Fine-grain Memory Object Representation in Symbolic Execution

Martin Nowack
Department of Computing
Imperial College London, UK
m.nowack@imperial.ac.uk

***Abstract*—Dynamic Symbolic Execution (DSE) has seen rising popularity as it allows to check applications for behaviours such as error patterns automatically. One of its biggest challenges is the state space explosion problem: DSE tries to evaluate all possible execution paths of an application. For every path, it needs to represent the allocated memory and its accesses. Even though different approaches have been proposed to mitigate the state space explosion problem, DSE still needs to represent a multitude of states in parallel to analyse them. If too many states are present, they cannot fit into memory, and DSE needs to terminate them prematurely or store them on disc intermediately. With a more efficient representation of allocated memory, DSE can handle more states simultaneously, improving its performance. In this work, we introduce an enhanced, fine-grain and efficient representation of memory that mimics the allocations of tested applications. We tested GNU Coreutils using three different search strategies with our implementation on top of the symbolic execution engine KLEE. We achieve a significant reduction of the memory consumption of states by up to 99.06% (mean DFS: 2%, BFS: 51%, Cov.: 49%), allowing to represent more states in memory more efficiently. The total execution time is reduced by up to 97.81% (mean DFS: 9%, BFS: 7%, Cov.:4%)—a speedup of 49x in comparison to baseline KLEE.**

***Index Terms*—symbolic execution; memory representation;**

I. Introduction

Dynamic symbolic execution (DSE), a method for analysing programs automatically, gained traction recently due to its many applications. It can be used for generating extensive test suites [1], finding bugs automatically [1], [2], [3], reverse-engineering software [4], or automating security analyses [5]. Despite its many applications, the two fundamental challenges remain: the state space explosion problem and high constraint solving costs. The former is due to a possibly infinite number of control flow paths through a tested application, which makes it difficult to validate each path. The latter arises with the constraints collected along each path. As a path becomes longer, the complexity of constraints increases so does its solving costs.

Even though different techniques have been proposed to cope with the state space explosion problem, like state merging [6] or directed exploration of the state space [7], a symbolic execution engine still needs to represent many *states* simultaneously in memory for efficiency. Each state mimics the memory representation of an application including allocated heap and stack memory at a specific moment of execution.

ASE 2019 ©2019 IEEE
Personal use of this material is permitted.
Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

```
1 char x[100] = {0, ..., 0};       // zero initialised
2 char *y = calloc(100, 1);        // zero initialised
3 int input = symbolic;            // symbolic input
4 if (input >= 100 || input < 1) {
5   x[0] = x[0] + 1;
6 } else {
7   x[input] = x[input] + 5;
8 }
```

Listing 1: Example of single-byte modifications of larger objects

The effective representation of the allocated memory of a tested application has received little attention. With symbolic input, the control flow can diverge if conditional branches depend on symbolic input values. Symbolic execution handles this by cloning the state (*branching*) to track control flows independently with constraint subsets of the input.

For example, in listing 1, the control flow at line 4 depends on the value of `input` from line 3. Therefore, the engine needs to follow the control flows independently (either line 5 or line 7). Each outcome will modify the array (`x[]`, line 1) in a different way. If, for example, the condition (line 4) is part of a loop, DSE will need to represent many slightly different copies of the state to reason about them. Handling a lot of states simultaneously can drastically increase the memory usage of a symbolic execution engine. If the maximum capacity is reached, states have to be saved or terminated prematurely [2], [4]. Copy-on-write techniques help to reduce the memory consumption, i.e. the same unmodified object (`y`, line 2) will be shared between states. If a state needs to modify an object (`x`, line 1), it creates a copy of the object and modifies the copy. However, with DSE, changes can range from a single bit to a whole object. With many states representing small changes in large objects, the overall memory consumption can be high for a symbolic execution engine that tracks changes with object granularity and unmodified memory is duplicated. In contrast, a DSE engine could track changes to memory in a very fine-grain way (e.g., byte-level), which would lead to additional computational tracking overhead. Especially, if a DSE engine needs to execute many concrete—but memory intensive—instructions.

In this paper, we introduce a layered representation of memory objects that can achieve a fine-grain but efficient tracking of changes. Beside substantial memory savings, we also show how it improves performance.

Our main contributions are:

1) A layered representation of memory which allows tracking changes efficiently (including properties like the initialisation status of each byte);
2) Different optimisations enabled by the layered representation;
3) An open-source implementation on top of the state-of-the-art symbolic execution engine KLEE [2];
4) A thorough evaluation of our contribution on GNU Coreutils, using multiple search strategies;

First, we will introduce the different requirements that a memory representation in the context of symbolic execution needs to fulfil (§II). We continue with a detailed description of our approach (§III) followed by a description of our implementation based on the state-of-the-art symbolic execution engine KLEE (§IV). We then evaluate our approach (§V) and finish with an overview of existing approaches (§VI) and our conclusion (§VII).

II. Overview

A. Background

We first recap how native applications handle memory allocations before we detail how symbolic execution mimics this behaviour of tested applications.

If we execute an application natively, it will allocate memory of a certain size. This can be either on the stack (Listing 1:1) or dynamically on the heap (e.g. `malloc()`, `calloc()`, Lst. 1:2). An allocation reserves part of free memory, which can be referred to by its address (*addr*). Using the address, the application can both read from the allocated memory and write to it arbitrarily. Multiple writes into the same memory object will update it with the newest value (e.g., `x[input] = x[input] + 5;`) [1]

Symbolic execution needs to reproduce the behaviour of an application. For that, it tracks the memory that an application has available and how it accesses it. However, with the many control flows that an application can have, symbolic execution needs to represent a large number of states simultaneously. This makes it hard to track the whole virtual address space (e.g. 4GiB for a 32bit) for all states. Instead, many state-of-the-art symbolic execution engines [2], [4] only track the allocated memory and its accesses.

Such an allocation can be represented as an array of data (A) and its associated metadata like size (n), domain (D), range (R) and also the base address of the allocation (the address of the first byte of the allocated memory):

$$A := \{v_0, v_1, ..., v_{n-1}\}$$

with size n and $\forall i, v_i \in R$, $i \in D$ and address $base$ referencing v_0.

[1]We assume a memory model that follows sequential consistency: i.e., read returns the value of a memory location that was written last to it.

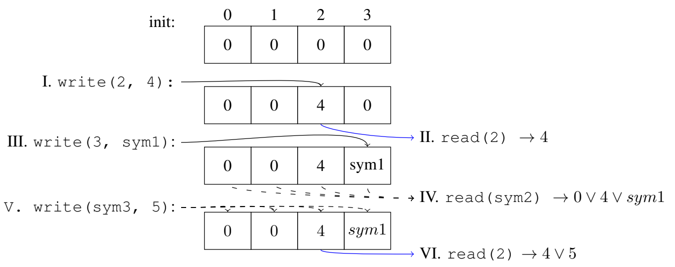


Fig. 1: Example of memory accesses of a 4 byte object. Each row reflects the memory object after a modifying operation on it.

Symbolic execution mimics how an application writes to an address (*addr*) or reads from it in two steps: first, searching for the right memory object (A) that is associated with the address (*addressing*); and second, accessing that object (*accessing*).

B. Addressing—Memory Object Lookup

For the *lookup*, symbolic execution engines search through a list of memory objects associated with the current state to find the single object that is associated with the address ($addr$). In this paper, we focus on languages that allow pointer arithmetic (like C or C++). Therefore, the address might point inside of an object and not only reference it. Hence, the object lookup will search for potential candidates by checking if the address ($addr$) is within the object's bounds with $base$ as the first byte of the object ($addr \in [base, base + width)$).

Assuming that the address refers to a correctly allocated object, to access the memory within that object, the correct index is calculated by using the actual address and subtracting the base address ($index := addr - base$).[2] If the DSE cannot resolve an address to an object, an out-of-bounds access has been detected. In case the address is a symbolic expression, DSE branches the state for each memory object the address can be resolved to.

C. Accessing—Memory Object Access

If a native application updates a memory location, it overrides the old value with the new one. Symbolic execution needs to model the same behaviour. However, in addition, it has to cope with symbolic values. That is, the $index$ or $value$ that is written or read can be symbolic.

We summarise the different access combinations in Tab. I and will discuss them now in more detail. As an example (Fig. 1), we assume the lookup of the memory object was successful, and we got a reference to a 4-byte memory object initialised with zeroes.

A concrete write with a concrete value will update this array (e.g., `write(2, 4);`). Reading from this array with a concrete index will return the latest value written (e.g, `read(2) = 4;`). If we write a symbolic value to a concrete index (e.g., `write(3, sym1);`), similarly, we fill the cell at index 3

[2]We assume that an allocation returns a non-zero-sized memory object if successful.

with the value $sym1$. Things become more complicated (and exciting) with symbolic indices. For example, if we read from a symbolic index (e.g., `read(sym2)`), the returned value depends on the actual value $sym2$ can be resolved to. In our example, assuming $sym2$ is inbounds ($0 \leq sym2 < 4$), it can be $v := 0 \vee 4 \vee sym1$. Similarly, if we write to a symbolic index, we do not know which exact location is modified (e.g., `write(sym3, 5);`), this could potentially override any location. This has an impact on all subsequent operations on this objects. For example, a read with a concrete index (`read(2)`) can return more complex expressions, e.g., `read(2) =` $4 \vee 5$`;`.

There are two main observations: First, write operations modify the current state of an object; second, read operations return a snapshot of the current state of the object.

D. The Impact of Branching

The implication for a symbolic execution engine is that it is not enough to manage memory objects with a byte-wise representation similar to the native execution of the tested application (which would be much faster). However, instead, it has to be able to memorise arbitrary changes with symbolic indices or symbolic values. For example, if a native application updates a four-byte array 100 times, the final size of the array will be four bytes. For a symbolic execution engine, this can be 4 bytes plus 100 times the representation of symbolic and concrete updates.

After a detailed look into handling memory accesses within a single state, we want to focus on how state branching intensifies the problem. While executing a state, the symbolic execution engine might reach an instruction which depends on a symbolic value. In that case, the engine might be forced to follow different subsets of the values domain. For that, it will create copies of a state, to be able to follow them independently, which is often referred to as *state branching*.

The cause of the branch is either due to symbolic data- or control-flow dependencies. Data dependencies result from a read or write operation that can resolve to multiple memory objects. In case of control-flow dependencies, it can be either conditional control flow changes with the condition depending on a symbolic expressions (e.g., Listing 1 line 4) or the control flow target can be a symbolic expression (e.g., dynamic function calls with a symbolic target: `call symbolic_value;`).

Following different control flows, symbolic execution needs to validate them first by checking the branch condition against the collected path constraints. It then follows the feasible path. Solving those queries can be computationally expensive. If multiple paths are possible, symbolic execution branches the state, which will create a copy of the state representation and add additional constraints that reflect one control flow.

TABLE I: Possible combinations of read/write accesses of a memory object with symbolic/concrete indices or values.

Method	Index Type	Value Type	Example
Read	Concrete	Conc./Sym.	value = `read(4);`
	Symbolic	Conc./Sym.	value = `read(sym1);`
Write	Concrete	Concrete	`write(3, 23);`
	Concrete	Symbolic	`write(4, sym2);`
	Symbolic	Concrete	`write(sym3, 5);`
	Symbolic	Symbolic	`write(sym4, sym5);`

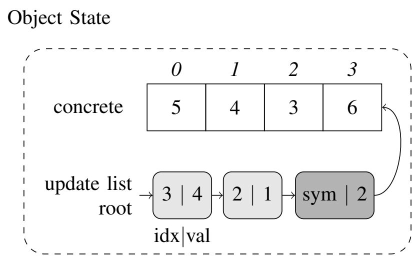


Fig. 2: KLEE's simplified representation of a memory object and its object state with an update list.

Equivalent to native execution, during the lifetime of a state, the state allocates many memory objects, and the number of memory objects in a state can be significant. To reduce performance penalties, memory objects are not copied with each branch but referenced by each state copy. If one of the states needs to update the values, the object state is copied entirely (copy-on-write) and updates applied to the state-specific copy. There is redundancy and wasted memory if the changes to an object in comparison to the object's size are small. For applications that branch heavily, these costs can be prohibitive.

E. Memory representation in the state-of-the-art symbolic execution engines

To understand how state-of-the-art symbolic execution engines handle memory, we analysed KLEE [2], an open-source symbolic execution engine for C and C++.[3] We will summarise in this section how KLEE handles memory before we introduce our approach (§III).

KLEE uses *states* that represent an execution path with all its allocated memory objects and path constraints. An allocation is handled as a byte-sized array by two entities: a *memory object* and an *object state*. The memory objects contain metadata like the address and the width of the allocated memory. The object state contains the actual content of the memory.

If a tested application accesses reads or writes from an address, KLEE looks up the memory object and the associated object state (object state, Fig. 2) with it. The simple case is when the index is concrete and the memory object has not been modified using a symbolic index before. In this case, a write can modify the object state in-place. And in the case that a memory object is shared between multiple states, a copy of the object state is created and modified. For read operations, the specific value of the object state is used. In the case an object was modified with a symbolic index, KLEE memorises

[3]https://klee.github.io

this update and all subsequent changes in a linked list (*update list*) with the most recent updates as the root of the list (Fig. 2).

If the memory object is shared, new modifications are as simple as adding a new item to the update list that is only referenced by the modifying state. The downside is that no update can be made in place. Instead, every modification is added as a new entry to the list. If objects are often modified with a concrete index and seldom with a symbolic (e.g., as it happens in tight loops) a lot of memory is used.

For read operations, KLEE creates a read expression that references the latest update item in the list.

In summary, the update list contains the temporal order of the writes but does not allow optimisations of them. Besides, it does not handle the spatial locality at all.

III. An optimised memory representation

In this section, we describe our layered memory object representation and the optimisations that improve runtime performance and reduce allocation space.

A. Memory Objects and their Updates—a Layered System

A sophisticated memory representation for a symbolic execution engine has the following requirements:

- Fast concrete memory accesses.
- Represents memory operations with symbolic indices or values.
- Extends the memory model to support uninitialised memory.
- Simplifies the reuse of shared views on memory objects and their changes while it preservs temporal and spatial locality.
- Strongly separates constraint generation and constraint solving to allow better utilisation of existing constraint solving optimisations (like caching).

With these requirements and challenges, we propose a *layered* representation of memory objects that can be shared between many states. We detail the basic ideas in Figure 3. It depicts one memory object *o* allocated by *state 1* and how it could evolve with each step (right side of this figure).

a) Layering: A *layer* is a description of an array that can represent each byte of a memory allocation.

The first concept is to group updates of a single memory allocation into layers. If a state needs to allocate memory, a first layer (*terminal layer*) is created to represent that memory object (I). The layer is referenced once. Therefore, it is unshared. With every write operation (e.g., `write(2, 23)`, II) to that memory object, the layer is updated. This continues until the current memory object's state needs to be preserved. For example, if the object is read with a symbolic index (III), the value is not resolved. Instead, the specific object state is referenced and becomes part of a constraint. The reference counter is increased to reflect this, and the object state becomes shared. Similar, this is done for branching, i.e. *state 2* branches from *state 1* (IV). As these states will share the same representation of the memory object, the reference counter will be increased. If one of those states needs to modify the object (e.g. state 1 `write(1,17)`), the representation for other instances (state 2 and `read(sym1)` of state 1) cannot be changed. The updating state handles this by adding a new layer (*update layer*) as part of the memory object that still references the old layer (V).

Over time, a tree of layers will represent the object's history and relevant views held by any state. Every layer represents changes of temporal locality, and every leaf layer presents modifiable views on that object. If a layer is shared—the reference counter is more than one—, the invariant is that no semantic changes are allowed to it.

b) Layer Types: We extend the basic concept and provide different kinds for each basic layer type (terminal and update) to reflect the different access patterns for symbolic execution and to implement additional features (e.g., detection of uninitialised memory accesses).

The terminal types can be either *symbolic* or *concrete only*. This depends on whether a symbolic memory object is allocated or a non-symbolic one. Initially, for a symbolic layer, every byte is symbolic. For a concrete layer, every byte is marked as uninitialised. Figure 3 shows the concrete only terminal type.

To explain the different types of update layers, we assume for a moment that a memory object is updated by consecutive write operations without being read or being shared by multiple states. The basic goal is to aggregate similar access types together. Depending on the index type and value type, different layers are used. We have: *concrete only* (V), *concrete index* (VIII), and *symbolic index* (X). With a concrete index, the byte to modify is explicitly specified. Based on the value type, if it is concrete, it will be part of a *concrete only* layer. This layer type allows the most compact memory representation possible (a simple byte array is sufficient). If the value is symbolic, we use a *concrete index* layer, and the value is directly associated with the concrete index. If the write is to a symbolic index, we use a *symbolic index* layer that is similar to the update list of the KLEE implementation. Similar access types allow reasoning about the structure in a similar way.

What if different types of writes to the same object are interleaved? For every write with a type different than the previous one, a new appropriate layer is added.

To summarise, layering has three objectives: First, to allow to share and reuse a specific view on a memory object inside a single state or between different states, which allows saving memory; second, to preserve the temporal order of write operations and preserve temporal locality—more recent writes will be in newer layers than older writes; and, third, allow spatial locality without violating temporal order. The last two objectives allow further optimisations.

B. Optimisations using a Layered Memory Updates

With this layer system in place, we can apply different optimisations for `read()` and `write()` operations:

- **index-based access**
- **in-place update**
- **conditional update**

- **layer invalidation**
- **optimised hashing**

Again, we will use Figure 3 to explain the different cases.

a) Index-based Access: Every layer (except the symbolic layer) allows index-based access. Using concrete indices, we directly access the correct position of the most recent layer for read or write operations. When reading a specific index and the leaf layer does not contain a value at the index, we traverse the stack of layers until a value at this index is found. For example for `read(2)` (VI), the most recent version of the object state is in (V) for state 1. At index 2, there is no value in this layer. But the next, older layer (II) contains a value (`23`) at this position, which is returned by the read operation. If none is found, e.g. reading at index 0 instead, it is a read of uninitialised memory. For writing, the most recent layer can be modified at the indexed position (if not shared), or a new layer will be added containing the intended write. If there is a symbolic index layer, the read will return a reference to this layer combined with the concrete index (similar to III). Symbolic index layers are more restrictive and do not allow direct index-based access. Still, if a set of consecutive bytes should be accessed, we know that each byte does not alias with any other from the same set. This is handled similarly for writing operations. A potential optimisation is to calculate the bounds of the index and memorise them as part of the layer. This makes the layer accessible for subsequent concrete index reads as they might access values outside of these bounds. For example, if a write operation with a symbolic index potentially modifies index 0 and 1 (`sym3` in X for state 2), a read at index 2 can still return value `sym2` from the previous write (VIII).

b) In-place Update: If, for a write operation, the most recent layer is not shared and a value was recently written to that index, the value can be updated in place. This keeps the most recent and relevant value without preserving any older non-relevant update. For example at stage II, a `write(2,24)` would result in updating the layer directly.

c) Conditional Update: Building on top of the index-based access, if we have a `write` operation with a concrete index (e.g. `write(2,23)`), we can check if the most recent value referenced by this index (II) is the same as the one we want to write (VII). If so, nothing needs to be written. This is especially useful if the layer containing the value is already shared, no new layer needs to be added. New layers will only contain semantical changes to an object.

d) Layer Invalidation: As we track the indices which have been modified by every layer, we can detect when the most recent layers describe the whole object (e.g. `write(0,5)`, IX). In that case, the referenced object state can be replaced by a single layer containing all the updates. The new layer will be unshared and for the previous layer, the reference counter will be decremented. This can potentially lead to freeing other layers and allowing further optimisations for remaining users.

e) Optimised Hashing: We maintain a hash sum of each object state that allows us to compare object states fast, independent of their internal tree representation. With the help

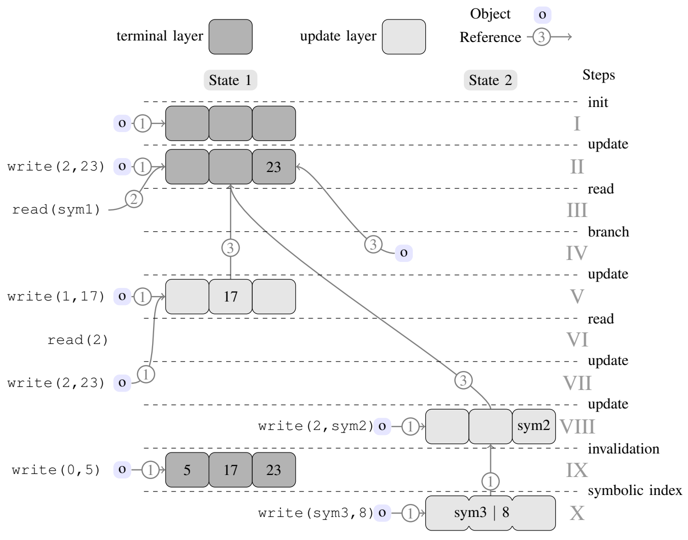


Fig. 3: Examples of different optimisations for layered objects. From top to bottom, each horizontal line separates an evolutionary step of the memory object.

of the layered structure and the index-based access, we can optimise the hashing. Every layer contains a hash sum that summarises the layer's updates and the one from previous layers. We update the hash with every update to the object state, we use the already stored value at the same index, undo its modifications on the hash sum and apply the modification using the new value. This way, we always have an up-to-date hash sum for an object state. The remarkable result of this is, that if we want to compare two objects and check if they are equivalent, they have to have the same hash sum even though they can be comprised of hugely varying structures that are the result of different executions. To handle hash collisions, if hash sums are equivalent, a byte-by-byte comparison can check their equivalence. As the hashing is composable, it allows other use cases. For example, to compare the divergence of two states of the same ancestor, only a subset of layers needs to be compared.

To summarise, the layered approach combined with the optimisations allows an efficient sharing of memory object changesets between states. The stack of layers preserves the temporal order. It fosters temporal and spatial locality, i.e. similar changes will be in close proximity in space and time and will typically use the same layer. Inside a layer, the temporal order is not important - except for the symbolic index layer type. For example, the order of `write(0,5); write(1, 17); write(2,23);` does not matter if subsets are not shared. This hashing allows for improving the equivalence check of objects vastly.

C. Object anonymity: Equivalence vs. Identity of Memory Objects

Another building block of our approach is to avoid that the information about memory objects of a state are tied to expressions used by the solver.

If different states allocate same-sized objects and generate equivalent constraints, we want to make sure that the solver gets an equivalent query. This allows us to cache solutions from a previous query and to reuse them.

To be able to do that, we use a technique based on De Bruijn Index [8] to allow to check for α-equivalence by checking for term equivalence. For that, we anonymise memory objects but also create and traverse expressions deterministically. While traversing a set of expressions as part of a solver query, we build up a list of anonymous memory objects (i.e. no address) by just remembering the order of their first occurrence and remembering their size. If an object is reencountered, we use the reference to its first occurrence. If we want to compare constraints from different states, we traverse the expression trees of the constraints in parallel and collect the anonymous objects and their order of occurrence on both sites. If the traversal showed a structural equivalence of the expressions, we know they are accessing equivalent expressions. Finally, we check the order of occurrences. If they are the same, we know expressions are semantically equivalent.

Therefore, if an assignment is valid for one set of constraints, it is also valid for another equivalent constraint set. If caches are structured such that they support anonymous objects in cached expressions, this can be used to make caching more efficient.

Independent of that, if an address of an object becomes part of an expression, two states will only generate the same expression, if the objects of each state refer to the same address.

IV. Implementation

We implemented our prototype on top of KLEE 1.4.0 [1], [9] replacing the old memory representation with our new one. The goal was to be feature compatible with KLEE and not to restrict its applicability.

A. Optimising the Layered Presentation of Memory Objects

We represent the memory objects with different layer classes. One key to our implementation is that every layer of a memory object is reference-counted using a smart pointer. Besides the automatic memory management, it provides us with the advantage to check if the layer is shared or not. This allows us to conditionally apply our optimisations §III-B.

Every layer type, except the symbolic layer and the symbolic index layer, contains a bitmap indicating which byte of this layer was already written. For the terminal all-concrete layer, we use an array with the exact size of the memory object as we noticed that many applications allocate memory and initialise it immediately before they use it further. This typically involves no symbolic expressions. For all update layers, we use a dense vector representation, which allows to only store the layer's updates. In combination with a bitmap array tracking at which index was written, we achieve a fast lookup.

The bitmap array allows us to implement two more features: tracking of uninitialised memory and layer merging. First, if we want to access a byte of an object, traverse every layer, check their modification bitmap but do not find this bit set, we can safely assume that the byte of the array is uninitialised. This allows us to efficiently find this type of failures—a threat for languages like C and C++. KLEE does not handle this type of failures as it initialises the value implicitly to a concrete random value. Second, the modified bitmap of consecutive layers can be merged efficiently; we check if all bits are set in the resulting bitmap. In this case, we do not need older layers, as their information has been overridden by the newer ones.

Possible Optimisations: We tried to keep our implementation as simple as possible. Still, we think there are many different optimisations to improve our general approach. For example, *small-* and *big*-size memory object optimisation: currently, we always allocate a full array plus appropriate bitmap for any allocation size. One could have a simplified version for small allocations (1-64 Byte, which happens quite often) to avoid the management overhead; or, a sparse version for large allocations (e.g. $> 1KiB$ - as they are often used for application local memory management like buffers and are not fully initialised).

B. Calling External Functions

One major challenge was the original design decisions made for KLEE[2]. For performance reasons, the execution states which resemble the tested application states are not fully decoupled from the address space of KLEE itself. The execution state's address space is not virtualised. Therefore, if a tested application executes an allocation, the returned address becomes part of that state but can also be directly accessed by KLEE itself. Still, for every memory access specific to this allocation, a memory object and a memory state will handle the operation. A state will never directly access the allocated space.

This approach has its advantages in case an external function is called: The addresses of allocated memory might be part of a function argument, e.g. for result buffers (`ext_funct(char * buf)`). Before calling an external function, KLEE copies every memory object and their content of a state to their real allocated memory buffers; it calls the external function; and, it updates the object states to reflect changes of real buffers by the invoked external function.

We had to extend this functionality as our memory state representation is layered. Therefore, before copying, we make a flattened copy of the original object state that we copy to the allocated memory; invoke the external functions, and compare the flat copy with the allocated memory. In case of changes, we copy them to the appropriate layer and continue execution.

C. Handling Anonymous Objects

To implemented the support for handling anonymous object III-C, we modified the traversing of expressions used by KLEE. In addition, we updated the caching infrastructure to handle anonymised objects.

V. Evaluation

We compare our implementation (*Fine-Grain Memory*) with the state-of-the art symbolic execution engine KLEE (*Baseline*) to answer the following research questions: *RQ1:* What is the performance impact of the fine-grain memory approach? *RQ2:* What is the impact on memory consumption?

A. General Experimental Setup

We used GNU Coreutils,[4] a suite of system applications that are part of most UNIX and Linux environments. Their variety exercises different aspects of a symbolic execution engine and they provide different patterns of impact (e.g., time spent in solving or time spent in generating constraints). We analyse each application of the GNU Coreutils with the original KLEE version (*Baseline*) and our new memory implementation (*Fine-Grain Memory*). For every application, we run the two different implementations for the same number of instructions that can be executed in roughly 30min. We use STP [10] 2.2.1 as our solver.

Every test runs as a single Docker instance without any parallel running application on one out of 50 equivalent host machines (Intel XEON E5405 with 2.00GHz and 8GB RAM).[5] Every experiment starts with the same clean state. We executed each experiment 3 times and took the mean to account for the impact of differences between execution time due to system variability.[6]

B. Deterministic Exploration

We aim for reproducible research. Still, we think that this is hard in the context of symbolic execution due to the state space explosion problem and the resulting non-determinism. With a reasonable time limit, only a subset of all possible paths can be solved. Out of the many execution paths that an application can have, every path is different: the instructions per path differ, and the associated costs of solving queries along a path vary. Therefore, if we compare two implementations, and each implementation is exploring different paths, it is hard to attribute differences to either the implementation or the path taken. For example, Fig. 4 shows a subset of applications we have tested and compares how many instructions three different search strategies can execute. Within a single application, the variety can be significant (e.g. for *id*), but also between different applications, the number of instructions varies significantly.

Ideally, we would like to fully explore an application. Unfortunately, this is often not possible, so we limit each experiment to 30 min. We employ a *controlled, deterministic exploration* of tested applications. In a nutshell, for each application and each searcher, we log every instruction that has been executed and by which state it has been executed. Furthermore, we limit each application to execute an exact number of instructions that can be finished in roughly 30 minutes.

[4]Version 6.11 https://www.gnu.org/software/coreutils/coreutils.html

[5]https://www.docker.com/

[6]The total execution time of the final results are worth 33 days of execution time on a single node

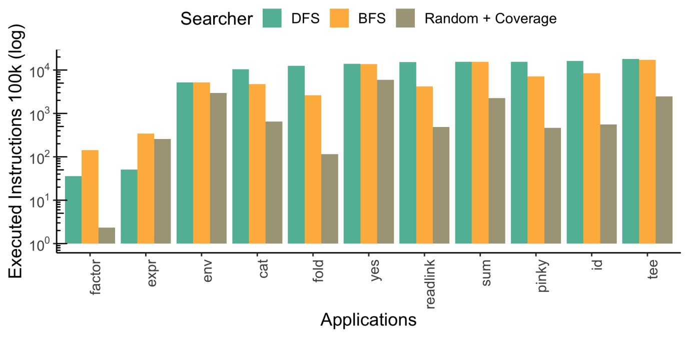


Fig. 4: Number of instructions executed for 30 min per application using different exploration strategies (depth-first, breadth-first and random plus coverage-driven) (30 min runtime).

We compare the logged instructions for our two implementations to check if they are identical. Applications can exhibit nondeterministic external behaviour (e.g., check date/time or free disk memory) on some execution paths. Therefore, we cannot compare all applications (89) of the GNU Core Utilities with all searchers like that. We restrict ourselves to the deterministic runs with an execution time of at least three minutes of the baseline run. Tab. II summarises how many applications per searcher could be executed deterministically.

This deterministic approach has two important advantages: First, we can attribute differences of measured aspects to the implementation and the approaches and, with that, we can rule out the possible impact of differently explored state space. Second, we can check if a new implementation behaves correctly with respect to the original one.

C. Experiment Variety

To show the effectiveness of our implementation, we use three different space exploration strategies as they show fundamentally different behaviour of a symbolic execution engine. The different exploration strategies are the following.

- **Depth-First-Search (DFS)**: Progress one state as much as possible until the tested application terminates or an error is detected. There is little state branching but the path constraints tend to be larger.
- **Breadth-First-Search (BFS)**: Prioritise the exploration of the width of the state space before exploring its depth. This leads to more state branching but smaller constraint sets per path.
- **Random + Coverage oriented**: Combination of two exploration strategies: a random selection of paths and a code-coverage-driven exploration of the state space. The behaviour varies between DFS and BFS.

With the potentially large variety of executed instructions between searchers for a single application (Fig. 4), we can see that different paths have different costs. Table II shows relevant properties of the test suite when executed with different searchers. Every searcher has applications where it executes most instructions (*Application with Max. Instructions*). Still,

TABLE II: Different search strategies and properties of the application

Searcher	Deterministic Applications	Max. Num. Instructions	Max. States	Apps with Max. Instructions
DFS	43	$1.8 * 10^9$	887	16
BFS	37	$1.7 * 10^9$	587893	11
Random + Coverage	40	$0.6 * 10^9$	500657	2

DFS can execute most of the applications with the highest number of instructions. *Random + Coverage* executes the least. One reason is that tracking the coverage-based information to decide which state to progress next is much more expensive than the other two searchers. Therefore, fewer instructions are executed as more time is spent in the state selection.

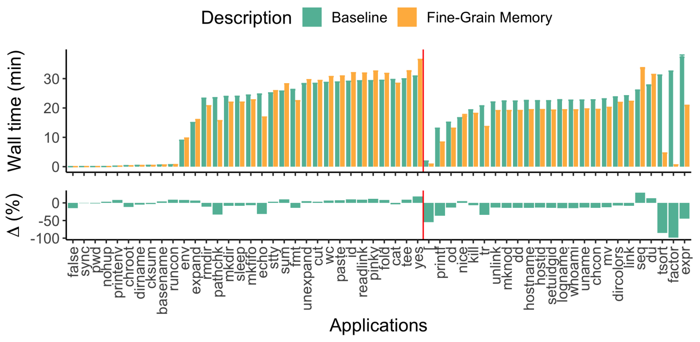
Another interesting property is the maximum number of states that co-exist during execution (*Max. States*), for DFS the number of states is three orders of magnitudes smaller than for BFS and Random + Coverage. Hence, the memory pressure is much higher for the two latter searchers. Also, every algorithm that needs to iterate over the whole set of states takes longer.

D. Impact on Execution Performance

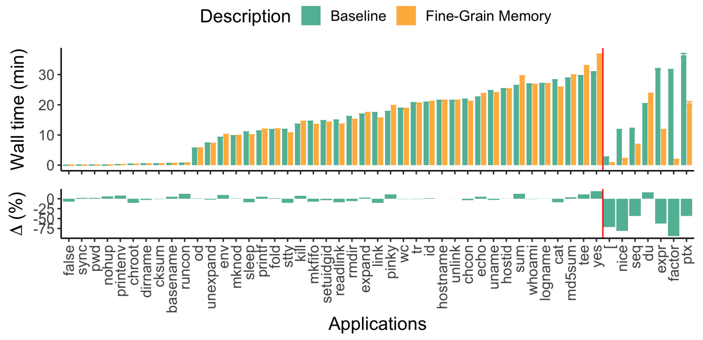
Fig. 5 shows the impact on execution time. For every searcher, we show how much time each application took to execute it deterministically. We divided each searcher result into two groups separated by a red bar. Applications of the *Baseline* where the executor spends more time in executing instructions are on the left side of the red bar; applications with more time spent in constraint solving are on the right side. Both categories stress different parts of a symbolic execution engine: instruction-intense applications increase the likelihood for memory accesses and branching of states and will exercise more memory-related operations; constraint solving often involves whole-object access for reasoning as part of the solver access. We can see for our experiments that the majority is constraint generation dominated. While DFS has the most constraint-solving dominated applications, BFS has the least.

The upper graphs show the absolute wall-time for each experiment, the lower graph shows the relative changes with respect to the baseline. For example, for depth-first search (Fig. 5a) *Fine-Grain Memory* has an 18% overhead for `yes` but reduces the execution time by 95% for `factor`. For constraint-generation dominated applications, the results are promising. We see runtime changes between by -2% to $+18.7\%$ for DFS (Fig. 5a) with similar results for BFS (Fig. 5b) . Random + Coverage (Fig. 5c) shows the most overhead between 40% and 50% (`pinky`, `readlink`, `id`). A major reason is that our prototype is not explicitly optimised to handle small array allocations for simplicity reasons. But it could be added in future work. Solver-intensive applications already benefit much more from our approach. Their runtime changes between -95% and $+9\%$.

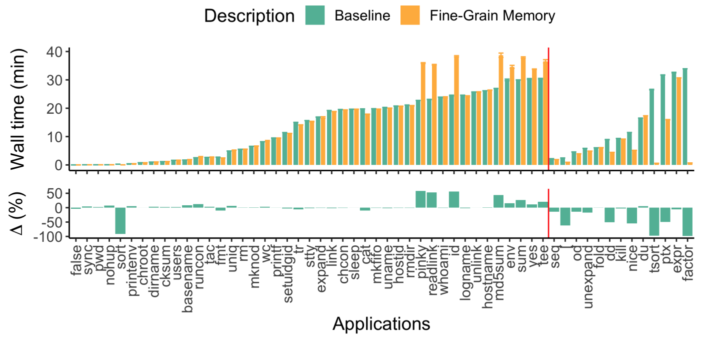
To understand better where those differences come from, we have a closer look at how much time every application spents in constraint generation (Fig. 6). The layered structure can add more time to the constraint generation. We measured



(a) Depth First Search



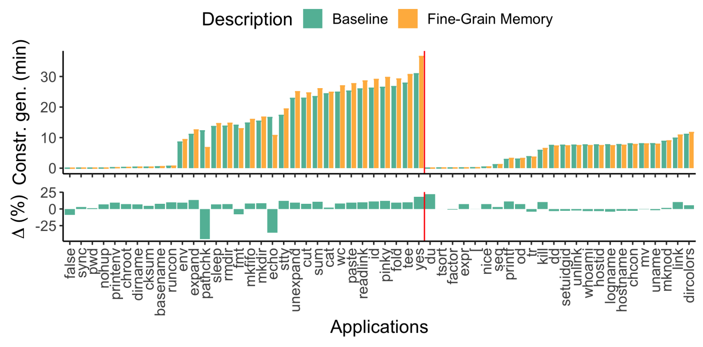
(b) Breadth First Search



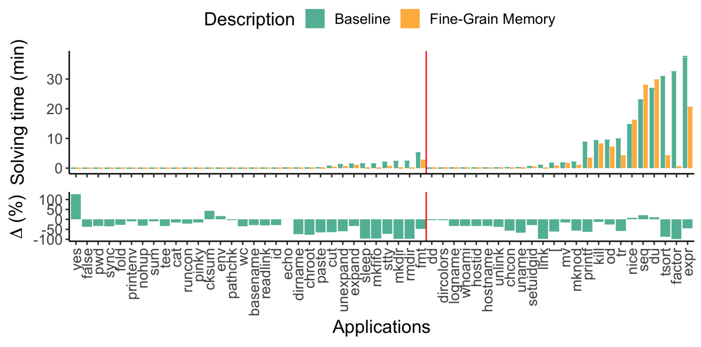
(c) Random + Coverage Search

Fig. 5: Execution time (s) of the *Fine-Grain Memory* implementation vs. *Baseline* implementation. Red bars separate experiments where *Baseline* spends more time generating instructions (left) from applications that spend more time in constraint solving (right).

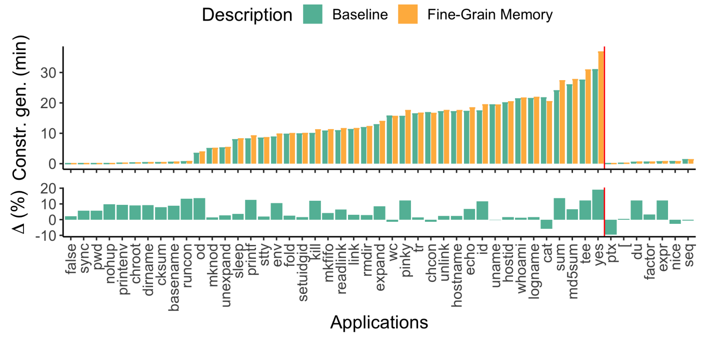
up to 21% for the constrain generation dominated applications. With the vast amount of instructions executed, this overhead accumulates. Still, we already observe reductions as well (e.g. for `pathchk`, `echo` DFS) that can utilise the index-based access.



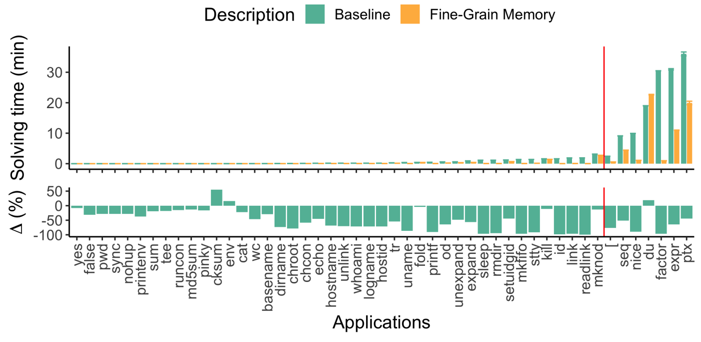
(a) Depth First Search



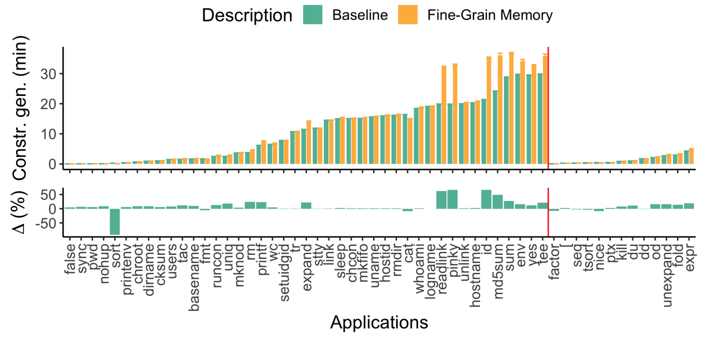
(a) Depth First Search



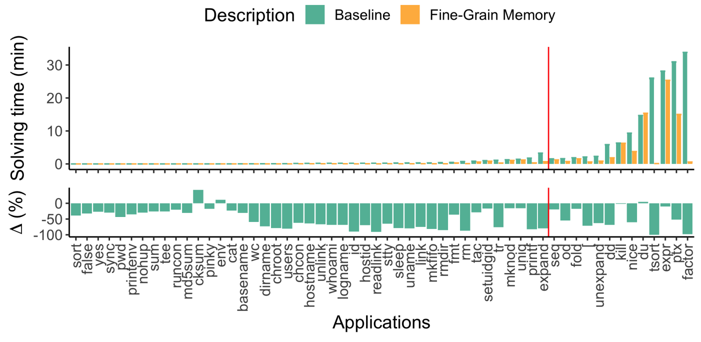
(b) Breadth First Search



(b) Breadth First Search



(c) Random + Coverage Search



(c) Random + Coverage Search

Fig. 6: Time spent in constraint generation in (s) of the *Fine-Grain Memory* implementation vs. *Baseline* implementation: upper bar chart shows absolute values (lower is better); lower bar shows relative changes (%)

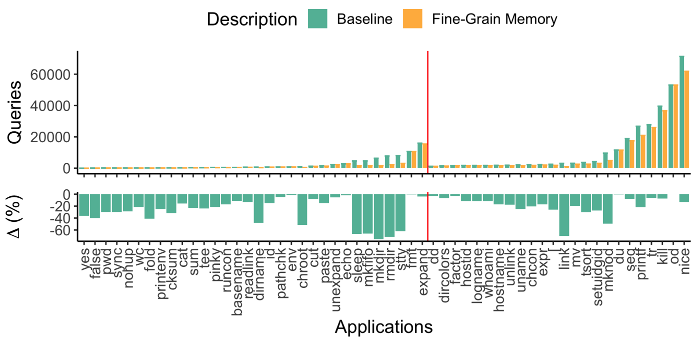
Fig. 7: Time spent in solving queries for different state space exploration strategies.

On the other hand, we save much more time by the reduced solver time. Looking at the graphs (Fig. 7), one can see that constraint-generation-dominated applications spend little time in solving. Still, the differences saved by our approach are significant.

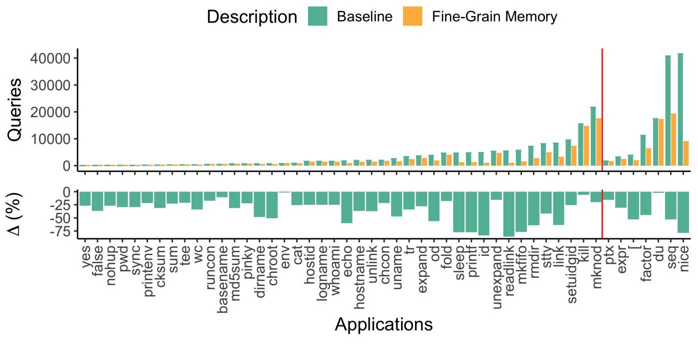
One building block of our memory implementation is the memory object anonymity (§III-C). If queries are detected as equivalent if they share the same structure, the more efficient a cache is for such solutions. For that, we look at the number of queries that are finally issued to a solver (Fig. 8), we see a noticeable reduction of query numbers (DFS, BFS: up to 95%). Therefore, we can conclude that the object anonymity implementation is effective. Still, the reduction does not fully explain all the solving time saved (III-A). The layered representation of memory objects allows the simplification of solver queries, which shows in the additionally saved solver time.

E. Impact on Memory Consumption

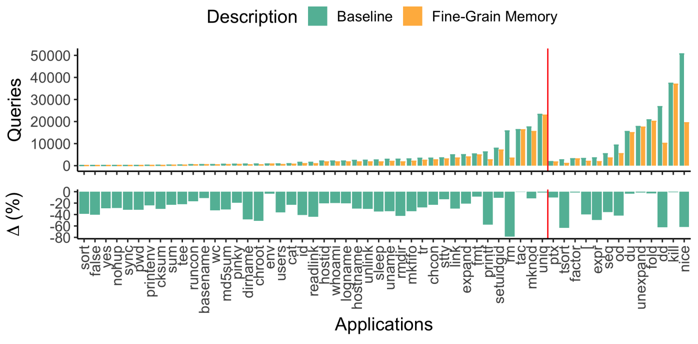
We can observe a positive impact on execution and solver time. Still, we want to measure the maximum allocated memory during the execution to quantify its impact. As a reminder, due to the deterministic path exploration, the number of concurrently active states was equivalent at all time for both implementations. The maximum memory we allowed for the states was 3072 MiB. We added a state size estimation which allows us to keep the same number of states for both implementations. Furthermore, we made sure that in case we



(a) Depth First Search

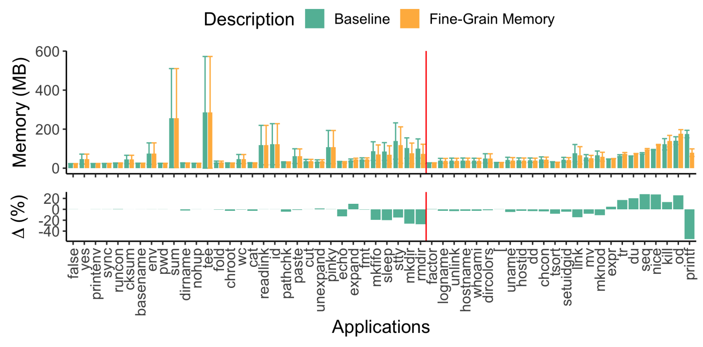


(b) Breadth First Search

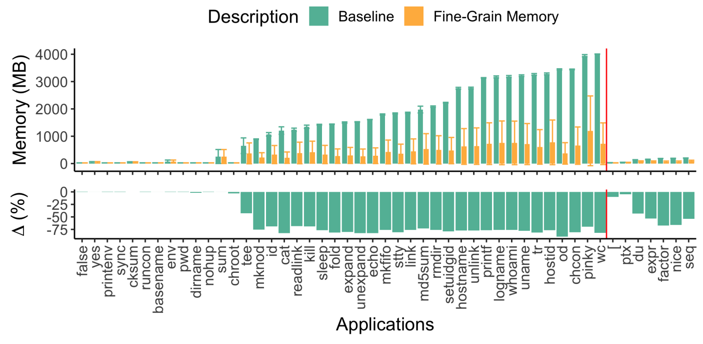


(c) Random + Coverage Search

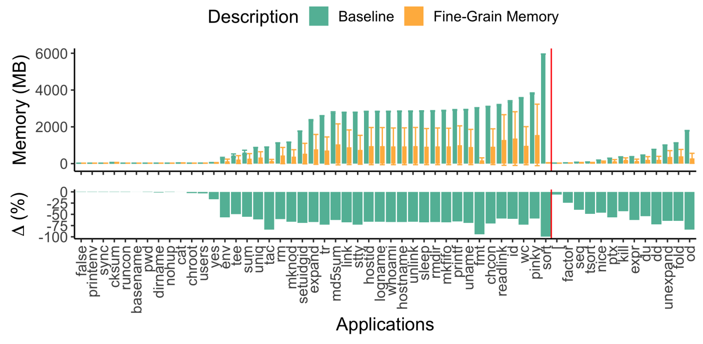
Fig. 8: Number of solver queries. This number excludes all the queries that are already solved by caching.



(a) Depth First Search



(b) Breadth First Search



(c) Random + Coverage Search

Fig. 9: Memory usage (MB) by applications for different DFS and BFS.

hit the limit with our estimated size, we kill the same states. The estimation for the state size was not always perfect but helped us to avoid reaching real memory limits and allowed the deterministic execution. We tracked the allocated memory as provided by the system allocator. Besides the memory for memory objects of each state, it also contains other memory allocations made by KLEE (*Baseline*) and our implementation (*Fine-Grain Memory*).

As one can see in Fig. 9b, the overall memory allocated highly depends on the search strategy. While on average, the consumed memory for *Baseline* is below 200 MiB for DFS (Fig. 9a), the memory consumed by BFS can be up to $20x$ higher. As one can see in Tab. II the size correlates with the number of states. In comparison, *Fine-Grain Memory* can add a little overhead if only a few states need to be represented (+20%, $< 50MiB$). In contrast, if a vast amount of states need to be present (like for BFS), the space reduction achieved by *Fine-Grain Memory* is substantial (up to 82%, $> 3GiB$) as more layers of memory objects can be shared between different states. To emphasise again, we limited the memory for both implementations to 3GiB to achieve a deterministic comparison. Even isf more memory is made available to both implementations, still, *Fine-Grain Memory* could represent more additional states with the extra memory than KLEE.

F. Handling Uninitialised Memory

Our new memory implementation supports the detection of the usage of uninitialised memory and terminates a state as soon as an operation depends on a previously read but uninitialised memory. We disabled that feature for the previous part of this section. We also ran the new implementation with uninitialised memory detection to validate that it works.

We found bugs in the runtime support library of KLEE itself and multiple in the CoreUtils suite.

G. Validity

Replacing an essential part of a symbolic execution engine like the memory representation can be an error-prone and challenging task. We added an extensive test suite to KLEE that allows us to check the memory behaviour. Furthermore, for the evaluation, we made sure that our experiments behave deterministically to foster reproducibility. For every setup and every application, we fully tracked the executed instructions for both implementations, the original KLEE and ours. We checked that both executed precisely the same instructions by the same states in the same order. We plan to publish our source code and experiment results to allow the community to reproduce our findings.

VI. Related Work

a) Representation of allocated memory objects: Various symbolic execution engines use different techniques to represent allocated memory. For Java, JFP-SE[11], only concrete memory objects are allocated. Symbolised values are managed via metadata (e.g. for each attribute of a class) associated with memory objects. This way, objects with partial changes are harder to represent efficiently than with our layered approach. For Mayhem [4], which analyses binaries, a notion of objects does not exist. Memory accesses can be arbitrary and associating them with a specific object is not always possible. Therefore Cha et al. apply two main optimisations: writes are always concrete and reads with symbolic index are handled as snapshots M of a memory region to which the symbolic index could be resolved. Similarly, our layered approach is not limited to objects, but could also be used for arbitrarily size memory regions.

b) Memory-efficient state representation: For EXE [1], the tested application and its address space is reflected by the process. In case states need to be branched, the process forks via POSIX call `fork()` and memory pages are shared between processes. The downside is that even for small changes, the copy-on-write of the operating system creates page-sized copies. KLEE [2] handled this issue by tracking memory based on allocated memory objects. This largely reduced the memory footprint.

Mayhem [4] employs state-suspension if needed. In case of too much memory usage, a subset of the states is saved on disk while it continues with the remaining ones. If no more states are left, previously-stored states are restored, and Mayhem continues to analyse them. This is an orthogonal approach to *Fine-Grain Memory* as we try to avoid spilling states on disk in the first place, to avoid performance penalties. Still, if memory pressure becomes too high due to the state-space explosion, state-suspension is a useful approach.

c) State Space Explosion: For handling the massive amount of states, different techniques have been proposed:

- State Merging: Using call-graph and control-flow information to merge similar states together [6] or the data-driven approach like in [12].
- Composition: Using summaries of modules (e.g. functions) to replace the invocation of those functions with symbolic summaries [13], [14], [15], [16]
- State space pruning: Removing irrelevant states as they are subsumed by existing ones [17]
- Targeted exploration: The state space is explored according to specific requirements, e.g. to cover yet uncovered instructions like proposed by Cadar et al. [1], [18], [7]

All those approaches are orthogonal to ours. Still, for any of those approaches, many states need to be explored in parallel eventually. Our memory efficient representation supports this.

VII. Conclusion and Future Work

Efficient representation of allocated memory is vital for symbolic execution. We have introduced *Fine-Grain Memory*, a method that represents memory in a layered structure to foster temporal and spatial locality and to preserve the temporal order of memory object modifications. Moreover, it allows additional optimisations that reduce the number the solver is invoked and therefore reduce the overall time spent in solving. Preliminary results show the usefulness of our approach by extensively reducing the memory footprint of a symbolic execution engine and improving its runtime performance. Our approach allows us to handle much more states simultaneously and avoids terminating them or storing and restoring them in the first place.

We plan to make our implementation open source and contribute relevant changes to KLEE.

Acknowledgment

We want to thank Cristian Cadar, Frank Busse, Timotej Kapus, Vesna Nowack and the anonymous reviewers for their valuable comments on our paper and their support. The UK EPSRC has generously supported this research via grant EP/N007166/1, EP/R011605/1 and funding by TU Dresden, Germany.

References

[1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Transactions on Information and System Security*, vol. 12, no. 2, pp. 1–38, Dec 2008. [Online]. Available: http://dx.doi.org/10.1145/1455518.1455522

[2] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI 2008. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756

[3] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, 2010. [Online]. Available: http://dx.doi.org/10.1145/1858996.1859035

[4] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 380–394.

[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, Feb. 2014.

[6] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient State Merging in Symbolic Execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254088

[7] K.-K. Ma, K. Yit Phang, J. S. Foster, and M. Hicks, "Directed Symbolic Execution," in *Static Analysis*, E. Yahav, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–111.

[8] N. G. de Bruijn, "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," *Studies in Logic and the Foundations of Mathematics*, vol. 133, pp. 375–388, Jan. 1994.

[9] KLEE developers. (2017, 07) KLEE 1.4.0. [Online]. Available: https://github.com/klee/klee/releases

[10] V. Ganesh and D. L. Dill, "A Decision Procedure for Bit-Vectors and Arrays," in *Computer Aided Verification*. Berlin, Heidelberg: Springer, Berlin, Heidelberg, Jul. 2007, pp. 519–531.

[11] S. Anand, C. S. Pasareanu, and W. Visser, "JPF-SE: A Symbolic Execution Extension to Java PathFinder." *TACAS*, pp. 134–138, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-71209-1_12

[12] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: multi-path symbolic execution using value summaries," *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, 2015. [Online]. Available: http://dx.doi.org/10.1145/2786805.2786830

[13] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 47–54. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190226

[14] S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381.

[15] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: ACM, 2010, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1706299.1706307

[16] R. Qiu, G. Yang, C. S. Pasareanu, and S. Khurshid, "Compositional Symbolic Execution with Memoized Replay," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 632–642.

[17] P. Boonstoppel, C. Cadar, and D. R. Engler, "RWset: Attacking Path Explosion in Constraint-Based Test Generation," in *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2008, pp. 351–366.

[18] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, June 2009, pp. 359–368.