Master's Thesis

Abstraction Refinement for Model Checking: Program Slicing + CEGAR

Thomas Lemberger

SoSy-Lab
LMU Munich, Germany
Supervisor: Prof. Dr. Dirk Beyer

March 9, 2018

Contents

List of Figures

List of Tables

Abstract

Program slicing and counterexample-guided abstraction (CEGAR) are two established approaches to program abstraction in software verification. Both are similar in their concept, with one main difference: Program slicing works on the syntactic level, while CEGAR works on the semantic level. Because of the complexity of software and the complex behavior of the two algorithms, the difference in the capabilities of the two is not clear. To contribute towards understanding this technique, we design a program slicing technique that is based on CEGAR and that performs program slicing on dynamically computed slicing criteria. We implement this technique in the widely used software verification framework CPAchecker, and extend it to be combine-able with any other, existing abstraction refinement based on CEGAR. As a proof of concept, we combine it with the CEGAR-based symbolic execution engine in CPAchecker.

In a next step, we extend the existing state-of-the-art, Llvm-based program slicer Symbiotic to use CPAchecker as a verification back-end. To combine theses two tools, we also implement a new Llvm front-end in CPAchecker that enables CPAchecker to analyze Llvm programs.

As a last step, to get a better understanding of the behavior of our analyses, we implement an abstract graph visualization technique, called pixel trees. Pixel trees allow users to grasp the structure of the program-state space that got explored by an analysis run, even for graphs that are too large to comprehend in a detail view.

We perform a thorough evaluation of all presented techniques. Through this, we are able to show that program slicing and CEGAR are two complementing technologies, and that their combination can have a significant impact on verification performance.

1 Introduction

In the last decade, model checking of programs has shown large improvements, even leading to its prominent use in large software companies, e.g, Facebook [1] or Linux [2]. Work has shown that model checking may actually surpass automatic testing in finding bugs in software [12].

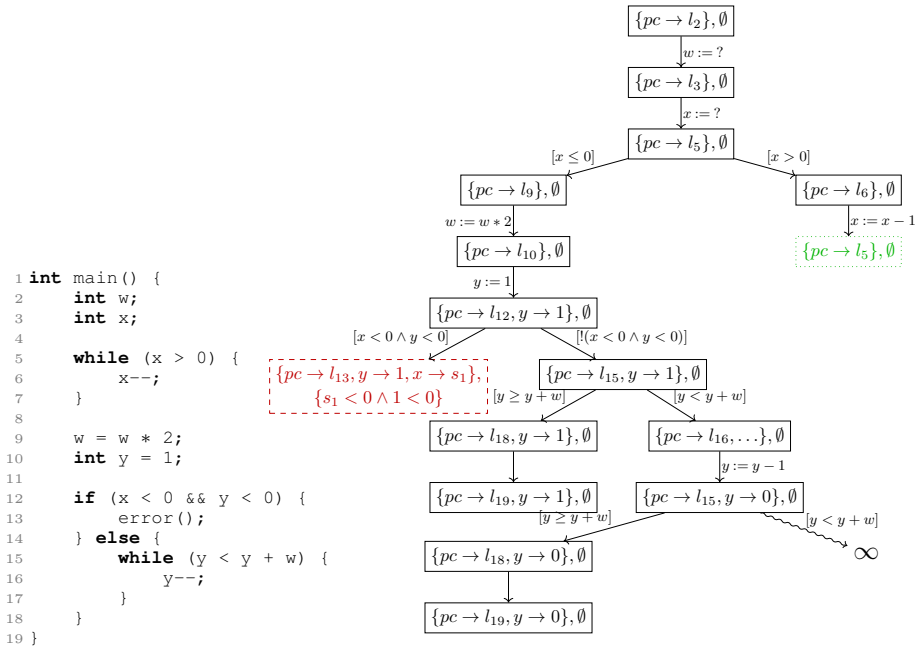
Most of such improvements in model checking are based on heuristics that exploit known or at least assumed code characteristics. Examples for such heuristics are program slicing [49], counterexample-guided abstraction refinement (CEGAR) [22], search heuristics [28] or bounded model checking [17]. But despite the success of them, it is still unclear how they affect code in detail. For two of these techniques, program slicing and CEGAR, it is even unclear how different their capabilities are.

This work tries to contribute towards understanding this. Program slicing and CEGAR both exploit code structure for abstraction. CEGAR uses the understanding that, to prove most error paths *infeasible*, only few information about the states of a program must be tracked. Program slicing, similarly, assumes that only few program operations determine the semantics that decide whether an error occurs. While CEGAR is an iterative approach that works on the semantic level, program slicing works on the syntactic level, only, and is usually a pre-processing step.

Figure 1 shows an example program and its analysis with symbolic execution with CEGAR. CEGAR is able to derive that x must not be tracked to prove function `error()` to be infeasible, and it can thus evade the first loop, but it then tracks y and infinitely unrolls the second loop that is controlled by y and w. Program slicing, on the other hand, would be able to derive that function `error()` can never be reached from the second loop, because it is not even syntactically reachable. But what if we have a program property that is not bound to one location, e.g., if we want to check a program for division by zero? In this case, slicing techniques may instrument the program with all possible error locations [21], but this may create a program slice that is significantly larger than necessary.

As a new approach, we design a CEGAR-based program slicing technique. First, we implement our own technique in CPAchecker, a state-of-the-art software verification framework. In CPAchecker, many analyses are implemented as *configurable program analyses*, a concept that allows easy combination of different techniques. Instead of using slicing as a fixed pre-processing step that requires us to know the slicing criterion (i.e., the target states that we don't want to encounter), we implement slicing as a dynamic analysis. To derive program slices, we use ourselves a CEGAR-based approach that builds an incremental program slice from encountered target states. Following the basic idea of slicing, we do not create a completely new program that misses program operations, but only replace the program operations in a program by `noop` operations during analysis, dynamically.

[1] http://fbinfer.com/ [2] http://linuxtesting.org/ldv



(a) Example program

(b) Infinite state space computed by a CEGAR-based technique

Figure 1: Running example

Through delegation, we allow an easy combination with other analyses that use, or do not use, CEGAR themselves. No code is necessary, slicing can be added to analyses through a single configuration file. With this, we are able to both easily combine program slicing and CEGAR-based techniques, and are not constraint to knowing our slicing criterion beforehand.

To get more data on the difference between program slicing and CEGAR, we do not only use our new, own implementation of program slicing, but also the successful slicing-based program verifier Symbiotic. This third-party tool ensures that our implementation does not have any major flaws leading to wrong conclusions. To allow the combination of Symbiotic and CPAchecker, we have to a) extend Symbiotic to accept CPAchecker as a verification back-end, and b) implement a Llvm-front-end into CPAchecker, because Symbiotic performs slicing on a Llvm transformation input programs that can't be translated back to C code.

We tackle one additional issue: While we are interested in the parts of the program-state space that were explored by a verification technique, these data become quickly hard to comprehend because of their sheer size. While an established graph representation (similar to Fig. 1b) for the program-state space exists, it is still hard to comprehend, and even to display, for large graphs. As an alternative, we provide a more abstract representation of graphs that we call *pixel trees*. The pixel tree of Fig. 1b is shown in Fig. 2. Through this representation, we are able to quickly comprehend even large computations of program-state spaces, and see their high-level differences in the blink of an eye.

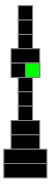


Figure 2: Pixel tree representation of the graph in Fig. 1b.

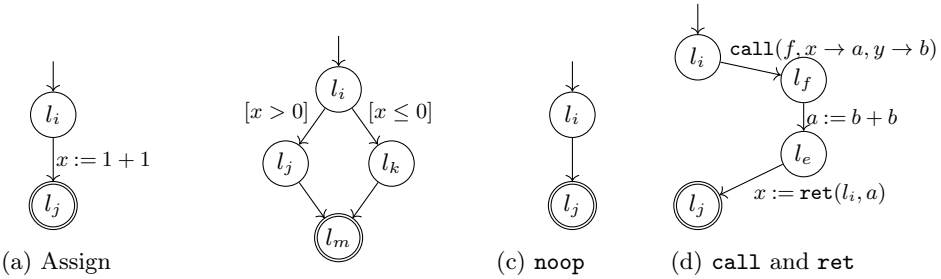
Contributions Our work provides the following contributions:

- We provide a generic program slicing algorithm based on CEGAR that can be used by all analyses in CPAchecker and that is able to derive program slices incrementally.
- We provide an Llvm front-end for CPAchecker, i.e., we a) enable analysis with CPAchecker for an additional programming language, and b) provide the possibility to combine CPAchecker with existing verification and program transformation techniques based on Llvm.
- We provide a new graph representation of the program-state space to help easily comprehend the high-level characteristics of large graph structures.
- We provide a thorough evaluation and analysis of verification results and verification behavior for our implementation, showing the benefits of combining program slicing with CEGAR techniques.

1.1 Related Work

Improvements to Symbolic Execution Next to program slicing and CEGAR, many other abstraction-based techniques exist that improve the performance of symbolic execution. Combinations with concrete execution, namely concolic execution [19, 31], or combinations with model checking [47] can improve the performance of symbolic execution, but are aimed at creating test cases, not exploring the full state space of a given program and proving it correct. This is also true for search heuristics that aim find program errors faster [19, 20]. Techniques that can be applied for formal verification are compositional execution [2, 30], invariant generation for unbounded loops [33], and invariant template generation [46], which is similar to loop invariants.

Slicing Techniques A vast amount of different program slicing techniques exists [40] and there are different techniques for constructing a dependence graph [38, 43] efficiently. The main technique of importance is dynamic slicing [35, 36], which performs slicing based on program run-time information and only preserves the semantics for a specific (set of) program input(s). Why we don't do the latter, through our CEGAR-based choice of slicing criteria, we also partly use run-time information to select our slicing criterion.



(a) Assign operation

(b) Assume operations

(c) `noop` operation

(d) `call` and `ret` operation

Figure 3: Example CFAs for the five program operations we support

Improvements to CEGAR Many approaches exist that try to improve the performance of CEGAR, by getting better interpolants [14, 15, 44] or optimizing the number of solver calls [5, 37]. These approaches are orthogonal to our approach of program slicing, which can delegate to techniques using these improvements. An alternative to CEGAR, when using interpolation, is lazy abstraction, also called IMPACT [16, 42].

2 Background

2.1 Proposition

We want to present all concepts in a way that puts focus on their specific characteristics and ignores unimportant technicalities. To do so, all concepts are formalized for a simple, inter-procedural and imperative programming language that supports two types of program operations: assume operations and assign operations. All program variables and all constant values range over arbitrary integers. Function parameters are passed by value and functions do have a single return value. There is no scoping, except for function parameters: These are only valid within a function. The set X is the set of all program variables of a program.

2.2 Formal Verification Basics

The definition of a partial function f is denoted by $\text{def}(f)$. The restriction of a partial function f to a new definition d is defined as $f_{|d} = \{x \to f(x) \mid x \in \text{def}(f) \cap d\}$.

2.2.1 Control-flow Automaton

We represent a program as *control-flow automaton* (CFA). A CFA A is defined as $A = (L, l_0, l_e, G)$. It consists of the set L of program locations, the program entry location $l_0 \in L$, the program exit location $l_e \in L$ and the set $G \subseteq L \times Ops \times L$ of edges. Each CFA has exactly one program entry location l_0 and exactly one program exit location l_e. Each edge $g = (l, op, l')$ describes the control flow from a origin location l to a target location l' through evaluation of a program operation op.

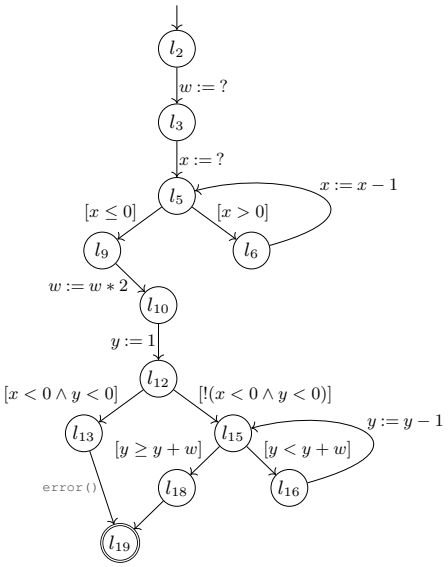
The set $\mathcal{P}$ of predicates represents all boolean expressions in first-order logic over program variables X and integers $\mathbb{Z}$. A program operation can be an assign operation $w := exp$ with program variable w and arithmetic expression exp, an assume operation $[p]$ with predicate $p \in \mathcal{P}$, a noop operation `noop`, a function call operation $\texttt{call}(f, a_1 \to p_1, \dots a_n \to p_n)$ with program variables a_1 to a_n and p_1 to p_n, or the return operation $w := \texttt{ret}(l, x)$ with program location $l \in L$ and program variables $w, x \in X$. The special symbol ? represents a non-deterministic, non-constant integer value. Expressions and predicates may only use program variables that are defined. To do so, program variables without an initial concrete value should be assigned the non-deterministic value ?. The interpretation of above program operations may differ, according to the application. We assume the following semantics: If the operation of an edge $g = (l, op, l')$ is an assign operation $op = w := exp$, control flow continues to l' after assigning the new value of expression exp to program variable w. If the operation is an assume operation $op = [p]$, control flow only continues from l to l', if p is true. If the operation is a noop operation $op = \texttt{noop}$, control flow continues from l to l' without any further effect. For `noop` operations, we omit the edge label. If the operation is a call operation $op = \texttt{call}(f, a_1 \to p_1, \dots a_n \to a_n)$, control flow continues from l to a function entry location l' of function f, after assigning the given function arguments a_1 to a_n to the n function parameters p_1 to p_n. And if the operation is a return operation $op = w := \texttt{ret}(l, x)$, then control flow continues from function exit location l to location l' after writing the value of function program variable x to w. Thus, the return operation carries information of the call site of a function. This makes it easy to track the calling context.

Figure 3 shows four small example CFAs. Figure 3a shows a CFA with a single assign operation $x := 1 + 1$. The edge $(l_i, x := 1 + 1, l_j)$ represents a transfer from program location l_i to program location l_j through the assignment of integer value 2 (because $1+1 = 2$) to program variable x. In this example, the set X of all program variables has to contain x, i.e., $\{x\} \subseteq X$. Figure 3b shows a CFA with two assume operations $[x > 0]$ and $[x \leq 0]$. The edge $(l_i, [x > 0], l_j)$ represents a transfer from program location l_i to program location l_j, under the condition $x > 0$.. Analogous, the edge $(l_i, [x \leq 0], l_k)$ represents a transfer from l_i to l_k under condition $x \leq 0$. This is called an *if-else-branching* (with condition $x > 0$). Figure 3c shows a CFA with a `noop` operation. The edge $(l_i, \texttt{noop}, l_j)$ represents a transfer from program location l_i to program location l_j without any further effect. Figure 3d shows a CFA with a function call. The `call` operation passes program variables x and y as arguments to a function f with parameters a and b. This function then stores the double of b in a, and it then returns a to location l_j of the callee. Then, x will be the double of y at l_j.

If $(l, op, l') \in G$, we say that l has an *outgoing edge* to l', and l' has an *in-going edge* from l. Program location l is a predecessor of program location l', and l' is a successor of l. Set $succs(l)$ describes all successors of a program location l. Set $preds(l')$ describes all predecessors of a program location l. A *path* $\alpha = \langle l_i \xrightarrow{op_i} l_{i+1} \xrightarrow{op_{i+1}} \dots \xrightarrow{op_{n-2}} l_{n-1} \xrightarrow{op_{n-1}} l_n \rangle$ from l_i to l_n is a sequence of connected CFA edges $g_i = (l_i, op_i, l_{i+1})$ to $g_{n-1} = (l_{n-1}, op_{n-1}, l_n)$

```
1 int main() {
2     int w;
3     int x;
4
5     while (x > 0) {
6         x--;
7     }
8
9     w = w * 2;
10    int y = 1;
11
12    if (x < 0 && y < 0) {
13        error();
14    } else {
15        while (y < y + w) {
16            y--;
17        }
18    }
19 }
```

(a) Source code representation



(b) CFA representation with entry location l_2 and exit location l_{19}

Figure 4: Example program

in the CFA. The length $|\alpha|$ of path α is the number of edges it consists of. For example, for the CFA in Fig. 3b, $|\langle(l_i, [x > 0], l_j), (l_j, \texttt{noop}, l_e)| = 2$. The distance $dist : L \times L \to \mathbb{N}$ is the length of the shortest path between two program locations. For example, $dist(l_i, l_e) = 2$. For every location $l \in L$, there is a path through A that starts at the program entry location l_0, ends in the program exit location l_e, and contains l. There is no edge $(l_e, op, l') \in G$ that starts at l_e. Every program location l has a) zero outgoing edges, if $l = l_e$, b) one outgoing edge, if the edge contains an assign operation, a call operation, a return operation or a `noop` operation, or c) two outgoing edges, if the edges contain an assume operation or an `noop` operation. A program location may never have more than two outgoing edges. If a program location has two outgoing edges, we call it a *branching location*. The number of in-going edges is not restricted.

CFAs can be used to represent programs of various programming languages by introducing additional program operations. A CFA representing our example program is shown in Fig. 4b. The function call `error()` is an external function and thus not interpreted. In the CFA, it is a `noop`-operation—but for better visualization, we still label the edge in the shown examples.

Function $uses : Ops \to 2^X$ returns all variables used in a given operation. An operation op uses a program variable x, i.e., $x \in uses(op)$, iff:

1. op is an assign operation $op = w := exp$ for some $w \in X$ and expression exp contains x,

2. op is an assume operation $[p]$ and predicate p contains x,

3. *op* is a call operation $\texttt{call}(f, a_1 \to p_1, \ldots, a_n \to p_n)$ and $x = a_i$ for any $1 \leq i \leq n$, or

4. *op* is a return operation $w := \texttt{ret}(l, x)$ and there is, in the CFA, a CFA edge $g' = (l, \texttt{call}(f, a_1 \to p_1, \ldots, a_n \to p_n), l')$.

The `noop` operator uses no program variables, i.e., $uses(\texttt{noop}) = \emptyset$. Function $uses : 2^{Ops}, \to 2^X$ returns all variables used in a given set of operations. Function $defs : Ops \to 2^X$ returns all variables defined in a given operation. An operation *op* defines a program variable x, i.e., $x \in defs(op)$, iff:

1. *op* is an assign operation $op = x := exp$ that assigns to x the value of some expression exp,

2. *op* is a call operation $\texttt{call}(a_1 \to p_1, \ldots, a_n \to p_n)$ and $x = p_i$ for any $1 \leq i \leq n$, or

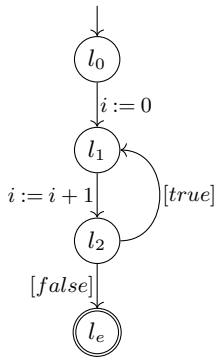
3. *op* is a return operation $x := \texttt{ret}(l, y)$ that assigns to x the return value y.

If *op* is an assume operation or `noop`, then $defs(op) = \emptyset$.

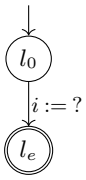
2.2.2 Reachability

A *concrete state* $\texttt{c} : \{pc\} \cup X \to \mathbb{Z}$ of a program assigns to the program counter *pc* and all program variables X concrete values. The set of all concrete states of a program is denoted by $\mathcal{C}$. A concrete state `c` of a program is *reachable*, if a path from the program entry exists on which `c` is assumed at some point. We call the set of all reachable concrete states of a program its *state space*. A *region* $\sigma \subseteq \mathcal{C}$ is a sub-set of concrete states of a program. Region σ is reachable, if at least one $\texttt{c} \in \sigma$ is reachable. The goal of the reachability problem is to prove whether a given *target region* σ^t is reachable. Every safety property of a program can be reduced to a reachability problem through instrumentation [4]. Thus, we can verify that a safety property holds by solving the corresponding reachability problem. If a target region is reachable, the safety property does not hold. If it is unreachable, the safety property does hold. If a safety property does not hold, we say that the program violates the safety property and that it is *unsafe*. If all safety properties of a program hold, we say that it is *safe*.

The number of possible concrete states of a program may be infinite. Figure 5 shows two CFAs with an infinite number of concrete states. Figure 5a may assume the infinite number of concrete states $\{\{pc \to l_0, i \to x\} \mid x \in \mathbb{Z}\} \cup \{\{pc \to l_1, i \to x\} \mid x \in \mathbb{N}_0\}$ due to the loop that never holds. And Fig. 5b may assume the infinite number of concrete states $\{\{pc \to l, i \to x\} \mid x \in \mathbb{Z}, l \in \{l_0, l_1\}\}$ since the non-deterministic assignment to i may be any concrete value.



(a) Infinite loop



(b) Non-deterministic value

Figure 5: Examples for a program with an infinite number of concrete states

2.2.3 Abstract Domain

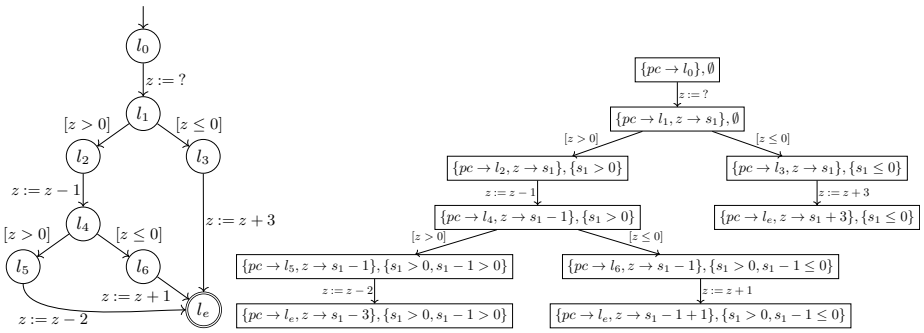
An *abstract domain* $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$ consists of the set $\mathcal{C}$ of concrete states, a semi-lattice $\mathcal{E} = (E, \sqsubseteq, \sqcup, \top)$ and a concretization function $[\![\cdot]\!]$. We call the elements E of $\mathcal{E}$ *abstract states*.

One can over-approximate the reachability of a concrete state c by abstracting from concrete states through abstract states. If we compute that an abstract state a is reachable, and $\mathsf{c} \in [\![a]\!]$, we assume that c is reachable. This may significantly reduce the number of computations necessary.

Abstract Reachability Graph An *abstract reachability graph* (ARG) represents the reachable abstract states of a program and their relation to each other. For some abstract domain with abstract states E, an ARG is defined as $AG = (E, a_0, R)$. Its nodes are abstract states E, it has an initial abstract state $a_0 \in E$, and edges $R \subseteq E \times Ops \times E$. An edge $(a, op, a') \in R$ exists, if the application of program operation op to a results in a'. Figure 6b shows an example ARG for the domain of symbolic execution.

2.3 Symbolic Execution

Symbolic execution [18, 25, 34] is a technique for exhaustive state-space exploration. Symbolic execution extends concrete execution with two components: a *symbolic memory*, and *path constraints*. The symbolic memory stores assignments of program variables to symbolic values $\mathcal{S}$. We define the set of all possible variable values in symbolic execution as $\mathcal{Z}_{\mathbb{C}_\mathbb{S}} = \mathbb{Z} \cup \mathcal{S}$. The partial function $v : X \to \mathcal{Z}_{\mathbb{C}_\mathbb{S}}$ contains all possible program variable assignments, both concrete and symbolic. We call $v(x)$ the *abstract variable assignments*. The set of all possible abstract variable assignments is $\mathcal{V}$. The path constraints $pc = \langle p_0, \ldots, p_n \rangle$ are a sequence of predicates $p_i \in \mathcal{P}$. The conjunction $\bigwedge_{p \in pc} p$ of path constraints describes the constraints on variable values. The set of all possible path constraints is $\langle P \rangle$. The abstract domain of symbolic execution is thus $\mathcal{E}_{\mathbb{SE}} = (\mathcal{C}, \mathcal{V} \times \langle P \rangle, [\![.]\!]_{\mathbb{SE}})$.



(a) CFA (b) ARG

Figure 6: Example for symbolic execution

To compute the state space of a program P, symbolic execution traverses through the CFA of P: It starts at l_0 with the initial set $v_0 = \{\}$ of no abstract variable assignments and the set $pc = \{\}$ of no path constraints. If an assign operation $w := exp$ is encountered, w is assigned the evaluation $\mathsf{eval}_v(exp)$ of exp. To compute $\mathsf{eval}_v(exp)$, each program variable x in p is replaced with its abstract variable assignment $v(x)$. If a non-deterministic value [3] occurs in exp, symbolic execution introduces a new *symbolic value*. If no (new or existing) symbolic value occurs in $\mathsf{eval}_v(exp)$, the expression can be evaluated to a single integer. Otherwise, it is stored as-is. Whenever a program branch is encountered, symbolic execution splits into two separate executions; one following each branch. If an assume operation $[p]$ is encountered, the evaluation $\mathsf{eval}_v(p)$ of predicate p is appended to the path constraints pc. If the conjunction of path constraints is unsatisfiable, the current program path is infeasible and symbolic execution of this program path stops. If the conjunction of path constraints is satisfiable, then it describes, through its satisfying assignments, the class of concrete values that may be assumed for non-deterministic values on the path so that the program takes that path.

Figure 6 shows an example CFA and the abstract states computed by the corresponding symbolic execution. At the non-deterministic assignment $z := ?$, a new symbolic value s_1 is introduced and assigned to z. Next, at the branching conditions $[z > 0]$ and $[z \leq 0]$, both paths are feasible. Thus, symbolic execution splits into two separate executions. The first execution takes the left branch and adds $\mathsf{eval}_v(z > 0) = s_1 > 0$ to its path constraints. The second execution takes the right branch and adds $\mathsf{eval}_v(z \leq 0) = s_1 \leq 0$ to its path constraints. The first execution then encounters assignment $z := z - 1$. Since z is assigned the symbolic value s_1, the new symbolic value $\mathsf{eval}_v(z - 1) = s_1 - 1$ is assigned to z.

[3] For example, user input is a typical non-deterministic value in practice.

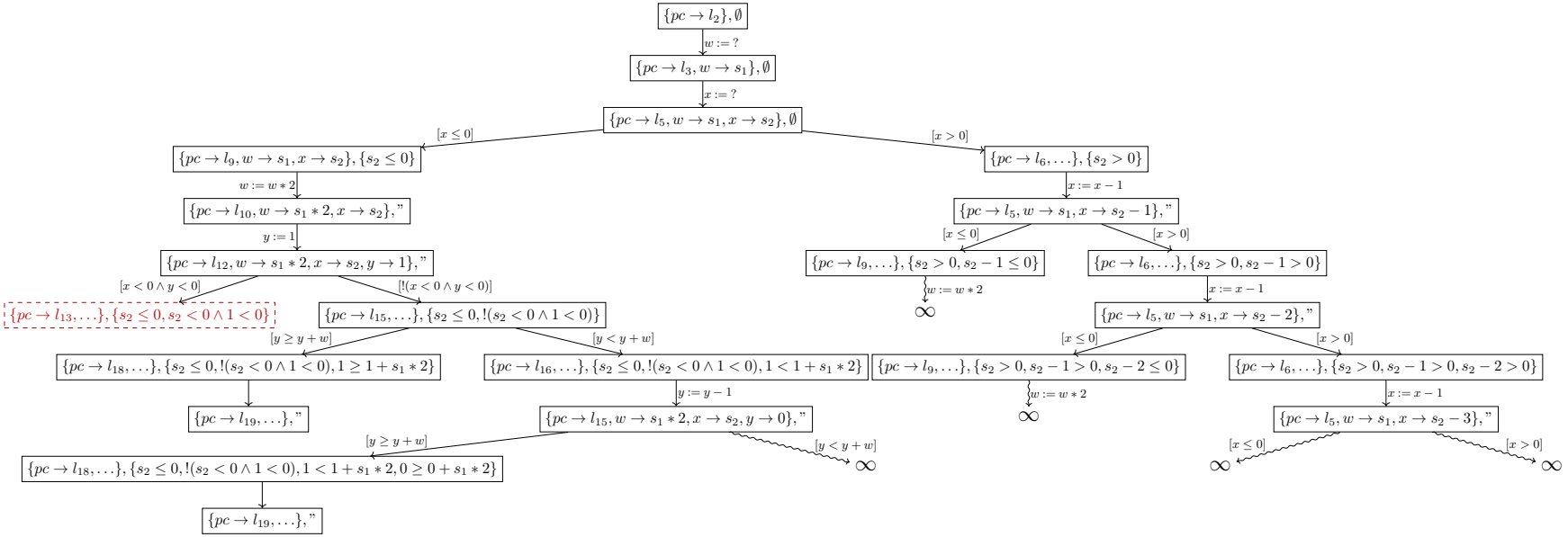


Figure 7: Part of the ARG computed by symbolic execution for our running example—the ARG has infinite elements.

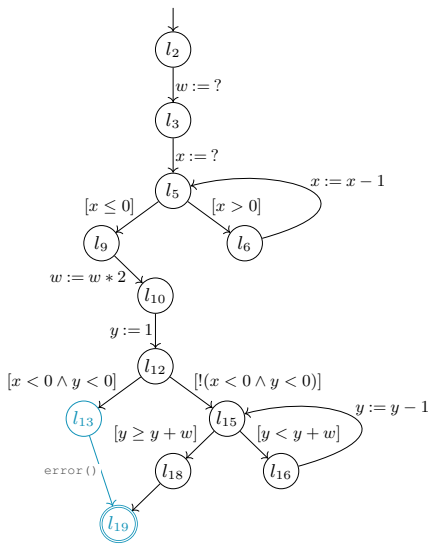
Symbolic execution continues in this fashion until every execution reaches the program exit location l_e. We can construct a set of test cases that covers all feasible paths of a program from the computed path constraints. For each execution, its final path constraints represent the set of values for which the program takes the same path through the program as the execution. Thus, if we take for each set of path constraints one satisfying assignment, we have exactly one variable assignment for each path through the program.

In general, symbolic execution may not always halt: if symbolic execution runs on our running example, it infinitely produces new states (Fig. 7). The running example (Fig. 4b) stays in the first loop until $x \leq 0$. Since x may be an integer of arbitrary size, it may stay in the loop for an arbitrary number of iterations. Every time the loop head l_5 is visited, symbolic execution splits into two execution runs: one execution leaves the loop and continues through the program (and encounters a similar problem at the next loop), the other execution stays in the loop and decreases x by 1. It then encounters l_5 again and splits again, since x could be 0 now, but may also still be positive. Symbolic execution continues this forever.

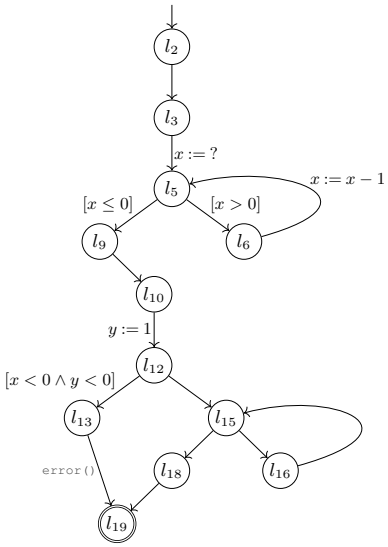
2.4 Program Slicing

As an example, we want to check the property that function `error` is never called. In our example program, `error` is only called at program location l_{13}. If we take a closer look at the program, we see that l_{13} is only reachable if $x < 0$ is true at l_{12} (Fig. 8a). If we then want to find out whether x can be less than 0, we see that the value of variable w has no influence on expression $x < 0$, and thus that we can ignore it. If we continue this process, we also find out that the right branch of l_{12}, including its potentially endless loop, can be ignored. A study [48] suggests that professional programmers perform such a backwards analysis throughout a program from a point of interest to the beginning of a program and only consider relevant parts of a program, whenever a program is hard to comprehend. Inspired by this human behavior, program slicing [50] takes an input program P and a slicing criterion c, and creates a slice of P on criterion c, denoted P/c. Slice P/c is a program that is a subset of P and behaviorally equivalent regarding slicing criterion c. We adjust this notion to our CFA representation of a program. Formally, program slicing takes an input CFA $A = (L, l_0, l_e, G)$ and a slicing criterion $c = \langle g,\ V \rangle$, with a CFA edge $g \in G$ and a set $V \subseteq X$ of program variables whose values are of interest. From these inputs, it creates a *slice* $A/c = (L, l_0, l_e, R)$ of A on slicing criterion c that has to fulfill the following two properties:

1. A/C can be obtained from A by deleting zero or more program operations. A program operation op at an edge $h = (l, op, l')$ is deleted by replacing h with $(l, \texttt{noop}, l')$.



(a) Original source code. The edge of special interest is highlighted.



(b) Program slice of the source code

Figure 8: Program slicing example

2. Whenever A exits on an input I, A/C also exits on I, and the values of all program variables in V at location l of program A are equivalent to the values of all program variables in V at location l of slice A/C. Note that program location l is the program location before executing the program operation op at edge $g = (l, op, l')$.

Figure 8b shows a possible slice for our example program on slicing criterion $< (l_{13}, \texttt{error()}, l_{19}), \{\} >$. The condition $[x < 0 \wedge y < 0]$ must be true to reach that edge. Since the values of program variable w have no influence on the values of x and y at l_{12}, all program operations on w can be removed. The same is true for all program operations at successor locations l_i with $i > 12$. The resulting slice only contains program operations relevant to our slicing criterion.

Program slicing is an abstraction of an original program that preserves the program semantics regarding a certain criterion. One trivial slice of any program P for an arbitrary slicing criterion is P itself. But just like with counterexample-guided abstraction refinement (CEGAR), we want to find an abstraction as coarse as possible, i.e., an abstraction that only contains program operations really necessary to preserve certain program semantics.

We use an algorithm [32] that computes smaller slices than the original program slicing algorithm [50]. This improved algorithm uses a *program dependence graph (PDG)* [29] to represent different types of dependencies between program operations. As a restriction, the algorithm only works on slicing criteria $c = \langle g,\ uses(op) \rangle$. i.e., a slicing criterion for a specific CFA edge $g = (l, op, l')$

always includes all program variables used by its program operation. This allows less concise slicing criteria, but produces more abstract slices for them.

2.4.1 Program Dependence Graph

A PDG is a program representation that connects program operations according to their dependences on other program operations, independent of their sequential order in a program. A PDG $DG_A = (N, E_D, E_C)$ for CFA $A = (L, l_0, l_e, G)$ is a directed graph that has the set $N = G \cup \{n_{entry}\}$ of nodes. Set N consists of the CFA edges G and a node n_{entry} that describes the program entry. The PDG DG_A has two different types of edges: the set $E_D \subseteq N \times N$ of flow-dependence edges and the set $E_C \subseteq N \times N$ of control-dependence edges.

Data Dependence For CFA edges g_1, g_2, edge $g_2 = (l_2, op_2, l_2')$ is *flow dependent* on $g_1 = (l_1, op_1, l_1')$ for variable $x \in X$, iff the following holds:

1. g_1 defines x, i.e., $x \in defs(op_1)$,
2. g_2 uses x, i.e., $x \in uses(op_2)$, and
3. there is a path from l_1' to l_2 without any new assignment to x, i.e., g_2 may use the definition of x at g_1.

Flow dependence can be computed in linear time. To get the list of reachable definitions of used variables at an operation, a reaching-definitions [1] analysis can be used. A reaching-definitions analysis computes the set RD_{l_k} of reachable definitions at each program location. A reachable definition $(l_i, x, l_j) \in RD_{l_k}$ says that at program location l_k, the definition of x at the program operation from program location l_i to program location l_j is reachable. This analysis can be augmented to compute flow dependences: At each CFA edge $g = (l_k, op, l_m)$ visited, the set of used program variables $uses(op)$ can be extracted by traversing through the AST of the expression of the edge's assignment or predicate, and collecting all program variables encountered. For each used program variable $x \in uses(op)$, and all reachable definitions $(l_i, x, l_j) \in RD_{l_k}$, CFA edge g is flow dependent on the CFA edge $(l_i, ., l_j)$.

If g_2 is flow dependent on g_1, then there is a flow-dependence edge from g_1 to g_2 in the PDG, i.e., $(g_1, g_2) \in E_D$.

Control Dependence For program locations $l, l' \in L$, l is *dominated* by l', if every path in A from l_0 to l contains l' and $l \neq l'$. Location l is *post-dominated* by l', if every path in A from l to l_e contains l' and $l \neq l'$. To compute post-dominators in a CFA, we can compute dominators on the reverse CFA.

We represent the post-dominator relations of the program locations in a CFA through a post-dominator tree $T = (L, l_e, E)$ with set L of program locations as nodes, program exit node $l_e \in L$ as root node and set $E \subseteq L \times L$ of tree edges. The post-dominator tree T determines for each program location the program locations it post-dominates and the program-locations it is post-dominated by.

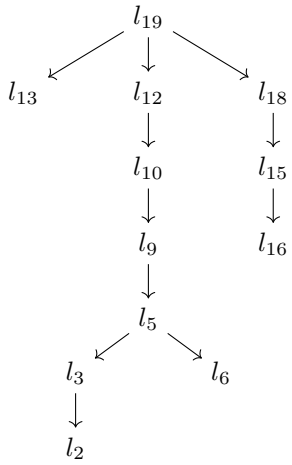


Figure 9: Post-dominator tree for running example

If a program location l post-dominates another program location l', then l is an ancestor of l' in T, i.e., there is a path in T from l to l'. If l is post-dominated by l', then l is a successor of l' in T. Since every program location is post-dominated by the program exit l_e, it is the root of the tree. The dominator tree of the program locations in a CFA is defined analogous. Figure 9 shows the post-dominator tree for our running example. The program exit node l_{19} is at the root of the tree - it post-dominates all other program locations. Program location l_{13} does not post-dominate any other program location, because every program location l_2 to l_{12} can take the alternate path $\langle l_{12} \to l_{15} \to l_{18} \to l_{19} \rangle$ through the CFA from l_{12} to l_{19}. Contrary, l_{18} post-dominates locations l_{15} and l_{16} because every path from either of them to l_{19} contains l_{18}. Program location l_{15} also post-dominates l_{16}. The program entry location l_2 can not post-dominate any other node, because it is for every path either the first node in it, or not contained in it. The ancestors of l_2 in the post-dominator tree describe all of the program locations that always have to be visited if a program execution halts.

A generic algorithm [38] for computing the dominator tree of all nodes of an arbitrary flow graph runs in $O(n\alpha(n))$, but we will later present an algorithm that fits our use-case of control dependence computation better.

CFA edge $g_2 = (l_2, ., l_2')$ is *control dependent* on $g_1 = (l_1, ., l_1')$, iff:

1. there exists a path P from l_1 to l_2 for which all $l \in P$ with $l \neq l_1, l \neq l_2$ are post-dominated by l_2, and
2. l_1 is not post-dominated by l_2.

If g_2 is control dependent on g_1, then there is a control-dependence edge from g_1 to g_2 in the PDG, i.e., $(g_1, g_2) \in E_C$. In addition, if a CFA edge g is not control dependent on any other CFA edge, then there is a control-dependence edge from the entry node n_{entry} to g, i.e., $(n_{entry}, g) \in E_C$.

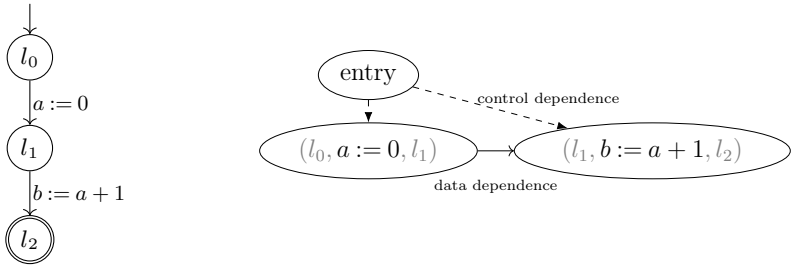


Figure 10: PDG for the CFA on the left

Figure 10 shows a small CFA and its PDG. We denote control-dependence edges with dashed arrows and flow-dependence edges with full arrows. CFA edge $g_2 = (l_1, b := a + 1, l_2)$ is flow-dependent on $g_1 = (l_0, a := 0, l_1)$, because

1. g_1 defines a
2. g_2 uses a, and
3. there is no re-definition of a between g_1 and g_2.

In addition, neither CFA edges g_1 or g_2 are control dependent on the other, so they are both control dependent on the entry node. Figure 11 shows the full dependence graph for our example program. Because of its size, we refrain from giving a detailed explanation of its construction.

System Dependence Graph For inter-procedural slicing, the program dependence graph can be extended to a *system dependence graph* [32] A system dependence graph [4] contains additional nodes: For each CFA edge g that contains an operation $\texttt{call}(a_1 \to p_1, \ldots, a_n \to p_n)$, the system dependence graph has an *actual-in* node for each function parameter p_i of the called function, in addition to the existing node for g itself. This allows a differentiation of dependencies to the individual function parameters. All actual-in nodes are control-dependent on g.

2.4.2 Slicing Algorithm

To compute a slice $A/c = (L, l_0, l_e, R)$ of $A = (L, l_0, l_e, G)$ on a single slicing criterion $c = \langle g,\ uses(op)\rangle$ with $g = (l, op, l')$, we first build the PDG $DG = (G, E_D, E_C)$ of A. Next, we use Algorithm 1 to compute the set H of CFA edges that g has a transitive flow- or control-dependence on, i.e., all CFA edges from which g is reachable in the PDG. It takes as input the CFA $A = (L, l_0, l_e, G)$, the PDG $DG = (G, E_D, E_C)$ corresponding to the CFA, and a single slicing criterion $c = \langle g, uses(op)\rangle$ with $g = (., op, .)$. For every CFA edge h visited, starting with $h = g$, it adds h to the set H of relevant CFA edges, marks h as visited, and adds to the *waitlist* of CFA edges to visit each CFA edge on

[4] We present a simplified version of the system dependence graph, adjusted to our notion of CFAs

Algorithm 1 Slice(A, DG, c). Adjusted worklist algorithm [32] for computation of relevant CFA edges

Input: CFA $A = (L, l_0, l_e, G)$, PDG $DG = (G, E_D, E_C)$ of A, slicing criterion $c = \langle g,\ uses(op)\rangle$ with $g = (., op, .)$
Output: set H of relevant CFA edges,
Variables: set *waitlist* of CFA edges to visit, CFA edges h, w

1: $H = \emptyset$
2: $waitlist = \{g\}$
3: **while** $waitlist \neq \emptyset$ **do**
4: pop h from $waitlist$
5: mark h as visited
6: $H = H \cup \{h\}$
7: **for all** $(w, h) \in E_D \cup E_C$ with w not marked visited **do**
8: $waitlist = waitlist \cup \{w\}$
9: **return** H

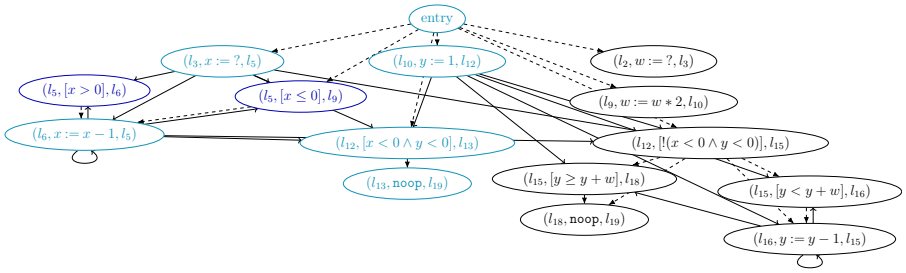


Figure 11: PDG of running example with relevant edges for previous slicing criterion highlighted

which h is flow- or control-dependent on and that was not yet visited. The algorithm returns H once it has visited all relevant CFA edges. This is the case if $waitlist = \emptyset$ after visiting a CFA edge. We then construct the set R of CFA edges of A/c using H. Set R contains all CFA edges of H, i.e., all CFA edges of the original CFA on which g has a transitive flow dependence or transitive control dependence. For each other edge $(l, op, l') \in G \setminus H$, set R contains a CFA edge $(l, \texttt{noop}, l')$. We can then build slice $A/c = (L, l_0, l_e, R)$ from R and the components of the original CFA $A = (L, l_0, l_e, G)$.

A slice $A/C = (L, l_0, l_e, R)$ of a CFA A on a set $C = \{c_1, \ldots, c_n\}$ of n slicing criteria c_1 to c_n can be computed by using the union H_1 to H_n of CFA edges relevant for the separate slicing criteria c_1 to c_n, i.e., $\bigcup_{i \in [1,n]} H_i$ and constructing the set R of CFA edges of A/C based on this union.

Figure 11 shows the PDG of our running example. We highlighted the CFA edges visited during computation of relevant CFA edges H for slice A/c with $c = \langle g,\ \{\}\rangle$ and $g = (l_{13}, \texttt{error()}, l_{19})$. The relevant CFA edges that $(l_{12}, [x < 0 \wedge y < 0], l_{13})$ is directly dependent on are highlighted in a light blue color. The two edges $(l_5, [x > 0], l_6)$ and $(l_5, [x \leq l_9], l_9)$ are dependences of

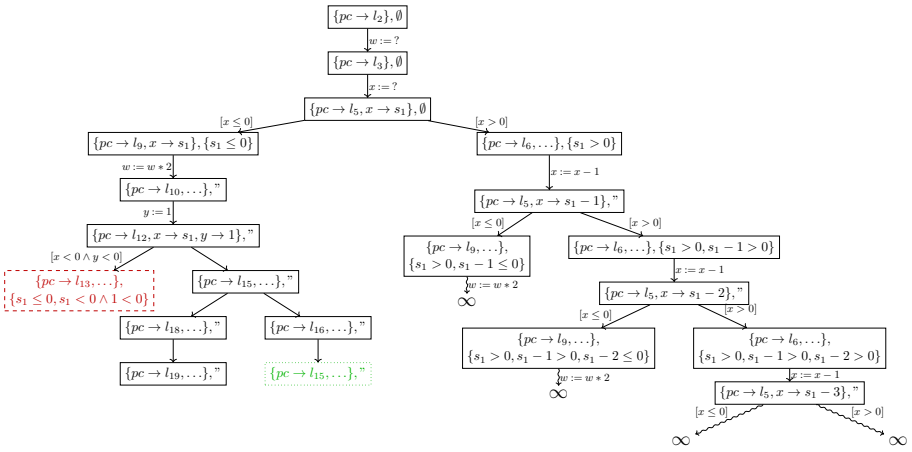


Figure 12: Part of the ARG computed by symbolic execution for the computed slice of our running example—the ARG still has infinite elements in the right subtree, but stops at the loop in the left subtree after one iteration due to the higher level of abstraction.

edge $(l_6, x := x - 1, l_5)$, and thus also part of the transitive dependences of g. These two nodes are highlighted in a dark blue color. Using this set of relevant CFA edges, we can construct R and get the slice A/c as seen in Fig. 8b. If we run symbolic execution on this slice, it computes the state space as shown in Fig. 12. In the left subtree of program location l_5, every node of the loop starting at l_{15} is only visited once due to the higher abstraction. When traversing this loop and visiting l_{15} the second time, analysis stops since no new information is computed (abstract state in green color and dotted lines). This way, the previously infinite amount of abstract states in this subtree is reduced to only 4 states. Symbolic execution still produces an infinite amount of states for the loop starting at l_5, though, since no additional abstraction happens there.

Performance

2.5 Counterexample-guided Abstraction Refinement

The strongest-post operator $\mathsf{SP}_{op_i}(\delta)$ for a program operation op_i and an abstract state δ represents the most concrete abstract state that can hold after applying op_i to δ. For a program path $\sigma = \langle l_0 \xrightarrow{op_0} \dots \xrightarrow{op_{n-1}} l_n \rangle$ and an initial restriction δ_0, we define the strongest-post operator $\mathsf{SP}_\sigma(\delta_0)$ as $\mathsf{SP}_\sigma(\delta_0) = \mathsf{SP}_{\langle l_1 \xrightarrow{op_1} \dots \xrightarrow{op_{n-1}} l_n \rangle}(\mathsf{SP}_{op_0}(\delta_0))$.

CEGAR [22] is a technique that aims to automatically derive an abstraction to a program that is appropriate for proving a program safe or unsafe. It starts with an initial abstraction of the program that may be too abstract for reliable verification. If a violation of a safety property is reported during verification, a counterexample can be given as potential proof. CEGAR checks whether this

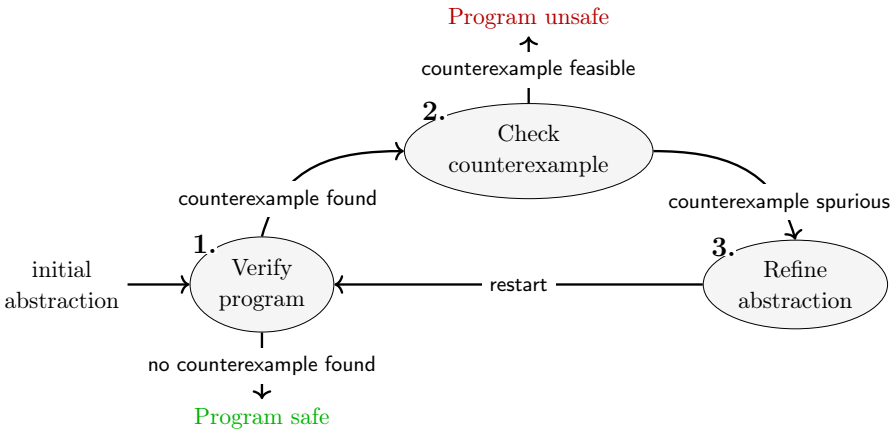
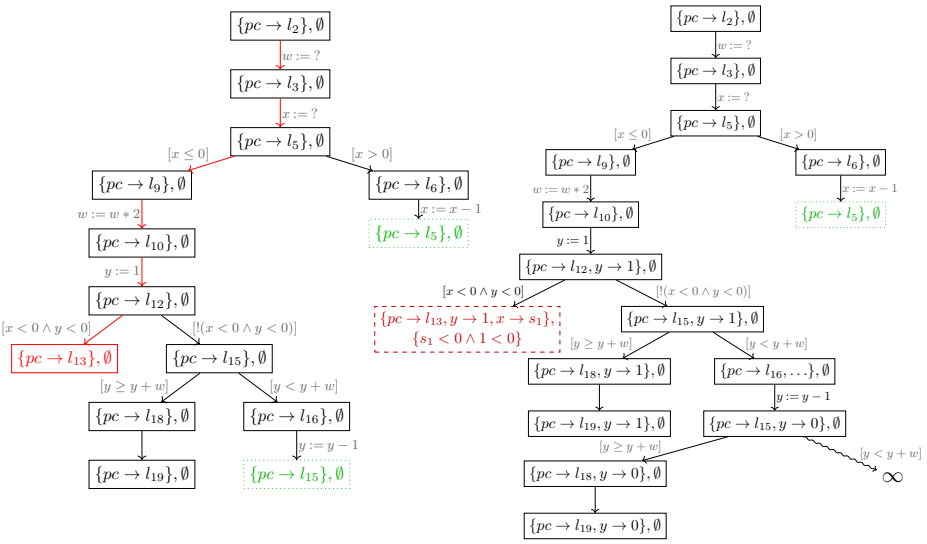


Figure 13: Concept of CEGAR

counterexample is feasible on the original program. If it is, the counterexample is a true counterexample and verification has found a property violation. If it is not, the counterexample is *spurious* and we have an example for which our abstraction is too abstract. CEGAR then refines the abstraction based on this spurious counterexample, i.e., makes its abstraction more concrete/less abstract. It then restarts verification with the new abstraction. Figure 13 illustrates this iterative approach. One technique of performing abstraction is *precision adjustment* [41]. A precision adjustment operator prec uses a given *precision* π and modifies each abstract state according to π. This happens directly after each abstract state is computed, so that the further computation is performed on the modified state. The concrete functionality of prec and the type Π of π depend on the abstract domain used by a verification technique. Since the concrete adjustment depends on π, π describes the abstraction.

One successful technique for abstraction refinement based on a counterexample is *Craig interpolation* [26]. CEGAR was originally created to tackle path explosion of symbolic model checking [24], but has since been extended to other domains, e.g., explicit-state model checking [13] and symbolic execution [11].

Figure 14 shows symbolic execution with CEGAR applied to our running example. In the first iteration (Fig. 14a), the initial precision of symbolic execution does track no variable assignments and no path constraints. As a result, the computed state-space is finite, but symbolic execution reports that the function call at location l_{13} is reachable, due to the abstraction. The reported error path is marked in the color red in Fig. 14a. CEGAR checks this error path with full precision of symbolic execution and finds that it actually is infeasible. Thus, precision is refined: since only $y{:=}1$ and the branching condition $x < 0 \wedge y < 0$ must be tracked to prove the function call at l_{13} unreachable, the new precision is $\pi_{\mathbb{SE}} = (\{y\}, \{x < 0 \wedge y < 0\})$. This precision tells symbolic execution to track all variable assignments to program variable y and the constraint $x < 0 \wedge y < 0$. CEGAR now starts a new symbolic execution run with this new $\pi_{\mathbb{SE}}$. This time,



(a) First iteration (b) Second iteration

Figure 14: Iterations of CEGAR

the function call to `error()` at location l_{13} is correctly computed as infeasible. But, since all assignments to program variable y are tracked, symbolic execution unrolls the loop starting at l_{15}; this again results in an infinite state-space.

2.6 Configurable Program Analysis

The concept of configurable program analysis (CPA) [7] is a flexible approach to formal program verification that combines the notions of model checking [23] and program analysis [1]. The CPA approach consists of the CPA+ algorithm (Algorithm 2) and CPAs. The CPA+ algorithm [8] is a waitlist-based reachability algorithm that provides a framework for state-space exploration and dynamic precision adjustment of the abstract domain. A CPA influences different aspects of the CPA+ algorithm and determines the shape of the explored state space.

2.6.1 CPA Algorithm

The CPA+ algorithm takes as input a CPA $\mathbb{D}$, an initial set R_0 of reachable abstract states (usually the initial abstract state e_0) and an initial set W_0 of frontier abstract states that have been computed, but that have not been visited yet (usually also the initial abstract state e_0). First, the set reached of all computed reachable abstract states and the waitlist of abstract states that must be visited are initialized with R_0 and W_0, correspondingly. Until the waitlist is empty, an element e and its precision π are picked (and removed) from the waitlist by some arbitrary strategy. The algorithm then computes all abstract successor states of e based on the transfer relation $\rightsquigarrow$ of the given CPA $\mathbb{D}$. Each abstract successor state e' is first adjusted according to the precision π, using the

Algorithm 2 CPA+$(\mathbb{D}, R_0, W_0)$, adapted from [39]

Input: a CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$, an initial reached set $R_0 \in E \times \Pi$, and an initial waitlist $W_0 \in E \times \Pi$, where E set of elements of D

Output: pair $(\mathsf{reached}, \mathsf{waitlist})$ of the set $\mathsf{reached} \subseteq E \times \Pi$ of computed reachable abstract states and their precisions and the set $\mathsf{waitlist} \subseteq E \times \Pi$ of frontier abstract states that haven't been visited yet, and their precisions

1: $\mathsf{reached} := W_0$
2: $\mathsf{waitlist} := R_0$
3: **while** $\mathsf{waitlist} \neq \varnothing$ **do**
4: choose and remove (e, π) from $\mathsf{waitlist}$
5: **for all** e' with $e \rightsquigarrow (e', \pi)$ **do**
6: $(\widehat{e}, \widehat{\pi}) = \mathsf{prec}(e', \pi, \mathsf{reached})$
7: **if** `isTargetState`$(\widehat{e})$ **then**
8: **return** $(\mathsf{reached} \cup \{(\widehat{e}, \widehat{\pi})\}, \mathsf{waitlist} \cup \{(\widehat{e}, \widehat{\pi})\})$
9: **for all** $(e'', \pi'') \in \mathsf{reached}$ **do**
10: $e_{new} := \mathsf{merge}(\widehat{e}, e'', \widehat{\pi})$
11: **if** $e_{new} \neq e''$ **then**
12: $\mathsf{waitlist} := (\mathsf{waitlist} \cup \{(e_{new}, \widehat{\pi})\}) \setminus \{(e'', \pi'')\}$
13: $\mathsf{reached} := (\mathsf{reached} \cup \{(e_{new}, \widehat{\pi})\}) \setminus \{(e'', \pi'')\}$
14: **if** $\neg\mathsf{stop}(\widehat{e}, \{e \mid (e, \cdot) \in \mathsf{reached}\}, \widehat{\pi})$ **then**
15: $\mathsf{waitlist} := \mathsf{waitlist} \cup \{(\widehat{e}, \widehat{\pi})\}$
16: $\mathsf{reached} := \mathsf{reached} \cup \{(\widehat{e}, \widehat{\pi})\}$
17: **return** $\mathsf{reached}$

precision adjustment operator prec of $\mathbb{D}$. The result is a more abstract state $\widehat{e}$ and its precision $\widehat{\pi}$. State $\widehat{e}$ may represent a target state, i.e., a state that is in the target region. If this is the case, the algorithm stops and returns the computed reached set and the remaining waitlist (both including the just computed $(\widehat{e}, \widehat{\pi})$). Otherwise, each pair $(\widehat{e}, \widehat{\pi})$ is used for two computations: First, each already computed, reachable state e'' for $(e'', \pi'') \in \mathsf{reached}$ is *merged* with $\widehat{e}$ using the merge operator of $\mathbb{D}$ and the current precision $\widehat{\pi}$. If the result e_{new} of the merge is different from e'', e'' is removed from the reached set and (potentially) from the waitlist and e_{new} with precision $\widehat{\pi}$ are added to both, even if e'' was already visited and not in the waitlist anymore. After performing this merge for all states of $\mathsf{reached}$, the algorithm checks whether $\widehat{e}$ is already covered by $\mathsf{reached}$. To do this, it uses the stop operator, whose behavior also depends on $\mathbb{D}$. If $\widehat{e}$ is not yet covered by $\mathsf{reached}$, it is added to both the waitlist and the reached set. The algorithm then continues with the next pair of state and precision in the waitlist, until a target state is found, or the full state space is explored (i.e., no more frontier states exist, the waitlist is empty).

2.6.2 CPA

A CPA [8] $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with dynamic precision adjustment consists of an abstract domain D, the set Π of precisions, a transfer relation $\rightsquigarrow$, the merge operator, the stop operator and the prec operator.

Abstract Domain For soundness, the abstract domain $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$ (c.f. Sect. 2.2.3) with semi-lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$ has to fulfill the following requirements:

1. $[\![\top]\!] = \mathcal{C}$
2. $\forall e, e' \in E : e \sqsubseteq e' \Rightarrow [\![e]\!] \subseteq [\![e']\!]$
3. $\forall e, e' \in E : [\![e \sqcup e']\!] \supseteq [\![e]\!] \cup [\![e']\!]$

Whenever we define a new abstract domain, we will not give a definition of the concretization function for conciseness.

Precision Set The set Π of precisions determines the precisions the CPA uses. The precision adjustment operator prec uses precisions, which decides which information is tracked by the analysis.

Transfer Relation The transfer relation $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns to each abstract state $e \in E$ all possible abstract successor states $e' \in E$ with precision $\pi \in \Pi$, based on a control-flow edge $g \in G$. If $(e, g, e', \pi) \in \rightsquigarrow$, we write $e \overset{g}{\rightsquigarrow} (e', \pi)$. If $g \in G$ with $e \overset{g}{\rightsquigarrow} (e', \pi)$, we write $e \rightsquigarrow (e', \pi)$.

For soundness, the transfer relation has to fulfill, that

$$\forall e \in E, g \in G : \bigcup_{e \overset{g}{\rightsquigarrow} (e', .)} [\![e']\!] \supseteq \bigcup_{c \in [\![e]\!]} \{c' \mid c \xrightarrow{g} c'\}$$

Merge Operator The merge operator $\mathsf{merge} : E \times E \times \Pi \to E$ weakens the information of the given second abstract state based on the first abstract state. The returned, weakened abstract state has the given precision. For soundness, merge may only weaken a state, i.e., return an abstract state that is equal to or more abstract than the given second abstract state. Formally,

$$\forall e, e' \in E, \pi \in \Pi : e' \sqsubseteq \mathsf{merge}(e, e', \pi) \ .$$

Two common merge operators are merge^{sep} and merge^{join}:

$$\mathsf{merge}^{sep}(e, e', \pi) = e'$$
$$\mathsf{merge}^{join}(e, e', \pi) = e \sqcup e'$$

Operator merge^{sep} does not weaken any abstract state—it returns the same abstract state that was given as second parameter; merge^{join} weakens the given second abstract state using the join operator $\sqcup$.

Stop Operator The stop operator $\mathsf{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ determines whether a given abstract state $e \in E$ with a precision $\pi \in \Pi$ is covered by a given set $R \in 2^E$ of abstract states. If e with π is covered by R, $\mathsf{stop}(e, R, \pi)$ returns *true*. Otherwise, it returns $false$.

Two common stop operators are stop^{sep} and stop^{join}:

$$\mathsf{stop}^{sep}(e, R, \pi) = \exists e' \in R : e \sqsubseteq e'$$
$$\mathsf{stop}^{join}(e, R, \pi) = e \sqsubseteq \bigsqcup R$$

Operator stop^{sep} checks every abstract state in R separately, while stop^{join} first joins all states in R and then checks whether e is smaller than that join. For soundness, stop has to fulfill that

$$\forall e \in E, R \in 2^E, \pi \in \Pi : \mathsf{stop}(e, R, \pi) = true \Rightarrow [\![e]\!] \subseteq \bigcup_{e' \in R} [\![e']\!]$$

Because of this requirement, operator stop^{join} can only be used with abstract domains for which $[\![e \sqcup e']\!] = [\![e]\!] \cup [\![e']\!]$ (so-called power-set domains).

Precision Adjustment The precision adjustment operator $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ computes a new abstract state and precision for a given abstract state $e \in E$ based on a given precision $\pi \in \Pi$ and a set $\mathsf{reached} \in 2^{E \times \Pi}$ of abstract states with their corresponding precisions.

For soundness, the computed new abstract state has to represent a subset of the concrete states the given abstract state represents, i.e.:

$$\forall e, \widehat{e} \in E, \pi, \widehat{\pi} \in \Pi, \mathsf{reached} \in 2^{E \times \Pi} : (\widehat{e}, \widehat{\pi}) = \mathsf{prec}(e, \pi, \mathsf{reached}) \Rightarrow [\![e]\!] \subseteq [\![\widehat{e}]\!] \ .$$

2.6.3 Composite CPA

The composite CPA $\mathbb{C} = (D_\times, \Pi_\times, \rightsquigarrow_\times, \mathsf{merge}_\times, \mathsf{stop}_\times, \mathsf{prec}_\times)$ allows the combination of multiple CPAs through composition. This allows a separation of concerns as well as the combination of strengths of different abstract domains. For the composition of two CPAs [5] $\mathbb{D}_1 = (D_1, \Pi_1, \rightsquigarrow_1, \mathsf{merge}_1, \mathsf{stop}_1, \mathsf{prec}_1)$ and $\mathbb{D}_2 = (D_2, \Pi_2, \rightsquigarrow_2, \mathsf{merge}_2, \mathsf{stop}_2, \mathsf{prec}_2)$, with $D_1 = (\mathcal{C}, \mathcal{E}_1, [\![\cdot]\!]_1)$ and $D_2 = (\mathcal{C}, \mathcal{E}_2, [\![\cdot]\!]_2)$, $\mathcal{E}_1 = (E_1, \top_1, \sqsubseteq_1, \sqcup_1)$ and $\mathcal{E}_2 = (E_2, \top_2, \sqsubseteq_2, \sqcup_2)$, the composite CPA $\mathbb{C}$ has abstract domain $D_\times = D_1 \times D_2 = (\mathcal{C}, \mathcal{E}_\times, [\![\cdot]\!]_\times)$ with semi-lattice $\mathcal{E}_\times = \mathcal{E}_1 \times \mathcal{E}_2 = (E_1 \times E_2, (\top_1, \top_2), \sqsubseteq_\times, \sqcup_\times)$. The precision $\Pi_\times$ is a composition of the precisions Π_1 and Π_2. For

1. composite transfer relation $\rightsquigarrow_\times \subseteq (E_1 \times E_2) \times G \times (E_1 \times E_2) \times \Pi_\times$,
2. composite merge operator $\mathsf{merge}_\times : (E_1 \times E_2) \times (E_1 \times E_2) \times \Pi_\times \to (E_1 \times E_2)$,
3. composite stop operator $\mathsf{stop}_\times : (E_1 \times E_2) \times 2^{E_1 \times E_2 \times \Pi_\times} \to (E_1 \times E_2) \times \Pi_\times$, and

[5] This notion can be generalized to an arbitrary amount of component CPAs.

4. composite precision adjustment operator

$$\mathsf{prec}_\times : (E_1 \times E_2) \times \Pi_\times \times 2^{(E_1 \times E_2) \times \Pi_\times} \to (E_1 \times E_2) \times \Pi_\times \ ,$$

the operations of the component CPAs $\mathbb{D}_1$ and $\mathbb{D}_2$ may be used. In addition, they may use (a) the strengthening operator $\downarrow$ and (b) the compare relation $\preceq$.

Strengthening Operator The strengthening operator $\downarrow : E_1 \times E_2 \to E_1$ computes a stronger abstract state of type E_1 by using information from a second abstract state of type E_2. It allows the use of a transfer relation $\rightsquigarrow_\times$ that is stronger than the simple composition of $\rightsquigarrow_1$ and $\rightsquigarrow_2$. The computed abstract state has to be more concrete than the first given abstract state, i.e., $\downarrow(e, e') \sqsubseteq_1 e$.

Compare Relation The compare relation $\preceq$: $E_1 \times E_2$ allows the comparison of two abstract states of different types.

Merge Operator A common merge operator for composite analysis is merge^{agree}, which merges the components of a composite abstract state based on the corresponding merge operators merge_1 and merge_2 under one condition: Both merges have to weaken the given second abstract state so that it is less or equal to both given abstract states. Formally,

$$\mathsf{merge}^{agree}((e_1, e_2), (e_1', e_2'), (\pi_1, \pi_2)) =$$
$$\begin{cases} (\mathsf{merge}_1(e_1, e_1', \pi_1), \mathsf{merge}_2(e_2, e_2', \pi_2)) & \text{if } \mathsf{merge}_1(e_1, e_1', \pi_1) \sqsubseteq_1 e_1, e_1' \\ & \text{and } \mathsf{merge}_2(e_2, e_2', \pi_2) \sqsubseteq_2 e_2, e_2' \\ (e_1', e_2') & \text{otherwise} \end{cases}$$

2.6.4 Location CPA

The location CPA [8] $\mathbb{L} = (D_\mathbb{L}, \widetilde{\Pi}, \rightsquigarrow_\mathbb{L}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ represents the syntactic position in a program. It can be used to analyze the syntactic reachability of program locations, and is mainly used in composition with other CPAs to track the program location and thus the control flow. This enables simpler other CPAs, since they don't have to handle location tracking. The location CPA consists of the following components.

Location CPA Domain The abstract domain $D_\mathbb{L} = (\mathcal{C}, L, [\![\cdot]\!]_\mathbb{L})$ consists of the set $\mathcal{C}$ of concrete program states, the semi-lattice L and concretization function $[\![\cdot]\!]_\mathbb{L}$. Semi-lattice $L = (L \cup \{\top_\mathbb{L}\}, \top_\mathbb{L}, \sqsubseteq_\mathbb{L}, \sqcup_\mathbb{L})$ consists of: The set $L \cup \{\top_\mathbb{L}\}$ of all program locations and a top element $\top_\mathbb{L}$ that represents all possible program locations; the less-or-equal relation $\sqsubseteq_\mathbb{L}$ that contains $l \sqsubseteq_\mathbb{L} l'$ if $l = l'$, or $l' = \top_\mathbb{L}$. the join operator $\sqcup_\mathbb{L}$ with

$$l \sqcup_\mathbb{L} l' = \begin{cases} l & \text{if } l = l' \\ \top_\mathbb{L} & \text{otherwise} \end{cases} .$$

Static Precision The precision $\widetilde{\Pi}$ of the location CPA only contains a single precision $\widetilde{\pi}$ that represents that all information is tracked: $\widetilde{\Pi} = \{\widetilde{\pi}\}$. We call this precision *static precision*.

Location Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{L}}$ has transfer $l \overset{g}{\rightsquigarrow}_{\mathbb{L}} (l', \widetilde{\pi})$ if $g = (l, \cdot, l')$.

Location CPA Merge The location CPA uses the merge operator merge^{sep} that never merges states.

Location CPA Stop The location CPA uses the stop operator stop^{sep} that considers each state separately.

Location CPA Precision Adjustment The location CPA never weakens states: $\widetilde{\mathsf{prec}}(l, \widetilde{\pi}) = (l, \widetilde{\pi})$.

2.6.5 Callstack CPA

The callstack CPA keeps track of the callstack in a program. It is a CPA $\mathcal{F} = (D_{\mathcal{F}}, \widetilde{\Pi}, \rightsquigarrow_{\mathcal{F}}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$ with the following components.

Callstack Domain The abstract domain $D_{\mathcal{F}} = (\mathcal{C}, \mathcal{E}_{\mathcal{F}}, [\![\cdot]\!]_{\mathcal{F}})$ consists of the set $\mathcal{C}$ of concrete program states, the semi-lattice $\mathcal{E}_{\mathbb{SE}}$ and the concretization function $[\![\cdot]\!]_{\mathbb{SE}}$.

An element of the semi-lattice $\mathcal{E}_{\mathcal{F}} = (\Xi \cup \{\top_{\mathcal{F}}\}, \top_{\mathcal{F}}, \sqsubseteq_{\mathcal{F}}, \sqcup_{\mathcal{F}})$ is a stack $\xi = [l_n, \ldots, l_m]$ of function call sites $l_n, \ldots, l_m \in L$, where elements are always added to the left, i.e., l_m is the function call site of the first function that was called and that was not returned from yet, and l_n is the function call site of the function we are currently in. The top element $\top_{\mathcal{F}}$ is a special element that represents that we could be in any function call stack. The less-or-equal relation contains $\xi \sqsubseteq_{\mathcal{F}} \xi'$, if $\xi' = \xi$ or $\xi' = \top_{\mathcal{F}}$. The join is defined as $\xi \sqcup_{\mathcal{F}} \xi' = \begin{cases} \xi & \text{if } \xi = \xi' \\ \top_{\mathcal{F}} & \text{otherwise} \end{cases}$.

Callstack Precision The precision $\widetilde{\Pi}$ of the callstack CPA is the static precision.

Callstack Transfer Relation The callstack transfer relation $\rightsquigarrow_{\mathcal{F}}$ has transfer $\xi \overset{g}{\rightsquigarrow}_{\mathcal{F}} (\xi', \widetilde{\pi})$ for $g = (l, op, l')$, if:

1. $op = [p]$ or $op = x := w$ and $\xi' = \xi$,
2. $op = \texttt{call}(f, \ldots)$ and $\xi' = [l] + \xi$, or
3. $op = w := \texttt{ret}(l_i, x)$, $\xi = [l_i, l_j, \ldots]$ and $\xi' = [l_j, \ldots]$.

Callstack CPA Merge The callstack CPA uses the merge operator merge^{sep} that never merges states.

Callstack CPA Stop The callstack CPA uses the stop operator stop^{sep} that considers each state separately.

Callstack CPA Precision Adjustment The location CPA never weakens states; it uses the static precision adjustment operator $\widetilde{\mathsf{prec}}$.

2.6.6 Symbolic Execution CPA

The symbolic execution CPA [11] implements symbolic execution in the CPA framework. It is a CPA $\mathbb{SE} = (D_{\mathbb{SE}}, \Pi_{\mathbb{SE}}, \rightsquigarrow_{\mathbb{SE}}, \mathsf{merge}^{sep}, \mathsf{stop}^{sep}, \mathsf{prec}_{\mathbb{SE}})$.

Symbolic Execution Domain The abstract domain $D_{\mathbb{SE}} = (\mathcal{C}, \mathcal{E}_{\mathbb{SE}}, [\![\cdot]\!]_{\mathbb{SE}})$ consists of the set $\mathcal{C}$ of concrete program states, the semi-lattice $\mathcal{E}_{\mathbb{SE}}$ and the concretization function $[\![\cdot]\!]_{\mathbb{SE}}$.

The elements of the semi-lattice $\mathcal{E}_{\mathbb{SE}} = (\mathcal{V} \times \langle P \rangle), \top_{\mathbb{SE}}, \sqsubseteq_{\mathbb{SE}}, \sqcup_{\mathbb{SE}})$ are a tuple (v, pc) of an abstract variable assignment $v \in \mathcal{V}$ and path constraints pc. The less-or-equal relation $\sqsubseteq_{\mathbb{SE}}$ contains $(v, pc) \sqsubseteq_{\mathbb{SE}} (v', pc')$ if both of the following holds:

1. $\text{def}(v') \subseteq \text{def}(v)$ and for all $x \in \text{def}(v') : v(x) = v'(x)$; and
2. $pc' \subseteq pc$.

This relation also implies the join $\sqcup_{\mathbb{SE}}$.

Symbolic Execution Precision Set The set $\Pi_{\mathbb{SE}} = 2^X \times 2^{\mathcal{P}}$ defines the precisions of the symbolic execution CPA as tuples $(\pi_X, \pi_{\mathcal{P}})$. Component π_X determines the abstract variable assignments to track, and $\pi_{\mathcal{P}}$ determines the constraints to track.

Symbolic Execution Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{SE}}$ contains transfer $(v, pc) \overset{g}{\rightsquigarrow}_{\mathbb{SE}} (v', pc', (\pi_X, \pi_{\mathcal{P}}))$ if one of the following three holds:

1. $g = (., w := exp, .)$, $pc = pc'$ and

$$v'(x) = \begin{cases} exp_{/v} & \text{if } x = w \text{ and for each program variable } x \\ & \text{that occurs in } exp,\ x \in \text{def}(v) \\ v(x) & \text{if } x \in \text{def}(v) \wedge x \neq w \end{cases}$$

   where $exp_{/v}$ is the evaluation of expression exp based on abstract variable assignment v. To evaluate exp, we replace the occurrence of every program variable $x \in \text{def}(v)$ in exp with its assignment $v(x)$. If no symbolic value occurs in $exp_{/v}$, the expression can be evaluated to a single integer. Otherwise, the expression is stored as symbolic value.

2. $g = (., [p], .)$, $v = v'$, $pc' = pc \cup p_{/v}$ and $\bigwedge_{p \in pc'} p \neq false$, where $p_{/v}$ is the evaluation of predicate p based on abstract variable assignment v, analogous to the evaluation of expressions. If the conjunction of path constraints is unsatisfiable, no successor state exists.

3. $g = (., op, .)$, op is a `noop`, call or return operation, $v' = v$ and $pc' = pc$.

Symbolic Execution Merge and Stop Operator The symbolic execution CPA does not merge abstract states when the control flow meets (i.e., it uses merge^{sep}). It checks each abstract state individually in the stop operator (stop^{sep}).

Symbolic Execution Precision Adjustment The precision adjustment operator $\mathsf{prec}_{\mathbb{SE}}$ adjusts both the abstract variable assignment and the path constraints according to a given precision:

$$\mathsf{prec}_{\mathbb{SE}}(v, pc) = (v_{|\pi_X}, pc \cap \pi_{\mathcal{P}})$$

for precision $\pi = (\pi_X, \pi_{\mathcal{P}}) \in \Pi_{\mathbb{SE}}$. It only keeps abstract variable assignments for program variables x that are in the first component of the precision, i.e., $x \in \pi_X$, and only path constraints p that are in the second component of the precision, i.e., $p \in \pi_{\mathcal{P}}$.

2.6.7 Reaching Definitions CPA

The reaching definitions CPA [6] tracks possibly active definitions of program variables. It is a CPA $\mathbb{RD} = (D_{\mathbb{RD}}, \widetilde{\Pi}, \rightsquigarrow_{\mathbb{RD}}, \mathsf{merge}^{join}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$.

Reaching Definitions Domain The abstract domain $D_{\mathbb{RD}} = (\mathcal{C}, \mathcal{E}_{\mathbb{RD}}, [\![\cdot]\!]_{\mathbb{RD}})$ consists of the set $\mathcal{C}$ of concrete states, the semi-lattice $\mathcal{E}_{\mathbb{RD}}$ and the concretization function $[\![\cdot]\!]_{\mathbb{RD}}$. The semi-lattice $\mathcal{E}_{\mathbb{RD}} = (2^{\mathsf{E}}, \top_{\mathbb{RD}}, \sqsubseteq_{\mathbb{RD}}, \sqcup_{\mathbb{RD}})$ consists of the set 2^{E} of all sets of reaching definitions, with $\mathsf{E} = X \times (L \times L)$. A reaching definition $(x, (l, l')) \in \mathsf{E}$ pairs a program variable x with a CFA edge that goes from l to l' and that contains a defining program operation. The top element $\top_{\mathbb{RD}} = \mathsf{E}$ represents all possible definitions for every program variable. the less-or-equal operator $\sqsubseteq_{\mathbb{RD}}$ contains $S \sqsubseteq_{\mathbb{RD}} S'$ if $S \subseteq S'$, i.e., if S contains a subset of the possible reaching definitions that S' contains, and the join operator $\sqcup_{\mathbb{RD}}$ is defined as $S \sqcup_{\mathbb{RD}} S' = S \cup S'$.

Reaching Definitions Precision Set The precision set of the reaching-definitions CPA is the static precision $\widetilde{\Pi}$. Only one precision exists.

Reaching Definitions Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{RD}}$ contains $S \overset{g}{\rightsquigarrow}_{\mathbb{RD}} (S', \widetilde{\pi})$, if one of the following is true:

1. $g = (., [p], .)$ and $S' = S$

2. $g = (l, op, l')$ with $op = w := exp$, and

$$S' = (S \setminus \{(w, k, k') \mid k, k' \in L\}) \cup \{(w, l, l')\}$$

3. $g = (l, op, l')$ with $op = \texttt{call}(f, a_1 \to p_1, \ldots, a_n \to p_n)$ and

$$S' = S \cup \{(a_i, l, l') \mid 1 \leq i \leq n\}$$

4. $g = (l, op, l')$ with $op = w := \texttt{ret}(l_i, x)$ and

$$S' = (S \setminus \{(a_i, k, k') \mid k, k' \in L, 1 \leq i \leq n\}) \cup \{(w, l, l')\}$$

with the corresponding CFA edge

$$g = (l_i, \texttt{call}(f, a_1 \to p_1, \ldots, a_n \to p_n), l_j) \ .$$

Reaching Definitions Merge Operator The reaching-definitions CPA uses merge^{join}—it always joins abstract states when the control flow meets.

Reaching Definitions Stop Operator The stop operator of the reaching-definitions CPA checks each abstract state individually (i.e., stop^{sep}).

Reaching Definitions Static Precision Adjustment The precision adjustment operator of the reaching-definitions CPA never performs abstraction; it is the static precision adjustment operator $\widetilde{\mathsf{prec}}(S) = S$.

2.7 Relevant Technology and Tools

2.7.1 LLVM

The LLVM [6] compiler infrastructure aims to provide a modern, SSA-based compilation strategy, aimed at a flexible set of source languages. The center of the LLVM project is the LLVM assembly language [7] (just called LLVM). All source languages are translated to LLVM as intermediate language. Compiler optimizations and other compiler steps are then performed on the LLVM representation. Since our work is based on analysis of C programs, we will focus on the aspects of LLVM relevant for C, only. LLVM code is stored in two different formats: Human-readable code, and so-called bitcode. LLVM is a static single assignment (SSA) based, type-safe program representation over a small set of low-level operations. Due to its assembly-like nature, understanding LLVM involves a significant amount of low-level technical knowledge. For our purpose, this is not necessary - thus, we will ignore technical details like data layouts, compiler information and metadata. Instead, we will give a short overview over the main components of the language that are necessary to understand its connection to C.

[6] https://llvm.org/ [7] https://llvm.org/docs/LangRef.html

```c
int x = 10;

int decr(int x) {
    return x - 1;
}

int main() {
    while (x > 0) {
        x = decr(x);
    }
}
```

(a) C program

```
@x = global i32 10, align 4

; Function Attrs: nounwind uwtable
define i32 @decr(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = sub nsw i32 %3, 1
  ret i32 %4
}

; Function Attrs: nounwind uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %2

; <label>:2:
  ; preds = %5, %0
  %3 = load i32, i32* @x, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %8

; <label>:5:
  ; preds = %2
  %6 = load i32, i32* @x, align 4
  %7 = call i32 @decr(i32 %6)
  store i32 %7, i32* @x, align 4
  br label %2

; <label>:8:
  ; preds = %2
  %9 = load i32, i32* %1, align 4
  ret i32 %9
}
```

(b) LLVM translation

Figure 15: Example of C program and corresponding LLVM translation

LLVM Values LLVM knows two types of identifiers: global (prefix `@`) and local (prefix `%`); and three different formats for identifiers: (1) Named values (prefix + string, e.g., `@main`), (2) unnamed values (prefix + unsigned integer, e.g., `%1`), and (3) constants (reserved words, constant values, e.g., `0`). A program variable can be a named value, as well as an unnamed value: Named values are derived from identifiers that exist in the source program, and unnamed values are used by the LLVM compiler as an easy way to avoid name conflicts when introducing temporary variables.

LLVM Program Structure An LLVM program consist of multiple *modules*. Each module is one translation unit and consists of (a) functions, (b) global variables and (c) symbol table entries. Each function consists of multiple *blocks*. Each block starts with a label and consists of a linear sequence of program operations. This sequence ends with either an unconditional jump to one block (`br` operation with one argument), or a conditional jump that selects one of two blocks (`br` operation with three arguments). Branching conditions and loops in a C program are represented through these conditional jumps.

Figure 15 shows an example C program and its corresponding LLVM translation. The `while` loop in the original C program (Fig. 15a, Lines 8–9) is represented by the two blocks with labels `2` and `5` in the LLVM program (Fig. 15b, Lines 18–27). Block `2`

describes the loop head: The value of global variable `@x` is loaded into temporary program variable `%3`, the value of the signed integer comparison (operation `icmp` with option `sgt`) of variable `%3` and constant value 0 is stored in temporary program variable `%4`, and a conditional jump is performed based on the value of `%4`: If `%4` is 1 (i.e., `@x` > 0), control jumps to block `5`, which contains a jump back to block `2` at its end. Otherwise, control leaves the loop and jumps to block `8`.

LLVM Type System LLVM knows the following types: the void type, the function type, and first-class types. The void type represents no value and has no size. The function type represents a function signature and consists of a return type and a list of formal parameter types. First-class types are all types that can be produced by program operations. There are single value types, aggregate types, the label type, the token type and the metadata type. The single value types of LLVM are:

1. Integer types with an arbitrary bit width between 1 and $2^{23} - 1$ bits, e.g., integer type `i32` of bit-width 32. Integer types do not have a signedness. Instead, program operations can interpret given integers as either signed, or unsigned. For example, in Fig. 15b, Line 20, the `icmp` operation is told to perform a signed integer comparison with option `sgt`.

2. Six different floating point types with a different bit width and a different mantissa bit width, e.g., `half` (16 bit), `float` (32 bit), `fp128` (128 bit).

3. The pointer type, representing a memory location, e.g. pointer type `i32 *` specifying a pointer to an integer with bit-width 32.

4. The vector type which represents a vector of elements of another single value data type, e.g. vector type `<2 x float>` specifying a vector of 2 32 bit floating point values.

The aggregate types of LLVM are:

1. The array type, which represents a fixed size of elements of a single arbitrary data type (excluding void and metadata) in sequential order. For example, the array type `[2 x i32]` specifies an array of size 2, with element type `i32`.

2. The structure type, which represents a collection of data members of arbitrary data types (excluding void and metadata). For example, the structure type `{i32, i18, i14}` specifies a structure with three members: one 32 bit integer, one 18 bit integer, and one 14 bit integer.

3. The opaque structure type, which represents a named structure type that does not have a body.

The label type represents a code label, the token type is a specific type carrying special information for the compiler, and the metadata type represents metadata.

LLVM Constants LLVM knows *simple constants* and *complex constants*. Simple constants are boolean, integer and floating point values (e.g., *true*, 100, and 1.5), as well as the null pointer and the token constant 'none'. Complex constants are structure, array and vector constants, as well as the zero initialization and metadata:

1. A structure constant is a comma separated list of elements, each preceded by its type, that specify a corresponding structure. A structure constant always has a structure type. For example, structure constant `{ i1 true, i32 100, float 1.5}` is a structure of type `{ i1, i32, float}` with member values `true`, `100` and `1.5`.

2. An array constant is a comma separated list of elements, each preceded by its type, that specify a corresponding array. An array always has an array type. For example, array constant `[i32 42, i32 24]` is an array of type `[2 x i32]` with element `42` at position 0 and `24` at position 1.

3. Analogous, vector constants specify a vector and always have a vector type, e.g., `<double 3.14, double 159.265>` is a vector of type `<2 x double>` with elements `3.14` and `159.265`.

4. The zero initialization constant `zeroinitialization` can be used to initialize a value of any type to zero, including aggregate types.

5. A metadata node is constant tuple without types, which contains some metadata about the module. Metadata may, for example, hold debug information.

LLVM Operations LLVM contains multiple low-level program operations (called *instructions*). Instructions are divided into the following instruction classes: terminator, binary, bitwise binary, memory, and other instructions.

Terminator Instructions Terminator instructions are used to transfer control or stop the program. They include the already seen `br` instruction that gives control to a different basic block. Other noteworthy instructions are the `ret` instruction that is similar to the `return`-statement of C and the `unreachable` instruction that represents an unreachable portion of code.

Binary Instructions Binary and bitwise binary instructions require two operands of the same type, execute an operation on them, and produce a single new value. The result value always has the same type as its operands. Common examples are arithmetic operations like `add` (sum of two operands), `sub` (difference of two operands), `mul` (product of two operands), `div` (quotient of two operands), as well as bitwise operations like `shl` (binary shift to the left of first operand by number of bits specified by second operand), `or` (binary or of two operands), `and` (binary and of two operand), etc.

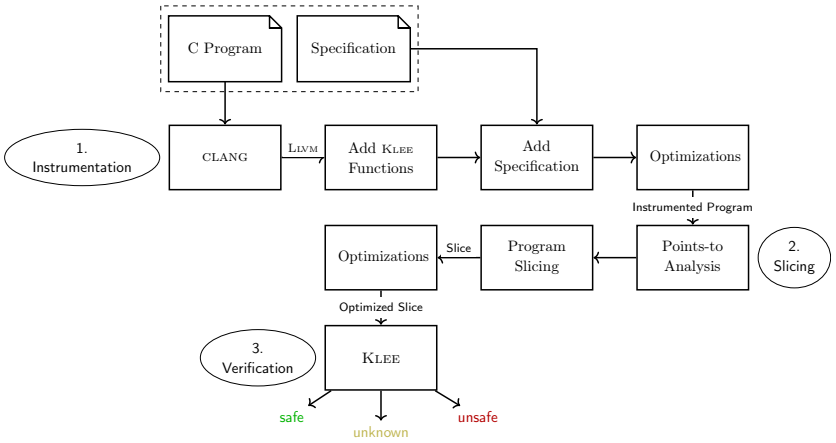


Figure 16: Workflow of SYMBIOTIC

Memory Instructions Memory instructions exist to extract and insert values into aggregate types (`extractvalue` and `insertvalue`), as well as allocate memory on the stack (`alloca`), read a value from a memory address (`load`), and write a value to a memory address (`store`).

Other Instructions Other instruction include the comparison of integer and float values (`icmp` and `fcmp`) and the function call instruction `call` that transfers control into a specified function and assigns the function parameters with the given arguments.

2.7.2 Symbiotic

SYMBIOTIC [21] is a formal verification tool for C programs that uses a combination [45] of instrumentation, static program slicing and symbolic execution. It can check C programs for any program specification that can be represented by a finite state machine. This allows SYMBIOTIC to check for, for example, reachability and memory violations, e.g., invalid memory accesses and memory overflows. Since the property that a program does not terminate can not be represented by a finite state machine, SYMBIOTIC can not check programs for termination. SYMBIOTIC uses LLVM as an intermediate language, in version 3.9.1. Figure 16 shows the verification steps of SYMBIOTIC: It first translates a given C program to LLVM using CLANG, instruments it to (a) make it compatible with the conventions of symbolic execution tool KLEE, and to (b) reflect the considered program specification. SYMBIOTIC then performs a first set of optimizations to improve the performance of the next steps. It uses a points-to-analysis to support the next step of intra-procedural program slicing [32]. A second set of optimizations is applied to the resulting program slice, before it is given to KLEE [20] for symbolic execution. SYMBIOTIC has a modular structure: The parser (CLANG) and the verification step (KLEE) can be replaced with other tools easily, and the instrumentation process can be adjusted as needed.

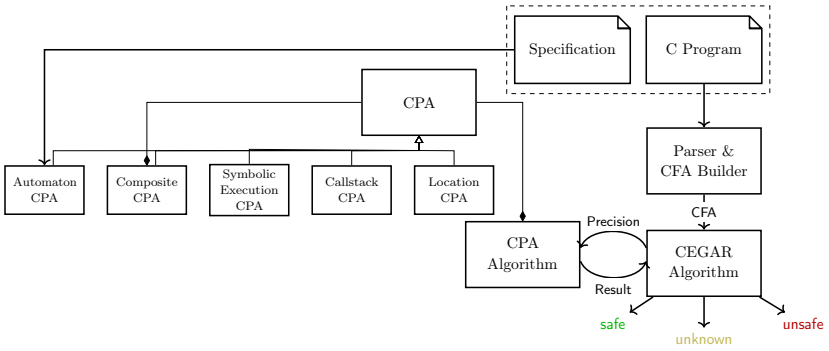


Figure 17: Workflow of CPAchecker

2.7.3 CPAchecker

CPAchecker [9] is a software verification framework for C programs [8] that is based on configurable program analysis. It provides a large amount of existing program analysis techniques implemented in the CPA framework, e.g., a predicate analysis [10], explicit-state model checking [13], and symbolic execution [11]. It also includes an implementation of the reaching-definitions CPA and, in addition to an implementation of the CPA algorithm, an implementation of the CEGAR algorithm (amongst others). The precision of a CPA is used as interface for storing the current abstraction.

As an intermediate representation, CPAchecker uses CFAs. Figure 17 shows the general verification steps of CPAchecker in a configuration that uses symbolic execution with CEGAR. A given C program is parsed and a CFA is created as intermediate representation. This CFA is given as input to the CEGAR algorithm. The CEGAR algorithm starts the CPA algorithm with an initial precision and gets a result, which consists of the ARG of the computed set of reachable abstract states and a verdict (target found/not found). If a target was found, the CEGAR algorithm extracts a counterexample from the ARG and checks whether it is feasible. If it is, the algorithm returns 'unsafe'. If it is not, the algorithm refines the precision with the counterexample. It then restarts the CPA algorithm with the new precision and repeats this process until the CPA algorithm returns that no target was found. If this is the case, the CEGAR algorithm returns 'safe'. (If a timelimit is reached, the running machine runs out of memory, or an unsupported program feature is encountered, the analysis returns 'unknown'.) The CPA algorithm itself uses multiple CPAs: The composite CPA is used to create a composition of the following CPAs: The location CPA to track program locations, the callstack CPA to track the function callstack of the program, the symbolic execution CPA that performs a symbolic execution on the program, and an automaton CPA, which represents the specification as a finite state automaton.

CPAchecker has a modular structure: The parser & CFA builder can be replaced by a parser for another programming language and a CFA builder that

[8] With some analyses, CPAchecker can also analyze Java programs.

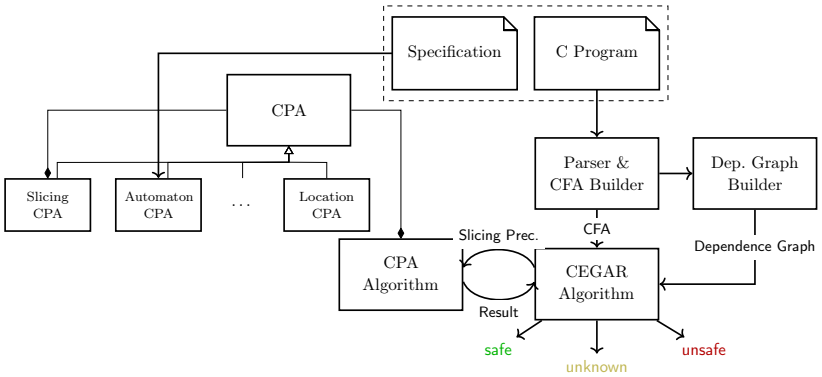


Figure 18: Workflow of CPAchecker with slicing

builds a CFA from that language's AST, and the CPAs can be replaced by any other combination of CPAs. In addition, multiple optimization options exist.

3 Iterative Slicing

Existing verification approaches that use slicing use it as a pre-processing step. For a set of known slicing criteria, the program slice of a given program is created and further analysis is performed on that slice. We don't do this for two reasons: 1. If a new, intermediate program is used for analysis, it may be difficult to create fitting back-references to the program under analysis (for example necessary for verification-result validation). And 2. the potential target region in a program may be significantly larger than the reachable target region; eager slicing may thus result in a slice that is larger than necessary.

Instead, we use a slicing CPA to perform slicing in tandem with the original verification technique on the original program. To create a program slice that is as small as possible, we start with an empty slice and add relevant edges on demand, through the use of CEGAR. Since we see slicing not as a removal of program operations, but only as a replacement with a `noop` operation, this can be done dynamically and without the need to construct a new CFA. The current program slice is defined by the *slicing precision*.

Figure 18 shows the workflow diagram of CPAchecker with slicing: In addition to the original workflow (Fig. 17), we create the dependence graph from the CFA. This is then used by the CEGAR algorithm in refinement of the slicing precision. The slicing CPA (on the left) uses this for dynamic program slicing.

3.1 Program Dependence Graph Construction

The system dependence graph contains one node for each CFA edge. In addition, it contains one actual-in node for each parameter of each CFA edge with a function call operation, and one actual-out node for each CFA

edge with a function return statement. The edges of the dependence graph are separately computed as follows.

3.1.1 Flow-Dependence Computation

To compute flow dependences in a CFA, we use the CPA algorithm with a *flow-dependence CPA*. For flow-dependence computation, it is necessary to track reaching definitions in the CFA—the flow-dependence CPA delegates this to a wrapped reaching definitions CPA. To keep track of the current program location and the function call stack, we combine the flow-dependence CPA with a location CPA and a callstack CPA. The flow-dependence CPA is a CPA $\mathbb{FD} = (D_{\mathbb{FD}}, \widetilde{\Pi}, \rightsquigarrow_{\mathbb{FD}}, \mathsf{merge}^{join}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$.

Abstract Domain The abstract domain $D_{\mathbb{FD}} = (\mathcal{C}, \mathcal{E}_{\mathbb{FD}}, [\![\cdot]\!]_{\mathbb{FD}})$ consists of the set $\mathcal{C}$ of concrete states, the semi-lattice $\mathcal{E}_{\mathbb{FD}}$ and concretization function $[\![\cdot]\!]_{\mathbb{FD}}$. The semi-lattice $\mathcal{E}_{\mathbb{FD}} = (E_{\mathbb{FD}}, \sqsubseteq_{\mathbb{FD}}, \sqcup_{\mathbb{FD}}, \top_{\mathbb{FD}})$ has the set $E_{\mathbb{FD}} = 2^{\mathsf{E}} \times (G \times X \times L \times L)$ of abstract states. An abstract state is a pair (S, d) of a set $S \subseteq \mathsf{E}$ of reaching definitions and a set $d \subseteq G \times X \times L \times L$ of flow dependences. A flow dependence $(g, x, l, l') \in d$ represents that CFA edge g is, through the use of x, flow-dependent on the CFA edge that goes from l to l'. The less-or-equal relation $\sqsubseteq_{\mathbb{FD}}$ contains $(S, d) \sqsubseteq_{\mathbb{FD}} (S', d')$ if $S \sqsubseteq_{\mathbb{RD}} S'$ and $d \subseteq d'$. The join operator $\sqcup_{\mathbb{FD}}$ for two states (S, d) and (S', d') is defined as the pairwise join of the reaching-definition states S and S' using the join of the reaching-definitions CPA, and the union of flow dependence sets d and d':

$$(S, d) \sqcup_{\mathbb{FD}} (S', d') = (S \sqcup_{\mathbb{RD}} S', d \cup d') \ .$$

The top element $\top_{\mathbb{FD}} = (\mathsf{E}, E_{\mathbb{FD}})$ is the abstract state with all possible reaching definitions and all possible flow dependences between CFA edges.

Precision Set The flow-dependence CPA uses the static precision $\widetilde{\Pi}$.

Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{FD}}$ has transfer $(r, d) \overset{g}{\rightsquigarrow}_{\mathbb{FD}} (r', d', \widetilde{\pi})$, if $g = (l, op, l')$, $r \overset{g}{\rightsquigarrow}_{\mathbb{RD}} r'$, and

$$d' = d \cup \{(g, x, l, l') \mid x \in \mathit{uses}(op) \wedge (x, l, l') \in r\} \ .$$

It does two things: It (a) performs the reaching-definitions transfer for the reaching-definitions state r, and (b) adds to the flow-dependence state d all flow dependences of g, computed from the variables used by op and the reaching definitions for these used variables.

Merge and Stop Operators The flow-dependence CPA always joins abstract states when the control flow meets, i.e., it uses merge^{join}. The stop operator considers each abstract state separately, i.e., stop^{sep}.

Algorithm 3 Doms(A)

Input: CFA $A = (L, l_0, l_e, G)$
Output: Mapping $D : L \to 2^L$ of program locations to their post-dominators
Variables: CFA node $l \in L$, set $\mathsf{waitlist} \subseteq L$ of CFA nodes,

```
1:  D(l) = L for all l ≠ l_0
2:  D(l_0) = ∅
3:  waitlist = {l_0}
4:  while waitlist ≠ ∅ do
5:    choose and remove l from waitlist
6:    for all (l, ·, l') ∈ G do
7:      δ_new = D(l') ∩ (D(l) ∪ {l})
8:      if δ_new ≠ D(l') then
9:        D(l') = δ_new
10:       waitlist = waitlist ∪ {l'}
11: return D
```

Precision Adjustment The flow-dependence CPA uses the static precision adjustment operator $\widetilde{\mathsf{prec}}$. It always returns the given abstract state and does not perform any adjustment.

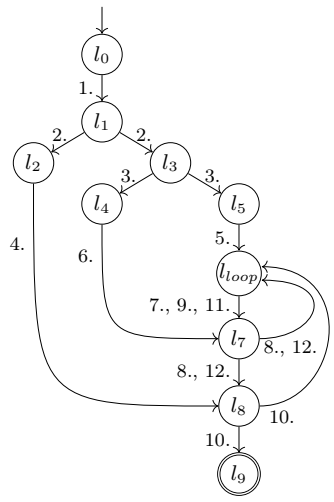
If we apply the CPA algorithm with a composition of the location CPA (to track the control flow), the callstack CPA (to track the callstack), and the flow-dependence CPA (to track flow dependences) to a CFA, the computed reached set contains all flow dependences in the CFA. For each such dependence, we add both CFA edges and the corresponding flow dependence edge to the dependence graph.

Augmentation for C Programs To analyze C programs, we have to take care of pointer aliasing and call-by-reference parameter passing of arrays. To do so, we add a points-to-analysis CPA to the composition of analyses, and handle array parameters and array arguments as additional return values at each return edge.

3.1.2 Post-Dominator Computation

To compute control dependences between CFA edges, we first have to compute the post-dominators of each program location in the CFA. To compute post-dominators, we can compute the dominators of each program location on the reverse CFA. Thus, we first present an algorithm for dominator computation, and then apply this algorithm to the reverse CFA.

For a given CFA $A = (L, l_0, l_e, G)$, Algorithm 3 computes the function $D : L \to 2^L$ that maps each program location to its set of dominators in the CFA. It performs a waitlist-based fix-point computation over the map D of candidate dominators for each $l \in L$. By definition, the entry location has no dominators: $D(l_0) = \emptyset$. For all other program locations $l \neq l_0$, we start with the full set of program locations as candidates: $D(l) = L$. The algorithm traverses through A, starting at the program entry location l_0. For each program location l that it visits, it computes new candidate dominators $D(l')$ for all successors l' of



l	$D(l)$			
	Init.	1st Visit	2nd Visit	3rd Visit
l_0	$\emptyset$			
l_1	L	$\{l_0\}$		
l_2	L	$\{l_0, l_1\}$		
l_3	L	$\{l_0, l_1\}$		
l_4	L	$\{l_0, l_1, l_3\}$		
l_5	L	$\{l_0, l_1, l_3\}$		
l_{loop}	L	$\{l_0, l_1, l_3, l_5\}$	$\{l_0, l_1, l_3\}$	$\{l_0, l_1\}$
l_7	L	$\{l_0, l_1, l_3\}$	$\{l_0, l_1\}$	
l_8	L	$\{l_0, l_1\}$		
l_9	L	$\{l_0, l_1, l_8\}$		

Figure 19: Example CFA and its dominator computation. The CFA edges are numbered in the order in which they are considered by Algorithm 3.

l. For a location l', dominators can only be locations that have been candidates in the past and that are either also dominators of the predecessor location l, or the predecessor l itself. Thus, the set of candidate dominators can be restricted to these locations. Formally, the new set of dominators δ_{new} is

$$\delta_{new} = D(l') \cap (D(l) \cup \{l\}) \ .$$

If the candidate dominators of l' change (i.e., the new set δ_{new} of candidate dominators is smaller then the previous one $D(l')$), then it is possible that the candidate dominators of all successors of l' may also need to change—thus, l' is added to the waitlist. If the waitlist is empty, all dominator sets are final and $D(l)$ represents the set of dominators for each $l \in L$. Figure 19 shows an example CFA and the intermediate steps of the dominator computation with Algorithm 3. All edges of the CFA contain `noop` operations—instead, the edges of the CFA are numbered in the order they are visited by Algorithm 3. The table shows the candidate dominators $D(l)$ for each program location l for each time l is chosen from the waitlist. Final dominators are marked green. The algorithm starts at l_0 with dominators $\emptyset$, looks at edge $(l_0, \texttt{noop}, l_1)$, computes the new candidate dominators $\{l_0\}$ for l_1, and adds it to the waitlist, because its candidate dominators changed. It then continues to l_1, visits edges $(l_1, \texttt{noop}, l_2)$ and $(l_1, \texttt{noop}, l_3)$, computes the new candidate dominators $D(l_2) = \{l_0, l_1\}$ and $D(l_3) = \{l_0, l_1\}$ and adds both l_2 and l_3 to the waitlist. Since l_2 and l_3 are on the same level, the selection strategy could choose both l_2 or l_3, first. In this example, we first choose l_3: From there, the algorithm visits edges $(l_3, \texttt{noop}, l_4)$ and $(l_3, \texttt{noop}, l_5)$, computes the new candidate dominators for l_4 and l_5, and adds both to the waitlist. The waitlist now contains l_2, l_4, l_5. Because of the reverse post-order selection strategy, the algorithm continues with l_2, visits $(l_2, \texttt{noop}, l_8)$ and computes the new candidate dominators $D(l_8) = \{l_0, l_1, l_2\}$.

Algorithm 3 continues in this fashion and visits l_8, l_{loop} and l_7. At l_7, the CFA contains the back-edge $(l_7, \texttt{noop}, l_{loop})$. Thus, the algorithm visits $(l_7, \texttt{noop}, l_8)$ and $(l_7, \texttt{noop}, l_{loop})$, computes the new candidate dominators $D(l_8) = \{l_0, l_1\}$ and $D(l_{loop}) = \{l_0, l_1, l_3, l_5\} \cap (\{l_0, l_1, l_3\} \cup \{l_7\}) = \{l_0, l_1, l_3\}$ and re-adds l_{loop} to the waitlist. It continues with l_{loop}, re-visits $(l_{loop}, \texttt{noop}, l_7)$ and computes the candidate dominators $\{l_0, l_1, l_3\} \cap (\{l_0, l_1, l_3\} \cup \{l_{loop}\}) = \{l_0, l_1, l_3\}$. Since the candidate dominators don't change for l_7, it is not added to the waitlist. Instead, the algorithm continues with l_8, again re-visits l_{loop}, which has now its final dominator set, re-visits l_7 and also adjusts its candidate dominators to the final set $\{l_0, l_1\}$, and then visits l_9 and halts.

We design a dominator CPA with which the CPA algorithm performs exactly the same steps as Algorithm 3. The dominator CPA is a component CPA that only works in composition with the location CPA, and is defined as $\mathbb{D} = (D_{\mathbb{D}}, \widetilde{\Pi}, \rightsquigarrow_{\mathbb{D}}, \mathsf{merge}^{join}, \mathsf{stop}^{sep}, \widetilde{\mathsf{prec}})$.

Abstract Domain The abstract domain $D_{\mathbb{D}} = (\mathcal{C}, \mathcal{E}_{\mathbb{D}}, [\![\cdot]\!]_{\mathbb{D}})$ consists of the set $\mathcal{C}$ of concrete states, the semi-lattice $\mathcal{E}_{\mathbb{D}}$, and the concretization function $[\![\cdot]\!]$. The semi-lattice $\mathcal{E}_{\mathbb{D}} = (\mathsf{D}, \sqsubseteq_{\mathbb{D}}, \sqcup_{\mathbb{D}}, \top_{\mathbb{D}})$ has elements $\mathsf{D} = 2^L$. In composition with a location CPA, δ represents the set of dominators for a program location. The less-or-equal relation $\sqsubseteq_{\mathbb{D}}$ contains $\delta \sqsubseteq_{\mathbb{D}} \delta'$, if $\delta \supseteq \delta'$. The join $\sqcup_{\mathbb{D}}$ computes the intersection of elements, i.e., $\delta \sqcup_{\mathbb{D}} \delta' = \delta \cap \delta'$. The top element $\top_{\mathbb{D}} = \emptyset$ is the empty set.

Precision Set The dominator CPA uses the static precision $\widetilde{\Pi}$.

Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{D}}$ has transfer $\delta \overset{g}{\rightsquigarrow}_{\mathbb{D}} \delta'$, if $g = (l, \cdot, l')$ and $\delta' = \delta \cup \{l\}$. The transfer relation always adds to program location l' the predecessor location l as (potential) dominator.

Merge Operator The dominator CPA always joins states when the control flow meets:

$$\mathsf{merge}^{join}(\delta, \delta') = \delta \sqcup_{\mathbb{D}} \delta' = \delta \cap \delta'$$

Through this intersection, the sets of potential dominators at each location are reduced to the dominators that are common to all sets.

Stop Operator The dominator CPA checks states separately (stop^{sep}).

Precision Adjustment The dominator CPA uses the static precision adjustment operator $\widetilde{\mathsf{prec}}$. It always returns the given abstract state and does not perform any adjustment.

Post-dominator CPA To compute post-dominators, we combine the dominator CPA with a backwards-location CPA. The backwards-location CPA

traverses through the CFA in reverse order. As initial abstract state, we have to provide $(l_e, \emptyset)$ to the CPA algorithm, so that it not only traverses the CFA backwards, but also starts at the program exit location. With these two adjustments, we can model a reverse CPA. The reached set computed by the CPA algorithm will contain abstract states that are tuples $(l, \delta) \in L \times 2^L$. Such a tuple represents that program location l has post-dominators δ.

Performance To build a PDG, we require to get the set of post-dominators per program location. If we compute the post-dominator tree, the computation of the set of post-dominators for a single program location is in $O(n)$, for n program locations in the CFA. If we compute the set of post-dominators for each program location, the total cost is in $O(n^2)$. The fastest algorithm for post-dominator computation known to us is in $O(n\alpha(n))$, but can only construct the post-dominator tree. Thus, the total time for computing the full sets for each location with this algorithm is in $O(n^3\alpha(n))$.

Instead, we can directly compute the set of post-dominators for each program location with Algorithm 3 in $O(n^2)$, if we implement the waitlist with a reverse post-order selection strategy.

Claim 1. *Algorithm 3 with a reverse post-order waitlist runs in the worst case in $O(n^2)$.*

Proof. Let's assume the waitlist is implemented in a way that program locations are chosen based on their location in the CFA: If a node is not the head of a loop, it is only selected if none of its transitive predecessors are in the waitlist. If a node is the head of a loop, it is only selected if none of its transitive predecessor that are not part of the loop are in the waitlist. This selection strategy requires the use of a sorted waitlist. Each insertion and removal in the waitlist is then in $O(\log n)$.

For a CFA $A = (L, l_0, l_e, G)$, one of the following cases is true for each program location $l \in L$, with predecessor set $preds(l)$:

C 1: The program location is the program entry ($l = l_0$).
The entry location l_0 has reached its fix point even before the first visit, because initially $D(l_0) = \emptyset$ (see Fig. 19)

C 2: The program location is not a loop head.
The program location reaches its fix point the next time it is visited after each predecessor has reached its fix point.

C 3: The program location is a loop head.
The program location reaches its fix point after each predecessor that is not in the governing loop has reached its fix point, and after the governed loop is traversed one more time (in the described reverse post order). If the loop contains other loops, the corresponding loop head and its members must be visited accordingly. See Fig. 19 and the loops starting at l_{loop} for an example.

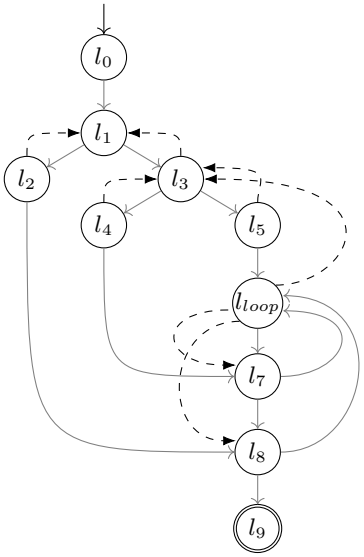


Figure 20: Example CFA and its control dependences (dashed lines)

For *C 1*, the algorithm requires 1 step to set $D(l_0) = \emptyset$. For *C 2*, program location l reaches its fix point in 1 step after its predecessors have reached their fix point. Thus, for a set of m program locations that are connected, that do not contain a loop and that start with entry location l_0, the algorithm requires m steps. For *C 3*, program location l reaches its fix point in m steps if the loop consists of m nodes and does not contain a nested loop. If the loop does contain nested loops, it requires at most $m * k$ steps for k nested loops. Since the maximum number of loops $k \leq m$ is less or equal to the number of nodes, the algorithm requires at most m^2 steps. If we combine these three steps, we require at most n steps for n connected program locations without any loops and n^2 steps for n connected program locations with loops. Thus, Algorithm 3 runs in $O(n^2)$. □

The only difference between a CFA A and its reverse A^- is, that each node in A may only have two successors, while each node in A^- may only have two predecessors, but an infinite amount of successors. Since we did not rely on the number of successors or predecessors in above proof, it is a valid proof that the performance of both dominator-, and post-dominator-computation with Algorithm 3 is in $O(n^2)$.

3.1.3 Control-Dependence Computation

CFA nodes can only be control dependent on branching nodes, i.e., nodes that have more than one successor. Thus, we only have to consider these for control dependence computation: For each branching node l_b in a CFA, we consider its two branches separately. Starting with the first node in a branch, we iterate

through the branch, and check for each node l that we visit, that it is not a post-dominator for l_b. If it is, the two branches of l_b meet at l, and we continue with the next branching node. If it isn't, we are still in one of the branches. In this case, we check that all nodes on the path from l_b to l are post-dominated by l. If they are, then l is control dependent on l_b and we add the corresponding control dependence edge to the dependence graph. If they aren't, then we have entered a nested loop. In this case, we continue without adding a control-dependence edge, because the nested branching node, which is control-dependent on l_b, will already create a transitive dependence. Figure 20 shows an example CFA with its control dependences as computed by our algorithm. The control dependences are marked as dashed lines, the CFA edges are grayed out. The example shows that program locations l_4 and l_5 have a transitive control dependence through l_3 on l_1, thus there is no need for tracking these control dependences explicitly.

3.2 Slicing CPA

We implement slicing not as a pre-processing step, but as component of the main analysis. The idea is the following: A slicing CPA wraps the original analysis and decides for each transfer $\overset{g}{\rightsquigarrow}$, whether the transfer's CFA edge $g = (l, op, l')$ is part of the current slice. If it is, the transfer of the wrapped analysis for g is performed as-is. Otherwise, the transfer is replaced with $\overset{g'}{\rightsquigarrow}$, where $g' = (l, \texttt{noop}, l')$.

The slicing CPA $\mathbb{SC} = (D_{\mathbb{SC}}, \Pi_{\mathbb{SC}}, \rightsquigarrow_{\mathbb{SC}}, \mathsf{merge}_{\mathbb{SC}}, \mathsf{stop}_{\mathbb{SC}}, \widetilde{\mathsf{prec}})$ for a wrapped CPA $\mathbb{D} = (D, \Pi, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec})$ with abstract domain $D = (\mathcal{C}, \mathcal{E}, [\![\cdot]\!])$ and $\mathcal{E} = (E, \sqsubseteq, \subseteq, \top)$ consists of the following components.

Abstract Domain The slicing CPA does not have to track any information itself. Thus, the abstract domain of the slicing CPA is the abstract domain of the wrapped CPA:

$$D_{\mathbb{SC}} = D$$

Precision Set The precision set $\Pi_{\mathbb{SC}} = 2^G \times \Pi$ consists of the CFA edges of the current program slice and the wrapped CPA's precision set Π.

Transfer Relation The transfer relation $\rightsquigarrow_{\mathbb{SC}} : E \times G \times E \times \Pi_{\mathbb{SC}}$ contains transfer $e \overset{g}{\rightsquigarrow}_{\mathbb{SC}} (e, (\pi_{\mathbb{SC}}, \pi))$ if $g = (l, op, l')$ and either

1. $g \in \pi_{\mathbb{SC}}$ and the transfer $e \overset{g}{\rightsquigarrow} (e, \pi)$ of the wrapped CPA exists, or
2. $g \notin \pi_{\mathbb{SC}}$ and the transfer $e \overset{g'}{\rightsquigarrow} (e, \pi)$ with $g' = (l, \texttt{noop}, l')$ of the wrapped CPA exists.

Merge Operator The merge operator $\mathsf{merge}_{\mathbb{SC}} : E \times E \times \Pi_{\mathbb{SC}} \to E$ delegates to the merge operator of the wrapped CPA:

$$\mathsf{merge}_{\mathbb{SC}}(e, e', (\pi_{\mathbb{SC}}, \pi)) = \mathsf{merge}(e, e', \pi) \ .$$

Algorithm 4 $\mathsf{refine}_{\mathbb{SC}}(A, DG_A, \sigma)$

Input: CFA $A = (L, l_0, l_e, G)$, dependence graph $DG_A = (N, E_D, E_C)$ of A, and infeasible path $\sigma = \langle l_0 \xrightarrow{op_0} \ldots \xrightarrow{op_{n-1}} l_n \rangle$
Output: precision $(\pi_{\mathbb{SC}}, \pi)$
Variables: Slicing criterion c.
1: $c = getCriterion(\sigma)$
2: $\pi_{\mathbb{SC}} = \mathsf{Slice}(A, DG_A, c) \cup \pi_{\mathbb{SC}}$
3: $[\pi = \mathsf{refine}_{\mathbb{D}}(\sigma)$ ◇ Call refinement procedure of wrapped CPA]
4: **return** $(\pi_{\mathbb{SC}}, \pi)$

Stop Operator The stop operator $\mathsf{stop}_{\mathbb{SC}} : E \times 2^E \times \Pi_{\mathbb{SC}}$ delegates to the stop operator of the wrapped CPA:

$$\mathsf{stop}_{\mathbb{SC}}(e, R, (\pi_{\mathbb{SC}}, \pi)) = \mathsf{stop}(e, R, \pi) \ .$$

Precision Adjustment The precision adjustment operator

$$\mathsf{prec}_{\mathbb{SC}} : E \times \Pi_{\mathbb{SC}} \times 2^{E \times \Pi_{\mathbb{SC}}} \to E \times \Pi_{\mathbb{SC}}$$

delegates to the wrapped CPA's precision adjustment operator:

$$\mathsf{prec}_{\mathbb{SC}}(e, (\pi_{\mathbb{SC}}, \pi), R) = (\widehat{e}, (\pi'_{\mathbb{SC}}, \pi'))$$

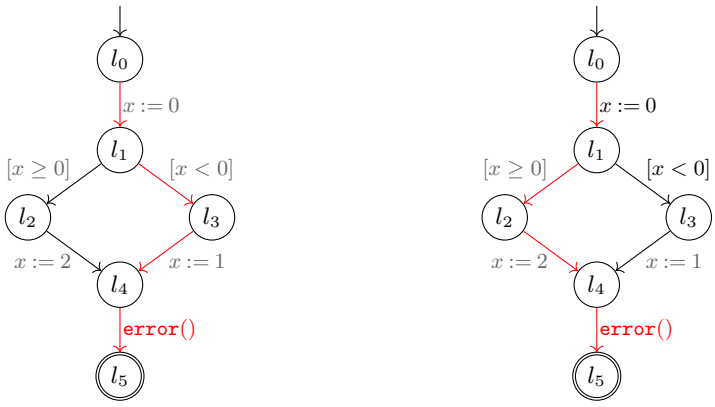
where $\mathsf{prec}(e, \pi, \widetilde{R}) = (\widehat{e}, \pi')$. Reached set $\widetilde{R} \in 2^{E \times \Pi}$ is the same as the original reached set $R \in 2^{E \times \Pi_{\mathbb{SC}}}$, but stripped of the slicing CPA's precision.

3.3 Slicing Refinement

Initially, our slicing algorithms assumes that no program operation is relevant—we run analysis with the empty slice $\pi_{\mathbb{SC}} = \emptyset$. With this, all CFA edges are replaced with corresponding `noop`-edges during analysis.

Slicing Refinement If an error path $\sigma = \langle l_0 \xrightarrow{op_0} \ldots \xrightarrow{op_{n-1}} l_n$ is found, we first get the program slice A/c for that target location and run the counterexample check $\texttt{checkPath}(\sigma, A/c)$ on that. The counterexample check $\texttt{checkPath}(\sigma, A/c)$ performs a counterexample check on the given target path σ based on the program slice A/c. To do so, it runs the slicing CPA on that target path, using the corresponding program slice. Since the error path σ ends in the target state, we take the last program operation op_{n-1} before the target state as a slicing criterion c, i.e., $c = \langle (l_{n-1}, op_{n-1}, l_n,\ X \rangle$. We use all program variables X for c, because we have no way of telling which program variables that occur in op_{n-1} are relevant for reaching the target state.

If the target path on that program slice is infeasible, we use refinement procedure $\mathsf{refine}_{\mathbb{SC}}$ (Algorithm 4) to refine the program slice: We use the same slicing criterion as for the feasibility check, i.e., $getCriterion(\sigma) = \langle (l_{n-1}, op_{n-1}, l_n),\ X \rangle$. We call this *target-location slicing*. We then call $\mathsf{Slice}(A, DG_A, c)$ (Algorithm 1)



(a) Invalid counterexample found with target-location slicing

(b) Valid counterexample found with target-path slicing

Figure 21: Example CFA that requires target-path slicing for correct counterexample

with CFA A, dependence graph DG_A and the slicing criterion c to get the relevant edges of A/c. These are added to our slicing precision $\pi_{\mathbb{SC}}$. This way, we are able to incrementally build a program slice for arbitrary program properties.

Slicing Refinement with valid Counterexamples If a target location is found and we want to produce a counterexample for verification-result validation, we have to use the counterexample check on the full path, i.e. $\texttt{checkPath}(\sigma, A)$, to make sure that only paths feasible on the original program are allowed. We also have to adjust our slicing criterion to contain all assume edges of an infeasible target path, i.e. $\mathit{getCriterion}(\sigma) = \langle \mathit{assumeEdges}(\sigma),\ X \rangle$. We call this *target-path* slicing. If we use target-location slicing, target paths that are actually infeasible due to contradicting program operations may be presented as counterexamples. Figure 21 shows an example CFA for which target-location slicing finds an invalid counterexample. Let's consider the program property that `error()` is never called. This is violated by the example CFA (called A in the following). The first iteration of our slicing CPA uses the empty program slice and finds the target path $\sigma = \langle l_0 \to l_1 \to l_3 \to l_4 \to l_5 \rangle$ (**??**). It then performs a counterexample check on the program slice $A/c(l_4, \texttt{noop}, l_5)X$, which only contains $(l_4, \texttt{noop}, l_5)$. Thus, the counterexample check returns that the target path is feasible and an invalid counterexample is returned. Instead, we have to perform our counterexample check on the full CFA A. If we do this, the counterexample check returns that the target path is infeasible, and we refine the slicing precision. Let's assume we use target-location slicing. Then, the resulting new precision would only contain $(l_4, \texttt{noop}, l_5)$, and the same infeasible target path would again be encountered in the next iteration of the analysis, leading to an infinite loop (or a failure). If we use target-path slicing, we add the slice for criterion $c\{(l_4, \texttt{noop}, l_5), (l_1, [x < 0], l_3)\}X$ to the slicing precision. This includes

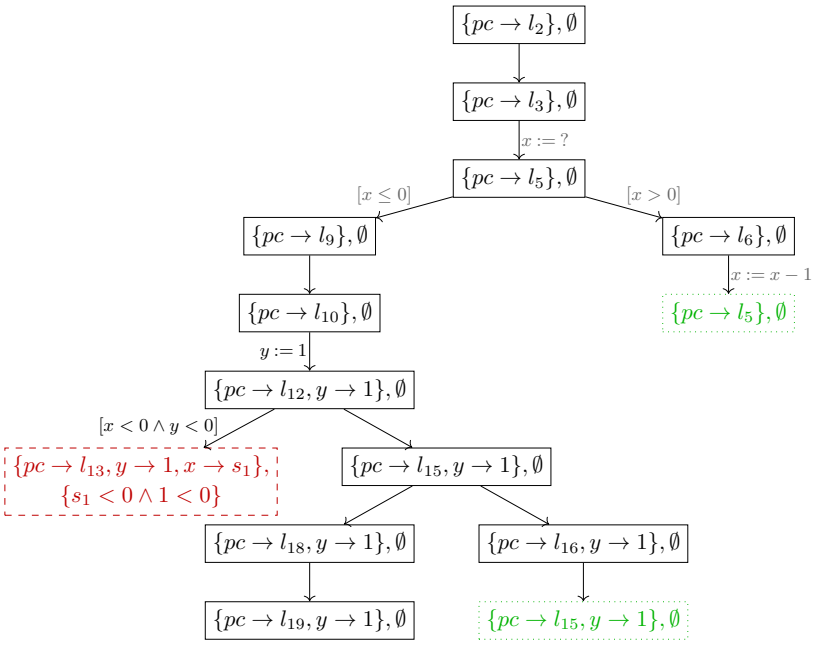


Figure 22: ARG created by symbolic execution, with a combination of CEGAR and program slicing, for our running example. Note that the state space is finite.

edges $(l_0, x := 0, l_1)$ and $(l_1, [x < 0], l_3)$, and thus the same target path will not be encountered again. Instead, the algorithm finds the feasible target path (**??**).

Combination with other Abstraction Refinements It is possible to combine the slicing refinement with other abstraction techniques based on CEGAR: Target-path slicing takes all CFA edges that *could* be relevant for proving the infeasibility of a program path, because it takes all CFA edges that have *any* influence on the feasibility of the target path. Thus, it describes an upper bound for possibly necessary information and any other precision derived from the same error path is compatible with it. All CEGAR techniques can be used with our slicing approach, freely. To do so, we delegate to the refinement procedure $\mathsf{refine}_{\mathbb{D}}$ of the CPA $\mathbb{D}$ wrapped by the slicing CPA (Algorithm 4, line 3). If the wrapped CPA does not use CEGAR, $\mathsf{refine}_{\mathbb{D}}$ always returns the static precision $\widetilde{\pi}$. Otherwise, it returns a refined precision for the wrapped CPA (that, by default, works with the current program slice). This allows us, for example, to combine program slicing with symbolic execution with CEGAR.

We consider our running example one more time. For this, we have already seen that symbolic execution fails to compute a finite state space that shows that the example is free of errors—even when using slicing or CEGAR (on their own). But when we combine both, we are able to derive that neither the first loop (thanks to CEGAR), nor the second (thanks to slicing) have to be unrolled. Fig. 22 shows the corresponding ARG computed by symbolic execution when using CEGAR and slicing. The ARG edges show

the program operations considered through slicing. Program operations that are ignored because of CEGAR are grayed out.

In addition to the combination of the CEGAR algorithm in CPAchecker and our newly implemented slicing CPA, we extended Symbiotic to use CPAchecker as a verification back-end. Since Symbiotic works on Llvm code, we have to create a corresponding front-end for CPAchecker to be able to verify the sliced Llvm programs.

3.4 LLVM Front-end in CPAchecker

To be able to use the optimized slice produced by Symbiotic (c.f. Sect. 2.7.2), with the CEGAR-based analysis of CPAchecker, we implemented an Llvm front-end in CPAchecker. The Llvm front-end consists of a) an Llvm parser that parses Llvm bitcode to an abstract syntax tree (AST), and b) a CFA builder, that transforms the AST to a CFA for analysis with the CPA algorithm (Fig. 18).

LLVM Parser We have two requirements for an Llvm parser: 1. Since CPAchecker is implemented in Java, it should be written in Java for easy integration. And 2. we want to use an official parser. Llvm does not guarantee backwards-compatibility between versions, so only if we use an official parser, we can make sure to be able to update to new Llvm versions in the future.

There is no official Llvm parser for Java, so we build, as a compromise, Java bindings that act as a bridge to the official parser: llvm-j[9]. llvm-j is a Java parser for Llvm bitcode that uses automatically generated Java bindings to the official C Llvm parsing library to make updates to new Llvm versions easy. To create these bindings, we use JNAerator[10]. Proxy classes are provided for the created bindings and the native C objects, so that the API of llvm-j doesn't change, even if the official C library does.

CFA Builder In addition to the widely used C front-end, CPAchecker also supports Java input programs [27]. This front-end uses dedicated Java CFA edges to map the Java semantics to the CFA. This creates the problem that Java CFA edges have to be explicitly supported by analyses. Thus, it is only possible to use a CPAchecker analysis with a Java program, if the developer of that analysis implemented it for Java—since this is barely the case, only few analyses can be used with Java programs, to this point. To prevent this situation from happening to our Llvm front-end, we restrict our Llvm front-end to that functionality of Llvm that can be mapped to C semantics, and build an original C CFA from Llvm programs. This loses information of Llvm that is meant for compiler optimization and does not allow us to analyze Llvm programs that were created from C++ code (because of missing support for exceptions, classes, etc.), but gives us the opportunity to apply all existing analyses in CPAchecker to the CFA created from Llvm programs.

[9] https://github.com/sosy-lab/llvm-j
[10] https://github.com/nativelibs4java/JNAerator

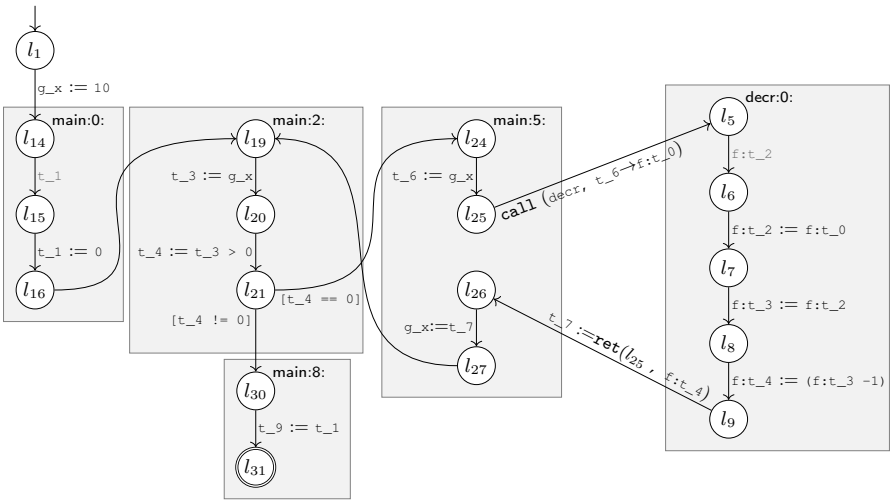


Figure 23: CFA for Llvm program from Fig. 15b

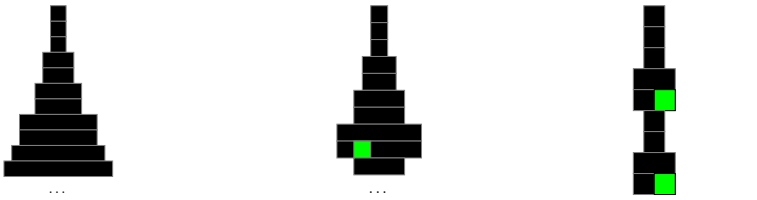
The CFA construction consists of three phases:

1. Create partial CFAs for global definitions
2. Create partial CFAs for each function
3. Link partial CFAs to final CFA

We first iterate over all global definitions in the parsed Llvm program and create CFA nodes and edges for these. We then visit each function definition individually: For each function, we first visit each basic block individually, create a partial CFA for each of it and then link them together. The CFAs of basic blocks have a fixed, sequential structure: They always start with a CFA node that represents the label of the basic block, and always end with either a (conditional) jump to other basic blocks or the function exit. The only time that the CFA of a basic block is not strictly sequential, is if the `select` instruction is used. This equals the ternary operator in C and is translated to an if-else statement with the corresponding variable assignments. This fixed structure of basic blocks makes it easy to link them: The last node of each basic block is linked to the labels of the following blocks through either a `noop` edge for an unconditional jump, or two assume edges for a conditional jump. If the end of a block is a function exit, no linking is done.

After creating the CFAs for each function definition, we give the partial CFAs (in C semantics) to a module of the existing CFA builder for C. This creates the final CFA.

Figure 23 shows the CFA for the example Llvm program from Fig. 15b. The program locations in the CFA refer to the line numbers in Fig. 15b. A rectangle frames each basic block, and entry nodes to basic blocks are additionally labeled with their function name and label (e.g., main:0:). The



(a) Start of ARG of Fig. 7 (b) Start of ARG of Fig. 12 (c) ARG of Fig. 22

Figure 24: Example pixel trees

global variable prefix `@` is replaced by `g_`, to be a valid C identifier (e.g., `@x` → `g_x`). The local variable prefix `%` is removed, and each temporary variable `%i` of LLVM is renamed to a program variable `t_i`. In the example, local variables of function `decr` are additionally prefixed with `f:`.

The CFA starts with the global definitions. The CFA edge of the last global definition (in the example edge $(l_1, \texttt{g\_x} := \texttt{10}, l_{14})$) ends in the entry node to the main function. From there, functions and basic blocks are built as described above.

3.5 Pixel Trees: Visualization of Analyses

While quantitative analyses of software verification algorithms are pointers towards the capabilities of algorithms, it is often vital to perform a qualitative analysis of the behavior of algorithms to get proper insights into their behavior. When we want to perform a qualitative analysis of a verification algorithm, we face a challenge: As soon as we step away from small examples and turn towards a real-world program, the explored state space (called *search space*) of the program quickly becomes too large to comprehend. While measures can be applied to 'measure' the search space of the program, these are often unintuitive and their proper interpretation is unclear. A visualization of the search space is necessary, that must be easy to comprehend by a human reader and still contain the information important to the user. For our application, the important information is the structure of the search space. Abstraction techniques like program slicing and CEGAR aim to explore as little of the state space as necessary to prove a program safe or unsafe. Thus, we want to know how many states were explored even though not necessary, and how the algorithm traversed through the search space.

To visualize this, we propose the structure of so-called *pixel trees*. The search space of an algorithm can be represented by a tree, e.g., by an ARG. The algorithm starts at one initial, given state: this state is the root of the ARG. From there, successors of that state are computed. Successors of the state are the direct children of it in the ARG, and each computation step is one additional edge in the ARG from the root. Thus, the distance of a state in the ARG to the root of the ARG is the number of computations necessary to reach it (in a tree, we can simplify this to the depth of a state in the tree). If

we represent the ARG as a traditional tree structure with information in the nodes (e.g., Fig. 7), it quickly becomes too big to comprehend.

Instead, we abstract the node labels and omit the edges and represent the ARG through continuous, horizontal lines. The length of each line represents the number of elements on that level: Each unit of length represents one state. The row of the line (starting from the top) represents the distance from the root. E.g., the root is represented by a line of length 1 and is the first row. If the root has two successors, the line in the second row will be of length 2. We do always align lines to the center, i.e., the position of a line does not guarantee any specific information about the parent. In addition to the line length and position, we color special states to transport additional information: Abstract successors that are already covered by the reached set, i.e., for which the stop operator held, and which were not added to the reached set, are still displayed and colored green. Target states are colored red, and states that were merged and replaced by another state are colored yellow.

Figure 24 shows the pixel trees for the ARG of our running example when using plain symbolic execution (Fig. 24a), when using symbolic execution with slicing (Fig. 24b), and when using symbolic execution with a combination of slicing and CEGAR (Fig. 24c). Since the first two ARGs have an infinite amount of states, we only display the pixel trees up to the last level displayed in the corresponding ARG figures. In the evaluation, we present the pixel trees created by different analyses (Fig. 26).

The creation of pixel trees is performant: it requires a single breadth-first search, where each node must only be visited once. While we focus on the creation of pixel trees for ARGs in this work, they can be used to represent any graph-like structure that contains exactly one entry or start node. They could, for example, also be used to visualize dependence graphs or CFAs.

4 Evaluation

4.1 Setup

Computing Resources We performed our experiments on machines with an Intel Xeon E3-1230 v5 CPU, with 3.4 GHz and 8 processing units. Each machine has 33 GB of memory. We used Ubuntu 16.04 with Linux kernel 4.4 as operating system. We limited each analysis run to 15 GB of memory, 4 processing units, and a time limit of 900 s. We always give CPU time with 2 significant digits.

Benchmark Tasks A verification task consists of an input program and a program specification/property to check the program against. For our experiments, we use a subset of the largest benchmark set of verification tasks for C to this date, the sv-benchmarks benchmark set[11]. The set has been used for each of the yearly iterations of the International Competition of Software Verification (SV-COMP) since 2012 [3], and is thus widely used and has a high probability of good quality. We use it in revision 604cca4.

The sv-benchmark set consists of multiple task categories and program property categories. For our evaluation, we use all benchmark tasks with a reachability property—the property defines that function `__VERIFIER_error` may not be called. Because symbolic execution in CPAchecker can not handle recursion, we exclude the category *Recursive* of verification tasks with recursive function calls.

In total, we use a set of 5 590 verification tasks for our experiments. Table 1 gives an overview over the size in lines of code (LOC) and the different characteristics of these tasks, sorted by their categories.

Tools We use CPAchecker in revision `r27515` [12] of the trunk and Symbiotic in the latest release, `spin-2018` [13]. For the combination of CPAchecker and Symbiotic, we use the same CPAchecker version, but a modified version of Symbiotic in our fork [14], revision `4509c83`. For reliable benchmarking, we use BenchExec in version `1.14` [15]. BenchExec allows the individual isolation of each verification run, and thus reduces the risk of measurement errors.

Availability All experimental data are available on our supplementary web page [16]. We also provide an artifact [17] that contains the experimental data and all tools necessary to repeat our results.

[11] http://github.com/sosy-lab/sv-benchmarks
[12] https://github.com/sosy-lab/cpachecker
[13] https://github.com/staticafi/symbiotic/tree/spin-2018
[14] http://github.com/leostrakosch/symbiotic
[15] https://github.com/sosy-lab/benchexec/tree/1.14
[16] http://www.cip.ifi.lmu.de/~lembergerth/slicing
[17] https://doi.org/10.5281/zenodo.1194263

Table 1: Overview of used tasks of sv-benchmarks benchmark set

Category	Tasks	LOC					Prominent C features
		Overall	Min.	Max.	Avg.	Median	
Arrays	167	7 275	14	1 161	44	36	C Arrays
True	123	5 700	14	1 161	46	36	
False	44	1 575	15	57	36	37	
BitVectors	50	10 511	13	696	210	39	Bit vector arithmetics
True	36	8 275	15	696	230	47	
False	14	2 236	13	636	160	32	
ControlFlow	94	183 904	94	22 300	2 000	1 600	Complicated control flow
True	52	100 866	94	22 300	1 900	1 100	
False	42	83 038	220	10 835	2 000	1 700	
ECA	1149	29 685 918	344	185 053	26 000	4 300	Lots of branching and many different dependencies between program variables
True	738	17 737 301	344	185 053	24 000	2 600	
False	411	11 948 617	566	185 053	29 000	4 800	
Floats	172	47 518	9	1 122	280	37	Floats (+ arithmetics)
True	141	46 548	9	1 122	330	50	
False	31	970	15	154	31	31	
Heap	181	142 378	11	4 605	790	650	C heap structures
True	110	88 544	11	4 576	800	500	
False	71	53 834	19	4 605	760	660	
Loops	163	10 026	14	1 647	62	25	C loops
True	111	5 991	14	476	54	26	
False	52	4 035	14	1 647	78	23	
ProductLines	597	1 160 305	838	3 789	1 900	1 000	Lots of branching because of software product line options
True	332	539 446	838	3 693	1 600	980	
False	265	620 859	847	3 789	2 300	3 000	
Sequentialized	273	580 463	330	18 239	2 100	1 100	Sequentialized, previously multi-threaded programs
True	103	255 265	330	18 239	2 500	1 200	
False	170	325 198	331	15 979	1 900	970	
LDV	2744	40 335 619	339	227 732	15 000	9 400	Linux device driver modules
True	2389	34 219 364	339	227 732	14 000	8 300	
False	355	6 116 255	1 389	85 772	17 000	13 000	
Total	5590	72 163 917	9	227 732	13 000	3 500	
True	4135	53 007 300	9	227 732	13 000	4 000	
False	1455	19 156 617	13	185 053	13 000	3 000	

Table 2: Verification results of different symbolic execution techniques

	SymEx	SymEx$_S$	SymEx$_C$	SymEx$_{SC}$	Symbiotic
Correct	787	1477	**2527**	2062	1709
Correct proof	203	**1088**	2058	1742	1075
Correct alarm	584	389	469	320	**634**
Incorrect	**23**	20	20	19	16
Incorrect proof	1	0	0	0	**8**
Incorrect alarm	**22**	20	20	19	8
Unknown	4780	4093	3043	3509	3865
Total	5590	5590	5590	5590	5590

4.2 Slicing and CEGAR in CPAchecker

First, we want to compare the performance of symbolic execution in CPAchecker with program slicing (SymEx$_S$), CEGAR (SymEx$_C$), and a combination of both (SymEx$_{SC}$). As a baseline, we use the symbolic execution implementation without CEGAR (SymEx), and Symbiotic with its symbolic execution-back-end Klee (Symbiotic). For slicing in CPAchecker, we always use target-path slicing.

Table 2 shows the results of these experiments. It shows the correct and incorrect verification results of each technique. A *correct proof* means that the technique correctly computed that the input program fulfills the reach property, i.e., it is safe. A *correct alarm* means that the technique correctly computed that the input program does not fulfill the reach property, and that it provided a counterexample for it. An *incorrect proof* means that the technique claims that the input program fulfills the reach property, even though it does not, and an *incorrect alarm* means that the technique claims that the input program does not fulfill the reach property (i.e., violates the property) even though the program is actually safe.

The results show that SymEx$_S$ can find significantly more proofs than plain symbolic execution, SymEx, but that it also finds significantly less property violations. SymEx$_S$ also performs worse than SymEx$_C$ in both finding proofs and alarms. SymEx$_S$ is comparable to Symbiotic in the number of found proofs, but it can only find half the amount of property violations (389 vs. 634). The combination of CEGAR and slicing, SymEx$_{SC}$, is better than both SymEx$_S$ and Symbiotic, but is still worse than SymEx$_C$. Of all techniques, SymEx$_{SC}$ finds the least amount of bugs, i.e., 320.

Figure 25 visualizes the performance of the different techniques. It shows for each technique, for an x-coordinate n, the CPU time (in seconds) required to solve the n-th fastest solved task. It reflects the data from the results table (Table 2), and additionally shows, that SymEx$_{SC}$ is also comparable in speed to the other techniques implemented in CPAchecker, in general. Symbiotic is very fast for the first, approximately, 1000 tasks, but its time consump-

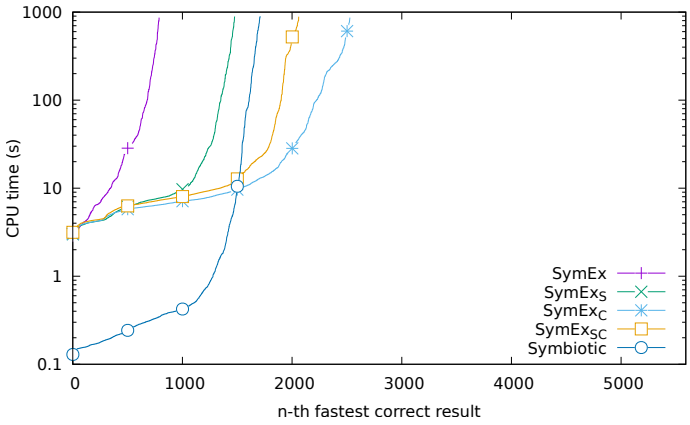


Figure 25: Quantile plot for the performance of the considered techniques on our benchmark set

Table 3: Different capabilities of the three symbolic execution techniques. Each cell shows the number of safe (t) and unsafe (w) verification tasks, that the technique of the corresponding row can solve, but that the technique of the corresponding column can't.

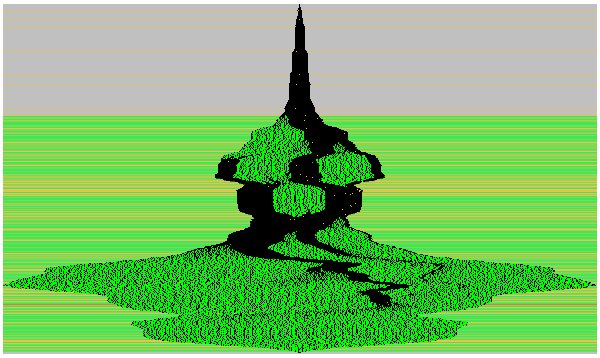
	SymEx	SymEx$_S$	SymEx$_C$	SymEx$_{SC}$
that		can't solve		
SymEx	–	18 t 202 w	94 t 189 w	96 t 286 w
SymEx$_S$	903 t 7 w	–	84 t 89 w	83 t 107 w
SymEx$_C$	1949 t 74 w	1054 t 169 w	–	318 t 149 w
SymEx$_{SC}$	1635 t 22 w	737 t 38 w	2 t 0 w	–
can solve				

tion increases steeply after that, and is slower than SymEx$_{SC}$ and SymEx$_C$ for the approximately last 200 tasks that it can solve.

We are not only interested in the overall performance of the different techniques, but want to go into more detail. In the following, we leave Symbiotic out because it is less comparable, due to its implementation outside of CPAchecker. Table 3 shows the difference in the capabilities of the techniques. Each cell shows the number of safe tasks without a property violation (t) and the number of unsafe tasks with a property violation (w), that the technique of the corresponding row can correctly solve and the technique of the corresponding column can not solve correctly. Most notably, the combination of slicing and CEGAR,

(a) SymExS (trimmed at both sides)

(b) SymExS

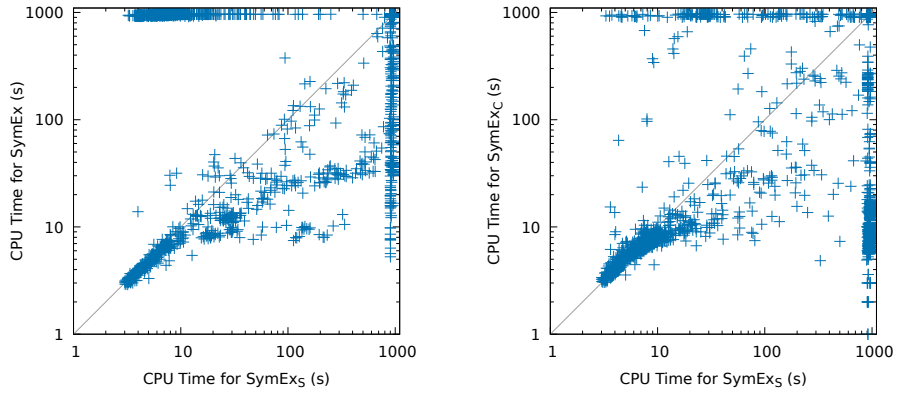


(c) SymExSC

Figure 26: Pixel trees for `Problem01_label44-false-unreach`

SymExSC, can only solve 2 additional tasks, compared to using CEGAR alone, and is able to solve a lot less tasks. SymExS, that uses slicing without CEGAR, on the other hand, has capabilities more different: While it also can't solve many of the tasks that SymExC can, it is able to proof 84 tasks safe and find property violations in 89 tasks that SymExC can't proof or find.

One of the two tasks that SymExSC can solve and that SymExC can't, can only be solved by SymExSC. It is a task from the *Linux Device Drivers* category, and we provide on our supplementary web-page the pixel tree of the ARG of its analysis with SymExS, SymExC and SymExSC. The second task is from the *ReachSafety-Loops* category, `trex01_true-unreach-call_true-termination.i`, and can be solved by slicing alone or slicing with CEGAR. To illustrate how different program slicing, CEGAR and the combination of both can behave on the same verification task, we present Fig. 26. It shows the pixel tree of the computed ARGs of SymExS, SymExC and SymExSC for verification task `Problem01_label44-false-unreach`. The trees are not in the same size ratio. All three tasks are able to solve the task and correctly produce an alarm, but compute significantly different ARGs. SymExS, which uses the full symbolic execution precision and only abstract through slicing, computes an ARG that is typical for path explosion: An exponential number of states is produced by the algorithm: the state space 'explodes' due to the high precision. Because of its breadth, the pixel tree is trimmed at both sides for better visibility. Since symbolic execution in CPAchecker uses breadth-first traversal through the search space, the target state is still found at some point. SymExS, in contrast, only tracks information that is really necessary, is able to abstract more states and creates a lean ARG. Interestingly, the combination of slicing and CEGAR, SymExSC, produces an ARG that is a mixture of the both previous ones. Apparently, CEGAR finds, through slicing,



(a) SymEx$_S$ vs. SymEx (b) SymEx$_S$ vs. SymEx$_C$

Figure 27: CPU time required by symbolic execution with slicing (SymEx$_S$) compared to plain symbolic execution (SymEx) and symbolic execution with CEGAR (SymEx$_C$)

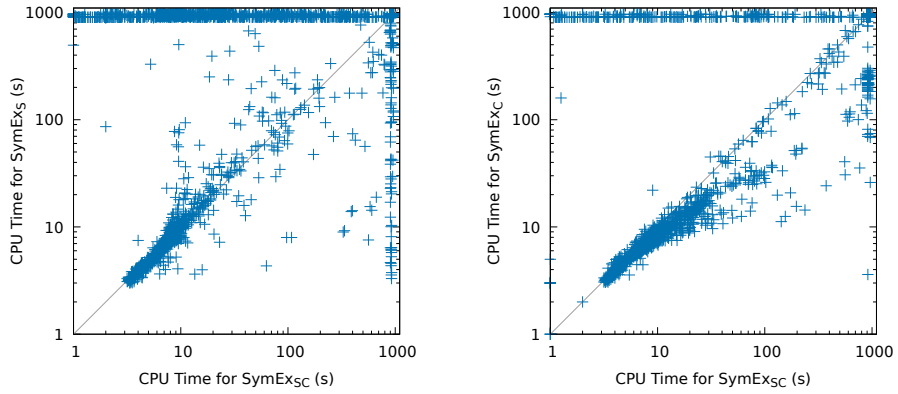
for this (specially selected) example worse target paths, or interpolants with information that is not as crucial, as when running CEGAR alone.

Figures 27a, 27b, 28a and 28b compare the CPU time required by the individual techniques for each verification task. The figures only show data for verification tasks that both of the corresponding techniques produced a *valid* result for. A result is valid, if is is either correct, or the tool ran into a timeout. This omits incorrect results, errors and early termination due to unsupported features. When we compare SymEx$_S$ to SymEx (Fig. 27a), we observe two interesting things: 1., SymEx$_S$ is for many tasks slower than SymEx (points below the diagonal), but 2. can solve a large amount of tasks below 10 seconds, for which SymEx takes more than 900 seconds and runs into a timeout (points in the upper left). Most of the tasks for which SymEx is faster than SymEx$_S$ are below 100 s. This means two things:

1. The technique with the higher abstraction (SymEx$_S$) is for tasks slower, for which the abstraction is not necessary (or even unhelpful) because the task requires a high precision or is simple enough that the technique with a lower abstraction (SymEx) can already solve it. And

2. the technique with the higher abstraction can, in turn, solve a significant amount of tasks a lot faster than the technique without abstraction, for which the abstraction helps because of a high, but irrelevant complexity in the program.

When we compare SymEx$_S$ to SymEx$_C$ (Fig. 27b), we see that slicing shows worse run-time performance than CEGAR. But SymEx$_S$ can also solve a significant amount of tasks that SymEx$_C$ can't solve. Because both techniques use abstraction with slightly different capabilities (as our running example through-



(a) SymExSC vs SymExS (b) SymExSC vs SymExC

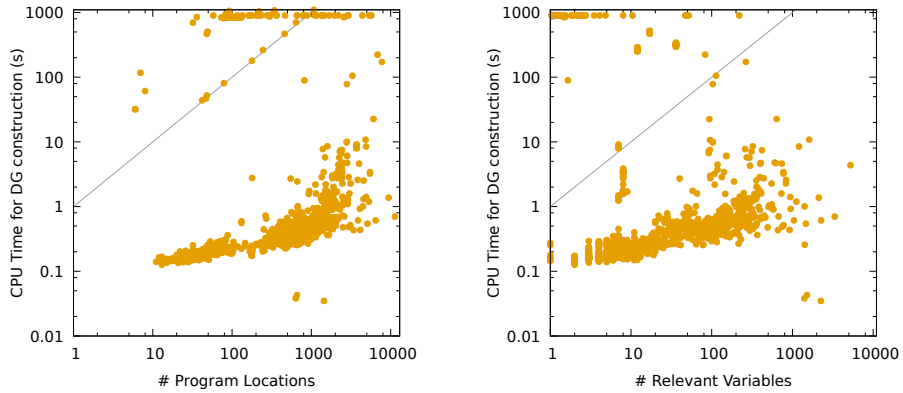
Figure 28: CPU time required by symbolic execution with slicing and CEGAR (SymExSC) compared to symbolic execution with slicing (SymExS) and symbolic execution with CEGAR (SymExC)

out this work showed), there are again tasks that only one of the two can solve successfully (points at the top and right edges of the figure).

The comparison of SymExSC to SymExS and SymExSC over only valid results shows the potential of the combination of program slicing and CEGAR. When we compare SymExS to SymExSC (Fig. 28a), there are a few tasks that SymExS can solve and SymExSC can't (points at right edge), but for the vast majority of tasks, SymExSC performs better. Figure 28b shows the comparison of SymExSC and SymExC: For some tasks, SymExSC, the a combination of both abstractions, is slower than the coarser abstraction technique SymExC (points below diagonal). But there is a significant amount of tasks that SymExSC can solve that SymExC can not. The time required by SymExSC for these is not always high, but is in the range of approximately 1 s to 900 s. This is an additional indicator that the combination with slicing does not only improve the time performance of CEGAR, but enables it to solve completely new types of problems.

Overhead of Slicing For most tasks, slicing itself produces little overhead. The slicing CPA only performs a single, hash-based check whether a given CFA edge is in the current program slice and then delegates to the wrapped analysis' transfer relation. Its merge operator and stop operator also simply delegate.

Two components may produce overhead: The creation of the dependence graph, and the refinement procedure computing a new slice. In the median, dependence graph construction requires 1.3 s of CPU time, but rises exponentially for large programs (Fig. 29), in general. Figure 29a shows on the x-axis the number of program locations of a verification task, and on the y-axis the CPU time required to construct its dependence graph. The maximum CPU time for dependence graph construction over tasks that can be solved is 65 s, For



(a) Number of program locations to construction time

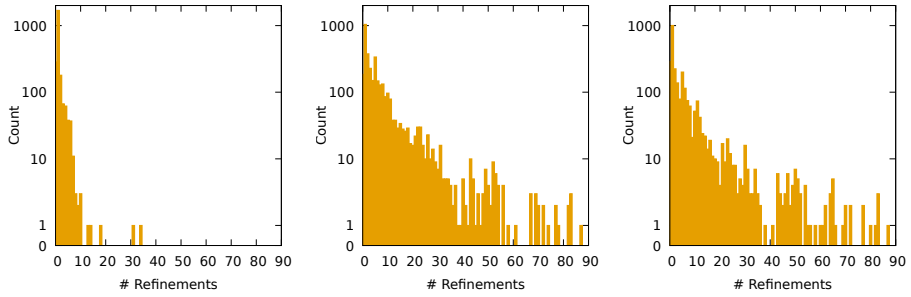
(b) Number of relevant variables to construction time

Figure 29: Indicators for the CPU time required for dependence graph construction

most tasks, the construction time is small and exponential to the number of program locations, but there is a set of tasks for which dependence graph construction takes even longer than the time limit of 900 s. For these, the number of program locations is no indicator, and they are a significant reason for a bad performance of the combination of slicing and CEGAR.

Another program property that may indicate the time required for dependence graph construction is the number of relevant variables in a program. A program variable is relevant, if it used in a branching condition, or if it is a transitive flow dependence to another program variable that is part of a branching condition. Figure 29b shows that over our experiments, only a small correlation exists: for an increasing number of relevant variables in a program, dependence graph construction time increases very weakly.

The second cause for overhead, the computation of new program slices, is also negligible for most tasks. Originally, our approach computes a new program slice at every refinement. Figure 30 shows on how many tasks how many refinements were performed, by slicing (SymEx_S), CEGAR (SymEx_C) and the combination of both (SymEx_SC). While slicing on its only requires a low amount of slices (one outlier at 31 refinements and one at 34, Fig. 30a), the more fine-grained abstraction of CEGAR requires a lot more refinements, up to 87 (Fig. 30b). This influences the combination of slicing and CEGAR (Fig. 30c). To optimize the slicing procedure over a high amount of refinements, we only compute a new slice if a slicing criterion is not in the existing slice. This reduces the amount of performed slice computations by 63 %, on average. On average, computation of a new slice requires 1.3 of CPU time, and 0.0025 s in the median. For large dependence graphs, computation time can increase significantly, though. The maximum CPU time required for computing a new slice was 180 s in our experiments.



(a) SymExS (b) SymExC (c) SymExSC

Figure 30: Counts of the number of refinements of each abstraction technique over all verification tasks. Fig. 30c has one outlier at coordinate (120, 1)

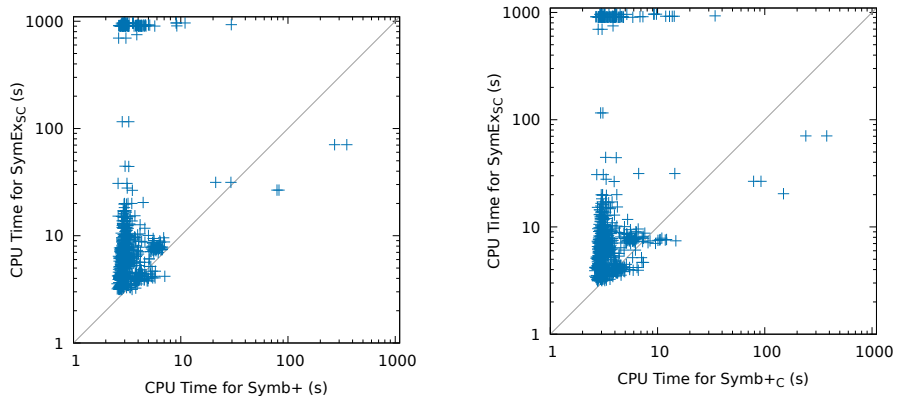
Table 4: Verification results of Symbiotic, Symb+, Symb+C and SymExSC

	Symbiotic	Symb+	Symb+C	SymExSC
Correct	1263	722	770	**1446**
Correct proof	813	604	650	**1190**
Correct alarm	**450**	118	120	256
Incorrect	5	13	**44**	2
Incorrect proof	0	**8**	**8**	2
Incorrect alarm	5	5	**36**	0
Unknown	1682	2215	2136	1502
Total	1867	1867	1867	1867

4.3 Slicing with Symbiotic

Next, we want to compare our own slicing procedure to the one implemented in Symbiotic. While our Llvm front end in CPAchecker aims to support all Llvm instructions necessary for C code, Symbiotic performs optimizations to the program code that create code that can't be clearly mapped to C code e.g., integers of arbitrary bit width. In addition, our front end is still a prototype. Because of this, we restrict our benchmark set to a set of verification tasks that can be parsed by CPAchecker. Of the original set of 5 590 verification tasks, we use 2 953 tasks that don't produce an error for any of the used techniques. Of these, 2 135 tasks are safe, and 815 contain a property violation.

Table 4 shows the results of our experiments for combinations of Symbiotic with CPAchecker. It shows the results for Symbiotic with Klee (Symbiotic), Symbiotic with CPAchecker with symbolic execution and without CEGAR (Symb+), Symbiotic with CPAchecker with symbolic execution and with CEGAR (Symb+C), and SymExSC. While the total numbers of Symb+ and Symb+C are significantly lower than the other two tools, Symb+ can solve 47 tasks that



(a) Symb+ vs. SymEx$_{SC}$ (b) Symb+$_C$ vs. SymEx$_{SC}$

Figure 31: CPU time per verification task of Symb+ and Symb+$_C$, compared to SymEx$_{SC}$

neither Symbiotic nor SymEx$_{SC}$ can solve, and Symb+$_C$ can solve 87 tasks that neither Symbiotic nor SymEx$_{SC}$ can solve.

Pre-processing through Symbiotic can also provide a major speed-up: both Symb+ and Symb+$_C$ are significantly faster for a set of verification tasks, as visualized by Figs. 31a and 31b. Even though our implementation is an incomplete prototype, it shows that the use of Symbiotic as a pre-processing step to CPAchecker can provide a significant advantage for some verification tasks.

4.4 Threats to Validity

Every benchmark set is biased, and the chances are high that this is also true for the sv-benchmark set. Even though it consists of a high number of tasks, these may not be representative for real-world software. That said, the sv-benchmark set is the largest benchmark set of C programs, to this date. Since the implementation was not only, but mostly tested on this benchmark set, it may be tuned towards it, and other, less frequent or absent C features may not be supported. This is also true for Symbiotic and the default configuration of symbolic execution in CPAchecker.

Measurements may be imprecise, but we ran all our experiments with BenchExec in isolated containers, and only used a single type of machine to produce our data. Thus, it is highly unlikely that significant measurement errors are present.

Our implementation is a prototype implementation and certainly contains bugs. While these may worsen the results of our analysis, it is highly unlikely that a high amount of correct results are the accidental result of bugs because CPAchecker runs a feasibility check with a predicate analysis on every found counterexample.

To increase the validity of our results, we did not only implement slicing in CPAchecker to improve comparability with other analyses in CPAchecker, but we also extended the third-party tool Symbiotic so that it supports CPAchecker. Thus, we used two completely different and independently developed program slicing tools.

5 Conclusion

In this work, we successfully designed and implemented the Slicing CPA in CPAchecker. As a novelty, this approach to program slicing uses CEGAR to derive program slices for dynamically computed slicing criteria. Thanks to the versatility of the CPA framework, this slicing approach can be freely combined with arbitrary analyses. It is, for example, possible to augment the flow dependence analysis with arbitrary other analyses to reduce the dependence graph and keep slices small. As a next step, we extended our CEGAR-based slicing CPA to allow the combination with arbitrary other CEGAR approaches. As an example, we combined it with the CEGAR-based symbolic execution analysis. While this decreased the overall performance of the analysis with CEGAR, it resulted in a significant speed-up for a large amount of tasks, and it performed significantly better than slicing on its own. We were able to provide, through an extensive evaluation and a detailed running example, proof that program slicing is not a mere subset of CEGAR, and that the combination of both techniques can yield vast performance improvements.

To increase confidence in our results, We extended the existing Llvm program slicer and program verifier Symbiotic to support CPAchecker as verification engine, and implemented an Llvm front-end to CPAchecker. This front-end allows, unlike other front-ends in CPAchecker, to apply all techniques for C programs that exist in CPAchecker to Llvm programs. Even though still a prototype, we were able to confirm our previous results with this combination of Symbiotic and CPAchecker.

To get a better understanding of the state space of a program that got explored by an analysis run, we implemented the creation of pixel trees. Pixel trees are an abstract representation of ARGs that can give a concise image of the structure of any graph with a single entry node.

Future work will explore the change in behavior of the combination of program slicing and CEGAR on symbolic execution and other abstract domains, and explore different further optimizations for the slicing CPA and dependence graph construction. A main part of this will be the adoption of more ideas from dynamic slicing [35].

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, LNCS 4963, pages 367–381. Springer, 2008.

[3] D. Beyer. Competition on software verification (SV-COMP). In *Proc. TACAS*, LNCS 7214, pages 504–524. Springer, 2012.

[4] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. SAS*, LNCS 3148, pages 2–18. Springer, 2004.

[5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *Proc. FMCAD*, pages 25–32. IEEE, 2009.

[6] D. Beyer, S. Gulwani, and D. Schmidt. Combining model checking and data-flow analysis. In E. M. Clarke, T. A. Henzinger, and H. Veith, editors, *Handbook on Model Checking*. Springer, 2017, to appear.

[7] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.

[8] D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.

[9] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.

[10] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.

[11] D. Beyer and T. Lemberger. Symbolic execution with CEGAR. In *Proc. ISoLA*, LNCS 9952, pages 195–211. Springer, 2016.

[12] D. Beyer and T. Lemberger. Software verification: Testing vs. model checking. In *Proc. HVC*. Springer, 2017, to appear.

[13] D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

[14] D. Beyer, S. Löwe, and P. Wendler. Refinement selection. In *Proc. SPIN*, LNCS 9232, pages 20–38. Springer, 2015.

[15] D. Beyer, S. Löwe, and P. Wendler. Sliced path prefixes: An effective method to enable refinement selection. In *Proc. FORTE*, LNCS 9039, pages 228–243. Springer, 2015.

[16] D. Beyer and P. Wendler. Algorithms for software model checking: Predicate abstraction vs. Impact. In *Proc. FMCAD*, pages 106–113. FMCAD, 2012.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. TACAS*, LNCS 1579, pages 193–207. Springer, 1999.

[18] R. S. Boyer, B. Elspas, and K. N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. In *Proc. ICRS*, pages 234–245. ACM, 1975.

[19] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, pages 443–446. IEEE, 2008.

[20] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224. USENIX Association, 2008.

[21] M. Chalupa, M. Vitovská, M. Jonáš, J. Slaby, and J. Strejček. Symbiotic 4: Beyond reachability (competition contribution). In *Proc. TACAS*. Springer, 2017.

[22] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[23] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT, 1999.

[24] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge (MA), 1999.

[25] L. A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*, pages 488–491. ACM, 1976.

[26] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[27] A. Driemeyer. Software-Verifikation von Java-Programmen in CPAchecker. Bachelor's Thesis, University of Passau, Software Systems Lab, 2012.

[28] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. ICSE*, pages 3–12. IEEE, 2007.

[29] 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.

[30] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54. ACM, 2007.

[31] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. PLDI*, pages 213–223. ACM, 2005.

[32] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[33] J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded symbolic execution for program verification. In *Proc. RV*, LNCS 7186, pages 396–411. Springer, 2011.

[34] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[35] B. Korel and J. W. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.

[36] B. Korel and J. Rilling. Dynamic program slicing methods. *Information & Software Technology*, 40(11-12):647–659, 1998.

[37] Y. Köroglu and A. Sen. Design of a modified concolic testing algorithm with smaller constraints. In *Proc. ISSTA*, pages 3–14. ACM, 2016.

[38] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[39] S. Löwe. Effective approaches to abstraction refinement for automatic software verification. PhD Thesis, University of Passau, Software Systems Lab, 2017.

[40] A. D. Lucia. Program slicing: Methods and applications. In *Proc. SCAM*, pages 144–151. IEEE, 2001.

[41] K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.

[42] K. L. McMillan. Lazy abstraction with interpolants. In *Proc. CAV*, LNCS 4144, pages 123–136. Springer, 2006.

[43] T. W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40(11-12):701–726, 1998.

[44] P. Rümmer and P. Subotic. Exploring interpolants. In *Proc. FMCAD*, pages 69–76. IEEE, 2013.

[45] J. Slaby, J. Strejcek, and M. Trtík. Checking properties described by state machines: On synergy of instrumentation, slicing, and symbolic execution. In *Proc. FMICS*, LNCS 7437, pages 207–221. Springer, 2012.

[46] J. Slaby, J. Strejcek, and M. Trtík. Compact symbolic execution. In *Proc. ATVA*, LNCS 8172, pages 193–207. Springer, 2013.

[47] T. Su, Z. Fu, G. Pu, J. He, and Z. Su. Combining symbolic execution and model checking for data flow testing. In *Proc. ICSE*, pages 654–665. IEEE, 2015.

[48] M. Weiser. Programmers use slices when debugging. *Commun. ACM*, 25(7):446–452, 1982.

[49] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357, 1984.

[50] M. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

München, den 9. März 2018

Thomas Lemberger