Handling Heap Data Structures in Backward Symbolic Execution

Robert Husák, Jan Kofroň, and Filip Zavoral

Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
{husak, zavoral}@ksi.mff.cuni.cz, jan.kofron@d3s.mff.cuni.cz

Abstract. Backward symbolic execution (BSE), also known as weakest precondition computation, is a useful technique to determine validity of assertions in program code by transforming its semantics into boolean conditions for an SMT solver. Regrettably, the literature does not cover various challenges which arise during its implementation, especially when we want to reason about heap objects using the theory of arrays and to use the SMT solver efficiently. Our contribution is threefold. First, we summarize the two most popular state-of-the-art approaches used for BSE, denoting them as disjunct propagation and conjunction combination. Second, we present a novel method how to model heap operations in BSE using the theory of arrays, optimized for incremental checking during the analysis and handling the input heap. Third, we compare both approaches with our heap handling implementation on a set of program examples, presenting their strengths and weaknesses. The evaluation shows that the conjunction combination approach with incremental solving is the most efficient variant, exceeding straightforward implementation of disjunct propagation in an order of magnitude.

Keywords: backward symbolic execution, weakest precondition, heap data structures, input heap, theory of arrays

1 Introduction

Symbolic execution is an established technique to explore semantics of programs, create tests with high code coverage and discover bugs [2]. To achieve that, it systematically explores the state space of the program reachable from the entry point, transforming the possible execution paths into boolean constraints. These constraints are usually passed to an SMT solver to determine the reachability of the corresponding paths. To reason about objects on the heap, many of the practically-usable tools [6, 13] use the theory of arrays [10], which can be handled by the most of the state-of-the-art SMT solvers [8].

If we are not interested in the exploration of the whole program and we want to inspect only one particular problematic place instead, we can use the *backward* variant of symbolic execution, sometimes referred to also as the *weakest precondition* analysis [7, 9]. As its name suggests, backward symbolic execution starts at the assertion of our interest and traverses the execution direction backwards. If it manages to reach the entry point and find an assignment satisfying the path constraints, it can provide us with a valuable test case. Otherwise, if no under-approximation is used and the assertion violation is proved to be unreachable, it is validated.

Listing 1.1: Sample C# code to demonstrate symbolic execution

```
1  void ScalarExample(int a) {
2      int b = 1;
3      int c = 2;
4      Debug.Assert(a != b);
5  }
6
7  void HeapExample(Node a) {
8      a.next = new Node();
9      Node b = a.next;
10     Debug.Assert(a != b);
11 }
```

Although backward symbolic execution can be very useful in many scenarios [7], it is not as popular in the literature as the forward variant. Therefore, many important design considerations and potential complications have to be rediscovered during each implementation. For example, when calling an SMT solver multiple times, it is often efficient for the subsequently analysed conjunctions to share a common prefix. As we illustrate in Section 2, that complicates the way how we handle the constraints, because we then should not alter the existing ones, only add new ones. We tackle this problem and other issues in our contributions:

1. We summarize the existing algorithms commonly used for backward symbolic execution in Section 3.
2. We present a novel way how to transform heap operations into boolean constraints in Section 4. These transformations fit into the mentioned algorithms and utilize performance enhancements of the state-of-the-art SMT solvers.
3. We compare the performance of all the presented approaches on a set of code examples in Section 5.

In Section 6 are mentioned the most important papers and tools related to our work, Section 7 concludes.

2 Problem

All the complications related to implementing a backward symbolic execution tool stem from its very nature. Forward symbolic execution starts with fixed a set of symbolic input variables and all the gradually constructed constraints can be essentially build from them. With backward symbolic execution, the situation is different, as the set of input variables constantly changes according to the variables encountered along the way. Every time a variable is read, it is added to this set; every time it is assigned to, it is removed from it.

We will illustrate the approaches on a simple method `ScalarExample` in Listing 1.1. The forward variant starts with a symbolic variable *a* assigned to `a`, on lines 2 and

3 it then assigns 1 to b and 2 to c. When it comes to the assertion a != b on line 4, it interprets it using the known values and negates the expression to discover any error inputs, resulting in a simple condition $a = 1$. The backward variant starts directly at the assertion, creating a condition $a = b$, where a and b are the input variables corresponding to their symbolic variables a and b, respectively. As the condition is not dependent on c in any way, it can safely skip its assignment on line 3. Next, the assignment of 1 into b on line 2 must effectively remove it from the set of input variables and replace it by 1 in the condition, resulting again in $a = 1$. Although this simple example does not demonstrate any significant differences, things become more complicated when heap operations and various efficiency optimizations are involved:

Constantly changing input heap: The rule with the constantly changing set of input variables applies to the input heap as well. Moreover, its analysis gets more complicated, because the objects from the input heap might get intertwined with the ones created during the analysed program run. Consider the assertion a != b on the line 10 in Listing 1.1. At first, the objects in the input heap might be possibly referenced by both a and b. The field read on line 9 causes b to be loaded from the current input heap. However, we cannot assume that the loaded reference is from the input heap as well, because it can always be assigned an explicitly allocated object, as on line 8. Therefore, we need to provide a way to correctly distinguish between the input heap and the explicitly allocated objects and to enable their various interactions.

Incremental solving: There are many usage scenarios of SMT solvers where we need to call them successively on similar formulas. E.g. in the case of symbolic execution we might want to explore two independent code branches sharing the same prefix. Therefore, a modern SMT solver can be usually used incrementally, with the possibility to cache certain knowledge between subsequent calls. To add and remove assertions they offer two useful mechanisms: an *assertion stack* and *assumptions*. The former enables us to use a stack-based system of scopes containing the particular assertions, with the ability to destroy all the data of the topmost scope while retaining the remaining ones. The latter works by adding every assertion a in the form of $l \implies a$, where l is a literal specified later during each call of the solver. Because we want to utilize these features to optimize our SMT calls, it is important that we construct the formulas respectively, possibly combining the strengths of both techniques.

3 Backward Symbolic Execution

3.1 Notation

Let us clarify the terminology and semantics of various formulas and symbols used in this paper. If we speak about a *function* or *mapping* $g : A \to B$, it is understood as a partial function, hence defined on the subset of its domain A. If $g(a)$ for $a \in A$ is not defined, we denote it as $g(a) = \mathit{undef}$. The function $g[a \to b]$ is defined to be the same as g, except for it maps a to b. This notation can be generalized for a set: $g[\{a_1, ..., a_n\} \to b] = g[a_1 \to b]...[a_n \to b]$.

As to the formalism used for SMT queries, many-sorted first-order logic is used. Because the meaning of *sort* in logic corresponds to the meaning of *type* in computer science, we will use these two names interchangeably, according to the context. The *signature* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{P})$ comprises a set of *sorts* $\mathcal{S}$, a set of *function symbols* $\mathcal{F}$ and a set of *predicate symbols* $\mathcal{P}$. *Symbolic variables* Σ_v, *terms* Σ_t, *atoms* Σ_a, and *formulas* Σ_f are derived from the signature, using the standard recursive way. The function $\textsc{Fresh}_{\Sigma_v}$ retrieves a symbolic variable not yet used in any context. In general, $\textsc{Fresh}_A$ retrieves a variable $a \in A$, which is not yet used in the analysis.

Let C be a set of classes contained in the analysed program. For each class $c \in C$, there is a corresponding set F_c containing all its fields. All the fields in the program are contained in $F = \bigcup_{c \in C} F_c$. To enable working with reference symbolic variables, we introduce a set of reference sorts $R = \{\sigma_c \mid c \in C\} \subset \mathcal{S}$. As an instrument to reason about types of fields and variables, we use function $t : V \cup F \to \mathcal{S}$. $F_{t(v)}$ is an abbreviation of F_c where $\sigma_c = t(v)$. Reference fields F_R and value fields F_V are defined as follows:

$$F_R = \{f \in F \mid t(f) \in R\} \qquad F_V = F \setminus F_R$$

Analogicaly, reference and value variables:

$$V_R = \{v \in \Sigma_v \mid t(v) \in R\} \qquad V_V = \Sigma_v \setminus V_R$$

Note that the sorts R representing reference variables are used only to ease formal description of heap operations. They are effectively replaced by functions on arrays and integers, as we describe in Section 4. All the reference variables are expected to point to objects on the heap, there is no notion of low-level pointers and of accessing stack variables by references.

To reason about a certain program, we expect it to be given as a control flow graph (CFG). Each node n contains at most one operation n.op and each edge $e = (n_1, \psi_e, n_2)$ is marked with a condition ψ_e. The possible operations follow: scalar assignment of term $v_t \leftarrow_s t$, reference assignment $v_t \leftarrow_r v_r$, reference comparison assignment[1] $v_t \leftarrow (v_r^1 = v_r^2)$, new object creation $v_t \leftarrow \texttt{new}\ T$, field read $v_t \leftarrow v_r.f$ and field write $v_r.f \leftarrow v_v$. Note that the last two operations can occur both for reference and variable fields. Assertions are modelled as edges to special nodes.

To keep the scope limited, this work does not address handling loops, interprocedural analysis or recursion. Instead, we expect that a suitable loop underapproximation or overapproximation was used during the CFG construction and so it is acyclic. Regarding interprocedural analysis, there are two basic approaches. Backward symbolic execution is often used as a subroutine of a larger algorithm which explores the call graph and combines the results from the analysis of different procedures [7]. The other way is to use procedure inlining during the CFG creation. Although not very scalable, it can be sufficient for certain use cases.

[1] We did not put reference comparison directly in the edge conditions so that we can describe its processing later in the unified manner with the other heap operations, see Section 4.

3.2 Algorithm

Whereas in forward symbolic execution we usually want to reasonably spread our analysis among the state space to achieve high code coverage [2], backward symbolic execution often works by gathering summaries towards the entry point [7, 1]. At least in the intraprocedural case it is a natural approach, as we are interested in finding a feasible path between the entry node and the target node.

```
BSE(cfg, n_trg)
 1: var states: node → state
 2: states[n_trg] ← state representing true
 3: for all node n in cfg sorted by reverse dependency on n_trg do
 4:     var deps ← {(ψ_e, n_dep, states[n_dep]) | edge (n, ψ_e, n_dep) in cfg}
 5:     states[n] ← Merge(n, deps)
 6:     if DoSolve() ∧ Solve(GetCondition(states, n)) = UNSAT then
 7:         states[n] ← state representing false
 8: return states
```

Fig. 1: Backward symbolic execution algorithm structure

An overall algorithm structure is shown in Fig. 1. Given a target node n_{trg}, it traverses *cfg* backwards towards the entry node and gathers useful information along the way. The information is stored in the *states* associative array. In the beginning, because we expect *cfg* to be acyclic, we can sort its nodes according to their topological order in the reversed *cfg*, skipping those not reachable from n_{trg}. This way, in the main loop we are sure that the dependent nodes were already processed. For each node, we gather the states of the directly adjacent nodes and their corresponding edge conditions into *deps*. Merge is a core function responsible for inferring the state of a given node according to its dependencies. DoSolve is a heuristic returning *true* for the entry node and possibly also during the exploration so that certain infeasible parts get pruned. GetCondition is used to gather the condition corresponding to a given state, returns false if n_{trg} is unreachable from that node. Eventually the algorithm retrieves all the computed states. The caller can then extract interesting pieces of information from it, such as a possible input driving the execution towards n_{trg}.

In the literature we have identified two main possible implementations of this algorithm. The first, listed in Fig. 2, is based on formulas in DNF and their propagation in the form of disjuncts [7]. MergeDisj merges the disjunctions in all the dependent nodes and enhances them by their corresponding edge conditions, simplifying the resulting formula by Simplify and passing it to ProcessOperationDisj. Simplify applies various techniques of reducing a disjunction size while maintaining its semantics. ProcessOperationDisj handles an assignment $v_t \leftarrow_s t$ by replacing the target variable v_t by the term t representing its value, GetConditionDisj simply returns the disjunction for the given node. Heap operations and the implementation of GetCondition for heaps is described in Section 4.

state: formula in DNF
MergeDisj(n, *deps*)
1: var *merged* ← disjunction of $\{\psi_e \wedge d \mid d$ disjunct in $\varphi, (\psi_e, n_{dep}, \varphi) \in deps\}$
2: **return** ProcessOperationDisj(Simplify(*merged*), n.op)
ProcessOperationDisj(*state*, $v_t \leftarrow_s t$)
1: **return** *state*$[v_t$ / $t]$
GetConditionDisj(*states*, n)
1: **return** *states*$[n]$

Fig. 2: Backward symbolic execution implementation using disjunct propagation

state: (node condition ψ_n, *vers*: $V_V \to \mathbb{N}$, *heap*)
MergeConj(n, *deps*)
1: var *mergedVers* ← merge *vers* in *deps* to get the highest of each entry
2: (var *mergedHeap*, var *heapJoinConds*) ← MergeHeaps(*heaps* in *deps*)
3: var ψ_{join} ← disjunction of
{versioned $\psi_e \wedge c_{n_{dep}} \wedge$ JoinVers(*vers*, *mergedVers*) $\wedge$ *heapJoinCond* for *heap*
$\mid (\psi_e, n_{dep}, \psi_{n_{dep}}, vers, heap) \in deps\}$
4: (var ψ_{op}, var *finalVers*, var *finalHeap*) ← ProcessOperationConj(*mergedVers*, *mergedHeap*, *n.op*)
5: **return** $(c_n \implies \psi_{join} \wedge \psi_{op}$, *mergedVers*, *mergedHeap*)
ProcessOperationConj(*vers*, *heap*, $v_t \leftarrow_s t$)
1: **if** $vers[v_t] = undef$ **then**
2: **return** (*true*, *vers*, *heap*)
3: **else**
4: var $oldVer \leftarrow vers[v_t]$
5: var $newVers \leftarrow vers[v_t \to oldVer + 1$, {unknown variables in $t\} \to 0]$
6: **return** ($v_t^{oldVer} = t$ versioned by *newVers*, *newVers*, *heap*)
GetConditionConj(*states*, n)
1: **return** $c_n \wedge$ conjunction of $\psi_{n'}$ where n' is reachable from n

Fig. 3: Backward symbolic execution implementation using conjunction combination

In Fig. 3 we see the second possible implementation [1]. Instead of propagating a set of disjuncts to the entry node, it associates each node n with a condition ψ_n describing its semantics and control flow. As seen in GetConditionConj, to reason about a whole path we can then pass a conjunction of these conditions to an SMT solver, which enables an efficient incremental usage. Because we can reason about mutable variables, our state contains also a map *vers* containing a version number for each encountered program variable. Unlike the previous case, we need to store certain information about a symbolic heap in each state, the details will be provided in Section 4.

MergeConj works as follows. Each node n is associated with a propositional variable c_n to express that the control flow reached it. The condition ψ_n is an implication with c_n on the left side. The right side consists of two parts: a join condition ψ_{join} and an operation condition ψ_{op}. The purpose of ψ_{join} is to model the branching of the control flow by creating a disjunction on the edge conditions where each disjunct redirects the

flow to the corresponding $c_{n_{dep}}$ and possibly synchronizes the variable versions of the dependent nodes using JoinVers. An operation condition, created by ProcessOperationConj handles an assignment by making the given variable under its current version equal to the given term and associating the variable with a new version. Notice that if we did not encounter v_t before, we can safely ignore the operation. Heap operation handling is described in Section 4, including the merging of heaps.

4 Modelling Heap Using Array Theory

4.1 Main Idea

Array theory enables SMT solvers to reason about heap memory in forward symbolic execution and concolic execution [6, 13]. Its axioms, in addition to those of theory of uninterpreted functions, follow [4]:

$$\forall a, i, j (i = j \Rightarrow read(write(a, i, v), j) = v)$$
$$\forall a, i, j (i \neq j \Rightarrow read(write(a, i, v), j) = read(a, j))$$

As we can see, array theory generalises the operations of the array data structure, with the only difference being the immutability of the array variables. In the forward variant of symbolic execution, a common approach is to associate an array with each defined field and represent all the references by integers [13]. Reading a value from an instance can be then naturally modelled by using the *read* operation on the corresponding reference and array. Writing a value is similarly performed by using the *write* operation to produce a new version of the particular array. To ensure that different allocations of new objects do not reference the same object, we can use an internal counter and increment it every time an allocation is performed (allocation site counting) [3]. To denote null references, 0 is usually used.

All these principles can be directly adopted for backward symbolic execution as well [7]. However, to our knowledge there is a serious problem not sufficiently tackled in the literature. Consider the method `HeapExample` in Listing 1.1. Apart from the instance created on line 8, there is also an instance passed as the parameter `a`. Because this instance was created before the method call, we must assert that it is distinct from the former. Furthermore, all the references from `a` in the beginning of the method must point either to null or to other input heap instances. Otherwise, an SMT solver might create an invalid model where the input heap contains a reference to the explicitly created instance before it even exists.

A natural approach used in our solution is to restrict all the input heap objects to be represented as negative integers. In the case of forward symbolic execution, we can remember the first version of the variable representing each field and then constrain it whenever we access it from any reference. In our example, we start with an input reference $a \leq 0$ and an array variable $next_0$ representing the field `next` in the beginning. On line 8 we assert $next_1 = write(next_0, a, 1)$, making $next_1$ the current version of the field. Nevertheless, when we access the field on line 9, we can retrospectively add a constraint $read(next_0, a) \leq 0$ to secure the shape of the input heap. As we only add constraints and never alter the existing ones, this approach is naturally efficient for incremental solving.

When trying to using this approach in backward symbolic execution, we encounter a major problem. As the input heap might change with every operation, we cannot use any single version of the array variable representing the given field. For example, if we decide to set the input heap constraint on line 9 as $read(next_0, a) \leq 0$, we prevent `a.next` to be assigned any explicitly created instance, which exactly happens on line 8.

As we explain below, we tackle this problem by creating a helper "input" array variable for each field and firmly asserting its equality with the current field variable version only when explicitly checking the condition. A similar solution is created also for the reference variables, as they face the same issue.

4.2 Operation Definitions

The implementation of heap operation handling for the disjunction propagation algorithm from Fig. 2 is shown in Fig. 4. To mark symbolic variables corresponding to reference variables and fields, we use the s superscript. For a reference variable v, v^s represents a symbolic integer variable; for a field f, f^s represents a symbolic array variable indexed by integers. The value sort of f^s is $t(f)$ if $f \in F_V$, integer otherwise. The semantics of a reference variable v is as follows. If $v^s = 0$, v is *null*; therefore, $null^s := 0$. If $v^s > 0$, v references an object explicitly created during the analysed part of the program. Otherwise, if $v^s < 0$, v references an object in the input heap, i.e. created in the not yet analysed code.

ProcessOperationDisjHeap(*state*, *op*)

1: **switch** *op* **do**
2: **case** $v_t \leftarrow_r v_v$
3: **return** $state[v_t^s \,/\, v_r^s]$
4: **case** $v_t \leftarrow (v_1 = v_2)$
5: **return** $state[v_t \,/\, (v_1^s = v_2^s)]$
6: **case** $v_t \leftarrow$ `new` T
7: **return** $state[v_t^s \,/\, \text{Fresh}_{\mathbb{N}^+}()]$
8: **case** $v_r.f \leftarrow v_v$
9: **return** $state[f^s \,/\, write(f^s, v_r^s, \text{Symb}(v_v))] \wedge v_r^s \neq 0$
10: **case** $v_t \leftarrow_s v_r.f$
11: **return** $state[v_t \,/\, read(f^s, v_r^s)] \wedge v_r^s \neq 0$
12: **case** $v_t \leftarrow_r v_r.f$
13: **return** $state[v_t^s \,/\, read(f^s, v_r^s)] \wedge v_r^s \neq 0 \wedge read(f^{in}, v_r^s) \leq 0$

GetConditionDisjHeap(*states*, *n*)

1: var *inputRefs* ← gather reference symbolic variables in *states*[*n*]
2: **return** $states[n] \wedge \bigwedge_{f \in F_R} f^s = f^{in} \wedge \bigwedge_{v \in inputRefs} v \leq 0$

Fig. 4: Heap operation modelling in the disjunct propagation approach from Fig. 2

We can see that assignments, comparisons and new object creations are implemented as simple replacements of the corresponding target variables in the existing

formula. $\textsc{Fresh}_{\mathbb{N}_+}$ ensures that each created object is represented by a distinct number. A field write replaces all the occurrences of the given field array variable f^s by an expression that writes the given value v_v to f^s on the index given by the instance v_r. Because v_v can be either a reference variable or a scalar value (term), we use a helper function Symb which optionally adds the s superscript if $v_v \in V_R$. Because the operation would not have been executed if v_r was *null*, we also add the condition $v_r^s \neq 0$.

When reading a value from a field, we distinguish between the scalar case $\leftarrow_s$ and the reference case $\leftarrow_r$. In the scalar case, we just replace the read variable by the formula representing array read and assert that v_r is not *null*. In the reference case, we also need to handle the aforementioned problems with input heap. Therefore, for each field f we create also a helper symbolic array variable f^{in}, which is never rewritten during any operation. By adding $read(f^{in}, v_r^s) \leq 0$ we ensure that any read from the input heap using v_r will always either be null or reference an input heap object. These variables are then used in GetConditionDisjHeap, where we associate all the constraints gathered for them with their corresponding fields. We also identify all the input heap references and constrain them to be ≤ 0 as well.

As we can see from the algorithms in Fig. 2 and Fig. 4, the disjunct propagation approach is straightforward to implement and the condition transformations directly correspond to the operations. However, its efficiency heavily depends on the implementation of formula handling, especially their substitution and simplification. The best result are supposed to be obtained by a custom implementation which reflects all the requirements of the particular project [7]. It is also possible to reuse existing solutions, for example the efficient algorithms for terms in Z3 using its API [8].

Nevertheless, even with the best implementation possible the conditions in certain programs can grow beyond a reasonable complexity, where every term substitution or simplification consumes too many resources. Therefore, we will now focus on the implementation of heap operations in Fig. 5 for the conjunction combination based algorithm shown in Fig. 3. Although the semantics regarding fields as array variables and references as integer variables remains, there are several differences making the operations more complicated. Because each condition is associated with the semantics of a single node and we cannot manipulate conditions for the already processed nodes, we are not allowed to use term substitution. Instead, we utilize a version-based mechanism similar to the implementation of assignment in ProcessOperationConj, where the version of the given variable is incremented and its equality with the particular term is added to the condition.

As a result, each node is also associated with a symbolic heap (η, α). The environment η contains all the current input heap reference variables and maps each of them either to 0 or to an integer symbolic variable. In the beginning of the analysis, η contains only the mapping from *null* to 0. The field version map α associates each field $f \in F$ with a non-negative integer representing the current version of its array symbolic variable. If $\alpha[f] = i$, the variable is denoted f^i. Initially, all fields have the version 0.

Let us proceed to the semantics of ProcessOperationConjHeap. The reference assignment $v_t \leftarrow_r v_v$ distinguishes three cases. If we have not yet encountered v_t, it is not contained in η and we are not interested in any value assigned to it. Otherwise, if we

heap: (environment $\eta : V_R \to \{0\} \cup \Sigma_v$, field versions $\alpha : F \to \mathbb{N}^0\}$
ProcessOperationConjHeap($vers$, (η, α), op)

1: var $\varphi \leftarrow true, vers' \leftarrow vers, \eta' \leftarrow \eta, \alpha' \leftarrow \alpha$
2: **switch** op **do**
3: **case** $v_t \leftarrow_r v_v$
4: **if** $\eta[v_t] \neq undef$ **then**
5: **if** $\eta[v_v] = undef$ **then**
6: $\eta' \leftarrow \eta[v_v \to \eta[v_t], v_t \to undef]$
7: **else**
8: $\eta' \leftarrow \eta[v_t \to undef]$
9: $\varphi \leftarrow \eta[v_t] = \eta[v_v]$
10: **case** $v_t \leftarrow (v_1 = v_2)$
11: $\eta' \leftarrow \textsc{Init}(\eta, v_1, v_2)$
12: $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
13: $\varphi \leftarrow v_t^{vers[v_t]} = (\eta'[v_1] = \eta'[v_2])$
14: **case** $v_t \leftarrow$ `new` T
15: **if** $\eta[v_t] \neq undef$ **then**
16: $\eta' \leftarrow \eta[v_t \to undef]$
17: $\varphi \leftarrow \eta[v_t] = \textsc{Fresh}_{\mathbb{N}^+}()$
18: **case** $v_r.f \leftarrow v_v$
19: $\eta' \leftarrow \textsc{Init}(\eta, v_r, v_v)$
20: $\alpha' \leftarrow \alpha[f \to \alpha[f] + 1]$
21: $\varphi \leftarrow (f^{\alpha[f]} = write(f^{\alpha'[f]}, \eta'[v_r], \textsc{Symb}(\eta', v_v))) \wedge (\eta'[v_r] \neq 0)$
22: **case** $v_t \leftarrow_s v_r.f$
23: $\eta' \leftarrow \textsc{Init}(\eta, v_r)$
24: $vers' \leftarrow vers[v_t \to vers[v_t] + 1]$
25: $\varphi \leftarrow (v_t^{vers[v_t]} = read(f^{\alpha[f]}, \eta'[v_r]) \wedge \eta'[v_r] \neq 0)$
26: **case** $v_t \leftarrow_r v_r.f$
27: $\eta' \leftarrow \textsc{Init}(\eta, v_r)$
28: $\varphi \leftarrow (\eta'[v_r] \neq 0)$
29: **if** $\eta[v_t] \neq undef$ **then**
30: $\eta' \leftarrow \eta'[v_t \to undef]$
31: $\varphi \leftarrow \varphi \wedge (\eta[v_t] = read(f^{\alpha[f]}, \eta'[v_r]) \wedge read(f^{in}, \eta'[v_r]) \leq 0)$
32: **return** $(\varphi, vers', (\eta', \alpha'))$

GetConditionConjHeap($states$, n)

1: var $inputRefs \leftarrow$ gather symbolic variables in η of the heap in $states[n]$
2: **return** GetConditionConjHeap($states$, n) $\wedge \bigwedge_{f \in F_R} f^{\alpha[f]} = f^{in} \wedge \bigwedge_{v \in inputRefs} v \leq 0$

Fig. 5: Heap operation modelling in the conjunction combination approach from Fig. 3

do not know v_v, we associate it with the variable[2] $\eta[v_t]$. If both v_t and v_v are known, we must assert the equality of their variables. Eventually, in all the cases we must remove v_t from η, because by being assigned to it was effectively removed from the set of input heap references. When comparing two references v_1 and v_2, we use a helper function Init, which associates them in η with fresh symbolic integer variables, if they are not

[2] We expect that *null* cannot be on the left side of the assignment.

already present there. Then, the scalar assignment of boolean term $\eta[v_t] = \eta[v_v]$ to v_t is performed, updating the version of v_t in *vers* accordingly. A new object creation is again modelled only if we have encountered the target reference variable v_t before. Its symbolic integer variable $\eta[v_t]$ is asserted to be equal with a fresh positive number and v_t is removed from η. A field write $v_r.f \leftarrow v_v$ needs to manipulate α by incrementing the version of f and using its two distinct versions to express the write. Note that due to the backward approach of our analysis, the version being written to is the current one.

Again, a field read operation is the most complicated one to model. In both scalar and reference cases we use Init to ensure that there is a symbolic integer variable corresponding to v_r, constrain it not to be equal to *null* by $\eta'[v_r] \neq 0$ and use *read* to model the read of the field from the heap. In the scalar case $\leftarrow_s$ we must also handle the assignment into v_t by increasing its version in *vers*. In the reference case $\leftarrow_r$ when we are interested in the reference stored in v_t, we also use the helper f^{in} array variable enabling us to constrain the input heap later in GetConditionConjHeap. MergeHeaps uses the same version map merging as MergeConj utilizing JoinVers. To merge environments with two or more distinct values corresponding to one reference variable, it is suitable to randomly pick one of them and constraint all the others to point to it.

4.3 Example

To demonstrate the operations on a real-life example, let us examine the assertion in Fig. 6, which corresponds to inspecting the reachability of the node n_{12}. In order not to make the example too complicated by merging, it is linear, hence the particular slice of the CFG is used instead of full graph. Notice that the heap operations from the code were decomposed into the atomic ones, producing helper variables such as tv_1, tn_2 or *tnv*.

The solution using the disjunct propagation approach is depicted in Table 1. Each row captures the current state of the condition computed for it, starting from n_{12} and going backwards to n_1. To simplify the notation, we do not use the *s* superscripts to denote symbolic variables, as all the variables in the condition are symbolic. As the reachability from n_{12} to n_{12} is trivial, the condition starts with *true*. Next, to reach it from n_{11}, the condition $tv_3 > tnv$ is added and the field read is performed, replacing *tnv* with $read(val, tn_2)$ and ensuring that tn_2 is not *null*. The next read into tn_2 is a reference one; therefore, $read(next^{in}, this) \leq 0$ is added. Notice that if we called GetConditionDisjHeap at this point, the condition $this = this^{in} \wedge this \leq 0$ would be temporarily added, ensuring that the input heap consisting of *this* is separated from the objects potentially created during the analysis. The helper variable tv_3 is replaced by its semantics in n_9. After the field write in n_8, $read(write(val, this, v), this)$ is simplified to v. Similarly, in n_7, all occurrences of $read(write(next, this, n), n)$ are replaced by n. Notice that now *next* is not the part of the formula and *this* and n are already constrained not to be *null*, so the operations in n_6 and n_5 do not have any effects. In n_2 is a newly created object assigned to n, resulting in replacing all the occurrences of n by 1. Finally, the condition for n_0 enhanced with input heap handling is passed to an SMT solver, proving the assertion by returning *UNSAT*.

Table 2 shows how the conjunction combination variant works. As its name suggests, the assertions created for all the relevant nodes are combined using conjunction.

```
1  class Node {
2    public int val;
3    public Node next;
4
5    public void PrependSmaller(int v) {
6      if (v < this.val) {
7        Node n = new Node();
8        n.value = this.val;
9        n.next = this.next;
10       this.next = n;
11       this.val = v;
12       Assert(this.val <= this.next.val);
13     }
14   }
15 }
```

(a) C# code

$n_1 : tv_1 \leftarrow_s this.val$
Condition: $v < tv_1$
$n_2 : n \leftarrow_r new\ Node$
$n_3 : tv_2 \leftarrow_s this.val$
$n_4 : n.val \leftarrow_s tv_2$
$n_5 : tn_1 \leftarrow_r this.next$
$n_6 : n.next \leftarrow_r tn_1$
$n_7 : this.next \leftarrow_r n$
$n_8 : this.val \leftarrow_s v$
$n_9 : tv_3 \leftarrow_s this.val$
$n_{10} : tn_2 \leftarrow_r this.next$
$n_{11} : tnv \leftarrow_s tn_2.val$
Condition: $tv_3 > tnv$
$n_{12} : true$

(b) CFG slice

Fig. 6: Sample C# code with heap objects and the corresponding CFG slice

Table 1: The verification of the assertion in Fig. 6 using disjunct propagation

n_{12}	$true$
n_{11}	$tv_3 > read(val, tn_2) \wedge tn_2 \neq 0$
n_{10}	$tv_3 > read(val, read(next, this)) \wedge read(next, this) \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_9	$read(val, this) > read(val, read(next, this)) \wedge read(next, this) \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_8	$v > read(write(val, this, v), read(next, this)) \wedge read(next, this) \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_7, n_6, n_5	$v > read(write(val, this, v), n) \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_4	$v > read(write(write(val, n, tv_2), this, v), n) \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_3	$v > read(write(write(val, n, read(val, this)), this, v), n) \wedge n \neq 0$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_2	$v > read(write(write(val, 1, read(val, this)), this, v), 1)$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0$
n_1	$v > read(write(write(val, 1, read(val, this)), this, v), 1)$ $\wedge\ read(next^{in}, this) \leq 0 \wedge this \neq 0 \wedge v < read(val, this)$

In order to determine the reachability from n_0, we must combine all the conditions in the table. Notice that for each node n_i there exists a helper c_i to express that the control flow reached it. Otherwise, the semantics of the operations is the same as in the former case.

Table 2: The verification of the assertion in Fig. 6 using conjunction combination

n_{12}	$c_{12} \implies true$
n_{11}	$c_{11} \implies c_{12} \wedge tv_3 > tnv \wedge tnv = read(val^0, tn_2) \wedge tn_2 \neq 0$
n_{10}	$c_{10} \implies c_{11} \wedge tn_2 = read(next^0, this) \wedge this \neq 0 \wedge read(next^{in}, this) \leq 0$
n_9	$c_9 \implies c_{10} \wedge tv_3 = read(val^0, this) \wedge this \neq 0$
n_8	$c_8 \implies c_9 \wedge val^0 = write(val^1, this, v) \wedge this \neq 0$
n_7	$c_7 \implies c_8 \wedge next^0 = write(next^1, this, v) \wedge this \neq 0$
n_6	$c_6 \implies c_7 \wedge next^1 = write(next^2, n, tn_1) \wedge n \neq 0$
n_5	$c_5 \implies c_6 \wedge tn_1 = read(next^2, this) \wedge this \neq 0 \wedge read(next^{in}, this) \leq 0$
n_4	$c_4 \implies c_5 \wedge val^1 = write(val^2, n, tv_2) \wedge n \neq 0$
n_3	$c_3 \implies c_4 \wedge tv_2 = read(val^2, this) \wedge this \neq 0$
n_2	$c_2 \implies c_3 \wedge n = 1$
n_1	$c_1 \implies c_2 \wedge v < tv_1 \wedge tv_1 = read(val^2, this) \wedge this \neq 0$
n_0	$c_0 \implies c_1$

5 Evaluation

We implemented the techniques into a development version of AskTheCode, an open-source tool for backward symbolic execution of C# code. In order to compare the efficiency of the aforementioned techniques, we prepared a set of simple programs which can be parametrized so that their complexity and validity of the assertions can vary. *Degree counting*(a, b) is a simple algorithm receiving a linked list as the input. Each its node contains an additional reference to another node and the algorithm calculates for each node its in-degree: the number of nodes referencing it. The assertion fails if it encounters a node whose in-degree is greater than its zero-based index in the list and also greater than a given number a. The second parameter b specifies the number of loop unwindings, i.e. the number of nodes inspected from the start of the list. As a result, the assertion is refutable if and only if $a + 2 \leq b$. Increasing b produces a larger CFG with also potentially more complicated conditions, but the counterexample might be easier to find due to a larger number of paths corresponding to it.

The measured time of analysis of each input variant is shown in Table 3. Notice that there are multiple approaches both to the disjunction propagation approach *Disj* and to the conjunction combination one *Conj*. Because we considered implementing a custom implementation of term simplification and efficient representation too complex, we decided to use the well-optimized terms available in the API of Z3. $Disj_{Set}$ uses a set of Z3 terms to represent the disjuncts in each state. Their uniqueness is ensured by the hash consing implemented in Z3. The simplification is performed for each term

separately. On the other hand, $Disj_{Z3}$ represents each state using a Z3 term and merging is performed by creating a disjunction of all the terms in the dependent nodes. $Disj_{Comb}$ is a combination of the two approaches. A state is represented as a Z3 term set, but the merging is performed by creating a disjunction term and putting it as a single item of the set. In $Conj_{Never}$, DoSolve always returns *false*, so no intermediate calls of the SMT solver are performed. An opposite extreme is $Conj_{Always}$, where DoSolve always returns *true*. In $Conj_{Loops}$, *true* is returned only for entry nodes of loops. The underlying solver is used incrementally, which enables it to reuse the information gained during the previous checks. We conducted the experiment on a desktop with Intel Core i7 and 6 GB of RAM.

Table 3: Performance evaluation, the times are in milliseconds

Test Case	$Disj_{Set}$	$Disj_{Z3}$	$Disj_{Comb}$	$Conj_{Never}$	$Conj_{Always}$	$Conj_{Loops}$
Degree counting$(0,3)$	372	1511	1342	298	1579	272
Degree counting$(1,3)$	388	1480	1344	60	1353	27
Degree counting$(2,3)$	360	1427	1328	47	1336	20
Degree counting$(1,4)$	2228	1994	1523	96	1788	38
Degree counting$(2,4)$	2255	1942	1530	86	1785	61
Degree counting$(3,4)$	2134	1724	1503	64	1743	28
Degree counting$(2,5)$	14108	3667	2766	130	2357	80
Degree counting$(3,5)$	13908	3764	2709	137	2325	79
Degree counting$(4,5)$	13760	3586	2440	81	2248	45
Degree counting$(3,6)$	80599	6765	4693	253	3313	384
Degree counting$(4,6)$	82475	6138	6008	336	3090	12
Degree counting$(5,6)$	82316	5107	3408	110	2799	84

From the results we can see that the most efficient solution for our problem was the usage of the conjunction combination approach enhanced with incremental solving with a reasonable number of the SMT solver calls. This supports our opinion that the effort put into the implementation to enable incremental solving is justified. The fact that the disjunct propagation approach did not reach as good results is to some extent caused by our limited effort invested into it. The results may drastically change in the case of a custom well-tuned simplifier. However, writing such simplifier might be a challenging feat, whereas the utilization of the incremental solving can efficiently move the problem on the well-optimized SMT solver.

6 Related Work

The disjunct propagation approach originates from Snugglebug [1], a tool using weakest preconditions to assess the validity of assertions in Java code. Snugglebug uses the algorithm for intraprocedural analysis, utilizing a custom-made simplifier over the propagated disjuncts. For interprocedural analysis, various other methods are used, such as directed call graph construction or tabulation. The SMT solver is utilized only at the

entry point, as many infeasible paths are rejected using the simplifier. The conjunction combination approach is used in UFO [1] as the under-approximation subroutine. UFO, however, does not handle heap objects.

Microsoft Pex [13] is a tool generating unit tests for .NET programs using dynamic symbolic execution. It executes the program with concrete inputs and observes its behaviour, using the Z3 SMT solver to generate new inputs steering the execution to uncovered parts of the code. It also uses the array theory to model heap operations, but the way it works with the input heap is different from our pure symbolic approach.

KLEE [6] is a symbolic virtual machine utilizing the LLVM [11] infrastructure, used mainly for C and C++ projects. It uses array theory not only to reason about heap operations, but also about pointers, low-level memory accesses etc. This differs from our approach, because we target only higher level languages with reference semantics, without the usage of pointers.

Symbolic execution tools JBSE [5] and Java StarFinder (JSF) [12] both employ lazy initialization to reason about heap objects, which lazily enumerates all the possible shapes of the heap. They differ by the languages used for specification of the heap objects' invariants. Whereas JBSE uses custom-made HEX, JSF utilizes separation logic. Although we use a different approach for the core of the heap operations, taking heap invariants into account might help us to prune infeasible paths and save resources.

7 Conclusion

In order to tackle the problem with missing information about how to implement backward symbolic execution, we presented the two most common approaches and the problems they face. We also described a method how to implement the handling of heap objects which works in both algorithms. Our evaluation shows that it the effort invested into the efficient utilization of an SMT solver can potentially bring significant performance improvements.

Acknowledgements

This work was supported by the project PROGRESS Q48, the Czech Science Foundation project 18-17403S and the grant SVV-2017-260451.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: TACAS (2012). https://doi.org/10.1007/978-3-642-28756-5_12
2. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (CSUR) **51**(3), 50 (2018)
3. Bjørner, N.: Engineering theories with z3. In: Yang, H. (ed.) Programming Languages and Systems. pp. 4–16. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer-Verlag, Berlin, Heidelberg (2007)

5. Braione, P., Denaro, G., Pezzè, M.: Jbse: A symbolic executor for java programs with complex heap inputs. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 1018–1022. ACM (2016)
6. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008), http://dl.acm.org/citation.cfm?id=1855741.1855756
7. Chandra, S., Fink, S.J., Sridharan, M.: Snugglebug: A powerful approach to weakest preconditions. SIGPLAN Not. **44**(6), 363–374 (Jun 2009), http://doi.acm.org/10.1145/1543135.1542517
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792734.1792766
9. Dinges, P., Agha, G.: Targeted test input generation using symbolic-concrete backward execution. In: 29th IEEE/ACM International Conference on Automated Software Engineering (ASE). ACM, Västerås, Sweden (September 15-19 2014)
10. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. pp. 12–17. SMT '08/BPR '08, ACM, New York, NY, USA (2008), http://doi.acm.org/10.1145/1512464.1512468
11. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization. pp. 75–. CGO '04, IEEE Computer Society, Washington, DC, USA (2004), http://dl.acm.org/citation.cfm?id=977395.977673
12. Pham, L.H., Le, Q.L., Phan, Q.S., Sun, J., Qin, S.: Testing heap-based programs with java starfinder. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 268–269. ACM (2018)
13. Tillmann, N., De Halleux, J.: Pex: White box test generation for .net. In: Proceedings of the 2Nd International Conference on Tests and Proofs. pp. 134–153. TAP'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1792786.1792798