“Symcrete” memory model with lazy initialization and objects of symbolic sizes in KLEE

1st Sergey Morozov
HSE University
16 Soyuza Pechatnikov Street, Saint Petersburg, 190121, Russian Federation
Email: morozov.serg901@gmail.com

2nd Aleksandr Misonizhnik
IT Solutions inc.
Saint Petersburg, Russian Federation
Email: misonijnik@gmail.com

3rd Dmitry Mordvinov
Saint Petersburg State University
7/9 Universitetskaya Emb., Saint Petersburg, 199034, Russian Federation
Email: mordvinov.dmitry@gmail.com

4th Dmitry Koznov
Saint Petersburg State University
7/9 Universitetskaya Emb., Saint Petersburg, 199034, Russian Federation
Email: d.koznov@spbu.ru

5th Dmitry Ivanov
Huawei Technologies Co., Ltd
Saint Petersburg, Russian Federation
Email: korifey@gmail.com

***Abstract*—Dynamic symbolic execution is a well-known technique for testing applications. It introduces symbolic variables — values with no concrete value at the moment of instantiation — and uses them to systematically explore the execution paths in a program under analysis. However, not every value can be easily modelled as symbolic: for instance, some values may take values from restricted domains or have complex invariants, hard enough to model using existing logic theories, despite it is not a problem for concrete computations.**

In this paper, we propose an implementation of infrastructure for dealing with such “hard-to-be-modelled” values. We take the approach known as symcrete execution and implement its robust and scalable version in the well-known KLEE symbolic execution engine. We use this infrastructure to support the symbolic execution of LLVM programs with complex input data structures and input buffers with indeterminate sizes.

I. Introduction

Dynamic symbolic execution is a software testing technique that allows exploring execution paths in a program under analysis, generates test coverage, and finding bugs in a given source code (e.g. out of bound memory errors or signed integer overflows) [1]. This is done by marking some program variables as *symbolics*, in other words, variables with no specific value. During analysis, a symbolic engine adds logical constraints to them, which possibly restrict values in different paths. To prove the satisfiability or unsatisfiability of a set of constraints, symbolic engines widely use SMT-solvers [2], such as Z3 [3], CVC5 [4], bitwuzla [5] and many others.

Encoding a set of values with logical constraints for each symbolic variable is one of the crucial ideas in symbolic execution. This approach enables keeping several program executions as a single *execution state* at the current position in the exploration path. All possible solutions for these constraints then become the values of symbolic variables in corresponding execution states. Since solving such formulas is an NP-hard problem, the performance and completeness of the solution heavily rely on the number and size of the logical formulas passed to the SMT-solver.

However, some values in a program can be hard to model by decidable logical constraints. The problem arises from the fact that the values of a variable may belong to a restricted domain. Such domains can have implicit and complex rules to encode in a logical formula. Let us provide some examples in which the described problem appears:

- *Objects with symbolic sizes*. Program under analysis may dynamically allocate memory on the heap (e.g. with `malloc(n)` in C language or `operator new[n]` in C++). If we treat the argument passed to that function as symbolic, we will allocate an object whose size may have different values depending on the current execution path (object with *symbolic size*). Consider an example presented in Listing 1.

Listing 1. Dynamic allocation

```
int foo(int n) {
  char *s = (char *) malloc(n);
  if (n == 1) {
    s[0] = 0;
  } else if (n > 1) {
    s[1] = 10;
  }
  return 1;
}
```

If we pass a symbolic argument to that function, we will allocate an object with symbolic size at the first line. Then the allocated object will have different sizes at the distinct branches of `if`-statement. Modelling objects with symbolic size might take many computational resources. Each allocated memory object is represented as a separate entity and cannot intersect with other objects. Naive modelling of these restrictions may result in SMT solvers needing to handle $\mathcal{O}(n^2)$ constraints, where n is the number of memory objects. Such modelling can significantly impact the performance of symbolic execution.

- *External calls*. During program exploration, the symbolic execution engine may meet calls to undefined or external functions, i.e. functions with no sources provided. As the engine does not have any information about the encountered function, it cannot properly model function behaviour to continue accurate analysis: for instance, the return value of this function may take a limited number of values. Interpreting return value as a symbolic value may be too excessive to model function behaviour, and the symbolic engine is doomed to lose precision in this case.

One possible behaviour is plain modelling of all such behaviours described in the bullets above. In this case, the engine *over-approximates* program behaviour, i.e. explores more paths than there are. Therefore it degrades performance and accuracy.

Another behaviour, taken, for instance, in KLEE symbolic execution engine [1], is to fix one possible solution during analysis. When the engine meets specific code constructions, it picks up the solution for all symbolic variables involved in one. Then it restricts taken variables with values from the received concrete solution for the following exploration. For instance, the constructs described above are modelled as follows:

- *Objects with symbolic sizes*. We might avoid performance issues by choosing one exemplar of a symbolic size fitting current constraints at the moment of the allocation. For example, while executing the `malloc(n)` statement at Listing 1, KLEE would choose some concrete value of n fitting the current path constraints, say, $n = 1$. But then, branchings on n would be evaluated only within this concrete assignment, leading to missed branches. In this case, KLEE misses covering the `s[1] = 10;` statement.
- *External calls*. Calls to external or undefined functions may be modelled as actual calls to these functions. As such functions might take arguments, which were marked as symbolic variables before, the symbolic execution engine needs to find a solution for them to satisfy previously added logical constraints. Return value then will be a constant value and can not be treated as a symbolic value.

In these cases, the engine explores fewer paths than actually exist. On the one hand, it leads to performance improvements, as the engine analyses less number of possible program behaviours. On the other hand, it impairs the engine's ability to find vulnerabilities in a program under analysis, leading to a non-exhaustive search through the program inputs space. In other words, this approach *under-approximates* program behaviours.

The idea that can be applied to resolve problems discussed above is to use a well-known approach of *symcrete*[1] [6, 7] execution. This feature allows a symbolic execution engine to mark variables as symbolic, but additionally keep a concrete value (*concretization*) for it satisfying some set of logical constraints. This concretization might be given by an algorithm different from the SMT-solver. Therefore, if such algorithms maintain some invariants inside, then they will be automatically satisfied for produced models.

The described idea gives several opportunities to the KLEE execution engine, but one of the most interesting is the support of objects with indeterminate sizes. It is achieved due to the property of allocators to allocate non-intersecting objects and the property of symcretes to keep concrete values fitting current constraints. Hence, we can dynamically maintain memory layout with no significant impact on performance. The feature of objects with symbolic sizes would increase the engine's precision for detecting buffer overflows and other memory issues in LLVM programs.

Symcretes should be fully compatible with the existing features of the symbolic virtual machine, such as lazy initialization [8, 9]. This technique enables the exploration of program behaviours with complex input data structures.

In summary, the main contributions of this paper are:

1) Implementation of the infrastructure of symcrete execution in KLEE.
2) Application of this infrastructure to model objects of symbolic sizes.
3) Application of this infrastructure to improve the currently existing mechanism of lazy initialization.

II. Background

Before discussing the main ideas of this paper, let us introduce the basic concepts of symbolic execution used throughout this paper.

A. Execution and forking

Dynamic symbolic execution executes a program with *symbolic* variables, i.e. values that represent all possible concrete program inputs. During program exploration, the execution engine operates with *execution states*, which can step over 1 instruction and fork. For these states, the symbolic execution engine maintains the inner representation of programs *memory model*. Also, every execution state maintains path constraints (PC) — a set of logical formulas describing the explored path. When the execution engine meets a conditional operator, it queries the solver with constraint and its negation, and forks state if solutions for both constraints exist. If only one statement is true, it does not fork and simply proceeds the execution of a reachable path.

[1] "Symcrete" = symbolic + concrete.

Take a look at the example in Listing 1: let `n` be a symbolic parameter of the function. In the beginning, path constraints are empty, and the inner memory representation contains only one record: $n \leftarrow \lambda$. After execution state meets the line `if (n == 1) { ... }`, it queries solver about the *validity* of PC with $\lambda = 1$ and PC with $\neg(\lambda = 1)$. As they are both satisfiable, it splits the current execution state into 2 states with the same objects in memory and path constraints $PC' = PC \wedge \lambda = 1$, $PC'' = PC \wedge \neg(\lambda = 1)$ correspondingly.

B. Memory model

Objects in memory have addresses, which represent their location in the symbolic engine's address space, sizes, representing the number of allocated bytes for their content in address space, alignment, which makes restrictions on an address (for instance in source code user can call `posix_memalign` and `memalign` functions), and contents, an array of (potentially symbolic) bytes. To handle all that information, symbolic engines maintain *memory model*, which stores required information about all currently existing objects: addresses, sizes, contents, and so on.

C. Constraints Representation

Every constraint in KLEE is an *expression*. Expression is a tree, each node of those is an operation, and children are operands. Every leaf of these trees is either constant or read from a *symbolic array*. A symbolic array is an array from the SMT theory of arrays, i.e. unbounded storage of symbolic integers, supporting both load and store operations. Each store operation creates a new version of an array with a value changed by a specified index, therefore arrays can be considered immutable.

For brevity, we use the term "array" instead of "symbolic array".

D. Validity Cores

A set of constraints with a statement may be *valid*, that is, no counterexample can be found for it, and *invalid* otherwise. To check the validity of expressions, the engine queries SMT-solver with a given set of assumptions and negation of the provided statement. If SMT-solver gives a solution that satisfies the received query, then a counterexample is found and the initial statement in the assumption of constraints from the set is invalid. Otherwise, it may return a *validity core*, a subset of constraints "explaining" the validity.

For instance, consider the set of assumptions $\{\lambda < 10, \lambda > \alpha\}$ and a statement $\lambda > 10$. We would like to check the validity of a statement within the assumptions, that is, the validity of the formula

$$\forall \lambda, \alpha : \ \lambda < 10 \wedge \lambda > \alpha \implies \ \lambda > 10.$$

To show it, we might prove that the negation is unsatisfiable, i.e.

$$\exists \lambda, \alpha : \ \lambda < 10 \ \wedge \ \lambda > \alpha \ \wedge \ \neg(\lambda > 10).$$

SMT-solver would find a satisfying assignment, for example, $\{\lambda \mapsto 1, \alpha \mapsto 0\}$. It means that we have found a counterexample for the initial statement.

In contrast, if we check a statement $\lambda < 11$ with the same assumptions, we would query the satisfiability of $\{\lambda < 10, \lambda > \alpha, \neg(\lambda < 11)\}$ and receive from SMT solver the "unsatisfiable" verdict. State-of-the-art SMT solvers can compute unsatisfiable cores, a subset of conflicting statements. In this case, one unsatisfiable core is $\{\lambda < 10, \neg(\lambda < 11)\}$. It can be converted to validity core: just take assumptions from the unsatisfiable core as-is, and convert the negated statements from the unsatisfiable core to the original ones. In our example, the validity core includes the assumption $\lambda < 10$ and the statement $\lambda < 11$.

E. Optimizing solvers

As mentioned above, solving logical formulas, which have been constructed during program analysis, is the NP-hard problem. Hence, the complexity of the formulas in the query and the number of such queries becomes a bottleneck of symbolic execution. To simplify the queries to the solver, execution engines apply many optimizations for logical constraints. One way to provide such optimizations is to use optimizing solvers — solvers, that can modify, separate, construct additional logical formulas, or even resolve received queries without calling an expensive SMT-solver. Such solvers can form a chain ending with the SMT-solver.

F. Pointer resolution

Many languages, like C or C++, allow storing addresses directly into locations and dereference them. The resolution of concrete pointers is trivial, but symbolic execution engines might encounter programs with symbolic pointers. Consider the example in Listing 2.

Listing 2. Pointer resolution

```
int x = 10;
int y = 20;

void bar(int *s) {
  *s = 0;
}
```

As we do not know, at which address pointer `s` should be resolved, we must check every possible memory object, including the pointer variable itself. To handle these cases, the vanilla KLEE engine makes a pointer resolution operation: it iterates over all existing memory objects in memory and attempts to dereference given pointer into them: query the solver if a formula $ptr + idx > address \ \cap \ ptr + idx + type\_size < address + size$, with the formulas from path constraints, where ptr is a dereferencing pointer, idx is a relative offset (e.g. if we access the array by some index, `ptr[10]` in C or C++ languages), $type\_size$ is the size of the type we are trying access through, $address$ is the address of the memory object we are trying to access, $size$ is the size of that memory object. If the pointer can be dereferenced to

the chosen memory object, KLEE forks the current execution state and modifies path constraints PC' of the received state with the above constraint.

In the example in Listing 2, pointer `s` can be resolved to at least 2 existing objects: `x` or `y`. After storing operation `*s = 0;` KLEE will maintain at least 2 execution states, in which 0 is written to `x` or `y`.

G. Lazy initialization

However, pointer resolution might not be enough to model all possible execution paths in a program. Suppose, you need to test a code for a linked list presented in Listing 3.

Listing 3. Linked list

```
typedef struct Node {
  int x;
  Node *next;
} Node;

int baz(Node l) {
  l.next->x = 1;
  assert((l.x + l.next->x) % 2 == 0);
}
```

In this code snippet `struct Node` contains a pointer to the next element in the linked list, which will be a symbolic value if we pass a symbolic argument to function `baz`. Consequently, pointer resolution at the line `l.next->x = 1` will proceed for the symbolic pointer in the same manner as described above. As we do not have any other objects of type `struct Node`, this code example will only test circular linked lists at most of length 1.

The problem here arises from the fact, that analysing program does not contain explicitly initialized additional linked list nodes. We will face a similar problem if we try to analyse any recursive data structures, like Binary Search Trees, Linked Lists, and so on.

To overcome described obstacle modern symbolic engines apply a technique called *lazy initialization*. This method allows initializing additional objects in memory, if so required, to explore more program behaviours. Return to the example at Listing 3: during pointer resolution the symbolic execution engine will allocate one more additional object of type `struct Node` to model linked list with length at least 2 and fail the assertion `assert((l.x + l.next->x) % 2 == 0);` (as for circular linked list we summed two equal numbers before).

III. Design principles

During infrastructure design, we agreed on a set of principles to create a maintainable and easily extensible framework. These principles are as follows: (a) clear separation of public and private interfaces, (b) recompute only the demanded values, and (c) concretization should always exist. Let's consider them in more detail.

a) Clear separation of public and private interfaces: One of the most important requirements for symcretes architecture was to keep the symcretes public interface as simple as possible. Thus, to prevent the developers from implementing complex logic in various spots of symbolic engine code, the public interface of symcretes infrastructure should only provide methods to add a symcrete value to the execution state and to receive a current concretization for symcrete. All the internal architecture of symcretes and any processing details made by its infrastructure should not be accessible from the symbolic engine code.

b) Recompute only demanded values: Since the symcrete variable is the symbolic variable paired with the concrete value fitting some constraint set, then this concrete value may become obsolete with the addition of a new statement. As it might be difficult to receive a new model for all symcrete variables in such situations, we require recomputing concrete values only for symcretes, which affects the validity of the query.

c) Concretization should always exist: At every moment we should be able to receive an actual model for the symcretes used in the current constraint set. In other words, symcretes architecture should be similar to the "Observer" pattern, where the observable object is the solver and it should provide a possibility to subscribe to the solver updates.

IV. Implementation

We have built our implementation on top of the KLEE of version 2.3 [10].

Followed by the principles described above we have separated symcretes and internal mechanisms to handle them, which we called *concretizing solver*. In our implementation symcrete is a pair of an array and a concrete value. To make a symcrete expressions we assign a read from created array to that expression. Concretization of symcretes represented by the map from such arrays to bits storages.

To distinguish different symcretes we equipped all arrays with a new characteristic — arrays sources. These sources should reflect how the current array has been received. For instance, an array that has been made to handle the addresses of memory objects should differ from arrays, that are used to handle the content of memory objects. Also, these sources can carry useful properties for algorithms, which are used to generate values for them. We will show the application of these properties below.

The main logic for symcretes located in concretizing solver. It is one of the optimizing solvers, that can modify and handle received queries properly. In particular, concretizing solver modifies each query with constraints over symcretes: it adds equalities in form of `(Eq (Read width 0 symcrete_array), Constant)`, where `Read width offset source` is the read expression of width `width` at offset `offset` and array `source` — and passes them to the underlying solver. However, such modifications are not enough to handle symcretes.

Let us consider the following example. Suppose, we have a symcretes values x and y with concretization $x = 5, y = 10$, query with the set of assumptions $[x \leq 10, y \leq 20]$ and the statement $x \leq y$. Concretizing solver at the preprocessing stage will make additional constraints $x = 5, y = 10$, and consequently, the query will transform into a new query with the set of assumptions $[x \leq 10, y \leq 20, x = 5, y = 10]$ and statement $x \leq y$. Note, that this query is valid according to "validity logic", as to compute validity we negate the statement, which results in $x > y$. Existing concretization can not satisfy all assumptions with negated statement.

Therefore, existing concretization might add constraints, which force a given theorem to become valid, despite the original query being invalid. To solve a such problem we process a symcretes *relaxation* after receiving a valid response from the solver. Symcretes relaxation is the algorithm, that aims to recompute values for symcretes to receive an invalid response if so exists.

To implement it according to our principles, we need to find all symcretes, that have inappropriate values (See principle "Recompute only required values"). Such values can be found in the validity core, which might be received from the solver. For that purpose, we extended the interface of KLEE's solver with functions, that may return validity cores on valid responses. Since then, we can process a relaxation after receiving a valid response with current concretization.

The relaxation algorithm is provided in Algorithm 1. More detailed, the core part of the algorithm is located in the `do { ... } while(...);` loop. It firstly constructs a concretized query by adding equality constraints on symcretes (line 5) and queries the solver with this query (line 6). If the response is already invalid, the loop can be completed (lines 7-9), and all we need is to assign appropriate values to symcretes, which have lost concretizations (lines 24-30). Otherwise, we will look at the validity core from the valid response and collect all symcrete arrays, those concretizations affected validity (this is done by collecting all arrays and filtering them by predicate `isSymcrete` at line 11). After that, we check if we removed concretization, which was not removed before (lines 15-17). If so, we continue the process. Otherwise, the current validity core proves, that the initial query is valid.

In the general case, the presented process can take more than 1 iteration. This might happen as SMT-solver does not guarantee to return all unsatisfiable sets of formulas from the given query: usually, they return any set of formulas that cannot be satisfied.

Let's see that in the example. For instance, we have symcretes x and y with concretizations 0 and 1 correspondingly, and statement $[x < y]$. The concretized query will have a form of $[x < y, x = 0, y = 1]$. Then we will query the solver with the statement $x \leq 0$. According to "validity logic" query will transform to a set of formulas $[x < y, x = 0, y = 1, x > 0]$, which can not be satisfied, and we can highlight at least 3 unsatisfiable subsets: $[x = 0, x > 0]$, $[x < y, x > 0, y = 1]$ and $[x < y, x = 0, y = 1, x > 0]$. SMT-solver can return any of these. If it returns the first subset, the algorithm will remove concretization only for x, but the query will remain valid. Then on the second iteration, the SMT-solver return the second subset of formulas from the presented subsets. Consequently, the algorithm will remove concretization for y and after that find a counterexample to the initial statement, say, $x = 1, y = 2$.

After removing all outdated concretizations for symcretes we need to assign new values to them. To do that we query the registered algorithms (lines 24-26). After receiving new concretizations we check if the solution for the entire query invalidates the received statement in the assumption of the given constraint set. If still not, we admit that the query is valid (lines 28-30). This can happen when concrete values for symcrete variables received from registered algorithms cannot

Algorithm 1 Relaxation algorithm

```
1:  function RELAX(query, symcretes)
2:      relaxationProceeded ← true;
3:      removedSymcretes ← [];
4:      do
5:          concretizedQuery ← query, symcretes;
6:          resp ← SOLVER.CHECK(concretizedQuery);
7:          if RESP.ISINVALID() then
8:              break;
9:          end if
10:         relaxationProceeded ← false;
11:         validSymcretes ← RESP.VALIDITYCORE().
                             ALLARRAYS().
                             FILTER(isSymcrete);
12:         if (validSymcretes \ symcretes).ISEMPTY() then
13:             break;
14:         end if
15:         relaxationProceeded ← VALIDSYMCRETES.
                                  INTERSECT(symcretes).
                                  ISEMPTY();
16:         removedSymcretes ← REMOVEDSYMCRETES.
                               UNION(validSymcretes);
17:         symcretes ← symcretes \ validSymcretes;
18:     while relaxationProceeded;
19:
20:     if ¬relaxationProceeded then
21:         return Valid
22:     end if
23:
24:     for sym ∈ removedSymcretes do
25:         sym ← GETVALUEBYSOURCE(sym.source);
26:     end for
27:
28:     concretizedQuery ← query, symcretes
29:     resp ← SOLVER.CHECK(concretizedQuery)
30:     return RESP.VALIDITY()
31: end function
```

provide values invalidating the query.

If the statement in the assumption of a set of given constraints is provably invalid, i.e. has a counterexample, then we store concretizations of symcretes involved in that query in a *concretization manager*. The concretization manager is the structure that stores concrete values for symcretes for all encountered invalid queries. It may be accessed from the symbolic execution engine to get the current concrete value for symcrete.

If we want to add a constraint without an explicit call to a solver, then we may lose the record to the concretization manager. In this case, we need to update it manually from the code location where the constraint is added.

Summing up all implementation details and principles, in KLEE to mark a variable as symcrete we need to create a new array. For that array, we need to specify its source. For arrays with such a source, we need to provide an algorithm which will be used to generate concrete values. To access the concrete value of the symcrete variable we may query the concretization manager with the constraint set and statement we are interested in.

In the next sections, we will show how we can use symcretes to support objects of symbolic sizes and improve the existing mechanism of lazy initialization.

A. Properties of objects of symbolic size

Before discussing the implementation of objects with symbolic sizes we need to discuss some of their properties. As we said before, every object has 3 main parameters: address from enclosing address space, size, and content. The content of memory objects can be considered independently from address and size, therefore we will not take it into account in the reasoning below.

Firstly, we may suppose, that addresses of objects with symbolic size may be considered as symbolic values. The idea comes from the fact, that two allocations with different sizes at the same location in source code will likely receive different addresses.

Secondly, we may assume that the size and address of one object are dependent values, i.e. changing of object's size may affect the address in the enclosing address space.

Also, we need present several requirements for our implementation:

1) it should allow to dynamically resize objects
2) if several states maintain the same objects with different actual sizes, they must appear identically
3) it should consume as less memory, as possible

The logic behind the first requirement can be seen in the example at Listing 4.

Listing 4. Reallocation

```
char *s = malloc(n);
if (n > 1) {
  if (n > 2) {
    s[n - 1] = 2;
  }
}
```

In the assumption of `n` to be a symbolic variable, at the first line, we allocate an object with symbolic size. The most inner `if`-statement must be reachable with the object of size at least 3 addressable by pointer `s`.

The second requirement says, that states containing the same object with different concretized sizes must keep its properties: ID, alignment, allocation site, address and size expressions, and so on. This requirement arises from the fact, that all actions are done with the specified object, and its properties can not be violated or become outdated. Hence, after state forks, we must be able to use old constraints with new ones to find a solution for addresses and sizes in different branches of execution.

The last requirement states, that our implementation should use as less memory as possible. More detailed, since SMT-solvers work with variables as with numbers without any additional information, they might give huge models for objects with symbolic size. That may cause performance issues. Another problem is that the test case, that the symbolic engine will generate to report a bug, also can be huge enough. Usually, users want to receive the smallest test case to find the issue, therefore we need to take care of that requirement.

B. Implementation of objects of symbolic size

As noted above, addresses of objects with symbolic sizes may be considered symbolic. Also, in the Section I, we have already noticed, that we can use symcrete variables in this case.

To use them we added a new array source, which we called `AddressSource` and an algorithm, that will be able to generate solutions for such arrays. We introduced an `AddressGenerator` interface for that purpose. It has only one method `allocate(addressArray, size)`. All the classes implementing `AddressGenerator` should provide appropriate (e.g. non-overlapping) addresses for specified address array `addressArray` from the arguments list each time the `allocate(addressArray, size)` method is called.

We implement this interface in `AddressManager` class, which provides an additional method `allocateMemoryObject(addressArray, size)`. This class is used in both concretizing solver and the execution engine. On call to `allocate` it allocates the memory, and ceiling size to the nearest power of 2. Then it creates a new memory object, that should copy all properties of the already existing memory object, that utilizes the same array as the address array and caches created object. It is also optimized for multiple allocations. Therefore, if the solver requests a size less than at least one of the cached memory objects, then it will return it (that optimizes memory consumption). Note, that in the worst case, this manager will use $2M$ bytes of memory, there $M = 2^{\lceil \log_2 S \rceil}$ and S is the size of the biggest memory object. An approach with the powers of 2 for allocated sizes has been chosen not to change

concretizations of addresses for all other states, that use the same memory object. This is because certain states may force expressions to take concrete values (for instance, during the execution of an external call), and changing of address value for a group of states will invalidate such states.

`allocateMemoryObject(addressArray, size)` method is used to receive a memory objects created at `allocate` method. These memory objects are required to update an address space of execution state after recomputation of concretization for symcretes in its path constraints.

Since now, as we can maintain objects with symbolic addresses, we may apply symcretes to handle the model for objects with symbolic size. For that, we introduce symcretes with array source *SizeSource*. Symcretes with such source will contain values, corresponding to the size of memory objects, and therefore, their sum should be minimized (as we said in the requirements above). We extended KLEE's solver interface with a minimization algorithm, that solves an optimization problem and computes minimal possible values for a expression. This is done by the binary search on the answer for a given expression with a set of given assumptions.

One more important thing about this implementation is that address symcrete can not become the reason for symcretes recomputation. It means, that if in the algorithm at the Listing 5 we received an address symcrete as a symcrete with a non-appropriate value and did not receive the size symcrete for the same object, we will not recompute the address and size. This is done for reasons that as we are using the system's allocator, we are not able to choose the values for addresses and ourselves. Hence, if some concretization for some addresses violates constraints, then it is likely constraints on addresses were added and we can not continue analysis for that execution path (except null check, in our implementation it is checked separately). For now, we cannot handle such situations properly, but for real-world problems, it covers most of the use cases.

Listing 5. Symbolic size allocation

```
unsigned n <- symbolic;
char *s = (char *)malloc(n);
if (s < 10) {
  exit(1);
}
if (n > 100000) {
  printf("Huge!");
} else {
  printf("Small!");
}
```

Let's see an example presented in Listing 5. In this example, we dynamically allocate memory objects of size `n`. At the moment of allocation `n` might take any possible value of type `unsigned`, and we do not know the exact size of allocated objects. As we are applying a minimization strategy for objects of symbolic sizes, the minimal possible value for the size of allocated objects is 0. Hence, before first `if`-statement exact size of allocated memory object in address space of enclosing execution state will be 0, and we will have 2 known symcretes: size and address with concretizations 0 and `$(malloc(0))` (return value of call to `malloc` function), correspondingly, and $PC = [n = s_{size}]$. Condition in the first `if`-statement adds constraint on the symcrete address of allocated memory object. Since then, in the unsatisfiable core we will have 2 constraints: $[s_{address} = \$(\texttt{malloc(0)}), s_{address} < 10]$. As it contains only symcrete for address, we say that we are not able to do anything if the current model is inappropriate. To execute the next `if`-statement we need to discuss one more optimization.

It may turn out, that from the given constraints we can deduce, that the size of the objects is a huge enough number. At Listing 5 size of the allocated object in the `then` branch of second `if`-statement might take values not less than 100001. If we try to get a model for such arrays in the execution engine, we will receive problems with performance and memory consumption. To solve such problems, we extended KLEE with structure *SparseStorage* — it is a byte buffer with the specified default value. To fill it we query the solver only about bytes in the array, that were used for reads that were applied to receive a model within this query. Is allowed to greatly reduce memory usage and increase performance.

Returning to the example, both branches of second `if`-statement are reachable with our execution state. In the `then` branch we will have an object of size 100001, and in the `else` branch — an object of size 0.

The last implementation detail is related to default values of uninitialized memory objects not marked as symbolic. In the real world almost always content of memory allocation consists of undefined bytes. In the initial KLEE implementation, this problem did not receive attention and all allocations were filled with 0 by default for objects with constant content. To save that semantics, we engaged Z3-functionality of constant arrays, i.e. arrays with a default value. Therefore, we introduced an additional array source *ConstantWithSymbolicSize*. This source indicates, that the underlying objects are a constant array (not symbolic), but have symbolic size. Therefore, in translation to the solver, it should receive a Z3's constant array with a default value specified in that source.

C. Improved lazy initialization

In Section II we described previously existing implementation of the lazy initialization mechanism within our fork of KLEE. In that implementation, we were forced to add additional constraints to restrict overlappings of lazily initialized memory object with any other objects. Once we added symcretes functionality, we may apply that technique to lazy initialization. The usage scheme is quite similar to the objects of symbolic size, but for now, we have explicitly defined symbolic address. Moreover, we can also use extensions with objects of symbolic size to lazily initialize memory objects as we do not know the exact size of the object, which we are dereferencing at the moment of lazy initialization. Thus, it turns out, that to lazily initialize a memory object all we need is to create a new object with symbolic size and add an

equality constraint between the symcrete address and address, which have been used for dereferencing.

V. Evaluation

A. Experiment

For evaluation of the described features we have used the test sets from TestComp-2022 competition [11]. Our main goal was to test the proposed approach implemented on top of the KLEE (KLEE-SYM) and make a comparison with the version of KLEE extended with lazy initialization (KLEE-LI).

We have used KLEE-LI based on the KLEE of version 2.3 with Z3 of version 4.12.1 as SMT-solver [12].

We have selected 5 different test sets with over 2000 tests per each — MemSafety-Arrays (MS-A), MemSafety-Heap (MS-H), MemSafety-LinkedLists (MS-LL), ReachSafety-Arrays (RS-A) and Termination-MainHeap (T-MH). Comparison has been made by the following metrics: instruction coverage (icov), branch coverage percentage (bcov), and numbers of found vulnerabilities (errs). Coverage has been measured with gcov [13] util.

Experiments were conducted on a workstation with CPU AMD Ryzen 7 3800X 8-Core with 16 gigabytes of RAM under the control of Linux. Execution of each test was bounded with 30 seconds timeout. As Z3 may receive complex queries, its execution time also has been bounded with 5 seconds timeout to prevent memory and time issues.

B. Results

Average results for tests in each source set are presented in Table I.

TABLE I
TestComp benchmarks average results

TestSet	KLEE-LI			KLEE-SYM		
	icov	bcov	errs	icov	bcov	errs
MS-A	71.8%	57.2%	346	79.5%	67.5%	680
RS-A	57.4%	45.0%	393	69.3%	61.5%	532
T-MH	91.2%	78.8%	317	90.1%	80.9%	215
MS-H	45.2%	46.2%	51	45.2%	45.7%	52
MS-LL	33.0%	30.2%	55	33.0%	30.2%	55

We can notice significant improvements at ReachSafety-Arrays and MemSafety-Arrays for all parameters. These test cases used dynamic allocations of blocks with indeterminate sizes and therefore received much better results in contrast with KLEE-LI. In addition, the amount of found vulnerabilities also increased since it became possible to explore more paths, that had been beyond the abilities of the engine before.

Nonetheless, we did not receive full coverage of these two test sets. One of the reasons that symbolic execution is sensible to strategies of path selection: these strategies navigate the engine through the exponential branching space. For presented test sets, the problems may come from constructions of a form presented in Listing 6.

Listing 6. Allocation and cycle

```
unsigned n <- symbolic;
char *s = (char *)malloc(n);
for (int i = 0; i < n; i++) {
  s[i] = i % 256;
}
if (s[n - 1] == 255]) {
  return 0;
}
return 1;
```

Our goal is to cover the `return 0` statement. But to do that KLEE-LI should get information, that this line is reachable only if 256 is a factor of `n`. As it cannot infer such information, it will brute force all possible variants on `n` until it will be able to reach the selected line of code. For larger programs, it may take a while to reach such statements.

On the other hand, we might see a slight deterioration in the instruction coverage and the number of errors detected on the Termination-MainHeap test set. This issue is connected to the imprecision of modelling the allocated buffer's contents: while in reality the memory of allocated buffers is guaranteed to be initialized, KLEE models the newly allocated buffers as filled with some fixed concrete value.

Also, we've collected additional statistics about verdicts for the generated tests (see Table II). We've calculated the amount of generated tests for each source set (column overall), the number of execution paths that have been halted because of the inability of the old version to maintain objects of symbolic size correctly (halted), and the number of solver errors happened during program exploration, e.g. timeouts, internal errors, etc. (serrs).

TABLE II
Tests generated for TestComp benchmarks

TestSet	KLEE-LI			KLEE-SYM		
	overall	halted	serrs	overall	halted	serrs
MS-A	801	455	0	681	0	1
RS-A	649	238	18	539	0	7
T-MH	539	222	0	216	0	1
MS-H	58	7	0	52	0	0
MS-LL	55	0	0	55	0	0

This table demonstrates that our approach has reduced the number of internal errors in KLEE and increased the amount of non-halted branches. For the last two test sets, we did not receive any improvements in instruction and branch coverage (Table I). However, for the test set MemSafety-Heap number of errors, that we classified as halted, decreased to 0. For the test set MemSafety-LinkedList, we've received identical results. The low percentage of coverage for these test sets is explained by a significant number of syntactically unreachable code in tested programs.

VI. Related works

Symbolic execution with symcrete variables is an already known approach. For instance, the authors of "Deferred Con-

cretization in Symbolic Execution via Fuzzing" [7] describe a similar approach, using symcretes to better approximate external calls with fuzzer (yet another application of symcretes).

Similar to symcrete variables ideas are also used in well-known techniques of *symcretic* [14] and *concolic* [15] execution. The idea behind these methods is to combine a symbolic and concrete execution to improve performance and increase code coverage in comparison with plain symbolic execution. Unlike execution with symcrete variables, these approaches use concrete values to guide an execution, while we use symcrete variables to increase the accuracy of symbolic execution analysis.

However, the memory model can be improved without a symcrete variables approach. For instance, authors of "A bounded symbolic-size model for symbolic execution" [16] propose an approach for memory modelling, where all constraints restricting memory objects overlapping are added explicitly. To solve a problem with excessive memory consumption the authors specify a bound on size for objects with symbolic sizes. On the one hand, such a way of modelling objects with symbolic size does not require additional queries to the solver to minimize object sizes, as memory consumption becomes the responsibility of the users. On the other hand, that bound may affect the completeness of a symbolic execution engine, i.e. restrict an engine from exploring possibly reachable paths, as in some cases user will have to guess the bound to achieve higher coverage. Therefore, memory consumption will increase and performance degrade.

Another possible implementation of objects with the symbolic size is presented in the work "Symbolic-size memory allocation support for Klee" [17]. It introduces a segmented memory layout approach for KLEE symbolic execution engine. The core difference is that this work proposes a memory model, where each memory allocation lies in its memory segment. In contrast, our implementation of objects with symbolic sizes does not significantly change the memory model of vanilla KLEE, and therefore still can be considered flattened. To resolve a problem with excessive memory consumption, the authors use the same methods as described in this article: size minimization to reduce overall memory consumption and sparse structures to keep only useful data for symbolic arrays.

VII. Conclusions

Accurate modelling of specific code constructions with logical constraints might be too complicated (recall the problem with external calls). We can make under or overapproximations to at least continue analysis, but with a significant loss of precision. To get things slightly better we apply the technique of symcrete variables — symbolic variables paired with concrete values for it, fitting the current constraint set.

We have proposed our implementation of dynamically recomputed symcrete values in KLEE for LLVM-programs analysis. For that, we have also enhanced the execution engine with the validity cores. Then we have shown how to engage this feature to model objects with symbolic size. To optimize the memory consumption problem, we have implemented a size minimization algorithm for objects with symbolic size and sparse storage to store only the affected solution bytes. Also, we have improved the existing mechanism of lazy initialization by address symcretization and interpretation of initialized object size as symbolic. We've also presented an implementation of this approach on top of KLEE and showed its effectiveness on several tests of Test-Comp competition.

Symcretes infrastructure is a powerful foundation for other improvements. For instance, we may use a similar approach to approximate the behaviour of external or undefined functions with fuzzers, as described in "Deferred Concretization in Symbolic Execution via Fuzzing" [7]. The return value and function arguments, in this case, should be marked as symcretes, and calls to that function generate concrete values for symcretes.

Another interesting idea is to use a symcrete infrastructure with a type system. This might be useful if we want to test a program, which operates with polymorphic objects. Types of such objects may be considered symbolic, and therefore we have uncertainty in calls to virtual functions and sizes of underlying objects. This uncertainty can be resolved with symcretes, as it seems that we can model such behaviours with objects with symbolic sizes and calls to undefined functions.

VIII. Acknowledgments

The work is supported by the grant of RNF № 22-21-00697.

References

[1] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[2] Clark Barrett and Cesare Tinelli. "Satisfiability modulo theories". In: *Handbook of model checking*. Springer, 2018, pp. 305–343.

[3] Leonardo de Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[4] Haniel Barbosa et al. "cvc5: A versatile and industrial-strength SMT solver". In: *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I*. Springer. 2022, pp. 415–442.

[5] Aina Niemetz and Mathias Preiner. "Bitwuzla at the SMT-COMP 2020". In: *arXiv preprint arXiv:2006.01621* (2020).

[6] Corina S Păsăreanu, Neha Rungta, and Willem Visser. "Symbolic execution with mixed concrete-symbolic solving". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 34–44.

[7] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. “Deferred concretization in symbolic execution via fuzzing”. In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2019, pp. 228–238.

[8] Misonijnik A. et al. “Automated testing of LLVM programs with complex input data structures”. In: *Proceedings of ISP RAS* 34.4 (2022), pp. 49–62.

[9] Sarfraz Khurshid, Corina S Păsăreanu, and Willem Visser. “Generalized symbolic execution for model checking and testing”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2003, pp. 553–568.

[10] Cristian Cadar and Daniel Dunbar. *KLEE*. Version 2.3. 2022. URL: https://github.com/klee/klee/tree/v2.3.

[11] Dirk Beyer. “Advances in Automatic Software Testing: Test-Comp 2022.” In: *FASE*. 2022, pp. 321–335.

[12] Leonardo de Moura and Nikolaj Bjørner. *Z3 4.12.1*. Version 4.12.1. 2023. URL: https://github.com/Z3Prover/z3/releases/tag/z3-4.12.1.

[13] Brian Gough and Richard M Stallman. “An Introduction to GCC for the GNU Compilers gcc and g++”. In: *Network Theory Ltd* 258 (2004).

[14] Peter Dinges and Gul Agha. “Targeted test input generation using symbolic-concrete backward execution”. In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 31–36.

[15] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A concolic unit testing engine for C”. In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[16] David Trabish, Shachar Itzhaky, and Noam Rinetzky. “A bounded symbolic-size model for symbolic execution”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021, pp. 1190–1201.

[17] Michael Šimáček. “Symbolic-size memory allocation support for Klee”. PhD thesis. Masarykova univerzita, Fakulta informatiky, 2018.