table 1. Numbers of decided benchmarks by the considered tool configurations. The columns *correct* and *wrong* present the numbers of correctly and incorrectly decided benchmarks, respectively. The columns marked with ✓ (resp. ✗) contain the number of benchmarks where the decision of the tool was that error locations are unreachable (resp. reachable). For each tool, the highest numbers of correctly decided benchmarks with and without reachable error locations are typeset in bold.

	No slicing				Standard CD				NTS CD			
	correct		wrong		correct		wrong		correct		wrong	
Tool	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗	✓	✗
CPAchecker	673	666	72	13	**976**	**690**	75	44	841	672	71	14
DIVINE	727	458	0	1	**804**	**610**	0	35	799	518	0	4
KLEE	799	**1173**	45	0	**1364**	1138	46	43	1102	1007	4	4
SeaHorn	2222	874	23	580	**2460**	**933**	32	627	2264	898	21	589
SMACK	**2165**	969	2	235	2076	**1059**	5	282	1984	1039	3	259

configurations with slicing decide in all but one case more benchmarks than the corresponding configuration without slicing. As expected, *Standard CD* usually decides more benchmarks than *NTS CD*. In fact, slicing helps the most in *ProductLines* subcategory (this holds namely for SMACK and DIVINE) and *LinuxDeviceDrivers64* subcategory (CPAchecker, KLEE, and SeaHorn). This is not surprising as other subcategories contain usually small programs, which were often designed for testing of some verification tool, and thus slicing does not significantly change the complexity of these benchmarks.

There are only two cases when a tool in *Standard CD* configuration correctly decided less benchmarks than without slicing. The first case is KLEE on benchmarks with a reachable error location. The reason is that there are about 40 such benchmarks in the *ECA* subcategory that can be decided by KLEE without slicing, but not by the other configurations as slicing runs out of memory. The second case is SMACK and the explanation for this case is provided later at the description of Figure 4.

The results also show that all configurations produce some incorrect answers. There are several potential sources of these answers including bugs in verification tools, bugs in slicing, and maybe also wrongly marked benchmarks. Nevertheless, one can clearly see that *NTS CD* produces very similar (and sometimes even lower) number of incorrect answers as the tools without any slicing. *Standard CD* produces noticeably more incorrect answers. Most of these incorrect answers are negative, i.e., the tool decides that some error location is reachable even if it is not. These answers are mostly caused by slicing away non-terminating loops and are distributed mainly in the category *Sequentialized*.

The positive contribution of program slicing can be observed also in Figure 4 that shows quantile plots of times of correctly decided benchmarks. From the plots, we can read how many benchmarks (on x-axis) would the tool decide if the

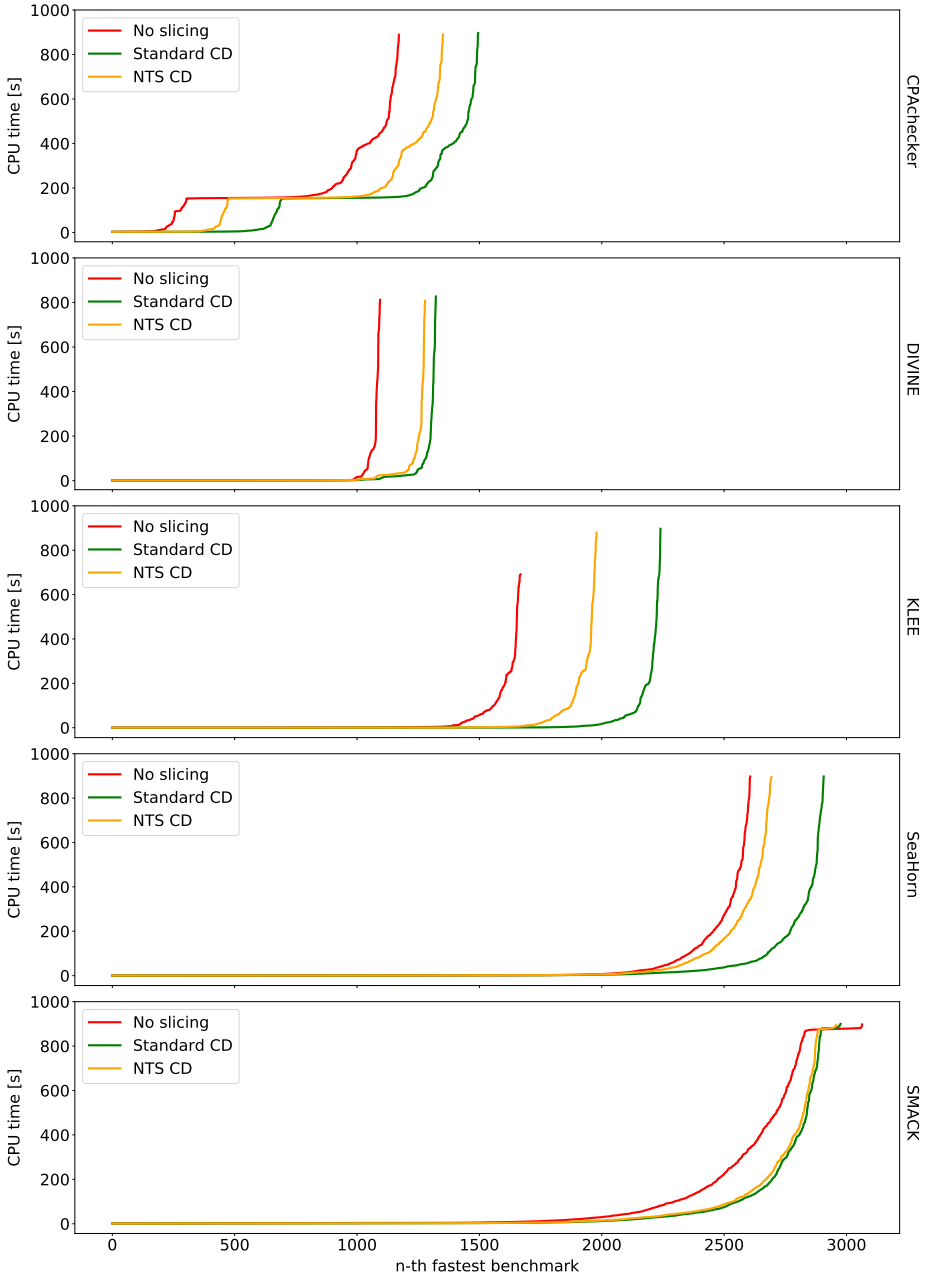


Fig. 4. Quantile plots of CPU time of correctly decided benchmarks. On x-axis are benchmarks sorted according to CPU time (on y-axis) in ascending order.

timeout would be set to the value on y-axis. We see the same pattern: *Standard CD* is the best (with the exception of SMACK), then *NTS CD* and then *No slicing* configuration.

The plots for SMACK have a very specific shape on its right end showing that many benchmarks are decided shortly before the timeout. This can be explained by a careful optimization of SMACK for SV-COMP: it seems that for many benchmarks the tool says shortly before the timeout that their error locations are unreachable unless it proves the opposite until then. Note that our experiments use the same benchmarks and time limit as the competition, and that this optimization can also explain the anomaly on the last line of Table 1. We manually checked the benchmarks on which SMACK behaved better without slicing and most of them crash after *exactly* 880 seconds with a compilation error when slicing is used.

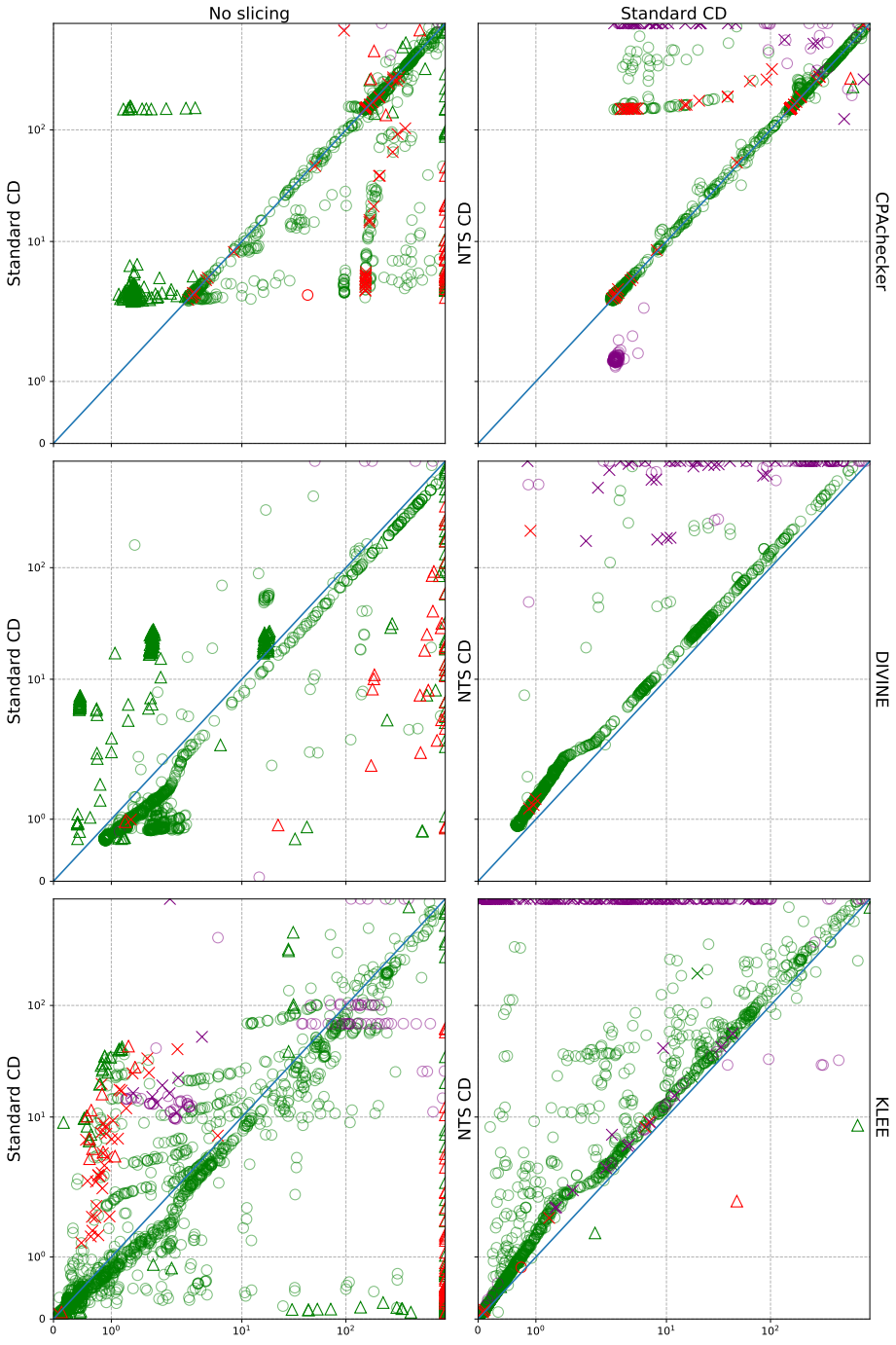
The positive effect of slicing to running times of the verification tools can be also seen in Figure 5, where scatter plots on the left compare *No slicing* configuration against *Standard CD* configuration, and scatter plots on the right compare *Standard CD* configuration against *NTS CD* configuration. The results of the configuration on x-axis are represented by shapes and on y-axis by colors. For x-axis, circle means correct result, cross wrong result, and triangle other result (timeout, error, unknown). For y-axis, green is correct result, red is wrong result, and purple is other result. For example, green circle means that the result was correct for both configurations and red triangle means that the result was other for the x-axis configuration but turned to wrong for y-axis configuration. To decrease the clutter in the plots, we omitted results of type other-other (purple triangles) as these are not very interesting.

From the scatter plots, we see that slicing usually helps decreasing the time of the analysis (green circles below the diagonal on the left scatter plots) or to decide new benchmarks that were not decided without slicing (green triangles on the left scatter plots). Slicing with standard control dependence (*Standard CD*) can introduce some wrong answers (red circles and triangles on the left scatter plots), which are mostly eliminated in *NTS CD* configuration (green or purple crosses on the right scatter plots). Using the *NTS CD* configuration, however, leads to increase of running times compared to *Standard CD* when deciding some benchmarks (green circles above the diagonal on the right).

5 Conclusion

We provided an intensive evaluation of the impact of program slicing on performance of software verification tools in reachability analysis. The experiments used a huge set of 6571 benchmarks, namely the *ReachSafety* category and *LinuxDeviceDrivers64* subcategory of SV-COMP 2019 benchmarks.

We confirmed several previous observations that program slicing can be effective in reducing the verification time. Further, we showed that it is worth considering the use of standard control dependence when slicing a program before its verification. This follows from the fact that the increase of the number


No slicing
Standard CD
Standard CD
NTS CD
CPAchecker
DIVINE
KLEE
0
10^0
10^1
10^2

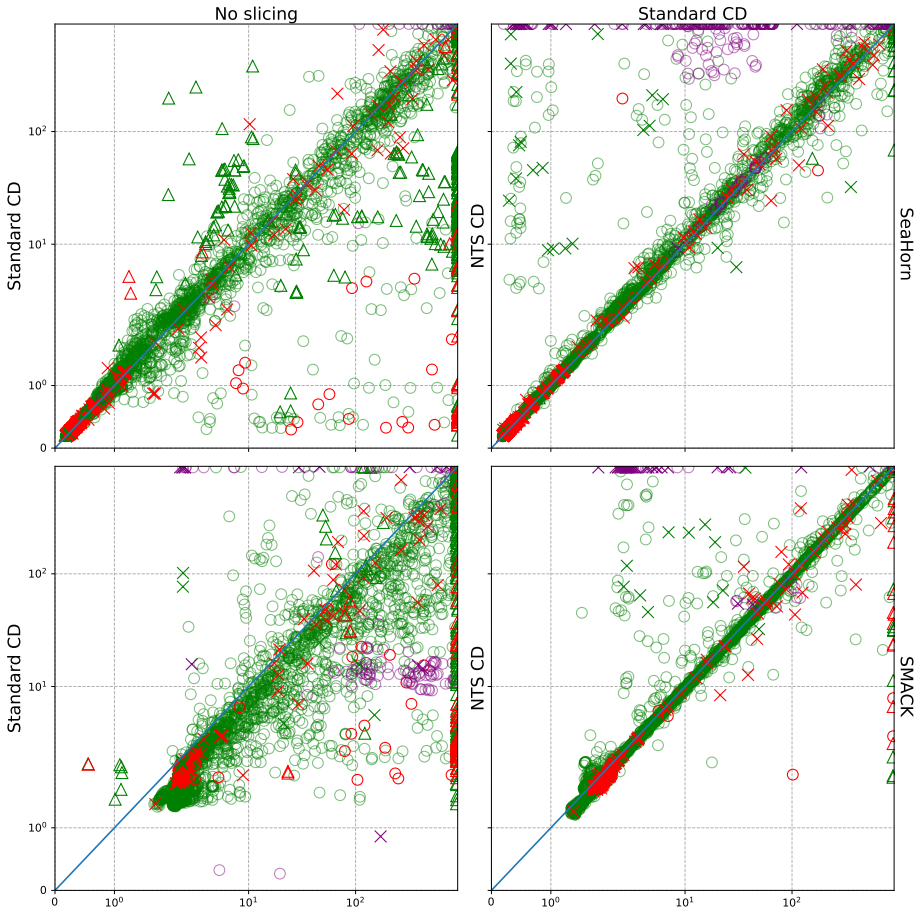


Fig. 5. Scatter plots that compare running time (in seconds) of *No slicing* and *Standard CD* configurations (left), and *Standard CD* and *NTS CD* configurations (right). Circle represents a correct result by the configuration on x-axis, cross a wrong result, and triangle other result (timeout, error, or unknown). Green color represents a correct result by the configuration on y-axis, red color a wrong result, and purple color other result. Purple triangles were omitted to reduce clutter.

of false positive answers is typically small (and we can get rid of them by trying to replay the found error trace, but we have not included this part in our experiments). Using non-termination sensitive control dependence may increase the analysis time compared to using standard control dependence, but it is still better than not using program slicing at all.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. URL `http://www.worldcat.org/oclc/12285707`.
2. J. D. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *SPIN'13*, *LNCS* 7976, pp. 99–116. Springer, 2013.
3. D. Beyer. Automatic verification of C and java programs: SV-COMP 2019. In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, *LNCS* 11429, pp. 133–155. Springer, 2019. URL `https://doi.org/10.1007/978-3-030-17502-3_9`.
4. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV'11*, *LNCS* 6806, pp. 184–190. Springer, 2011.
5. D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: requirements and solutions. *STTT*, 21(1):1–29, 2019.
6. D. W. Binkley and K. B. Gallagher. Program slicing. *Advances in Computers*, 43: 1–50, 1996.
7. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI'08*, pp. 209–224. USENIX Association, 2008.
8. M. Carter, S. He, J. Whitaker, Z. Rakamaric, and M. Emmi. SMACK software verification toolchain. In *ICSE'16*, pp. 589–592. ACM, 2016.
9. M. Chalupa, M. Vitovská, and J. Strejček. SYMBIOTIC 5: Boosted instrumentation - (competition contribution). In *TACAS'18*, *LNCS* 10806, pp. 442–446. Springer, 2018.
10. O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC'12*, pp. 1284–1291. ACM, 2012.
11. O. Chebaro, P. Cuoq, N. Kosmatov, B. Marre, A. Pacalet, N. Williams, and B. Yakobowski. Behind the scenes in SANTE: a combination of static and dynamic analyses. *Autom. Softw. Eng.*, 21(1):107–143, 2014.
12. M. B. Dwyer, J. Hatcliff, M. Hoosier, V. P. Ranganath, Robby, and T. Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, pp. 73–89, 2006.
13. J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
14. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *CAV'15*, *LNCS* 9206, pp. 343–361. Springer, 2015.
15. M. Harman and R. M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001.
16. J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.
17. M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE'01*, pp. 54–61. ACM, 2001.
18. S. Horwitz, T. W. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

19. F. Ivancic, I. Shlyakhter, A. Gupta, M. K. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C programs using F-SOFT. In *ICCD'05*, pp. 297–308. IEEE Computer Society, 2005.
20. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *CAV'05*, *LNCS* 3576, pp. 301–306. Springer, 2005.
21. F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3): 256–274, 2008.
22. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In *CAV'12*, *LNCS* 7358, pp. 427–443. Springer, 2012.
23. T. Lange, M. R. Neuhäußer, and T. Noll. Speeding up the safety verification of programmable logic controller code. In *HVC'13*, *LNCS* 8244, pp. 44–60. Springer, 2013.
24. C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, pp. 75–88. IEEE Computer Society, 2004.
25. H. Lauko, V. Štill, P. Ročkai, and J. Barnat. Extending DIVINE with symbolic verification using SMT - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics*, *LNCS* 11429, pp. 204–208. Springer, 2019.
26. X. Li, H. J. Hoover, and P. Rudnicki. Towards automatic exception safety verification. In *FM'06*, *LNCS* 4085, pp. 396–411. Springer, 2006.
27. A. D. Lucia. Program slicing: Methods and applications. In *SCAM'01*, pp. 144–151. IEEE Computer Society, 2001.
28. D. P. Mohapatra, R. Mall, and R. Kumar. An overview of slicing techniques for object-oriented programs. *Informatica (Slovenia)*, 30(2):253–277, 2006.
29. V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *ESOP'05*, *LNCS* 3444, pp. 77–93. Springer, 2005.
30. H. Sabouri and M. Sirjani. Slicing-based reductions for Rebeca. *Electr. Notes Theor. Comput. Sci.*, 260:209–224, 2010.
31. F. Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.
32. D. Trabish, A. Mattavelli, N. Rinetzky, and C. Cadar. Chopped symbolic execution. In *Proc. of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pp. 350–360. ACM, 2018. URL `https://doi.org/10.1145/3180155.3180251`.
33. S. Vasudevan, E. A. Emerson, and J. A. Abraham. Efficient model checking of hardware using conditioned slicing. *Electr. Notes Theor. Comput. Sci.*, 128(6): 279–294, 2005.
34. S. Vasudevan, E. A. Emerson, and J. A. Abraham. Improved verification of hardware designs through antecedent conditioned slicing. *STTT*, 9(1):89–101, 2007.
35. L. Wang, Q. Zhang, and P. Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In *SCAM'08*, pp. 165–173. IEEE Computer Society, 2008.
36. M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.
37. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.