Backward Symbolic Execution with Loop Folding

Marek Chalupa and Jan Strejček

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

Abstract. *Symbolic execution* is an established program analysis technique that aims to search all possible execution paths of the given program. Due to the so-called path explosion problem, symbolic execution is usually unable to analyze all execution paths and thus it is not convenient for program verification as a standalone method. This paper focuses on *backward symbolic execution (BSE)*, which searches program paths backwards from the error location whose reachability should be proven or refuted. We show that this technique is equivalent to performing *k-induction* on control-flow paths. While standard BSE simply unwinds all program loops, we present an extension called *loop folding* that aims to derive loop invariants during BSE that are sufficient to prove the unreachability of the error location. The resulting technique is called *backward symbolic execution with loop folding (BSELF)*. Our experiments show that BSELF performs better than BSE and other tools based on k-induction when non-trivial benchmarks are considered. Moreover, a sequential combination of symbolic execution and BSELF achieved very competitive results compared to state-of-the-art verification tools.

SAS Artifact ★ Validated

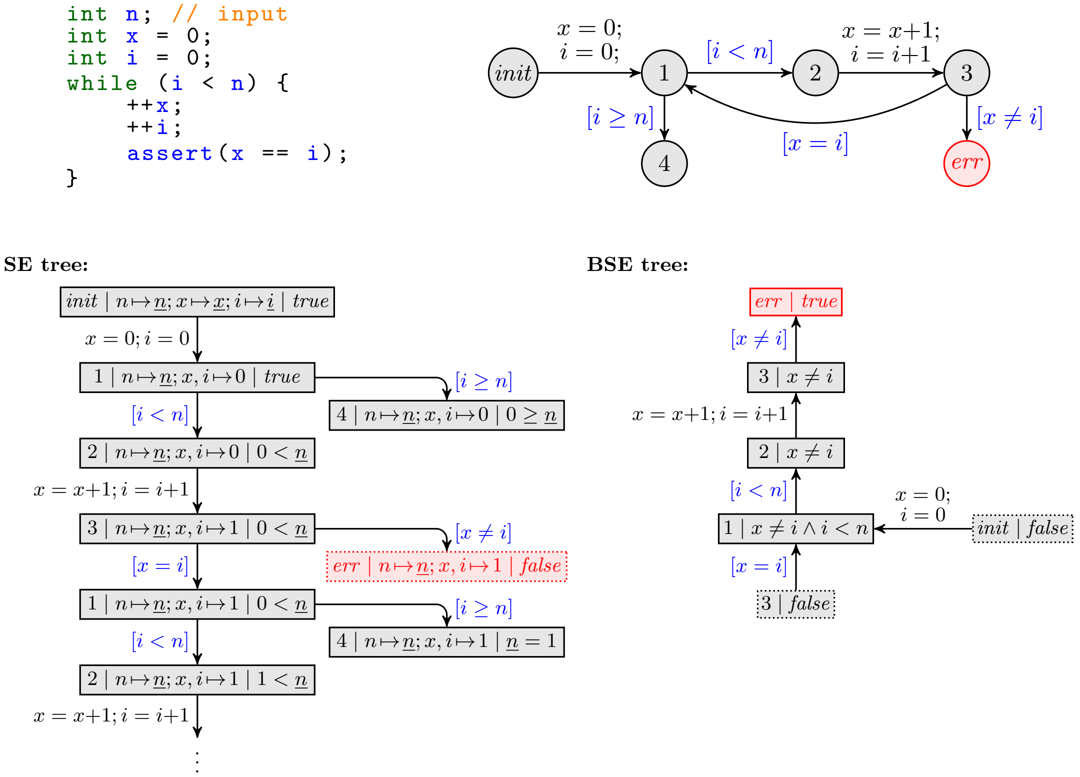
SAS Artifact ★ Extensible

SAS Artifact ★ Available

1 Introduction

Symbolic execution (SE) [56] is a widely used technique for static program analysis. In principle, SE runs the program on symbols that represent arbitrary input values with the aim to explore all execution paths. This approach is inherently doomed to suffer from the *path explosion* problem. In other words, it typically runs out of available resources before finishing the analysis as the number of all execution paths is often very large or even infinite. Moreover, some execution paths may be infinite, which is another obstacle that makes SE fail to completely analyze the program.

Many techniques modifying or extending SE have been introduced to mitigate the path explosion problem. Some of them try to reduce the set of considered execution paths [20, 74, 81, 83] or process multiple execution paths at once [46, 58, 76]. Others focus on the efficient processing of program loops [75, 39, 80], computation of reusable function summaries [38, 3, 70], or they do not symbolically execute nested [63] or library [61] functions as these are assumed to be correct. There are also approaches combining SE with other established techniques like predicate abstraction [40], counterexample-guided abstraction refinement (CEGAR) [16], or interpolation [47, 64]. We refer to a recent survey [6] for more information about symbolic execution and its applications.



```
int n; // input
int x = 0;
int i = 0;
while (i < n) {
     ++x;
     ++i;
     assert(x == i);
}
```

$init$
$x = 0;$
$i = 0;$
1
$[i < n]$
2
$x = x{+}1;$
$i = i{+}1$
3
$[i \geq n]$
4
$[x = i]$
$[x \neq i]$
err
SE tree:
$init \mid n{\mapsto}\underline{n}; x{\mapsto}\underline{x}; i{\mapsto}\underline{i} \mid true$
$x = 0; i = 0$
$1 \mid n{\mapsto}\underline{n}; x, i{\mapsto}0 \mid true$
$[i \geq n]$
$4 \mid n{\mapsto}\underline{n}; x, i{\mapsto}0 \mid 0 \geq \underline{n}$
$[i < n]$
$2 \mid n{\mapsto}\underline{n}; x, i{\mapsto}0 \mid 0 < \underline{n}$
$x = x{+}1; i = i{+}1$
$3 \mid n{\mapsto}\underline{n}; x, i{\mapsto}1 \mid 0 < \underline{n}$
$[x \neq i]$
$err \mid n{\mapsto}\underline{n}; x, i{\mapsto}1 \mid false$
$[x = i]$
$1 \mid n{\mapsto}\underline{n}; x, i{\mapsto}1 \mid 0 < \underline{n}$
$[i \geq n]$
$4 \mid n{\mapsto}\underline{n}; x, i{\mapsto}1 \mid \underline{n} = 1$
$[i < n]$
$2 \mid n{\mapsto}\underline{n}; x, i{\mapsto}1 \mid 1 < \underline{n}$
$x = x{+}1; i = i{+}1$
⋮
BSE tree:
$err \mid true$
$[x \neq i]$
$3 \mid x \neq i$
$x = x{+}1; i = i{+}1$
$2 \mid x \neq i$
$[i < n]$
$1 \mid x \neq i \wedge i < n$
$x = 0;$
$i = 0$
$init \mid false$
$[x = i]$
$3 \mid false$

Fig. 1. Top part presents a simple program (left) and the corresponding control-flow automaton (right) with *err* location representing the assertion violation. The bottom part shows the infinite SE tree (left) starting in the initial location *init*, and the BSE tree (right) starting in the error location *err*.

Our original research goal was to study possible combinations of SE and *k-induction* [79] for program verification, in particular for the *error location reachability* problem, i.e., the problem to decide whether there exists an execution of the program that reaches an error location. k-induction has been introduced as a technique for checking safety properties of symbolic transition systems by induction with respect to the length of paths in the system. It has been also adapted to model checking software [30, 13, 36, 71], where the induction is typically led with respect to the number of loop iterations. We show that in the context of error location reachability problem, k-induction applied to control-flow paths of a given program corresponds to *backward symbolic execution* with the breadth-first search strategy. This is the first result of the paper.

Backward symbolic execution (BSE) [23, 43, 29] is the backward version of SE: it starts at the program location whose reachability is to be determined and symbolically executes the program backwards until it either reaches the initial location or all analyzed paths become infeasible. Similarly, as in the case of SE, this process may never terminate.

Let us illustrate the difference between SE and BSE on a very simple example. Assume that we want to verify the validity of the assertion in the program in Figure 1 (top left). In other words, we need to decide the error location reachability problem for the location *err* in the corresponding control-flow automaton (top right). SE assigns to each variable v the symbol $\underline{v}$ representing its input value. Further, SE gradually builds the *SE tree* (bottom left) of paths starting in *init*. Each node in the SE tree is labelled with a triple $l \mid m \mid \phi$ of the current program location l, the memory content m, and the *path condition* ϕ, which is the weakest precondition on input values that makes the program follow the path leading from the tree root to the node. Whenever a path becomes infeasible, i.e., its path condition becomes unsatisfiable, SE stops executing this path (we draw such nodes dotted). Clearly, the assertion is valid iff the tree does not contain any node that is labelled with *err* and a satisfiable path condition. The assertion in Figure 1 is valid, but SE cannot prove it as the SE tree is infinite.

BSE works similarly, but it proceeds from the error location backward to *init*. In other words, instead of computing the weakest precondition of paths that start in *init*, it computes the weakest precondition of paths that end in *err*. Note also that because BSE directly computes the precondition, it does not need to keep the contents of memory. For the program in Figure 1, the BSE tree (bottom right) is finite and because there is no feasible path from *init* to *err*, it proves that the assertion is valid.

Now consider a similar program given in Figure 2. This time, the results of SE and BSE are switched: the SE tree (bottom left) is finite and implies the validity of the assertion, but the BSE tree (bottom right) is infinite and thus inconclusive.

The main difference between examples in Figures 1 and 2 from the BSE perspective is the position of the assertion. In both cases, BSE first processes the negation of the assertion. But only in the example with the assertion inside the loop, it is processed again and this time in the positive form, which makes the path infeasible. This illustrates that a valid assertion inside a program loop may be a loop invariant that is able to prove its own validity (it is inductive).

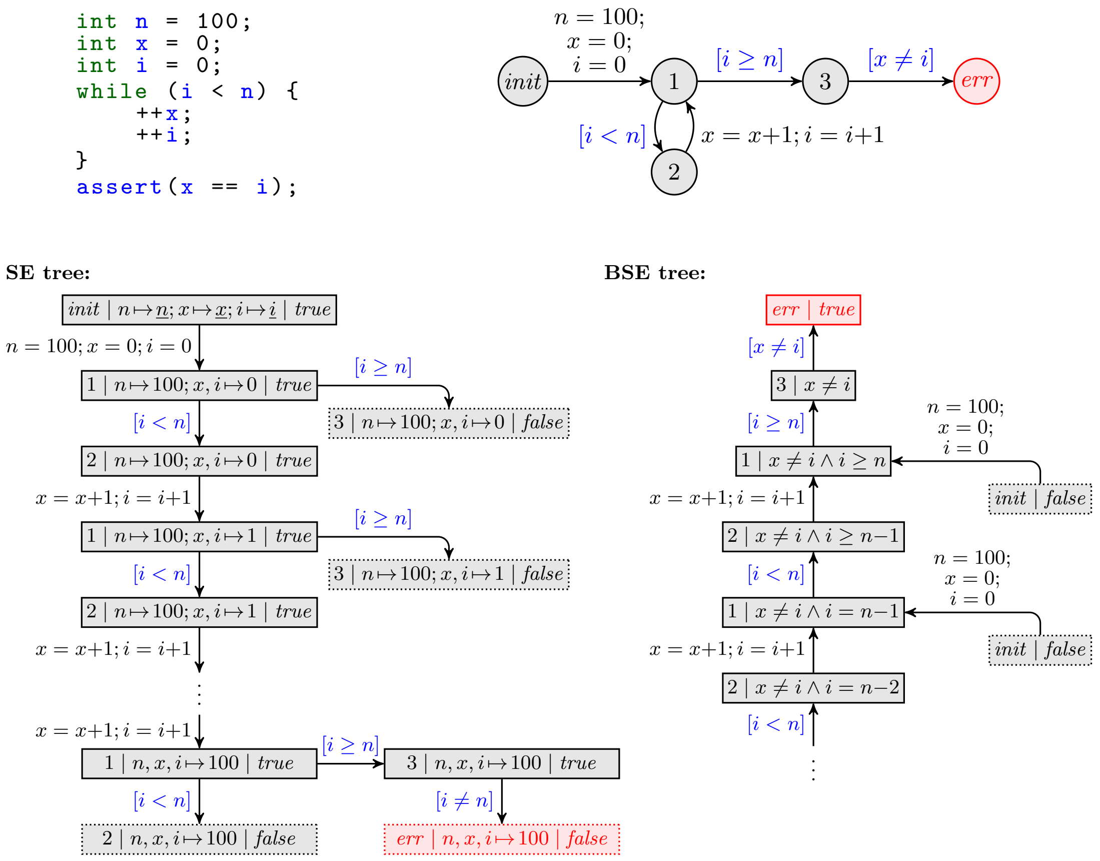
A standard solution to checking assertions that are not strong enough to prove their own validity is to use externally generated invariants [12, 19, 21, 44]. In this work, we address this issue by extending BSE with *loop folding* that attempts to infer inductive invariants during BSE. *Backward symbolic execution with loop folding (BSELF)* is the second result presented in this paper.

We have implemented both BSE and BSELF. Our experimental evaluation shows that BSELF is significantly more efficient than BSE and other tools implementing k-induction on non-trivial benchmarks. Further, our experiments indicate that a sequential combination of SE and BSELF forms a powerful tool fully comparable with state-of-the-art verification tools.

The paper is organized as follows. The next section provides necessary definitions. Section 3 studies BSE, k-induction, and the relation between them. The algorithm BSELF is described in Section 4 and the experimental evaluation is presented in Section 5. Finally, Section 6 discusses related work.

2 Preliminaries

In this paper, we consider programs represented as *control-flow automata (CFAs)* [14]. A CFA $A = (L, \mathit{init}, \mathit{err}, E)$ consists of a finite set L of *program locations*, an *initial location* $\mathit{init} \in L$, an *error location* $\mathit{err} \in L \smallsetminus \{\mathit{init}\}$, and a finite set $E \subseteq (L \smallsetminus \{\mathit{err}\}) \times \mathit{Ops} \times L$ of edges between program locations which are labeled with operations. An operation is either an *assumption* (denoted in figures with blue text in square brackets, e.g., $[x > 5]$) or a sequence of assignments (e.g., $x = y + 10; y = 5$). If a location has two or more outgoing edges, then all these edges have to be labeled with assumptions such that any valuation satisfies at most one of these assumptions. The error location has no outgoing edges.



```
int n = 100;
int x = 0;
int i = 0;
while (i < n) {
     ++x;
     ++i;
}
assert(x == i);
```

Fig. 2. Top part presents a simple program (left) and the corresponding control-flow automaton (right) with *err* location representing the assertion violation. The bottom part shows the infinite SE tree (left) starting in the initial location *init*, and the BSE tree (right) starting in the error location *err*.

A *control-flow path* or simply a *path* π in a CFA is a nonempty finite sequence of succeeding edges $\pi = (l_0, o_0, l_1)(l_1, o_1, l_2)\dots(l_{n-1}, o_{n-1}, l_n) \in E^+$. Locations l_0 and l_n are *start* and *target locations* of the path and we refer to them with $sl(\pi)$ and $tl(\pi)$, respectively. By $Locs(\pi)$ we denote the set of locations on π, i.e., $Locs(\pi) = \{l_i \mid 0 \leq i \leq n\}$. A path with start location *init* is called *initial*. A path is called *error path* if its target location is *err* and it is called a *safe path* otherwise. The *length* of the path π is denoted by $|\pi|$ and is equal to the number of edges on π, i.e. $|\pi| = n$.

We assume that our CFAs are *reducible* [48], i.e., every loop has a single entry location. The entry location of a program loop is called the *loop header*. We further assume that there are no nested loops in CFAs. Given a loop header h, by $LoopPaths(h)$ we denote the set of paths corresponding to a single iteration of the loop. Formally, $LoopPaths(h)$ is the set of all paths π such that $sl(\pi) = tl(\pi) = h$ and h does not appear inside π (i.e., $\pi = (h, o_0, l_1)(l_1, o_1, l_2)\dots(l_{n-1}, o_{n-1}, h)$ where $l_1, l_2, \dots, l_{n-1} \neq h$). We extend the *Locs* notation to loops identified by their headers such that $Locs(h) = \cup_{\pi \in LoopPaths(h)} Locs(\pi)$.

To simplify the presentation, we assume that programs manipulate only variables of a fixed bit-width integer type. A *program state* (or simply a *state*) is fully specified by a pair (l, v) of the current program location l and the current valuation v of program variables. An *initial state* consists of the initial location *init* and an arbitrary valuation. Given an edge $(l, [\psi], l') \in E$ labelled with an assumption ψ, we write $(l, v) \xrightarrow{(l,[\psi],l')} (l', v)$ for each valuation v satisfying ψ. Given an edge $(l, soa, l') \in E$ labelled with a sequence of assignments *soa*, we write $(l, v) \xrightarrow{(l,soa,l')} (l', v')$ for all valuations v and v' such that v' arises from v by performing the sequence of assignments. We generalize the notation to paths: given a program path $\pi = (l_0, o_0, l_1) \ldots (l_{n-1}, o_{n-1}, l_n)$, we write $(l_0, v_0) \xrightarrow{\pi} (l_n, v_n)$ whenever there exist valuations $v_1, v_2, \ldots, v_{n-1}$ satisfying $(l_i, v_i) \xrightarrow{(l_i,o_i,l_{i+1})} (l_{i+1}, v_{i+1})$ for each $0 \le i < n$. A path π is *feasible from a state* (l, v) if $(l, v) \xrightarrow{\pi} (l', v')$ holds for some state (l', v'). A path is called *feasible* if it is feasible from some program state. Note that if two paths are feasible from the same program state, then one of these paths has to be a prefix of the other.

In this paper, we study the *error location reachability* problem, i.e., the problem to decide whether a given CFA contains a feasible initial error path. The CFA is called *correct* if there is no such path. If the CFA is not correct, then any feasible initial error path is called an *error witness*.

In the following, we often work with a set of states that have the same program location and their valuations are models of some formula ϕ over program variables. Such a set is denoted as (l, ϕ) and it is formally defined as

$$(l, \phi) = \{(l, v) \mid v \textit{ satisfies } \phi\}.$$

A state (l', v') is *reachable from* (l, ϕ) if there exist $(l, v) \in (l, \phi)$ and a path π such that $(l, v) \xrightarrow{\pi} (l', v')$. A set (l, ϕ) is called *inductive* if each state reachable from (l, ϕ) with the location l is again in (l, ϕ). A set (l, ϕ) is an *invariant* if it contains all states with the location l that are reachable from $(\mathit{init}, \mathit{true})$.

Given a formula ϕ and a path π, let $\pi^{-1}(\phi)$ denote the weakest precondition of ϕ for the sequence of the operations along π. Formally, $\pi^{-1}(\phi)$ is a formula that is satisfied by a valuation v if and only if $(sl(\pi), v) \xrightarrow{\pi} (tl(\pi), v')$ for some v' satisfying ϕ. The formula $\pi^{-1}(\phi)$ can be computed from π and ϕ for example by symbolic execution of the path. Clearly, a path π is feasible if and only if $\pi^{-1}(\mathit{true})$ is satisfiable.

In general, we work with quantifier-free first-order formulas over a decidable theory. Each such a formula can be transformed into *conjunctive normal form (CNF)*, which is a conjunction of *clauses*, where each clause is a disjunction of *literals* and a literal is an atomic formula or its negation. We assume that there exists a decision procedure $sat(\phi)$ which returns *true* if ϕ is satisfiable and *false* otherwise. We say that two formulas are *disjoint* if their conjunction is unsatisfiable.

3 Backward Symbolic Execution and *k*-Induction

This section recalls backward symbolic execution and k-induction [79]. Moreover, it shows that the two techniques provide equivalent results when applied to the error location reachability problem.

3.1 Backward symbolic execution (BSE)

Backward symbolic execution (BSE) [23], sometimes also called *symbolic backward execution* [7], computes the weakest preconditions [27] of control-flow paths by a slightly different process than

```
Input: CFA $A = (L, init, err, E)$
Output: $correct$ if $A$ is correct, an error witness $\pi$ otherwise

$workbag \leftarrow E \cap (L \times Ops \times \{err\})$
while $workbag \neq \emptyset$ do
    $\pi \leftarrow$ pick a path of the minimal length from $workbag$
    $workbag \leftarrow workbag \smallsetminus \{\pi\}$
    if $\pi$ is feasible then
        if $\pi$ is initial then return error witness $\pi$
        $workbag \leftarrow workbag \cup \{e\pi \mid e \in E \wedge tl(e) = sl(\pi)\}$
return $correct$
```

Algorithm 1: The backward symbolic execution (BSE) algorithm.

SE. In particular, paths are explored in the opposite direction: from the error location towards the initial location. BSE either shows that all error paths are infeasible, or it finds a feasible initial error path, or it runs forever. We assume that paths are explored in the shortest-first order. A high-level formulation of BSE is provided in Algorithm 1.

In the beginning, *workbag* is set to contain all paths of length 1 leading to the error location. Until *workbag* is empty, it takes a path from *workbag* and checks its feasibility. If the path is infeasible, it is discarded. In the opposite case, we check whether the path is also initial and report it as an error witness if the answer is positive. Otherwise, we prolong the path by each edge leading to its start location (i.e., we prolong it in the *backward* direction) and put all these prolonged paths to *workbag*. If *workbag* gets empty, it means that all initial error paths have an infeasible suffix and thus they are infeasible. Because each iteration picks a path of the minimal length from *workbag*, BSE invoked on an incorrect CFA eventually reports an error witness even if the number of feasible error paths is infinite. Unfortunately, there are correct programs for which BSE does not terminate as illustrated in Figure 2. More specifically, BSE does not terminate on correct CFAs with infinitely many feasible error paths.

Theorem 1. *Let P be the set of all feasible error paths of a CFA A. BSE executed on A*

- *returns an error witness if P contains an initial path;*
- *returns* correct *if P is finite and contains no initial path;*
- *does not terminate if P is infinite and contains no initial path.*

Proof. We start with a simple observation. Let $\pi \in P$ be a path of length n and, for each $0 < i \leq n$, let π_i be the suffix of π of length i. As π is a feasible error path, each suffix π_i is also a feasible error path and thus $\pi_i \in P$. Path π_1 of length 1 is inserted to *workbag* during its initialization. For each $0 < i < n$, when π_i is processed by BSE, either it is initial and reported as an error witness, or π_{i+1} is inserted to *workbag*.

Assume that A is incorrect, i.e., P contains an initial path. Let $\pi \in P$ be a feasible initial error path of the minimal length. Hence, no proper suffix of π is initial. The observation implies that for all $0 < i < n$, processing of π_i inserts π_{i+1} to *workbag*. When $\pi = \pi_n$ is processed, it is returned as an error witness unless other error witness is found sooner.

Now assume that A is correct, i.e., P contains no initial path. Note that *workbag* can contain only error paths of length one, paths in P, and paths of the form $e\pi$ such that $\pi \in P$. Hence if P

is finite, there are only finitely many paths that can appear in *workbag*. Moreover, every path can be inserted to *workbag* at most once. Altogether, we get that finiteness of P implies that *workbag* gets empty after a finite number of iterations and BSE returns *correct*.

If P is infinite and contains no initial path, the observation implies that every $\pi \in P$ eventually gets to *workbag*. Since every iteration of BSE removes only one path from *workbag* and P is infinite, BSE does not terminate. □

Note that usual implementations of BSE do not check the feasibility of each path in *workbag* from scratch [23, 24] as Algorithm 1 does. Instead, they gradually build the BSE tree of all feasible error paths as shown in Figures 1 and 2 by computing the weakest preconditions incrementally: for a path of the form $e\pi$, the value of $(e\pi)^{-1}(\mathit{true})$ is computed from the previously computed $\pi^{-1}(\mathit{true})$ using the relation $(e\pi)^{-1}(\mathit{true}) = e^{-1}(\pi^{-1}(\mathit{true}))$. We employ this incremental approach in Section 4 where we extend BSE with loop folding.

3.2 *k*-induction

The *k-induction* [79] technique uses induction to prove the correctness of transition systems. Adapted to CFAs, it is sufficient to prove these two statements for some $k > 0$ to show that the CFA is correct:

(Base case) All feasible initial paths of length at most k are safe.
(Induction step) Each feasible path of length $k+1$ that has a safe prefix of length k is also safe.

If the base case does not hold, then there exists a feasible initial error path and the CFA is not correct. To prove the induction step, we consider each feasible safe path π of length k and check that all paths that arise by prolonging π with an edge leading to the error location are infeasible. If this check fails, we cannot make any conclusion. Thus we try to prove both statements again for $k+1$. If we check the statements for $k = 1, 2, \ldots$, the base case can be simplified to checking only paths of length (exactly) k. The whole process is formalized in Algorithm 2.

The k-induction algorithm applied to an incorrect CFA eventually returns an error witness of the minimal length. When applied to a correct CFA, it either returns *correct* or it does not terminate.

Theorem 2. *Let P be the set of all feasible error paths of a CFA A. k-induction executed on A*

- *returns an error witness if P contains an initial path;*
- *returns* correct *if P is finite and contains no initial path;*
- *does not terminate if P is infinite and contains no initial path.*

Proof. Assume that A is incorrect, i.e., P contains an initial path. Let $\pi \in P$ be a feasible initial error path of the minimal length and let $n = |\pi|$. For each $0 < k < n$, the base case holds as all feasible initial paths of the length k are safe (due to the minimality of $|\pi|$) and the induction step cannot be proven as the suffix of π of length $k+1$ is a feasible error path with a safe prefix of length k. Hence, k-induction reaches the iteration for $k = n$ where the base case identifies π or another feasible initial error path of length n as an error witness.

Now assume that A is correct, i.e., P contains no initial path. The base case clearly holds for each k. The induction step holds for k if and only if all paths in P have length at most k. If P contains a path π of length at least $k+1$, then it also contains the suffix of π of length $k+1$. This suffix is a feasible error path with a safe feasible prefix of length k. Hence, the induction step fails

```
Input: CFA $A = (L, init, err, E)$
Output: $correct$ if $A$ is correct, an error witness $\pi$ otherwise

$k \leftarrow 1$
while $true$ do
    foreach initial path $\pi$ of length $k$ do                         // base case
        if $\pi$ is feasible and $tl(\pi) = err$ then return error witness $\pi$

    $inductionstepfail \leftarrow false$                                 // induction step
    foreach safe path $\pi$ of length $k$ do
        if $\pi$ is feasible then
            foreach $e = (tl(\pi), o, err) \in E$ do
                if $\pi e$ is feasible then
                    $inductionstepfail \leftarrow true$

    if $\neg inductionstepfail$ then return $correct$
    $k \leftarrow k + 1$
```

Algorithm 2: The algorithm for k-induction on control-flow paths.

for k. If all paths in P have length at most k, then all feasible paths of length $k+1$ are safe and the induction step holds. To sum up, if P is finite, the k-induction returns *correct* in the iteration where $k = \max\{|\pi| \mid \pi \in P\}$. If P is infinite, the induction step always fails as for any k there exists a path in P longer than k and thus k-induction does not terminate. □

Note that when k-induction is applied to finite transition systems instead of CFAs, the incompleteness can be fixed by restricting the induction step only to acyclic paths [79].

3.3 Equivalence of BSE and *k*-induction

Theorems 1 and 2 imply that BSE and k-induction return an error witness or the value *correct* in identical cases. On an incorrect CFA, both algorithms detect an error witness of the minimal length. On a correct CFA with a finite set P of all feasible error paths, the k-induction terminates for $k = \max\{|\pi| \mid \pi \in P\}$ and the longest path processed by BSE has length at most $k+1$ (as k-induction in fact checks paths of length $k+1$ for the given k).

If we look once again at Algorithm 2, we can see that the induction step can be simplified. Instead of analysing each feasible path π of length k and checking whether it can be prolonged into a feasible error path of the form πe, we can directly look for all error paths of length $k+1$ and check their feasibility. This form of the k-induction algorithm gets closer to BSE. The main difference is that BSE checks only the feasibility of error paths of length $k+1$ that have a feasible suffix of length k. Hence, we can see BSE as an optimized version of the k-induction algorithm.

4 BSE with Loop Folding (BSELF)

This section introduces our extension of BSE called *backward symbolic execution with loop folding (BSELF)*. Loop folding targets the incompleteness of BSE. Similar to other verification techniques, we approach this problem by using invariants that constraint the state space analyzed by BSE. Instead of relying on external invariant generators, we compute the invariants directly in BSELF. That allows us to compute disjunctive invariants which can be hard to discover for invariant-generation algorithms [73, 68, 44, 82].

Before describing BSELF in detail, we give a brief description of its functioning. BSELF is searching the program backwards from err as regular BSE. The difference comes when it runs into a loop, i.e., when it finds a feasible error path π with $tl(\pi) = err$ and $sl(\pi) = l$ where l is a loop header. Normal BSE would continue the backward search, unwinding the loop. BSELF, instead, attempts to fold the loop – infer an inductive invariant from which is the path π infeasible. The loop folding successively generates *invariant candidates*. An invariant candidate is a formula ξ such that the set of states (l, ξ) is inductive and π is infeasible from (l, ξ), i.e., $\xi \wedge \pi^{-1}(true)$ is unsatisfiable. The generation continues until either some invariant candidate is shown to be an actual invariant or a pre-set bound is reached, in which case we give up the current loop folding attempt. If an inductive invariant from which π is infeasible have been found, the search on π is terminated. Otherwise, BSELF continues BSE as if no loop folding took place. Irrespective of the result of the folding, we remember all the generated invariant candidates (if any) in an auxiliary set O_l so that we can recycle work if we hit l again on some path. The check of whether an invariant candidate is an invariant is performed by a nested call of BSELF. This way, we automatically handle sequentially chained loops.

The idea behind loop folding is the following. We start with an *initial invariant candidate* ξ_0 that we derive from π and/or previously computed invariant candidates stored in O_l. The set of states (l, ξ_0) is inductive and the program contains no nested loops, so ξ_0 must describe a set of states in which the program may be during some *last* iterations of the loop (this, in fact, holds for any invariant candidate). So if (l, ξ_0) is not an invariant, there is a possibility that adding states from previous iterations will make it an invariant. Thus, we compute the set of states in which the program may be one iteration before entering (l, ξ_0) and try to overapproximate these states to cover more than just one previous iteration of the loop. This step provides us with a new invariant candidate. If it gives rise to an invariant, we are done. Otherwise, we repeat the process to obtain a new invariant candidate and so on.

Precisely speaking, loop folding does not extend one invariant candidate all over again. Every invariant candidate can be extended to several new invariant candidates. Given an invariant candidate ξ, we first compute the pre-image of (l, ξ) along every path of the loop (thus we get $|LoopPaths(l)|$ pre-images). Every non-empty pre-image (l, ψ) from which is π infeasible is then overapproximated to one or more sets (l, ψ') such that $\psi' \vee \xi$ is a new invariant candidate (i.e., $\psi \implies \psi'$, $(l, \psi' \vee \xi)$ is inductive, and $(\psi' \vee \xi) \wedge \pi^{-1}(true)$ is unsatisfiable). Therefore, loop folding generates a tree of invariant candidates instead of a single sequence of invariant candidates.

We note that giving up loop folding is an important part of the design of BSELF. It has several effects: first, it constraints the time spent in computations that could stall the algorithm for a long time, e.g., nested calls of BSELF that check the invariance of invariant candidates. Second, remember that we store all invariant candidates generated for a loop l in O_l. After we give up a folding of the loop l on π, the next trial of folding l on a path derived from π will use also invariant candidates generated on other paths than π. Symmetrically, attempts to fold the loop l on other paths than π will use the invariant candidates computed during the loop folding of l on π. Finally, during loop folding, we never merge a newly generated invariant candidate into a previously generated candidate, i.e., we preserve the tree structure of candidates during loop folding. This tree structure is forgotten by storing candidates to O_l and thus further attempts to fold the loop l can

```
Procedure $BSELF(loc, \phi_0, infoldloop)$
    Input: location $loc \in L$, formula $\phi_0$ over program variables, a boolean $infoldloop$
    Output: $correct$ meaning that no state of $(loc, \phi_0)$ is reachable from $(init, true)$, or
            $incorrect$ meaning that a state of $(loc, \phi_0)$ is reachable from $(init, true)$, or
            $unknown$ meaning that the procedure finished without decision

    initialize $queue$ with $(loc, \phi_0, \emptyset)$
    while $queue$ is not empty do
        $(l, \phi, visited) \leftarrow$ pop item from $queue$

        if $\neg sat(\phi)$ then continue
        if $l = init$ then return $incorrect$

        if $l$ is a loop header then                    // start of the loop folding extension
            if $FoldLoop(l, \phi, visited)$ then return $correct$
            if $infoldloop$ then return $unknown$
            $visited \leftarrow visited \cup \{l\}$         // end of the loop folding extension

        foreach $e = (l', o, l) \in E$ do
            push $(l', e^{-1}(\phi), visited)$ to $queue$
    return $correct$
```

Algorithm 3: The main procedure of the BSELF algorithm.

merge these candidates generated for different paths through the loop and thus find invariants that need such a merging (see the last example of Subsection 4.5).

4.1 The BSELF algorithm

The pseudocode of BSELF is shown in Algorithm 3. To shorten the notation, we assume that an input CFA $(L, init, err, E)$ is fixed. Further, for each loop header l, the algorithm uses

- a global variable O_l initially set to $\emptyset$, which stores constructed *invariant candidates* at l, and
- a parameter $\kappa_l \geq 0$ which bounds the effort to infer an invariant at l in a single visit of l.

These global variables and parameters appear only in procedure *FoldLoop*. To decide the error location reachability problem, one should call $BSELF(err, true, false)$.

If we ignore the loop folding extension, Algorithm 3 is just an efficient version of Algorithm 1. The difference is that preconditions are now computed incrementally along individual edges of CFA instead of executing whole error paths. Since we lost the information about the length of paths, we use a first-in first-out *queue* instead of a workbag to achieve the shortest-path search order. The parameter *infoldloop* and sets *visited* have an effect only inside *FoldLoop* procedure. Indeed, Algorithm 3 executed with $infoldloop = false$ never returns *unknown*.

Before discussing the central procedure *FoldLoop* presented in Algorithm 4, we describe how it is used in the main BSELF loop. Whenever BSELF hits a loop header l with the states (l, ϕ) (further called the *error states*), we call the procedure *FoldLoop* which attempts to find an invariant (l, ρ) that proves the unreachability of the current error states, i.e., such that $\rho \wedge \phi$ is unsatisfiable. If the procedure succeeds, we return *correct*. If it fails, we check whether $infoldloop = true$. If it is the case,

```
Procedure $FoldLoop(l, \phi, visited)$
  Input: location $l \in L$, formula $\phi$ over program variables, and set $visited \subseteq L$
  Output: $true$ if an invariant disjoint with $(l, \phi)$ is found; $false$ otherwise

  $\psi \leftarrow InitialInvariantCandidate(l, \phi, visited)$
  if $\neg sat(\psi)$ then return $false$
  $workbag \leftarrow \emptyset$
  for $\psi' \in Overapproximate(l, \psi, false, \psi, \phi)$ do
    $workbag \leftarrow workbag \cup \{(\psi', \psi', \kappa_l)\}$
    $O_l \leftarrow O_l \cup \{\psi'\}$                    // update known invariant candidates

  while $workbag \neq \emptyset$ do
    $(\psi, \xi, k) \leftarrow$ pick item from $workbag$
    $workbag \leftarrow workbag \setminus \{(\psi, \xi, k)\}$
    $fail \leftarrow false$                                // check if $(l, \xi)$ is an invariant
    foreach $e = (l', o, l) \in E$ outside any loop do
      if $BSELF(l', e^{-1}(\neg\xi), true) \neq correct$ then
        $fail \leftarrow true$
        break
    if $fail = false$ then return $true$
    if $k > 0$ then                                        // extend the candidate
      foreach $\pi \in LoopPaths(l)$ do
        $\psi' \leftarrow \pi^{-1}(\psi)$
        if $\neg sat(\psi' \wedge \phi)$ then
          for $\psi'' \in Overapproximate(l, \psi', \psi, \xi, \phi)$ do
            $workbag \leftarrow workbag \cup \{(\psi'', \psi'' \vee \xi, k - 1)\}$
            $O_l \leftarrow O_l \cup \{\psi'' \vee \xi\}$  // update known invariant candidates
  return $false$
```

Algorithm 4: The procedure $FoldLoop(l, \phi)$ looking for invariants disjoint with (l, ϕ).

then BSELF was called from inside *FoldLoop* and we return *unknown*. This is to ensure progress and avoid stalling in nested calls of *FoldLoop*. Finally, if $infoldloop = false$, we update *visited* with l to remember that we have visited this loop on the current path and continue searching paths like in regular BSE.

Now we turn our attention to the procedure *FoldLoop* (Algorithm 4). In the following, by an *invariant candidate at location* l we mean a formula ξ such that (l, ξ) is disjoint with (l, ϕ) and inductive, i.e., each state with location l reachable from (l, ξ) is also in (l, ξ). We talk just about an invariant candidate if l is clear from the context. The procedure *FoldLoop* maintains a *workbag* of triples (ψ, ξ, k), where ξ is an invariant candidate at l, ψ is the latest extension of ξ (i.e., the last set of states added to ξ), and k is the remaining number of extensions of this candidate that we allow to try. Initially, we set k to κ_l, so every candidate is extended maximally κ_l times.

First, we ask the procedure *InitialInvariantCandidate* for an initial invariant candidate ψ at l. Then we call the procedure *Overapproximate* that returns a set of overapproximated candidates. That is, each ψ' returned from *Overapproximate* is again an invariant candidate and (l, ψ') is a superset of (l, ψ). Then we put the triples (ψ', ψ', κ_l) to *workbag* and remember ψ' also in O_l for possible future attempts of folding this loop.

In every iteration of the main cycle, a triple (ψ, ξ, k) is picked from *workbag*. Then we check whether the corresponding candidate ξ is an invariant. As the candidate is inductive, it is sufficient to check that ξ holds whenever we enter the loop header l from outside of the loop. Hence, we consider all edges $e = (l', o, l)$ that enter the loop from outside and call $\mathit{BSELF}(l', e^{-1}(\neg\xi), \mathit{true})$ to detect if ξ always holds when entering l. If the answer is *correct* for all considered edges, then no state in $(l, \neg\xi)$ is reachable from $(\mathit{init}, \mathit{true})$ and we found an invariant (l, ξ) disjoint with (l, ϕ).

Otherwise, if $k > 0$ then we try to extend the candidate by new states. Specifically, we take every loop path $\pi \in \mathit{LoopPaths}(l)$ and compute the precondition $\psi' = \pi^{-1}(\psi)$ with respect to ψ (the previous extension of the candidate). Note that the set $(l, \psi' \vee \psi)$ is again inductive as (l, ψ) is inductive and all executions of the program from (l, ψ') must end up in $(l, \psi) \subseteq (l, \psi \vee \psi')$. If ψ' is disjoint with ϕ, then ψ' is also an invariant candidate. We put the triples corresponding to overapproximations of this candidate to *workbag* and update the known candidates in the O_l set. Intuitively, the described process of extending a candidate corresponds to computing the set of states in which the program is one iteration before getting into ψ along $\pi \in \mathit{LoopPaths}(l)$ and overapproximating this set to cover not just one, but possibly multiple previous iterations of the loop (along any path).

The main cycle is repeated until either an invariant is found or *workbag* gets empty. Even if we fail to find an invariant in a particular call to *FoldLoop*, it is possible that we find one when BSELF reaches l again. This is because the procedure *InitialInvariantCandidate* (which is described later in detail) not only reuses candidates stored in O_l to recycle the work, but it can even merge several candidates originally computed for different paths through the loop and from different attempts of folding the loop with the header l.

To make the description of loop folding complete, it remains to describe the procedures *InitialInvariantCandidate* and *Overapproximate*.

4.2 The computation of the initial invariant candidate

The procedure *InitialInvariantCandidate* computes the initial invariant candidate during loop folding. It gets the current error states (l, ϕ) and the set *visited* of loop headers where the loop folding failed during the exploration of the current path, and produces a formula ψ such that the set (l, ψ) is disjoint with (l, ϕ) and inductive.

Let Π_e be the set of safe paths starting in l that exit the loop without finishing a single iteration. Formally, Π_e contains the paths $\pi_e = (l_0, o_0, l_1)(l_1, o_1, l_2) \dots (l_{n-1}, o_{n-1}, l_n)$ such that $l_0 = l$, $l_1, \dots, l_{n-1} \in \mathit{Locs}(l) \smallsetminus \{l\}$, and $l_n \notin \mathit{Locs}(l) \cup \{\mathit{err}\}$. Further, we set

$$\psi_e = \neg\phi \wedge \bigvee_{\pi_e \in \Pi_e} {\pi_e}^{-1}(\mathit{true}).$$

Note that ψ_e is an invariant candidate as it is disjoint with ϕ and inductive because it enforces that the loop is left without finishing any other iteration (and we cannot reach the loop again as the program has no nested loops).

The procedure *InitialInvariantCandidate* works as follows. If $l \notin \mathit{visited}$, then BSELF tries to fold this loop for the first time during exploration of the current path. In this case, ψ_e seems to be a reasonable invariant candidate. However, we would like to recycle the work from previous loop foldings on l (executed on different paths), so we extend ψ_e with all possible candidates stored in

O_l that are disjoint with ϕ and subsume ψ_e. Hence, the procedure returns the formula

$$\psi_1 = \psi_e \vee \bigvee_{\substack{\xi \in O_l \\ \psi_e \implies \xi \\ \neg sat(\xi \wedge \phi)}} \xi.$$

If $l \in$ *visited*, then BSELF previously failed to fold this loop during exploration of the current path. In this case, we combine the candidates stored in O_l. More precisely, we define formulas

$$\psi_2 = \bigvee_{\substack{\xi \in O_l \\ \psi_e \implies \xi \\ \neg sat(\xi \wedge \phi)}} \xi \qquad \text{and} \qquad \psi_3 = \bigvee_{\substack{\xi \in O_l \\ \neg sat(\xi \wedge \phi)}} \xi$$

where ψ_2 is a formula that joins all candidates stored in O_l that are disjoint with ϕ and subsume ψ_e. If $sat(\psi_2) = true$ (i.e., we found some suitable candidates in O_l) we return it. Otherwise, we return ψ_3 which gives up on subsumption and just gathers all the candidates stored in O_l that are disjoint with ψ. Note that there may be no such candidates and therefore ψ_3 can be just *false*.

4.3 Overapproximation of an inductive set

The procedure $\mathit{Overapproximate}(l, \psi', \psi, \xi, \phi)$ gets the current error states (l, ϕ), an invariant candidate ξ together with its last extension ψ, and the newly suggested extension ψ'. The procedure produces a set of extensions ψ'' that are overapproximations of ψ' (i.e., $\psi' \implies \psi''$) and they are valid extensions of ξ. Formula ψ'' is a *valid extension* of ξ if the following two conditions hold.

1. $(l, \psi'' \vee \xi)$ is disjoint with (l, ϕ). As (l, ξ) and (l, ϕ) are always disjoint, the condition holds if and only if $\psi'' \wedge \phi$ is unsatisfiable.
2. $(l, \psi'' \vee \xi)$ is inductive. As (l, ξ) is inductive, it is sufficient to check that after one loop iteration starting from (l, ψ'') we end up in $(l, \psi'' \vee \xi)$. This condition holds if and only if

$$\bigvee_{\pi \in LoopPaths(l)} \big(\psi'' \wedge \pi^{-1}(\neg(\psi'' \vee \xi))\big)$$

is unsatisfiable.

Note that Algorithm 4 ensures that the value of ψ' is always a valid extension of ξ.

Our overapproximation procedure works in several steps. In the first step, we collect relations that are implied by ψ' (sometimes together with ψ). Specifically, we derive these kinds of relations:

Type 1 Equalities of the form $x = c$, where x is a program variable and c is a constant.

Type 2 Linear equalities of the form $x \pm y = a$ or $x \pm y = a \cdot z$, where x, y, z are program variables and a is a constant.

Type 3 Relations $a \leq x \leq b \;\wedge\; x \equiv 0 \pmod{b-a}$ for a program variable x and constants $a < b$ such that either $\psi' \implies x = a$ and $\psi \implies x = b$, or $\psi \implies x = a$ and $\psi' \implies x = b$.

Type 4 A formula μ' created from a sub-formula μ of ψ' by the substitution of x with y (or vice versa), where x, y are program variables or constants such that $\psi' \implies x = y$ and $\mu \neq \mu'$.

To collect these relations, we use satisfiability queries. For example, to check whether ψ' implies the relation $x - y = a \cdot z$ for some a, we first check the satisfiability of $\psi' \wedge (x - y = A \cdot z)$ where A is an uninterpreted constant. If the answer is positive, we get a model that assigns some value a to A. Now we check the satisfiability of $\psi' \wedge (x - y = A \cdot z) \wedge A \neq a$. If it is unsatisfiable, then ψ' implies $x - y = a \cdot z$.

In the second step, we create a formula ρ by conjoining the relations of type 1 to ψ' and transforming this new formula to CNF. Note that ρ is equivalent to ψ'. The rest of the overapproximation procedure tries to overapproximate $\rho \wedge R$ for every relation R of type 2–4, yielding potentially many valid extensions of ψ'. To reduce the number of considered relations, we use only those that are not implied by any other relation. Additionally, we try also $R = \mathit{true}$ which leads to overapproximating plain ρ.

Given a relation R, we try to drop clauses of ρ while keeping $\rho \wedge R$ a valid extension. Note that at the beginning, $\rho \wedge R$ is again equivalent to ρ. Let us choose a clause c in ρ and let $\delta = \rho_{-c} \wedge R$, where ρ_{-c} denotes the formula ρ without the clause c. Note that δ is an overapproximation of ρ. If δ is also a valid extension of ξ, we replace ρ with ρ_{-c}. Otherwise, we keep clause c in ρ. We repeat this process until no clause can be dropped. Finally, let ρ' be the formula $\rho \wedge R$.

The fourth step tries to relax the inequalities in ρ'. It tries to replace each inequality $e_1 \leq e_2$ (resp. $e_1 < e_2$) in ρ' with $e_1 \leq e_2 + r$ (resp. $e_1 < e_2 + r$) where r is a constant as large as possible such that the modified ρ' is a valid extension. We search this constant r using the bisection method. If we find such an $r \neq 0$, we must also check that the modified formula is an overapproximation of ψ'. Note that it does not have to be the case, for example, due to integer overflows. If the modified ρ' is an overapproximation, we keep it and continue with the next inequality. A crucial point is to apply this step also to equalities by taking each equality clause $e_1 = e_2$ as $(e_1 \leq e_2) \wedge (e_1 \geq e_2)$.

The last step is similar to the third one: we drop clauses from the current ρ' as long as the formula is a valid extension of ξ. In contrast to the third step, now we try to drop also clauses that were originally in R and thus were not dropped in the third step. The resulting formula has to be a valid extension of ξ and an overapproximation of ψ' by construction.

Similarly, as with filtering relations, we now filter the computed extensions and return only those that are not implied by any other extension.

Note that the result of overapproximating steps are sensitive to the order in which the clauses are processed and to the order in which inequalities are relaxed.

4.4 Optimizations

In our implementation, we use also two optimizations of BSELF. The first optimization is that when we try to fold a loop for the first time, we continue BSE until we unwind the whole loop once along every loop path before we start the actual folding. If the current error becomes infeasible during unwinding, we directly return *true*. This way we avoid loop folding of loops that are easily verifiable by pure BSE.

The second optimization is that we annotate loop headers with generated loop invariants which are then used in BSE. This has no effect on the algorithm as the invariants are always stored also in O_l sets – should the invariant make a path infeasible during BSE, it will get to the initial inductive candidate during loop folding and is discovered again. However, there is the overhead of overapproximating and checking the invariance again.

```
 1  int x; // input
 2  int i = 0;
 3  assume(x == 1);
 4  while (i < 1000000) {
 5      if (i == 5)
 6          --x;
 7      ++x;
 8      ++i;
 9  }
10
11  assert(x == i);
```

```
 1  int n = 1000000;
 2  int x = 0;
 3  int i = 0;
 4  while (i < n) {
 5      ++i;
 6  }
 7  while (x < n) {
 8      ++x;
 9  }
10
11  assert(x == i);
```

```
 1  int x; // input
 2  int y; // input
 3  int n; // input
 4  assume(x >= 0 &&
 5         x <= y && y < n);
 6  while (x < n) {
 7      ++x;
 8      if (x > y)
 9        ++y;
10  }
11  assert(y == n);
```

Fig. 3. Three programs verifiable with BSELF.

4.5 Examples

In this subsection, we give examples of running BSELF on the program from Figure 2 and three programs in Figure 3 that all trigger a different behavior of BSELF.

Program in Figure 2. In this program, BSELF first hits the loop with the error states $(1, \phi) = (1, x \neq i \wedge i \geq n)$. There are no stored invariant candidates, so the initial invariant candidate is inferred as $\psi_e = ((x = i \vee i < n) \wedge i \geq n)$. It is simplified and overapproximated to $x = i$, which is directly identified as an invariant.

Figure 3 (left). In this program, BSELF computes the initial invariant candidate and overapproximates it to $\xi_0 = (1000000 \leq i \wedge x = i)$. It is not an invariant, so BSELF tries to extend it. Although the loop has two paths, the only possible pre-image of ξ_0 is $x = i \wedge i = 999999$. The later equality is relaxed to $999999 \leq 999993 + i$ which simplifies to $6 \leq i$ and ξ_0 is extended with $\psi_1'' = (6 \leq i \wedge x = i)$ to $\xi_1 = ((1000000 \leq i \wedge x = i) \vee (6 \leq i \wedge x = i))$. This is still not an invariant, but the extension of ξ_1, which is computed as an overapproximation of the pre-image of ψ_1'', is $(i = x - 1 \wedge i \leq 5)$ which together with ξ_1 forms an invariant.

Figure 3 (middle). This program shows that BSELF is not constrained to one loop only. Let us call the first loop $L4$ and the other loop $L7$ and set $\kappa_{L4} = \kappa_{L7} = 1$. The loop folding at $L7$ starts with the candidate $i = x \wedge x \geq n$. The nested call of BSELF to check whether this candidate is invariant leads to folding the loop $L4$ with the initial inductive candidate $i \geq n \wedge i = x$. This folding fails after one extension (because the limit on the number of extensions $\kappa_{L4} = 1$ is hit), the nested instance of BSELF terminates and the top-level instance of BSELF continues extending the candidate at $L7$ to $(i = x \wedge x \geq n) \vee (n = i \wedge x < n)$. This set of states is again checked for invariance, leading to folding the loop $L4$ which succeeds after 1 extension with the invariant $(i \geq n \wedge (x = i \vee n = i) \wedge (n = i \vee x \geq n) \wedge (x < n \vee x = i)) \vee (i < n \wedge (x < n \vee x = 1 + i))$ that in turn proves that $(i = x \wedge x \geq n) \vee (n = i \wedge x < n)$ is invariant at $L7$.

Figure 3 (right). Assume that $\kappa_{L6} = 1$. BSELF starts folding $L6$ with the error set $y \neq n \wedge x < n$ from which it derives the initial candidate $\xi_0 = (y = n)$. Two extended candidates are generated, namely $\xi_1 = (y = n \vee (n > x \wedge n - 1 \leq y \wedge y \leq n))$ and $\xi_2 = (y = n \vee (y = x \wedge y \leq n))$. Neither of these candidates is an invariant and we hit the limit on the number of extensions, therefore the folding fails. However, ξ_1 and ξ_2 were stored into O_{L6}. BSELF continues unwinding the loop, hitting its header two more times on different paths. In both cases, the initial invariant candidate is drawn from O_{L6} and it is $\psi = \xi_1 \vee \xi_2$ as both these sets are disjunctive with the new error states. In one case ψ is overapproximated to $y = n \vee n \leq y$ and in the other it is overapproximated to $n - 1 \leq y \vee y = x \vee 1 + x \leq y \vee y = n$. The overapproximations are different because they were done with respect to different error states. However, both are identified as invariants and BSELF terminates.

5 Experimental Evaluation

We have implemented BSE and BSELF[1] in the symbolic executor SlowBeast [1]. SlowBeast is written in Python and uses Z3 [65] as the SMT solver. It takes LLVM [59] bitcode as input.

As BSELF aims to improve BSE on programs with loops, our evaluation uses the benchmarks of the category *ReachSafety-Loops* from the *Competition on Software Verification (SV-COMP)* 2021 [10][2]. Every benchmark is a sequential C program with explicitly marked error locations. The category contains 770 benchmarks out of which 536 are safe and 234 are unsafe.

In experiments with BSELF, BSE, and SE, we compile each benchmark with Clang to LLVM, inline procedure calls, and flatten nested loops. Even after this preprocessing, some of the benchmarks do not meet the assumptions of BSELF, which is designed primarily for integer programs and does not support the reading of input inside loops. In such cases, loop folding may fail and BSELF falls back to performing BSE. In experiments with BSELF, we set the parameter κ_l to $2 \cdot |\mathit{LoopPaths}(l)| - 1$ for each loop header l.

We first compare BSELF against BSE and then we compare both these techniques to state-of-the-art verification tools. All experiments were conducted on machines with *AMD EPYC* CPU with the frequency 3.1 GHz. For each tool, the run on a benchmark was constrained to 1 core and 8 GB of RAM and 900 s of CPU time. We used the utility Benchexec [17] to enforce resources isolation and to measure their usage.

5.1 Comparison of BSELF and BSE

First, we turn our attention to the comparison of BSELF and BSE. The scatter plot in Figure 4 (left) shows the running time of BSE and BSELF on all benchmarks that were decided by at least one of the algorithms. We can see that BSELF can decide many benchmarks that are out of the scope of BSE (green crosses on the top). Not surprisingly, there are also benchmarks where BSE beats BSELF as computing invariants has non-negligible overhead (red crosses and black crosses under the diagonal line). The quantile plot on the right shows that BSELF performs better on the considered benchmark set than BSE.

The observation from the plots is confirmed by the total numbers of decided benchmarks in Table 1. BSELF was able to solve 65 more safe benchmarks. On unsafe instances, BSE performs better which is expected as BSELF focuses on proving the correctness rather than on finding bugs.

[1] The artifact with implementation and experiments infrastructure can be found at https://doi.org/10.5281/zenodo.5220293.

[2] https://github.com/sosy-lab/sv-benchmarks, commit `3d1593c`

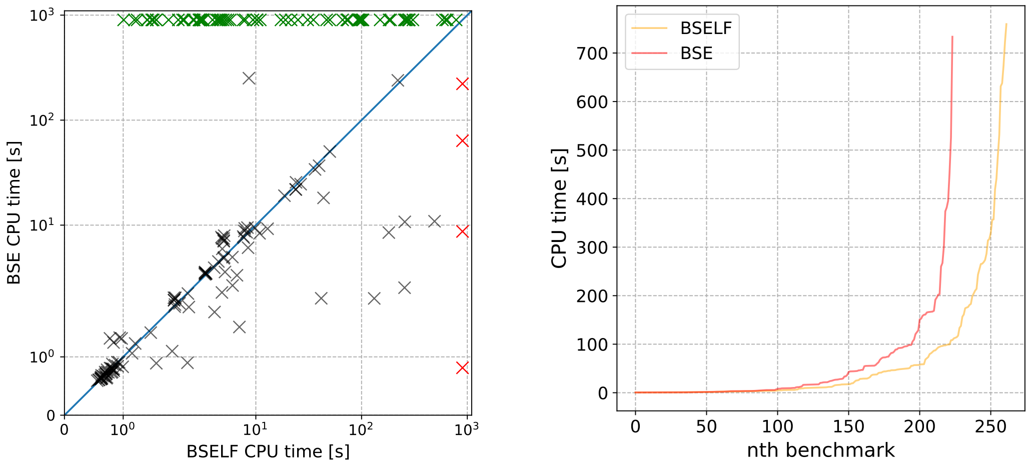


Fig. 4. The left plot provides comparison of running times of BSELF and BSE on benchmarks solved by at least one of them. Green crosses represent benchmarks decided only by BSELF, red crosses are benchmarks decided only by BSE, and black crosses are benchmarks decided by both algorithms. The right plot shows how many benchmarks each algorithm decides with the timeout set to the value on y-axis.

5.2 Comparison of BSELF to state-of-the-art tools

Now we compare BSELF to state-of-the-art tools that can use k-induction and to tools that performed well in the *ReachSafety-Loops* category in SV-COMP 2021. The first set of tools is formed by CPAchecker [15] and ESBMC [35, 37]. These tools combine *bounded model checking (BMC)* with k-induction, where the induction parameter is the number of iterations of loops instead of the length of paths. To solve the incompleteness problem of k-induction, both tools use invariants [12, 37]. We used the configuration `-kInduction-kipdrdfInvariants` of CPAchecker (referred to as CPA-kind) that employs external invariants from interval analysis (continuously generated in parallel to k-induction) in combination with invariants inferred from counter-examples to k-induction with a PDR-like procedure [12, 11]. This configuration performed the best among the configurations of CPAchecker that we tried. ESBMC was run with the `-s kinduction` option (referred to as ESBMC-kind). In this setup, ESBMC computes invariants using interval analysis, injects them as annotations into the program, and then runs BMC with k-induction [37].

Tools that performed well in SV-COMP 2021 are Divine [8], UAutomizer [49], VeriAbs [2], and another configuration of CPAchecker called CPA-seq. We call these tools collectively as *sv-comp tools*. Divine is a control-explicit data-symbolic model-checker. UAutomizer models programs as automata and reduces the verification problem to deciding a language emptiness (internally implemented using interpolation-based predicate analysis with CEGAR). VeriAbs is a software verifier that uses a portfolio of techniques selected heuristically according to a given program. One of the techniques is also BMC with k-induction. CPA-seq combines several approaches including value analysis and predicate abstraction. All these tools were run in their settings for SV-COMP 2021. We also created a configuration SE+BSELF where we run SE for 450 seconds and for the remaining 450 seconds we run BSELF (if SE did not decide the result).

Table 1. The total number of benchmarks and non-trivial benchmarks solved by the tools. The tools are divided into three groups: tools that use k-induction, other unbounded tools, and bounded tools. The column *safe* (*unsafe*) reports the number of solved benchmarks with an unreachable (reachable, respectively) error location. The column *wrong* shows the number of wrong decisions by the tool.

	All (770)			Non-trivial (299)		
	safe	unsafe	wrong	safe	unsafe	wrong
BSE	144	80	0	64	1	0
BSELF	209	53	0	104	0	0
CPA-kipdr	245	125	2	58	0	2
ESBMC-kind	284	161	3	42	9	3
SE+BSELF	373	161	0	102	0	0
CPA-seq	318	151	2	72	4	2
Divine	316	152	2	63	10	2
UAutomizer	255	126	0	126	5	0
VeriAbs	412	199	0	136	32	0
CPA-BMC	255	142	2	0	0	2
SE	273	142	0	0	0	0

Finally, we ran BMC and SE on all benchmarks with the purpose to tell apart benchmarks that are easy to handle by simple state space enumeration. Benchmarks that can be easily decided neither with BMC (we used BMC implementation from the tool CPAchecker) nor with SE (we used SE from SlowBeast) are further dubbed as *non-trivial*. Out of the 770 benchmarks, 299 benchmarks were non-trivial.

Table 1 shows that all approaches but BSE outperform BSELF when compared on all benchmarks. However, this superiority is mostly caused by the ability to decide easy tasks by entirely unwinding loops. The configuration SE+BSELF that runs SE before BSELF shows that it is the case. If we compare the tools on non-trivial benchmarks, BSELF is able to solve more benchmarks than the other k-induction-based tools and is surpassed only by UAutomizer and VeriAbs in the comparison of all tools. SE+BSELF is highly competitive with sv-comp tools. Indeed, the only tool that performs better is VeriAbs, which is not that surprising as it selects a suitable verification technique (including BMC with k-induction) for each program.

Table 2 provides the cross-comparison of individual tools on non-trivial benchmarks by any of the approaches. Among k-induction-based tools, BSELF dominates CPA-kind and ESBMC-kind in these numbers, which suggests that loop folding is a stronger invariant generation technique than those used by these tools.

6 Related Work

Related work on symbolic execution was discussed in Section 1. *Backward symbolic execution* [23], or *symbolic backward execution* [7] has been paid less attention in the area of automatic code verification than its forward counterpart. Its roots can be tracked to backward symbolic analysis of

Table 2. Cross-comparison on non-trivial benchmarks. Numbers in rows show how many benchmarks the tool in the row decided and the tool in the column did not.

	BSE	BSELF	CPA-kipdr	ESBMC-kind	SE+BSELF	CPA-seq	Divine	UAutomizer	VeriAbs
BSE	-	3	28	50	4	27	49	15	31
BSELF	42	-	56	88	2	56	63	38	34
CPA-kipdr	21	10	-	46	11	8	34	4	15
ESBMC-kind	36	35	39	-	36	27	34	18	18
SE+BSELF	41	0	55	87	-	55	62	37	33
CPA-seq	38	28	26	52	29	-	41	5	20
Divine	57	32	49	56	33	38	-	28	7
UAutomizer	81	65	77	98	66	60	86	-	50
VeriAbs	134	98	125	135	99	112	102	87	-

protocols by Danthine and Bremer [26], and Holzmann et al. [51]. Chandra et al. [23] use interprocedural BSE with function summaries [77] and path pruning to find bugs in Java programs. Chen and Kim use BSE in the tool STAR [24], basically following the approach of Chandra et al., to compute the precondition of a program crash – BSE is guided by the given crash report to reproduce a bug. Arzt et al. [4] use BSE in a very similar manner. None of these works consider loop invariants.

Although BSE on its own is not very popular in automatic software verification, its principal foundation – the weakest precondition – is cherished in deductive verification [27, 9, 60, 32].

Our work was motivated by finding a synergy of symbolic execution with k-induction. The first use of k-induction is attributed to Sheeran et al. [79] who used it to model check hardware designs. Many other model checking approaches follow up on this work [18, 66, 72, 54, 45, 57]. The k-induction scheme has been transferred also to software model checking, where it is usually applied only to loops [30, 36, 12, 22].

Our technique infers loop invariants. There are plenty of works on this topic [25, 55, 19, 5, 41, 42, 21, 44, 43, 78, 28, 52, 67, 31], but a relatively few of the works target disjunctive invariants [41, 42, 43, 78, 28, 67, 82] that arise naturally in loop folding in BSELF.

Loop acceleration computes the reflexive and transitive closure of loop iterations [50] or its supersets or subsets [62, 34]. It can be used to infer or help to infer inductive invariants [50, 52, 62] or, in general, for the verification of safety properties with model checking or abstract interpretation. BSELF could benefit from accelerators to speed up BSE and loop folding.

Similar to loop acceleration is *loop summarization* [82] which deals with inferring *loop summaries* [39]. A loop summary is a relation that associates a set of output states (a post-condition) of the loop to a given set of input states (a pre-condition) of the loop [38, 39]. With loop summaries, one is able to skip the execution of loops and directly apply the loops' effect instead of unwinding them [80]. Such an application would directly help BSE(LF) in scaling on programs with loops as it removes the need to unwind/fold loops.

Inferring relations when overapproximating inductive sets in BSELF is similar to the use of predicates in *predicate abstraction* [33, 53].

7 Conclusion

In this work, we showed that performing k-induction on control-flow paths is equivalent to running backward symbolic execution (BSE) with the breadth-first search strategy. Then we introduced *loop folding*, a technique to infer disjunctive invariants during BSE that can help to solve a new class of benchmarks that were previously out of the scope of BSE. We compared BSE with loop folding (BSELF) with pure BSE, several k-induction-based tools, and also with the state-of-the art tools that performed well on the *ReachSafety-Loops* category in SV-COMP 2021. Compared to each of these tools, BSELF is able to solve benchmarks that the other tool is not, which makes it a valuable addition to the portfolio of program verification approaches.

In the future, we want to explore the possibilities of using the information from failed induction checks and try different overapproximation methods, e.g., some PDR-like procedure.

Bibliography

[1] SlowBeast (online), https://gitlab.fi.muni.cz/xchalup4/slowbeast, accessed: 15-08-2021

[2] Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P., Datar, A., Kumar, S., Venkatesh, R.: Veriabs : Verification by abstraction and test generation. In: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019. pp. 1138–1141. IEEE (2019), https://doi.org/10.1109/ASE.2019.00121

[3] Anand, S., Godefroid, P., Tillmann, N.: Demand-driven compositional symbolic execution. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. LNCS, vol. 4963, pp. 367–381. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_28

[4] Arzt, S., Rasthofer, S., Hahn, R., Bodden, E.: Using targeted symbolic execution for reducing false-positives in dataflow analysis. In: Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015. pp. 1–6. ACM (2015), https://doi.org/10.1145/2771284.2771285

[5] Awedh, M., Somenzi, F.: Automatic invariant strengthening to prove properties in bounded model checking. In: Proceedings of the 43rd Design Automation Conference, DAC 2006. pp. 1073–1076. ACM (2006), https://doi.org/10.1145/1146909.1147180

[6] Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. 51(3), 50:1–50:39 (2018), https://doi.org/10.1145/3182657

[7] Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. 51(3), 50:1–50:39 (2018), https://doi.org/10.1145/3182657

[8] Baranová, Z., Barnat, J., Kejstová, K., Kučera, T., Lauko, H., Mrázek, J., Ročkai, P., Štill, V.: Model checking of C and C++ with DIVINE 4. In: Automated Technology for Verification and Analysis - 15th International Symposium, ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer (2017), https://doi.org/10.1007/978-3-319-68167-2_14

[9] Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'05. pp. 82–87. ACM (2005), https://doi.org/10.1145/1108792.1108813

[10] Beyer, D.: Software verification: 10th comparative evaluation (SV-COMP 2021). In: Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021. LNCS, vol. 12652, pp. 401–422. Springer (2021), https://doi.org/10.1007/978-3-030-72013-1_24

[11] Beyer, D., Dangl, M.: Software verification with PDR: an implementation of the state of the art. In: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020. LNCS, vol. 12078, pp. 3–21. Springer (2020), https://doi.org/10.1007/978-3-030-45190-5_1

[12] Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Computer Aided Verification - 27th International Conference, CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer (2015), https://doi.org/10.1007/978-3-319-21690-4_42

[13] Beyer, D., Dangl, M., Wendler, P.: Combining k-induction with continuously-refined invariants. CoRR abs/1502.00096 (2015), http://arxiv.org/abs/1502.00096

[14] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. Int. J. Softw. Tools Technol. Transf. 9(5-6), 505–525 (2007), https://doi.org/10.1007/s10009-007-0044-z

[15] Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Computer Aided Verification, CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer (2011)

[16] Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016. LNCS, vol. 9952, pp. 195–211 (2016), https://doi.org/10.1007/978-3-319-47166-2_14

[17] Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. STTT 21(1), 1–29 (2019)

[18] Bjesse, P., Claessen, K.: Sat-based verification without state space traversal. In: Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000. LNCS, vol. 1954, pp. 372–389. Springer (2000), https://doi.org/10.1007/3-540-40922-X_23

[19] Bjørner, N., Browne, A., Manna, Z.: Automatic generation of invariants and intermediate assertions. Theor. Comput. Sci. 173(1), 49–87 (1997), https://doi.org/10.1016/S0304-3975(96)00191-0

[20] Boonstoppel, P., Cadar, C., Engler, D.R.: Rwset: Attacking path explosion in constraint-based test generation. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008. LNCS, vol. 4963, pp. 351–366. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_27

[21] Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects Comput. 20(4-5), 379–405 (2008), https://doi.org/10.1007/s00165-008-0080-9

[22] Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Static Analysis - 22nd International Symposium, SAS 2015. LNCS, vol. 9291, pp. 145–161. Springer (2015), https://doi.org/10.1007/978-3-662-48288-9_9

[23] Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: a powerful approach to weakest preconditions. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009. pp. 363–374. ACM (2009), https://doi.org/10.1145/1542476.1542517

[24] Chen, N., Kim, S.: STAR: stack trace based automatic crash reproduction via symbolic execution. IEEE Trans. Software Eng. 41(2), 198–220 (2015), https://doi.org/10.1109/TSE.2014.2363469

[25] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, POPL 1978. pp. 84–96. ACM Press (1978), https://doi.org/10.1145/512760.512770

[26] Danthine, A., Bremer, J.: Modelling and verification of end-to-end transport protocols. Computer Networks (1976) 2(4), 381–395 (1978), https://www.sciencedirect.com/science/article/pii/037650757890017X

[27] Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976), https://www.worldcat.org/oclc/01958445

[28] Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013. pp. 443–456. ACM (2013), https://doi.org/10.1145/2509136.2509511

[29] Dinges, P., Agha, G.A.: Targeted test input generation using symbolic-concrete backward execution. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14. pp. 31–36. ACM (2014), https://doi.org/10.1145/2642937.2642951

[30] Donaldson, A.F., Kroening, D., Rümmer, P.: Automatic analysis of DMA races using model checking and k-induction. Formal Methods in System Design 39(1), 83–113 (2011), https://doi.org/10.1007/s10703-011-0124-2

[31] Fedyukovich, G., Bodík, R.: Accelerating syntax-guided invariant synthesis. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018. LNCS, vol. 10805, pp. 251–269. Springer (2018), https://doi.org/10.1007/978-3-319-89960-2_14

[32] Filliâtre, J.: Deductive software verification. Int. J. Softw. Tools Technol. Transf. 13(5), 397–403 (2011), https://doi.org/10.1007/s10009-011-0211-0

[33] Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2002. pp. 191–202. ACM (2002), https://doi.org/10.1145/503272.503291

[34] Frohn, F.: A calculus for modular loop acceleration. In: Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020. LNCS, vol. 12078, pp. 58–76. Springer (2020), https://doi.org/10.1007/978-3-030-45190-5_4

[35] Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: ESBMC 5.0: An industrial-strength C model checker. In: 33^{rd} ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18). pp. 888–891. ACM, New York, NY, USA (2018)

[36] Gadelha, M.Y.R., Ismail, H.I., Cordeiro, L.C.: Handling loops in bounded model checking of C programs via k-induction. STTT 19(1), 97–114 (2017), https://doi.org/10.1007/s10009-015-0407-9

[37] Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics. LNCS, vol. 11429, pp. 209–213. Springer (2019), https://doi.org/10.1007/978-3-030-17502-3_15

[38] Godefroid, P.: Compositional dynamic test generation. In: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007. pp. 47–54. ACM (2007), https://doi.org/10.1145/1190216.1190226

[39] Godefroid, P., Luchaup, D.: Automatic partial loop summarization in dynamic test generation. In: Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011. pp. 23–33. ACM (2011), https://doi.org/10.1145/2001420.2001424

[40] Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010. pp. 43–56. ACM (2010), https://doi.org/10.1145/1706299.1706307

[41] Gopan, D., Reps, T.W.: Lookahead widening. In: Computer Aided Verification, 18th International Conference, CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer (2006), https://doi.org/10.1007/11817963_41

[42] Gopan, D., Reps, T.W.: Guided static analysis. In: Static Analysis, 14th International Symposium, SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer (2007), https://doi.org/10.1007/978-3-540-74061-2_22

[43] Gulwani, S., Juvekar, S.: Bound analysis using backward symbolic execution. Tech. Rep. MSR-TR-2009-156, Microsoft Research (2009)

[44] Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: Computer Aided Verification, 21st International Conference, CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer (2009), https://doi.org/10.1007/978-3-642-02658-4_48

[45] Gurfinkel, A., Ivrii, A.: K-induction without unrolling. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017. pp. 148–155. IEEE (2017), https://doi.org/10.23919/FMCAD.2017.8102253

[46] Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: Runtime Verification, 9th International Workshop, RV 2009. LNCS, vol. 5779, pp. 76–92. Springer (2009), https://doi.org/10.1007/978-3-642-04694-0_6

[47] Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010. pp. 71–82. ACM (2010), https://doi.org/10.1145/1706299.1706309

[48] Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. J. ACM 21(3), 367–375 (1974), https://doi.org/10.1145/321832.321835

[49] Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Computer Aided Verification - 25th International Conference, CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer (2013), https://doi.org/10.1007/978-3-642-39799-8_2

[50] Hojjat, H., Iosif, R., Konecný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Automated Technology for Verification and Analysis - 10th International Symposium, ATVA 2012. LNCS, vol. 7561, pp. 187–202. Springer (2012), https://doi.org/10.1007/978-3-642-33386-6_16

[51] Holzmann, G.J.: Backward symbolic execution of protocols. In: Protocol Specification, Testing and Verification IV, Proceedings of the IFIP WG6.1 Fourth International Workshop on Protocol Specification, Testing and Verification. pp. 19–30. North-Holland (1984)

[52] Jeannet, B., Schrammel, P., Sankaranarayanan, S.: Abstract acceleration of general linear loops. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14. pp. 529–540. ACM (2014), https://doi.org/10.1145/2535838.2535843

[53] Jhala, R., Podelski, A., Rybalchenko, A.: Predicate abstraction for program verification. In: Handbook of Model Checking, pp. 447–491. Springer (2018), https://doi.org/10.1007/978-3-319-10575-8_15

[54] Jovanovic, D., Dutertre, B.: Property-directed k-induction. In: 2016 Formal Methods in Computer-Aided Design, FMCAD 2016. pp. 85–92. IEEE (2016), https://doi.org/10.1109/FMCAD.2016.7886665

[55] Karr, M.: Affine relationships among variables of a program. Acta Informatica 6, 133–151 (1976), https://doi.org/10.1007/BF00268497

[56] King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976), https://doi.org/10.1145/360248.360252

[57] Krishnan, H.G.V., Vizel, Y., Ganesh, V., Gurfinkel, A.: Interpolating strong induction. In: Computer Aided Verification - 31st International Conference, CAV 2019. LNCS, vol. 11562, pp. 367–385. Springer (2019), https://doi.org/10.1007/978-3-030-25543-5_21

[58] Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12. pp. 193–204. ACM (2012), https://doi.org/10.1145/2254064.2254088

[59] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO'04. pp. 75–88. IEEE Computer Society (2004), https://doi.org/10.1109/CGO.2004.1281665

[60] Leino, K.R.M.: Efficient weakest preconditions. Inf. Process. Lett. 93(6), 281–288 (2005), https://doi.org/10.1016/j.ipl.2004.10.015

[61] Li, G., Ghosh, I.: Lazy symbolic execution through abstraction and sub-space search. In: Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013. LNCS, vol. 8244, pp. 295–310. Springer (2013), https://doi.org/10.1007/978-3-319-03077-7_20

[62] Madhukar, K., Wachter, B., Kroening, D., Lewis, M., Srivas, M.K.: Accelerating invariant generation. In: Formal Methods in Computer-Aided Design, FMCAD 2015. pp. 105–111. IEEE (2015)

[63] Majumdar, R., Sen, K.: Latest: Lazy dynamic test input generation. Tech. Rep. UCB/EECS-2007-36, EECS Department, University of California, Berkeley (2007)

[64] McMillan, K.L.: Lazy annotation for program testing and verification. In: Computer Aided Verification, 22nd International Conference, CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer (2010), https://doi.org/10.1007/978-3-642-14295-6_10

[65] de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008), https://doi.org/10.1007/978-3-540-78800-3_24, doi: 10.1007/978-3-540-78800-3_24

[66] de Moura, L.M., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification (extended abstract, category A). In: Computer Aided Verification, 15th International Conference, CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer (2003), https://doi.org/10.1007/978-3-540-45069-6_2

[67] Nguyen, T., Kapur, D., Weimer, W., Forrest, S.: Using dynamic analysis to generate disjunctive invariants. In: 36th International Conference on Software Engineering, ICSE 2014. pp. 608–619. ACM (2014), https://doi.org/10.1145/2568225.2568275

[68] Popeea, C., Chin, W.: Inferring disjunctive postconditions. In: Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference. LNCS, vol. 4435, pp. 331–345. Springer (2006), https://doi.org/10.1007/978-3-540-77505-8_26

[69] Qi, D., Nguyen, H.D.T., Roychoudhury, A.: Path exploration based on symbolic output. ACM Trans. Softw. Eng. Methodol. 22(4), 32:1–32:41 (2013), https://doi.org/10.1145/2522920.2522925

[70] Qiu, R., Yang, G., Pasareanu, C.S., Khurshid, S.: Compositional symbolic execution with memoized replay. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015. pp. 632–642. IEEE Computer Society (2015), https://doi.org/10.1109/ICSE.2015.79

[71] Rocha, W., Rocha, H., Ismail, H., Cordeiro, L.C., Fischer, B.: Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017. LNCS, vol. 10206, pp. 360–364 (2017), https://doi.org/10.1007/978-3-662-54580-5_23

[72] Roux, P., Delmas, R., Garoche, P.: SMT-AI: an abstract interpreter as oracle for k-induction. Electron. Notes Theor. Comput. Sci. 267(2), 55–68 (2010), https://doi.org/10.1016/j.entcs.2010.09.018

[73] Sankaranarayanan, S., Ivancic, F., Shlyakhter, I., Gupta, A.: Static analysis in disjunctive numerical domains. In: Static Analysis, 13th International Symposium, SAS 2006. LNCS, vol. 4134, pp. 3–17. Springer (2006), https://doi.org/10.1007/11823230_2

[74] Santelices, R.A., Harrold, M.J.: Exploiting program dependencies for scalable multiple-path symbolic execution. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010. pp. 195–206. ACM (2010), https://doi.org/10.1145/1831708.1831733

[75] Saxena, P., Poosankam, P., McCamant, S., Song, D.: Loop-extended symbolic execution on binary programs. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009. pp. 225–236. ACM (2009), https://doi.org/10.1145/1572272.1572299

[76] Sen, K., Necula, G.C., Gong, L., Choi, W.: MultiSE: multi-path symbolic execution using value summaries. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015. pp. 842–853. ACM (2015), https://doi.org/10.1145/2786805.2786830

[77] Sharir, M., Pnueli, A., et al.: Two approaches to interprocedural data flow analysis. New York University. Courant Institute of Mathematical Sciences . . . (1978)

[78] Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Computer Aided Verification - 23rd International Conference, CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer (2011), https://doi.org/10.1007/978-3-642-22110-1_57

[79] Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000. LNCS, vol. 1954, pp. 108–125. Springer (2000), https://doi.org/10.1007/3-540-40922-X_8

[80] Slaby, J., Strejček, J., Trtík, M.: Compact symbolic execution. In: Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013. LNCS, vol. 8172, pp. 193–207. Springer (2013), https://doi.org/10.1007/978-3-319-02444-8_15

[81] Wang, H., Liu, T., Guan, X., Shen, C., Zheng, Q., Yang, Z.: Dependence guided symbolic execution. IEEE Trans. Software Eng. 43(3), 252–271 (2017), https://doi.org/10.1109/TSE.2016.2584063

[82] Xie, X., Chen, B., Zou, L., Liu, Y., Le, W., Li, X.: Automatic loop summarization via path dependency analysis. IEEE Trans. Software Eng. 45(6), 537–557 (2019), https://doi.org/10.1109/TSE.2017.2788018

[83] Yi, Q., Yang, Z., Guo, S., Wang, C., Liu, J., Zhao, C.: Eliminating path redundancy via postconditioned symbolic execution. IEEE Trans. Software Eng. 44(1), 25–43 (2018), https://doi.org/10.1109/TSE.2017.2659751