A Deterministic Memory Allocator for Dynamic Symbolic Execution

Daniel Schemmel
Imperial College London, UK

Julian Büning
RWTH Aachen University, Germany

Frank Busse
Imperial College London, UK

Martin Nowack
Imperial College London, UK

Cristian Cadar
Imperial College London, UK

Abstract

Dynamic symbolic execution (DSE) has established itself as an effective testing and analysis technique. While the memory model in DSE has attracted significant attention, the memory allocator has been largely ignored, despite its significant influence on DSE.

In this paper, we discuss the different ways in which the memory allocator can influence DSE and the main design principles that a memory allocator for DSE needs to follow: support for external calls, cross-run and cross-path determinism, spatially and temporally distanced allocations, and stability. We then present KDAlloc, a deterministic allocator for DSE that is guided by these six design principles.

We implement KDAlloc in KLEE, a popular DSE engine, and first show that it is competitive with KLEE's default allocator in terms of performance and memory overhead, and in fact significantly improves performance in several cases. We then highlight its benefits for use-after-free error detection and two distinct DSE-based techniques: MoKlee, a system for saving DSE runs to disk and later (partially) restoring them, and SymLive, a system for finding infinite-loop bugs.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging

Keywords and phrases memory allocation, dynamic symbolic execution

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.9

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.13`

Funding This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement no. 819141 and 966733).

Acknowledgements We would like to thank Jordy Ruiz and the anonymous reviewers for their valuable feedback on the paper.

1 Introduction

Dynamic symbolic execution (DSE) [11] is a software testing technique that relies on systematically exploring the execution paths that a program might take, using a constraint solver to reason about the feasibility of each path.

An important component of a DSE engine is its memory model, which has received significant attention from the research community [5, 10, 16, 24, 32]. However, one component of the memory model has been largely ignored: the memory allocator. But the memory

allocator plays a key role in DSE, with a direct influence on its ability to find memory-safety bugs and ensure deterministic execution, which is needed by several DSE-based techniques. Furthermore, the memory allocator can impact the performance and memory consumption of DSE, particularly in the EGT [9] style of DSE, where multiple program paths are concurrently kept in memory.

The memory allocator influences DSE's ability to find memory-safety violations because such errors can sometimes be detected only under certain memory layouts. For instance, a DSE engine like KLEE [8] may miss a buffer overflow if the pointer overflows into another memory object or may miss a use-after-free error if another object is allocated at the same location before a freed pointer is incorrectly dereferenced. In both cases, the final read or write appears to target valid memory, causing the engine to overlook the error.

The memory allocator influences deterministic execution in two ways: across runs and across paths. Most obviously, a non-deterministic allocator may impact determinism across two otherwise identical DSE runs (*cross-run determinism*). Less obviously, the memory allocator may also impact determinism across paths if the allocation decisions on one path influence the allocation decisions on another path (*cross-path determinism*). Determinism is important in DSE in many different scenarios. For instance, it makes it possible to compare multiple DSE configurations, such as using different constraint solvers [22] or search heuristics [6]. It also facilitates debugging when one needs to execute the same path repeatedly. Determinism is also important for scenarios that rely on re-runs, such as saving a run to disk and later restoring it [7], or re-executing events in the context of partial-order reduction for multi-threaded code [27]. Finally, determinism is also key for scenarios that compare memory contents across paths, such as finding infinite loops [26] and pruning redundant paths [4].

In this paper, we present KDAlloc, a memory allocator specifically targeting DSE. KDAlloc is carefully designed to achieve cross-run and cross-path determinism, maximise the probability of finding memory-safety bugs, keep a low memory and performance overhead, and allow the interaction with the outside environment. (The latter is an important distinguishing characteristic of DSE, and refers to its ability to interact with uninstrumented/unavailable code, such as external libraries and the operating system.)

We implement KDAlloc in KLEE [8], a popular DSE system based on the EGT [9] approach of keeping multiple paths concurrently in memory. We first show that KDAlloc is generally competitive with KLEE's default allocator in terms of performance and memory overhead, and that using it sometimes leads to significant performance gains. We then highlight its benefits for use-after-free error detection and in improving two distinct DSE-based techniques: MoKlee [7], a system for saving DSE runs to disk and later (partially) restoring them, and SymLive [26], a system for finding infinite-loop bugs.

In summary, the main contributions of this paper are:

1. An investigation of the main properties desirable from a memory allocator in a DSE context.
2. The design of KDAlloc, a new memory allocator for DSE, and its implementation into KLEE, a popular DSE system. KDAlloc is available as open source, together with an associated artifact.
3. An evaluation of KDAlloc in terms of performance and memory overhead, and its effectiveness on three scenarios: detection of use after free, MoKlee and SymLive.

The rest of the paper is structured as follows. §2 introduces some background information concerning DSE and its interaction with the memory allocator. §3 defines desired properties of an allocator for DSE engines. §4 presents the high-level design of KDAlloc, while §5

discusses several implementation details. §6 presents our experimental evaluation, on a diverse set of 18 benchmark suites with a total of 94 applications; and on two case studies: MoKlee and SymLive. Finally, §7 discusses related work and §8 concludes.

2 Background

Dynamic symbolic execution (DSE) executes programs on *symbolic inputs*. On each explored path, DSE maintains a *symbolic store* mapping variables to symbolic expressions and a *path condition (PC)* which describes the inputs following that path. For example, the symbolic store might map a program variable `x` to the symbolic expression $\lambda + 1$, while the symbolic input λ is constrained by the PC such that $\lambda \geq 0 \wedge \lambda < 100$.

Whenever execution reaches a branch point that depends on the symbolic input (e.g., `if (x == 0)` with $\texttt{x} \mapsto \lambda + 1$), the path is forked into two new paths: one following the `then` side, where the branch condition is added as a conjunct to the PC ($\text{PC}' = \text{PC} \wedge \lambda + 1 = 0$), and one following the `else` side, where the negation of the branch condition is added as a conjunct to the PC ($\text{PC}'' = \text{PC} \wedge \lambda + 1 \neq 0$). If either of these new PCs is unsatisfiable, execution does not continue along that path, as no concrete input exists which would cause the program under test to take that path through the program. In the example, `x` can never be zero. The symbolic store maps `x` to the symbolic expression $\lambda + 1$, which is constrained to be in the interval $[0, 100)$, meaning that $\lambda + 1$ has to be at least 1. Therefore, the `then` case (with $\text{PC}' = \lambda \geq 0 \wedge \lambda < 100 \wedge \lambda + 1 = 0$) cannot be triggered.

In the EGT [9] variant of DSE, all paths under exploration are kept in memory as *symbolic states*. Each symbolic state stores all the information necessary to continue execution on that path. In particular, each symbolic state has its own address space: globals, stack and heap.

One distinguishing characteristic of DSE is its ability to interact with the outside environment, such as external libraries and the operating system. In order to be able to perform such an *external function call*, a state needs to share its address space with the address space of the external library. This imposes an important limitation on the way memory is managed by the DSE engine, and thus on the memory allocator. For instance, KLEE manages this by having all states allocate memory in the unique address space of the KLEE process. Note that while two states with the same parent state may both have an object allocated at the same address, the object contents are unique to each state and stored in separate, internal memory buffers. However, before an external function call, the object contents are copied to their assigned allocation address in the address space of the KLEE process, so that external functions can work as expected. Similarly, after the external call completes, any changes made by the external code are propagated back from the KLEE address space to the internal memory buffers associated with the current state.

3 Design Principles

Traditional memory allocators [2, 13, 17, 18] are primarily concerned with performance and memory consumption. To achieve this, allocators try to keep the overhead of the allocator's operations low and to reduce memory fragmentation. However, allocator performance and memory consumption are not the primary considerations in a DSE context, as other operations overshadow them. Instead, we identify six key principles that need to guide the design of a memory allocator for DSE:

3.1 Support for External Calls

As discussed in §2, the ability to interact with the external environment is one of the main strengths of DSE. In order to be able to perform an external function call, a state needs to share its address space with the address space of the external function. Therefore, a memory allocator for DSE needs to manage not only the virtual address spaces associated with each state, but also the global address space which is used by external code.

3.2 Cross-run Determinism

One important property of a DSE engine is to have multiple identical runs behave in the same way. As discussed in the introduction, this makes it possible to compare multiple DSE configurations, e.g. with different constraint solvers [22]; facilitates debugging; and enables applications that rely on re-runs, such as saving a run to disk and later restoring it [7].

One reason for which different runs may behave differently is that program behaviour can depend on the memory layout. One simple example is the `memmove` function further described in §6.5, which changes behaviour depending on the relative locations of the source and destination addresses.

To remove this source of non-determinism, a DSE engine needs to use a cross-run deterministic memory allocator.

▶ **Definition 1** (Cross-run Determinism). *A DSE engine or memory allocator is* cross-run deterministic *iff its behaviour is the same for each run that is initialised in the same way.*

For instance, KLEE provides a simple deterministic memory allocator which internally allocates (via `mmap`) a large memory region at a fixed address, and then serves allocations from this region, never freeing objects. KLEE's deterministic allocator is cross-run deterministic, but it is often unusable in practice since it never frees memory.

Of course, a cross-run deterministic memory allocator does not suffice to have a cross-run deterministic DSE engine, as the latter may have other types of non-determinism (e.g., through the interaction with the environment).

3.3 Cross-path Determinism

As discussed in §2, the EGT style of symbolic execution stores multiple symbolic states in memory. If these symbolic states were to influence one another, the result of the analysis could suddenly depend on the order in which symbolic states are analysed. This is undesirable, as it makes it difficult to re-execute individual paths in isolation, which is needed in e.g. a debugging context or for selectively re-executing program paths.

▶ **Definition 2** (Cross-path Determinism). *A DSE engine is* cross-path deterministic *iff the behaviour of one symbolic state does not impact the behaviour of another.*

For a DSE engine to be cross-path deterministic, the memory of each symbolic state must be managed independently. Otherwise, any allocation would impact the shared memory allocator state and might change the memory allocation pattern of another symbolic state.

For instance, KLEE provides two allocators: the default one which simply uses the underlying system allocator, and the deterministic allocator described above. Since the allocator state is shared across states, none of them is cross-path deterministic, although the latter is cross-run deterministic.

By contrast, the memory allocator in EXE [10] is cross-path deterministic, because symbolic forking in EXE uses the UNIX system call `fork`, which duplicates the allocator's state.

3.4 Spatially Distanced Allocations

As noted above, traditional memory allocators aim to reduce memory fragmentation, in order to decrease the working set of a program and improve cache locality. On the other hand, by performing compact allocations, they also make it more probable that out-of-bounds accesses point to other valid objects.

For DSE, the latter is a much more important consideration. First, and as argued before, performance is a secondary aspect given the other large overheads of DSE. Second, addresses generated by the allocator are only actually used for the duration of an external function call, so the effect of the increased fragmentation becomes much less pronounced.

By contrast, finding buffer overflows is an important application of DSE. As most common out-of-bounds errors result in accesses that are in close proximity to their target object (e.g., off-by-one array indices), accessing them should not result in a valid access to a different object. Instead, allocations should be separated as far as possible to enable the detection of such overflows.

Our benchmark results, which we believe to be typical for a 2 h run of KLEE, had up to 758 million allocations and up to 412 MiB live in the whole symbolic execution engine, with up to 134 MiB live in any symbolic state. In a 64-bit address space (even when considering the 48-bit physical address space that can actually be used), there is a lot of room to spatially distance those allocations.

We note that the DSE engine can implement other mechanisms to find buffer overflows, which do not depend on the memory layout. For instance, EXE [10] tracks referent objects for all pointers. However, in the context of DSE, such mechanisms are unfortunately fragile, because such tracking information is lost while executing external code.

3.5 Stability

A desired property of a memory allocator for DSE is to have allocations as stable as possible with respect to slight changes in the allocation pattern. For example, if two paths allocate the same objects except that one of them temporarily allocates an extra object, a stable allocator would give the same memory addresses to the common objects on the two paths.

Stability ensures that similar paths have similar memory layouts, improving the effectiveness of approaches that compare memory across states [26, 27]; and that similar paths generate similar queries, improving cache hit rates [8].

Unfortunately, stability is in direct conflict with the objective of having temporally distanced allocations, which we discuss next.

3.6 Temporally Distanced Allocations

While spatially distancing allocations is important for buffer-overflow detection, temporally distancing them is important for finding use-after-free errors.

At one extreme, KLEE's deterministic allocator never frees memory, and thus never reuses it, reliably detecting all use-after-free errors. At the other end, an allocator that eagerly reuses memory would miss most use-after-free errors when the DSE engine has no additional mechanisms for tracking referent objects.

```
1  int *i = malloc(sizeof(int));
2  if (sym)
3    *i = 7;
4  else
5    *i = 42;
6  int *k = calloc(1, sizeof(int));
7  printf("ext. call: %d/%d", *i, *k);
```

Global allocator disallows k1 = k2
KDAlloc allows k1 = k2

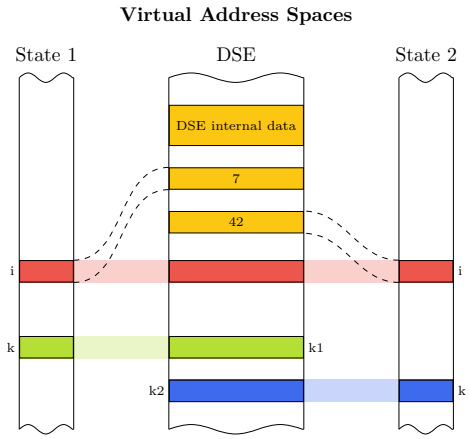


Figure 1 Code example with two paths illustrating how the state virtual address spaces are managed with a global memory allocator vs. KDAlloc.

As mentioned above, temporally distancing allocations is in direct conflict with the stability goal, as delaying memory reuse makes similar paths have different memory layouts. For example, if an object is allocated and freed again without any other memory allocation in between, it may be reused instantly iff allocations are not temporally distanced.

4 Design

Our allocator is guided by all six design principles discussed in §3. In this section, we first introduce the high-level design behind our allocator and discuss how it aligns with these design principles.

4.1 State Virtual Address Spaces

DSE engines like KLEE handle the allocated memory of all states via a global allocator. By contrast, KDAlloc is designed to manage the allocated memory for each state individually. This means the same virtual addresses can be used by different states, even for objects that are newly allocated by each state. This independence is a key element to achieve cross-path determinism and increase stability.

Figure 1 shows the difference between the way memory addresses are handled by a global allocator vs. KDAlloc. The program has two paths, encoded as two states by the DSE engine. State 1, shown on the left, runs the path that takes the `then` side of the branch, while State 2, shown on the right, runs the path that takes the `else` side.

Initially, there is a single state, State 1, which runs the `malloc` at Line 1. The DSE engine invokes the allocator and returns address i. When the symbolic condition at Line 2 is reached, another state, State 2, is forked. Immediately after the fork, both states share the same address space. When State 1 executes Line 3, it writes value 7 at the object allocated at address i. In a regular program execution, the address space (set of assigned addresses) and associated memory (the place where data is stored) are tightly connected, but these are handled separately in DSE. Therefore instead of placing value 7 at address i, State 1 places it into a separate memory location associated with State 1, as part of the DSE internal data. When State 2 executes Line 5, the write is handled in a similar way, by placing value 42 into a separate memory location associated with State 2.

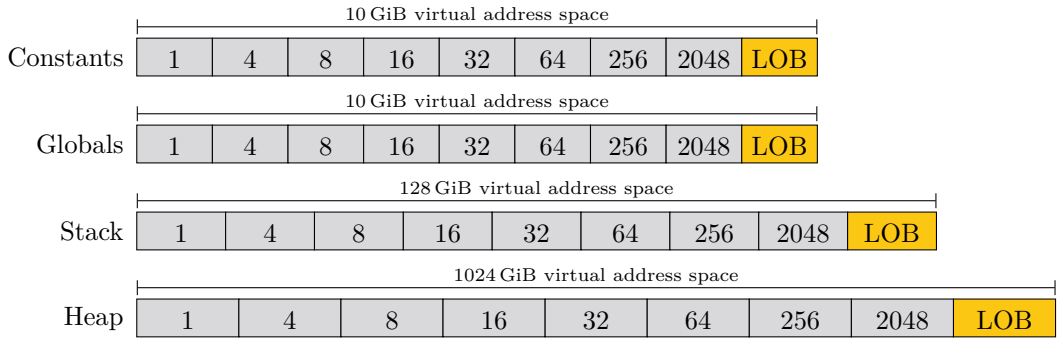


Figure 2 We decouple constants, globals, stack and heap by using different allocator instances. Each of these instances allocate virtual address space in the form of a large `mmap`ed area in main memory that is not backed by physical memory. (Virtual) sizes and base addresses for these are configurable with defaults suitable for common use. Each allocator instance is divided into 9 bins. The first 8 bins manage objects up to a maximum size (1–2048 bytes), while the last one, the large object bin (LOB), manages objects larger than 2048 bytes.

After these assignments, both states go on to execute the `calloc` at Line 6. It is here where the allocator makes a key difference. When a global allocator is being employed, as in KLEE, the two states will always receive *distinct* addresses, $k_1 \neq k_2$. By contrast, KDAlloc *can* return the identical addresses $k_1 = k_2$, as it manages addresses individually for each state. Furthermore, KDAlloc aims to return the same addresses when available, in order to increase stability.

When an external function call is invoked, as at Line 7, the assigned addresses need to be populated with the current set of values, so that external calls can find them in the expected place. In our example, the contents from the internal representation (in yellow) are copied to the concrete space at addresses i (for both states), k_1 (for State 1) and k_2 (for State 2).

In KDAlloc, the process is the same. The difference is that the concrete memory is backed by `mmap` and shared across the states, allowing the same addresses to be used between multiple states without them being related.

4.2 Memory Regions

For each state, KDAlloc maintains four memory regions: one for constants, one for globals, one for the stack and one for the heap. The reason for this separation is to increase stability: For instance, a change in the dynamic allocations should not impact stack allocations and vice versa.

4.3 Object Bins

We further divide these memory regions into bins for objects of a certain size: The first bin is for objects that have a single byte, the second for objects that are larger than 1 byte but not more than 4 bytes, the third one for objects that are larger than 4 bytes but not more than 8 bytes and so on up to objects of no more than 2048 bytes. A separate memory bin called the *Large Object Bin (LOB)* manages objects larger than 2048 bytes. The reason for having per-size bins is again to increase stability: In this way, a change in the allocation of objects within a certain size range does not impact the allocations for objects of other size ranges. In contrast, the reason for having a LOB is that for most programs only a minority of allocations are of large size.

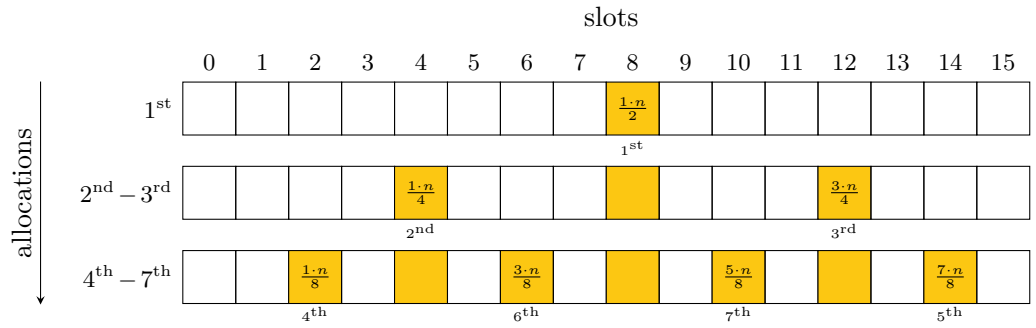


Figure 3 The slot allocator scatters object allocations throughout its bin to maximise inter-object distances. The figure shows a bin with $n = 16$ slots, and 7 allocations.

The bins managed by KDAlloc for each state are illustrated graphically in Figure 2. They are grouped into allocator instances (one each for constants, globals, stack, and heap). We determine the size of per-size bins by dividing the virtual address space assigned to its allocator instance by the total number of bins in the instance (i.e., 9) and rounding down to the next power of two. The LOB takes up the remaining virtual address space. The heap allocator instance, for example, defaults to a region that is subdivided into 8 per-size bins of 64 GiB each, while the LOB spans 512 GiB.

Each bin has its own allocator: the ones handling smaller objects use a *slot allocator* (§4.4), while the LOB uses a *large object allocator* (§4.5).

4.4 Slot Allocator

Each bin except for the LOB uses a slot allocator. A slot allocator divides the bin into n slots of equal size. For instance, a bin for objects of up to size 8 bytes is divided into n equal slots of 8 bytes each.

Typical slot allocators would allocate slots consecutively, to reduce fragmentation. As discussed in §3.4, a more important consideration in DSE is having spatially distanced allocations to prevent that out-of-bounds accesses point to valid objects. A common approach is to add fixed-size *redzones* [19,28] around allocations. Redzones are deliberately unallocated regions that trigger an error on any access. How far beyond the object bounds an access will always be detected as out-of-bounds depends on the redzone size.

As we are not held back by traditional fragmentation concerns, our algorithm aims to maximise the distance between allocations and thus the redzone sizes. The basic idea is to map the slots to a binary tree that is traversed level by level to identify the next free slot. Inside each level, nodes are traversed from the outside to the inside, to prevent the generation of long, monotonically increasing sequences of addresses. The root of this tree is in the middle slot. Every node divides the space of the parent node into two equally-sized spaces with the node itself marking the split.

Figure 3 illustrates this graphically. It depicts a bin with 16 slots, showing how slots are assigned over time. The first allocation is done in the middle slot, slot 8 (row 1 in the figure). Then, the second allocation is done in the middle of the left-side empty region, on slot 4, while the third allocation in the middle of the right-side empty region, on slot 12 (row 2). Finally, allocations 4 to 7 are done in the middle of the remaining regions, at slots 2, 14, 6 and 10 (row 3). With each level, the distance between objects becomes smaller.

On each allocation, this virtual binary tree is checked for slot availability in increasing level order, and the first available slot is always returned. Even if slots are allocated up to a level k, slots on lower levels may become available when objects are freed.

4.5 Large Object Allocator

For large allocations, a slot allocator is not ideal, as finding an appropriate slot size is difficult. If the size chosen is too small, large allocations will not fit, while if the size chosen is very large, most of the slot will usually be left unused.

To address this, our large object allocator takes another approach to manage the LOB. For every allocation, the largest free space is split equally, with the allocation placed at the split point. Similarly to the slot allocator, this algorithm keeps the distance between neighbouring objects as large as possible whenever an allocation occurs. On a deallocation, the freed space is merged with the regions to its left and right and returned to the allocator.

4.6 Quarantine

KDAlloc uses a quarantine [19, 28] to maximise the chance of finding use-after-free errors. That is, when an object is freed, instead of making the freed space available right away, a pointer to the object is instead placed in a quarantine region of a fixed size. When the quarantine becomes full, space is returned on a FIFO basis. This differs from other quarantine implementations in that the condition for releasing an object from the quarantine is not the total size of the quarantined objects, but rather just their number. We chose this method as the quarantine only governs the reuse of addresses and, unlike in quarantine implementations for general purpose allocators, quarantined objects do not take up any space beyond the metadata itself.

Each of the slot and large object allocators uses its own quarantine, to prevent deallocations in different bins from interacting with each other.

Choosing the quarantine sizes involves several trade-offs. Most obviously, larger quarantines take more space. On the other hand, larger quarantines can lead to the detection of more use-after-free errors (see §6.4), particularly those with a larger temporal distance between the deallocation and the invalid use. Quarantines also impact stability. For instance, consider again the example from §3.5, where two states allocate the same objects except that one of them allocates an extra object which is immediately freed. Without a quarantine, after the temporary allocation, the two paths will continue to have identical address spaces. However, with a quarantine the address spaces will start to differ, as the freed address cannot be reused immediately.

5 Implementation

We implemented KDAlloc as an alternative allocator in KLEE [8], a popular open-source DSE engine for LLVM bitcode.[1] Our implementation is written in C++ and has around 2K lines of code (excluding empty lines).

KDAlloc builds on the deterministic allocator from prior work on combining DSE with a partial-order reduction to symbolically analyse multi-threaded programs [27]. While that paper has only a superficial description of the deterministic allocator it uses ([27], §3.4), the allocator is made available as open source.[2]

[1] We use KLEE version 2.2.

[2] `https://github.com/por-se/por-se/blob/5786633/include/pseudoalloc/pseudoalloc.h`

KDAlloc reuses the basic structure of bitmap-based sized bins combined with a region-based large object bin, but significantly improves upon that allocator in many ways: Fork performance and memory usage are significantly improved by the addition of copy-on-write semantics, especially for the large object bin, which has been redesigned to use a singular treap with node-level copy-on-write (§5.5) instead of combining multiple `std::map`s. For the slot allocator, the scatter function has been improved to not generate long runs of monotonically increasing addresses (§4.4) due to visiting the implicit binary tree in level-order. We also completely redesigned how memory pages are returned to the OS after being used in external function calls (§5.3), as exploratory experiments showed the simple idea of always returning all pages to the OS immediately after an external function call leads to significant performance penalties in some cases.

5.1 Allocator Instances

The global and the constant memory regions are each associated with exactly one allocator that does not fork during the lifetime of KLEE, as the memory allocation of constants and globals does not change during the runtime of the program. We distinguish between globals and constants, because globals need to be written out on a per-state basis when an external function call occurs, while constants can be written only once. Note that, just like KLEE, we assume external calls are well-behaved, e.g., we do not implement any techniques to ensure that external function calls do not access memory outside their valid allocations. The heap and stack memory regions are used to initialise the heap and stack allocators when the initial state is created from scratch. After this has been done once, the allocators are forked when the state is forked. The allocator state can be forked efficiently by using a bin-level copy-on-write (CoW) mechanism. Initially, all bins are unallocated. Once an allocation happens, the respective bin is created and owned by the allocator. When a symbolic state is forked, so are the allocator states, which leads to both allocators having a shared CoW reference to each allocated bin. When an object is (de-)allocated in such a shared bin, the bin is copied and after that owned. If only one other allocator references the CoW bin, it regains exclusive access.

KDAlloc uses size information for deallocations, which KLEE provides as part of each memory object. It is not fundamentally necessary to use sized deallocations; if another symbolic execution engine does not store the size of the allocation in a similar manner, the address itself could be used to look up the appropriate bin, at which point a slot allocator knows the size of the allocation by default (1 slot) and the large object allocator can deduce the size by finding the empty spaces before and after the allocated object.

5.2 Virtual Memory Regions

The virtual memory regions used to back external function calls are created at user-provided base addresses. They are mapped as read and write, but non-executable, anonymous, private and without physical backing pages. Similarly, mainline KLEE's deterministic allocator allocates memory in much the same way, except not enforcing the non-backing of pages.[3]

The `fork` call is slow, especially when large amounts of memory are involved [1, 21]. To ensure that the impact of the allocator change on KLEE's usage of `fork` (to decouple solvers from the main process) is as small as possible, the memory region is marked as `MADV_DONTFORK`. This means that the region will not be available in child processes.

[3] `https://github.com/klee/klee/blob/7d85ee8/lib/Core/MemoryManager.cpp#L70-L71`

5.3 External Function Calls

Mainline KLEE creates an initial buffer when a new memory object is created by the program under test, which is used to acquire a memory address that can be used for future external function calls (i.e., calls to uninterpreted functions). When KDAlloc performs an external function call, it copies the data out to the virtual memory region instead. Since the allocator performs allocations with maximal distance, it will usually not share memory pages between allocations. This means, that a one byte allocation will probably require a full page (usually 4096 B) for the duration of the external function call. After the external function call has been performed, any changes are copied back into the symbolic state, and the physical pages are not needed anymore.

To reduce memory consumption, we inform the OS that it may reclaim the physical pages at its leisure by using `madvise` with `MADV_DONTNEED`. In Linux, this operation is, however, relatively costly as its runtime depends on the size of the mapping, not just the number of active pages. As many external function calls only require a comparatively small number of pages, we run `madvise` only once the number of active pages exceeds twice the average number of pages needed for a single external function call (with a minimum of 1024 pages/4 MiB). To compute the average of pages needed for one external call, we use an exponential moving average: $\text{avg}_{i+1} = \frac{3 \cdot \text{avg}_i}{4} + \frac{\text{call}_i}{4}$. We approximate the number of pages needed for a single external function call based on the number of involved objects, by assuming that no two objects share a memory page, which is often the case as we maximise inter-object spacing. The total number of pages can be easily determined by using `getrusage`. We exclude allocations in the global constants mapping from this mechanism, as they do not change in between external function calls.

5.4 The Slot Allocator

The slot allocator is used for allocations smaller or equal to 2048 B (§4.4). It uses two different methods of naming a slot, the *position* and the *index*. The position is the actual address of the allocation relative to the base address of the associated virtual memory region. The index reflects the node position while traversing the binary tree (§4.4) and is used as an index into a bitmap that stores whether the slot is currently allocated or not.

Spreading out allocated objects maximises the distances between allocations. When a maximum of n allocations were live at the same time (including those in quarantine), the distance between any two allocations and between the beginning or end of the bin and any allocation is at least $size/2^{\lceil \log_2(n+1) \rceil} - slotsize$. The slot allocator asserts that the distance will always be greater than zero, which ensures that no two directly adjacent slots will be used, which in turn ensures that a slot-allocator administrating slots of size s will always leave at least s bytes in between any two allocations and to the allocator using the region before it (slot allocators are assigned ascending regions, so it will also have at least s bytes unused after the last allocated slot).

While it may seem like this method wastes a lot of memory to increase robustness, this is only the case when viewed from the program under test and for the duration of external function calls, when objects of a single state are copied into the virtual memory regions. By transforming between position and index, the bitmap is kept compact while the managed objects are spread out.

5.5 The Large Object Allocator

As explained in §4.5, the large object allocator manages a number of free regions and performs allocations in the middle of the largest such free region, splitting it into two. It manages memory in blocks of 4096 B and ensures that at least one such 4096-B block remains unused in between any two allocated objects.

Since objects can be of very different sizes, the large object allocator does not utilise a simple bitmap, but rather uses a treap, i.e., a data structure that is both a search tree and a max heap, to store free regions. The addresses of the free regions are used as keys in the search tree and their sizes are used as priorities to organise the max heap. This enables fast lookups of regions by their address as well as quick identification of the largest one. However, as there may be multiple equally-sized regions, and therefore multiple regions that contend for being the largest one, we also use a perfect hash of the addresses as secondary priorities. Thus, every priority is unique, and the treap assumes a unique shape.

To allocate new memory, the root of the treap (which is also the top of the heap) is removed. The object is situated in the middle of that free region, and the newly generated redzones are reinserted into the treap. If the object is larger than the largest free region, or not large enough to retain redzones of at least 4 KiB before and after, the allocation fails.

To free an allocation, the address of the allocation is used to find the free regions immediately before and after the object to be freed. These regions are removed from the treap and combined back into one region, covering both redzones and the object, before being reinserted.

To save memory, each node of the treap is shared using reference-counted copy-on-write. Since each node contains references to its children, this means that sub-treaps can be shared between multiple allocator states.

5.6 Quarantine

The quarantine is a per-bin FIFO queue implemented as a ring buffer that only allocates when it is needed (i.e., only when the quarantine is neither zero nor infinite) and in use. Each bin uses its own quarantine queue to prevent deallocations in different bins from flushing previous deallocations in otherwise unrelated bins. This trade-off comes at a slight increase in memory overhead, as each bin needs a k-element quarantine buffer to give a global guarantee that at least the last k deallocations are being delayed by the quarantine.

In §4.6, we discussed the various trade-offs involved in choosing the quarantine sizes. Based on initial experiments, we have decided on a default value of 8 slots per quarantine, but kept the size configurable.

6 Evaluation

In this section, we evaluate the overheads and benefits of KDAlloc. After presenting the experimental setup (§6.1), we measure the memory overhead and performance impact of KDAlloc by comparing it to the default allocator of mainline KLEE (§6.2) and explore the root cause of the solver time improvements that were observed in some of the benchmarks (§6.3). We then evaluate the benefits KDAlloc provides in the context of use-after-free error detection (§6.4) and two KLEE-based projects: MoKlee (§6.5) and SymLive (§6.6).

Table 1 KLEE benchmarks.

Suite	Version	Apps
GNU awk	5.1.0	`awk`
GNU bc	1.07.1	`bc`
GNU Binutils	2.35.1	9 tools[4]
GNU Coreutils	8.32	68 tools
GNU datamash	1.7	`datamash`
GNU Diffutils	3.7	`diff`
GNU Findutils	4.7.0	`find`
Libtasn1	4.16.0	`asn1Decoding`
libTIFF	4.1.0	`tiffdump`, driver[5]
libxml2	2.9.10	driver[6]
GNU M4	1.4.18	`m4`
GNU Make	4.3	`make`
ImageMagick	7.0.10-45	`magick`
oSIP	5.2.0	driver[5]
GNU sed	4.8	`sed`
SQLite	3340000	`sqlite3`
tcpdump	4.9.3	`tcpdump`
Vorbis Tools	1.4.0	`oggenc`

Table 2 SymLive benchmarks.

Suite	Version	Apps
BusyBox	1.27.2	`hush`, `sed`, `yes`
GNU Coreutils	8.25	`ptx`, `tail`, `yes`
GNU regex	0.12	driver
GNU sed	4.4	`sed`
Toybox	0.7.5	`sed`, `yes`

Table 3 MoKlee benchmarks.

Suite	Version	Apps
GNU Binutils	2.33	`readelf`
GNU Coreutils	8.31	87 tools
GNU Diffutils	3.7	`diff`
GNU Findutils	4.7.0	`find`
GNU Grep	3.3	`grep`
libspng	`#2079ef6`	driver
tcpdump	4.9.3	`tcpdump`

6.1 Experimental Setup

To minimise the non-deterministic impact of the environment, we ran all experiments on a cluster of homogeneous machines (Intel Core i7-4790 @ 3.6 GHz with 16 GiB RAM) with an equivalent OS/library setup (Ubuntu 18.04). Each experiment is executed inside a Docker container that allows reproducing a similar execution setup on each machine. For mainline KLEE and SymLive we use LLVM 11.0 and Z3 4.8.8, whereas MoKlee is linked against LLVM 3.8 and uses Z3 4.8.4.

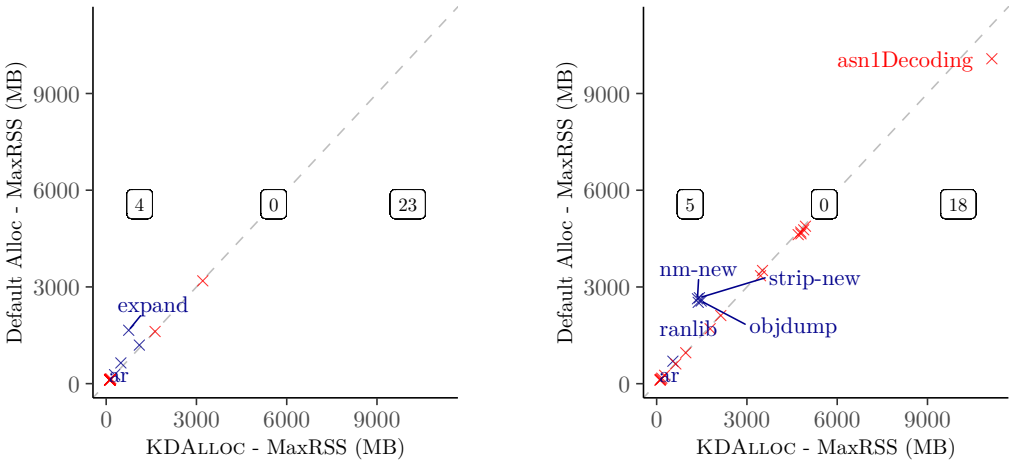
All of our experiments are available at `https://doi.org/10.5281/zenodo.6540857`.

6.2 Memory Consumption and Performance

We implemented KDAlloc as an additional allocator in mainline KLEE 2.2 and tested it against a diverse set of 18 benchmark suites ranging from basic system utilities to databases or image processing tools. From each suite, we chose the most representative applications and test drivers made available in recent KLEE-related publications. In total, we tested 94 applications (Table 1). However, we only used a subset of the GNU Coreutils applications: We excluded all applications that interfere with our test setup (e.g. `chmod`, `truncate`), crash KLEE due to a known bug in the Z3 front-end (e.g. `ptx`), are very similar or aliases to other tools (e.g. `dir`), or are not compatible with our deterministic thresholds described below (e.g. `fmt`).

Comparing two alternative implementations is a non-trivial endeavour, especially with EGT-style symbolic execution engines. Every change in exploration, memory allocation, environment, state termination due to memory pressure etc. can cause completely different executions and invalidate the comparison. Hence, it is of utmost importance to eliminate

[4] Binutils: `addr2line`, `ar`, `elfedit`, `nm`, `objdump`, `ranlib`, `readelf`, `size`, `strip`
[5] Driver from: `https://figshare.com/articles/code/ESEC_FSE_2020_PSPA_artifact/12410231`
[6] Driver from: `https://github.com/davidtr1037/klee-mm-benchmarks`



(a) DFS – 4 points above the diagonal, 23 points below the diagonal and none on the diagonal.

(b) RndCov – 5 points above the diagonal, 18 points below the diagonal and none on the diagonal.

Figure 4 MaxRSS for KDAlloc vs the default allocator using different searchers for runs that are deterministic. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue and points below the diagonal are red. The number of points above, below and on the diagonals are noted in each graph.

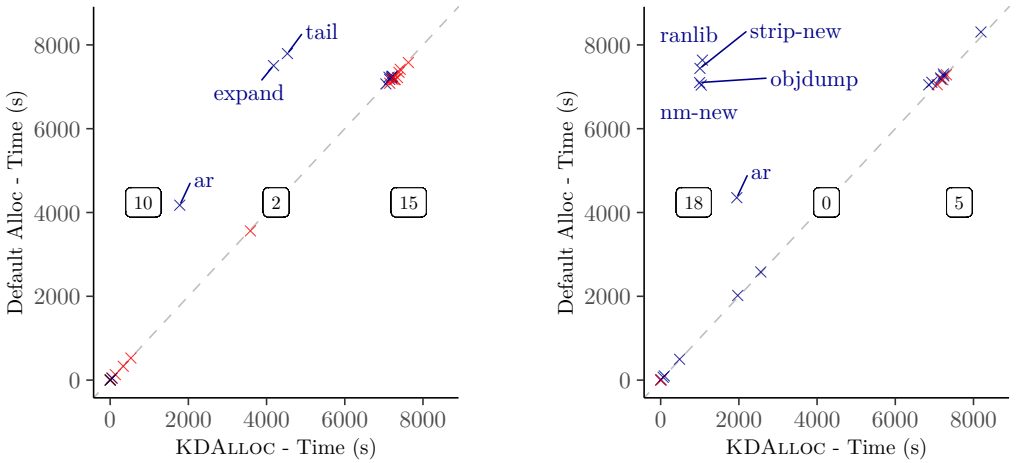
most sources of non-determinism. We tried to mitigate the issue by slightly modifying KLEE: First, we replaced all timer-based thresholds by instruction-based thresholds. Second, we similarly replaced memory-based thresholds by thresholds based on the number of states. Third, we added more statistics, such as the number of allocations or external function calls, to evaluate and compare executions.

Initially, we ran each application for 2 h with the default 2 GB memory limit, unmodified thresholds, and the default allocator. From these runs we derived suitable values for the more deterministic thresholds for each application. Instead of running experiments for 2 h with a 2 GB memory limit and logging intervals of e.g. 30 s, we can now use much more precise descriptions and run an application for n instructions, re-compute coverage every x instructions, update logs every y instructions and terminate states when we reach a maximum of z states.

For a few applications such as `fmt`, we were not able to find a working threshold for the maximum number of active states, as KLEE's performance for these applications degrades from 100k active states to 1-10 states over time. A low threshold leads to an immediate termination of KLEE whereas a high threshold causes memory exhaustion before the targeted run time.

After acquiring the thresholds for KLEE's default exploration strategy (RndCov), a combination of random-path traversal and a distance-based approach to target uncovered code [8], we repeated the process for the simpler depth-first search strategy (DFS).

With these more deterministic configurations, we ran each experiment with both exploration strategies (DFS/RndCov) and both allocators five times. We consider an application deterministic (or *comparable*) under a search strategy when all ten runs, five for each allocator, show the same values for core statistics, in particular the same number of instructions, covered instructions, allocations, queries, and external calls.



(a) DFS – 10 points above the diagonal, 15 points below the diagonal and 2 on the diagonal.

(b) RndCov – 18 points above the diagonal, 5 points below the diagonal and none on the diagonal.

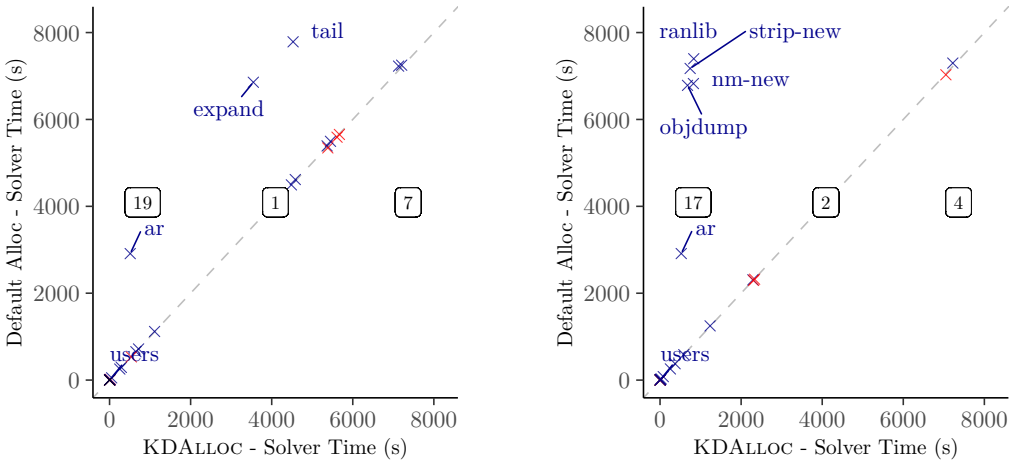
Figure 5 Execution time for KDAlloc vs the default allocator using different searchers for runs that are deterministic. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue and points below the diagonal are red. The number of points above, below and on the diagonals are noted in each graph.

To understand the memory consumption of KDAlloc, we recorded the maximum resident set size (MaxRSS) for all runs, as reported by the Linux kernel. As seen in Figure 4a for DFS and Figure 4b for RndCov, the two allocators generally use the same amount of memory. There are several outliers where the differences are slightly larger (but still small): For most of them KDAlloc consumes less memory, but there is also one case (`asn1Decoding` for RndCov) where KDAlloc consumes more. The reason KDAlloc can consume more memory is through its more expensive metadata when many small allocations are involved. On the other hand, it can consume less memory since it only uses the buffers corresponding to the addresses it returns when an external call is performed (in contrast, the default allocator uses `malloc` to reserve that space).

To better understand the nature of our benchmarks, we measured several statistics across all benchmarks and both search heuristics. KDAlloc had to handle up to 758M allocations (median 11M), allocate up to 5157 MiB (median 97 MiB), and manage up to 5.5M live allocations (median 9154) and 412 MiB (median 625 KiB) of live memory in a single experiment run. A single state during symbolic execution could perform up to 758M allocations (`yes` with DFS) (median 2706), allocate up to 4896 MiB (median 57 KiB), and keep 7899 allocations (median 640) or 134 MiB (median 16 KiB) live.

In Figure 5a (DFS) and Figure 5b (RndCov) we compare the execution time when KDAlloc is used (x-axis) with the execution time when the default allocator is used (y-axis). For most of the tested applications, the execution time is similar. This shows that KDAlloc does not impose any significant performance overhead. In contrast, all outliers (with a relative difference of at least 10%) are above the diagonal indicating that KDAlloc can lead to significant speedups for certain benchmarks.

To explain the potential cause of those positive outliers, we analysed the solving time, shown in Figure 6a for DFS and Figure 6b for RndCov. The graphs confirm that the main reason for the speedup achieved by KDAlloc for those benchmarks is the time spent solving constraints.



(a) DFS – 19 points above the diagonal, 7 points below the diagonal and 1 on the diagonal.

(b) RndCov – 17 points above the diagonal, 4 points below the diagonal and 2 on the diagonal.

Figure 6 Solver time using different searchers for runs that are deterministic with and without KDAlloc. Points with relative difference of at least 10% are labelled. Points above the diagonal are blue, points below the diagonal red and points exactly on the diagonal are black. The number of points above, below and on the diagonals are noted in each graph.

6.3 Solver Time Improvements

At the beginning, we suspected a higher cache-hit rate as the main reason for KDAlloc speeding up constraint solving. Due to higher stability (§3.5), KDAlloc leads to more hits in KLEE's caches (the query cache and the counterexample cache [8, 22]). If query expressions share common addresses, we may have more identical queries and solutions are also more likely to be reused, which can have a great impact on caching efficacy [20, 31]. Indeed, the outlier benchmarks show an increased number of hits. However, the relative difference is small, at just over 0.01% of all queries, and did not explain the overall speedup.

Therefore, we hypothesised that the different addresses returned by KDAlloc may make a large number of queries easier to solve. To investigate this, we tracked all the solver calls for KDAlloc and the default allocator and compared their execution time for the `expand` benchmark (DFS).

In both configurations, KLEE issued a total of 1,274,757 queries. Of these, 758,079 queries were different, depending on the allocator. We could match all but 2604 of those queries by mapping the constants that relate to the non-deterministic addresses returned by the default allocator to the corresponding constants that relate to the deterministic addresses returned by KDAlloc. The remaining differences were minor and due to sources of non-determinism that were neither addressed by our work nor had an impact on the deterministic execution of `expand`.

While for each individual query the absolute time difference is only a few milliseconds, many of the queries with deterministic addresses can be solved roughly twice as fast as their non-deterministic counterpart. With such a large number of queries, these differences add up and explain the overall solver speedup.

We manually inspected the 100 queries that show the most significant absolute time improvement. These queries involve bounds checks and constraints on single bytes of a symbolic file (originating from `expand`'s main loop) and show high similarity. For these

```
(Extract w32 0
    (Add w64 0xFFFFDDBC00000000 (Select w64 C 0x0000000000000000 0x0000224400000000)))
↓ Extract(Add): (Extract (Add x y)) → (Add (Extract x) (Extract y))
(Add w32 (Extract w32 0 0xFFFFDDBC00000000)
    (Extract w32 0 (Select w64 C 0x0000000000000000 0x0000224400000000)))
(Extract w32 0 0xFFFFDDBC00000000) → 0x00000000 and ↓ Extract(Select)
(Add w32 0x00000000
    (Select w32 C (Extract w32 0 0x0000000000000000) (Extract w32 0 0x0000224400000000)))
(Extract w32 0 0x0000000000000000) → 0x00000000
(Extract w32 0 0x0000224400000000) → 0x00000000
(Add w32 0x00000000 (Select w32 C 0x00000000 0x00000000))
(Select w32 C 0x00000000 0x00000000) → 0x00000000
(Add w32 0x00000000 0x00000000) = 0x00000000
```

(a) KDAlloc's address structure can enable offset reasoning independent of base addresses.

```
(Extract w32 0
    (Add w64 0xFFFFAAAAA7290C00 (Select w64 C 0x0000000000000000 0x0000555558D6F400)))
↓ Extract(Add): (Extract (Add x y)) → (Add (Extract x) (Extract y))
(Add w32 (Extract w32 0 0xFFFFAAAAA7290C00)
    (Extract w32 0 (Select w64 C 0x0000000000000000 0x0000555558D6F400)))
(Extract w32 0 0xFFFFAAAAA7290C00) → 0xA7290C00 and ↓ Extract(Select)
(Add w32 0xA7290C00
    (Select w32 C (Extract w32 0 0x0000000000000000) (Extract w32 0 0x0000555558D6F400)))
(Extract w32 0 0x0000000000000000) → 0x00000000
(Extract w32 0 0x0000555558D6F400) → 0x58D6F400
(Add w32 0xA7290C00 (Select w32 C 0x00000000 0x58D6F400))
```

(b) The same simplification is not possible with address-dependent constants from the default allocator.

Figure 7 Example of query simplification enabled by KDAlloc. Steps performed by Z3's pre-processing stage are shown in red. The highlighted simplification is only observed with KDAlloc.

queries, we found that the time differences can be explained by the initial pre-processing stage of Z3 [12], which was able to greatly simplify the queries generated when using KDAlloc before invoking the core solver.

The top of Figure 7a shows a fragment of one of the queries that depends on the allocator used and where the address-dependent constants, `0xFFFFDDBC00000000` and `0x224400000000`, were obtained using KDAlloc. The top of Figure 7b shows the same fragment, but with the address-dependent constants stemming from the default allocator: `0xFFFFAAAAA7290C00` and `0x555558D6F400`. The queries are given in the KQuery format,[7] where `(Select [width] [condition] [true-expr] [false-expr])` is an *if-then-else* operation that evaluates to either `true-expr` or `false-expr` (both of the same given `width`) depending on whether `condition` evaluates to *true* or *false*. The `(Extract [width] [index] [expr])` operation evaluates to the `width` least-significant bits taken from `expr`, omitting the first `index` bits. In both fragments, we omit a complex expression denoted by `C`.

In these fragments, only the lower 32 bits of the result are significant (`Extract w32 0`). Using simple rewriting rules, the `Extract` operation is pushed down to the constants. In the KDAlloc case, this simplifies the expression enough to remove the `Select`, leaving a single constant. In the case of the default allocator, since the two operands that are to be chosen based on `C` are not the same, the `Select` cannot be removed.

[7] `https://klee.github.io/docs/kquery/`

Listing 1 Whereas KLEE with KDAlloc and a quarantine can detect the use after free at Line 9 reliably, KLEE with the default allocator can only detect it when the freed space is not reused in the meantime, e.g. by the second `strdup` or internal KLEE data structures.

```
1  char *mallocfree() {
2      char *s = strdup("A");
3      free(s);
4      char *t = strdup("B");
5      return s;
6  }
7  int main(void) {
8      char *s = mallocfree();
9      puts(s);
10     return 0;
11 }
```

Why do addresses issued by KDAlloc follow those specific patterns? First, the virtual address spaces are configured to start at addresses with all of the lower 32 bits set to zero. Next, the way we size our bins (see §4.3), along with the algorithm we use to assign slots (see §4.4), results in base addresses that are a sum of high powers of two for every object. As a result, all enquiries about the lower 32 bits of pointers can trivially be rewritten into reasoning about offsets, without involving the base address.

We also confirmed that STP [15] (another major solver used by KLEE) is able to perform a similar simplification during its initial pre-processing stage and shows similar improvements in solving time. We can thus conclude that the addresses returned by KDAlloc can have a positive effect on query solving time apart from caching.

6.4 Detection of Use-after-free Errors

KLEE can find many memory access violations. Its detection, however, is primarily focused on out-of-bounds accesses. To detect these, KLEE checks for every memory access whether the address does not point to a valid object, or in the case of a symbolic address whether it is possible not to point to a valid object. In such a case, an out-of-bounds error is reported.

If the program under test frees an allocation, its entry is removed from the list of valid objects in the current state. While many use-after-free (and the related double-free) errors can be found using this mechanism (i.e. the address does not resolve to a valid object), KLEE has been shown to sometimes miss even the simplest cases[8] or to depend critically on subtle aspects such as compilation flags[9] for their detection.

KLEE's use-after-free error detection is fragile as it depends on the freed space not being reused by a subsequent allocation. An example of this is shown in Listing 1. Without compiler optimisations enabled, KLEE usually fails to detect the use after free at Line 9, as the underlying default allocator usually reuses the address for the allocation of the string `"B"` or one of the local variables in the `strdup` or `puts` calls. With Line 4 removed or optimisations enabled, KLEE is able to find the bug.

KDAlloc, on the other hand, provides a quarantine for deallocated objects (§4.6). As long as objects remain quarantined, any use-after-free error is guaranteed to be detected. In addition, KDAlloc greatly enhances the comprehensibility of reported use-after-free errors, which in baseline KLEE are reported as out-of-bounds accesses. If an address is not resolved to a valid object, but the location is still marked as allocated in the allocator metadata, we conclude that it must be in quarantine, and thus it is a use-after-free error.

[8] `https://www.mail-archive.com/klee-dev@imperial.ac.uk/msg02998.html`
[9] `https://github.com/klee/klee/issues/1434`

Listing 2 An implementation of the ANSI C function `memmove` as found in the uClibc[10] library. The pointer values in the comparison at Line 4 depend on the values returned by the underlying allocator. The default allocator returned different values in the re-execution and caused a divergence.

```
1  void *memmove(void *dest, const void *src, size_t n) {
2    char *s = (char *) dest;
3    const char *p = (const char *) src;
4    if (p >= s) {
5      while (n) {
6        *s++ = *p++;
7        --n;
8      }
9    } else {
10     while (n) {
11       --n;
12       s[n] = p[n];
13     }
14   }
15
16   return dest;
17 }
```

Implementing a quarantine for the default allocator is possible [28], but would incur a significant space penalty for several reasons. First, only a small portion of the quarantined objects are actually objects from the program under test, as most are allocated by the DSE engine. Thus, the quarantine would have to be several times as large to provide similar benefits. Second, this would be a global quarantine, meaning that it would have to be even larger to provide similar per-state guarantees, even if the symbolic states are visited uniformly. Finally, such a quarantine causes memory fragmentation, as memory cannot be reused immediately; in the case of KDAlloc, this causes no additional memory pressure, as it only fragments the emulated address space of the program under test.

6.5 MoKlee

MoKlee [7] is an extension of KLEE implementing a variant of memoised symbolic execution [33]. MoKlee provides the ability to save an ongoing DSE run to disk and then to (partially) restore it back into memory via a fast replay process. More exactly, MoKlee saves to disk metadata, such as constraint solver results and path information, and re-uses this information during replay to remove the constraint solving cost. MoKlee can optionally filter out fully explored path subtrees, in a mode called *path pruning*.

This approach is applicable to real-world software as long as the engine is able to detect divergences [7]. Divergences occur when a DSE engine explores different code paths in different runs, although the controllable inputs are the same. This happens due to values that are read from the environment (e.g. date/time strings or disk usage) or, more relevant for our approach, when the execution relies on memory addresses. For instance, the branches taken to traverse a hash table with pointer values as keys depend on the addresses returned by the underlying allocator. If the allocator is non-deterministic and the allocation order changes in

[10] https://www.uclibc.org/

subsequent runs, the insertion order in the hash table changes and branch decisions cannot be re-used. KLEE's `mmap`-based deterministic allocator is shown to reduce divergences significantly, e.g. for `find`, but it is not suitable for most applications as it cannot free or re-use memory [7].

We ported KDAlloc to MoKlee, and added some statistics (e.g. number of external calls) and an instruction-based threshold for the coverage computation. Most of the other changes that were necessary for mainline KLEE could be mimicked by similar flags that were already available in MoKlee. As benchmarks we re-used the 93 applications (Table 3) provided with the MoKlee artifact.[11] The setup is similar to the mainline KLEE experiment: We start with 2 h memoisation runs using a 2 GB memory limit, as described in the MoKlee artifact, to find deterministic thresholds for the two exploration strategies (RndCov, DFS). After that, we re-run the memoisation pass for all applications with both exploration strategies and *both* allocators, but using deterministic thresholds. A total of 66 applications for DFS, respectively 38 applications for RndCov, had the same relevant statistics and hence executed the same paths with high probability across both allocators. In short, these runs are *comparable*.

To measure the influence of both allocators on divergences, we re-executed all memoised runs with and without path pruning using both exploration strategies as done in the MoKlee paper. We only omit the results for the non-pruning re-execution of memoised DFS runs with the RndCov search strategy, as the wide-and-deep shape of DFS execution trees often cause state explosions and hence state terminations in many benchmarks, making a comparison meaningless.

We refer to each experiment by the search strategies used during the memoised run and during replay. For instance DFS/RndCov denotes the experiment where the original memoised run used DFS, and the replayed run used RndCov.

Firstly, we evaluate the experiments based on comparable memoisation runs. Due to the nature of these tools, the number of divergences is low and we only observed them in three applications when re-executed without path pruning: `nohup` (RndCov/DFS) diverges in a comparison that checks the maximum number of open file descriptors, whereas `shred` (RndCov/DFS) diverges in a comparison of timestamps. Both applications diverge in exactly the same locations with both allocators and the root causes cannot be prevented by our allocator. However, `du` (DFS/DFS) only diverges with the default allocator. The reason for that is a pointer comparison in the implementation of the standard C function `memmove` (Listing 2) to copy bytes between memory areas. Memory areas are allowed to overlap, in contrast to `memcpy`, and most C libraries save a temporary buffer by comparing the source and destination pointers (Line 4). Depending on their order, the algorithm starts copying either from the front or the back of the areas to prevent overwriting the overlapping area too early. In our experiment, the default allocator returned different addresses for `p` and `s` during the re-execution, changing the order of both pointers in memory and causing a divergence that got detected by MoKlee. When a divergence occurs, MoKlee removes the memoised subtree and the affected subtree needs to be re-explored. In the case of `du`, a significant 32.5% of the memoised instructions were lost that way. With KDAlloc's deterministic allocation on the other hand, both pointers retrieved the same values during re-execution and the complete memoised run could be re-used.

[11] `https://zenodo.org/record/3895271`

Table 4 Number of source locations with diverging behaviour in applications with differing memoisation runs across allocators. Different numbers are observed between allocators for the first three benchmark suites.

	DFS		RndCov	
Suite	MoKlee	KDAlloc	MoKlee	KDAlloc
Coreutils	22	12	42	32
Findutils	1	0	1	1
Libspng	0	0	1	0
Binutils	0	0	0	0
Diffutils	0	0	0	0
Grep	0	0	1	1
Tcpdump	0	0	0	0

Secondly, we evaluate the remaining non-comparable applications where the memoisation runs between allocators differ. Here, we count the number of unique source locations where divergences occur across re-execution runs (with and without path pruning). As can be seen in Table 4, with KDAlloc divergences occur in significantly fewer locations for both search strategies, showing that KDAlloc is effective in preventing memory-related divergences.

Furthermore, in the original MoKlee paper `find` benefited most from a deterministic allocator. Our experiments confirm that observation. Starting from a RndCov memoisation run and re-executing that run without path pruning, KLEE has to terminate 3407 paths due to divergences with the RndCov exploration strategy (3545 for DFS) whereas KDAlloc does not observe a single divergence with RndCov and only 3 with DFS. In summary, KDAlloc reduces the number of diverging paths and hence improves the effectiveness of memoised symbolic execution.

6.6 SymLive

SymLive [26] is an open-source[12] KLEE extension that uses symbolic execution to find paths that lead to infinite loops. It hashes the symbolic states and checks for repeating hashes along each path. If such a repetition is found, the program transitions back into a previous program state: Assuming deterministic program execution, the program under test has entered an infinite loop, which SymLive reports if it violates a generic liveness property.

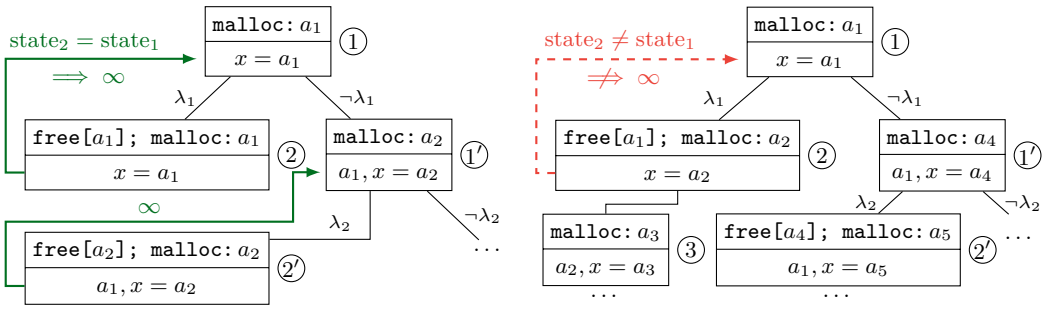
The original implementation of SymLive employs KLEE's default allocator, which is good enough in many practical applications [26]. The default allocator, however, is not cross-path deterministic (§3.3), which may prevent SymLive from detecting an infinite loop in one state if another state prevents address reuse.

The program in Listing 3 is one such example. It conditionally leaks allocations and prevents forked states from terminating. Its infinite behaviour can be reliably detected using KDAlloc with the quarantine disabled (the desired configuration in this context), but not using the default allocator. The example revolves around two pointers, `x` and `p`. The former is initially bound to a memory object, the latter is made symbolic (Line 2). The infinite loop (Lines 3–6) reassigns `x` with a freshly allocated memory object (Line 5). If `x == p`, the previous pointee is freed just before (Line 4); otherwise it is leaked.

[12] `https://github.com/COMSYS/SymbolicLivenessAnalysis`

Listing 3 Leaking memory objects in an infinite loop can hinder its detection in SymLive without cross-path deterministic allocation.

```
1  int main(void) {
2    void *x = malloc(1), *p = klee_symbolic_ptr();
3    while (1) {
4      if (x == p) free(x);
5      x = malloc(1);
6    }
7  }
```



(a) SymLive with KDAlloc. **(b)** SymLive with default allocator.

Figure 8 States observed by SymLive, using (a) KDAlloc or (b) default allocation, while executing the example from Listing 3. States are shown with heap operations (top) and allocated addresses (bottom).

Following the `then` branch keeps the number of memory objects stable; following the `else` branch leads to one leaked memory object per iteration. All paths along which `p` eventually equals a pointer returned from the `malloc` at Line 5 can form an infinite loop, where that pointer is repeatedly deallocated and allocated again at the same address. As only this address is ever deallocated, this loop requires immediate address reuse (similar programs can be crafted for any finite delay in address reuse). KDAlloc can easily detect this liveness violation when the quarantine is disabled, while KLEE's default allocator fails due to cross-path interference.

In Figure 8, we visualise the different behaviours of SymLive with KDAlloc (Figure 8a) and the default allocator (Figure 8b). In both cases, state (1) corresponds to the initial `malloc` (Line 2), and shows a_1, the returned address. The symbolic state is then forked at the branch in Line 4. If the branching condition is false, the new state, state (1′), differs from state (1) only by one leaked object and the constraint from the symbolic fork. On the other hand, if the condition is true, the behaviour differs between the two allocators.

In the case of KDAlloc, the allocator's internal state was forked along with the state. Thus, state (2) can reuse a_1 after freeing it, even though the address is still allocated in other states. State (2) therefore compares equal to state (1) and the infinite loop is detected.

In contrast, the default allocator uses a single, global allocator instance, so a_1 can only be reused once it has been deallocated in every symbolic state. Since the addresses are leaked, this never happens, and state (2) therefore cannot compare equal to state (1), thus no infinite loop will be detected. As a result, execution continues into state (3) (and beyond), with `malloc` returning another address in each iteration. Irrespective of the allocator, state (1′) behaves the same as state (1), as the leaked address has no further impact until all possible

allocations have been performed. Thus, with KDAlloc states (1′) and (2′) compare as equal, as do equivalent states on any further path. However, with the default allocator, states (1′) and (2′) do *not* compare as equal, nor do equivalent states on any further path.

While this example is crafted specifically to showcase the importance of cross-path determinism, it is plausible that non-deterministic allocation can cause infinite-loop bugs to be missed or at least delay their detection significantly in real-world programs. While this impact is hard to measure, given its low overhead, it is preferable to use KDAlloc with SymLive to enable or speed up infinite-loop detection.

To confirm that KDAlloc does not break any other (possibly implicit) requirement of SymLive, we ran it with KDAlloc on the applications from SymLive's artifact[13] (Table 2) where it was able to detect infinite-loop bugs. For these experiments we stayed close to the original configuration and used a memory limit of 10 GB. The experiments include `toybox`, a package that re-implements (among others) many Coreutils and `sed`. Repeating these experiments, we quickly noticed a problem with `toybox`, which failed with KDAlloc. The reason for this turned out to be simple: `toybox` uses a dispatch mechanism that allows users to call many tools through a single binary, e.g. `./toybox sed`. As part of this mechanism, `toybox` implements a safeguard that tries to detect whether the stack depth is too high. This detection, however, is implemented by comparing the integer representation of two pointers to stack variables from different stack frames, which gives an implementation-defined result as per the C standard [14]. In our design, which is fully standard-compliant in this regard, we do not follow a linear, stack-like order when allocating such variables. Additionally, we try to maximise redzones around each allocation. Together, this leads `toybox`'s safeguard to bail out when we use KDAlloc. All `toybox` utilities can also be built as standalone binaries, avoiding the dispatch along with its problematic safeguard. We used this (deviating from the original setup) for all our `toybox` runs.

Our extension was able to find infinite loops in all applications with the default exploration strategy. For the various `sed` implementations, infinite loops were only found for the shorter of two possible symbolic inputs, as, with the longer inputs, SymLive runs into memory and time limits (set to 24 h) irrespective of the allocator. We refrained from further exploring which allocator works better in that case, as KLEE frees up memory by (non-deterministically) terminating states once its memory limit is reached.

7 Related Work

As a core concept in programming, memory allocation is a well-researched topic [2,13,17,18,30]. In fact, many of the building blocks used to create KDAlloc are well-known methods in this area of research. For example, separating allocations by size to quickly find the best-fitting unallocated region [17], combining multiple equally-sized objects into one large run [13], using bitmaps to denote allocated/free slots in a larger run of equally-sized objects [3, 13], spacing objects further apart in memory [3, 25, 28], delaying deallocations in a quarantine zone [25, 28], and segregating heap data and metadata [3] are all well-established techniques for designing memory allocators.

However, to the best of our knowledge, these building blocks have never been combined in such a way, as to fit and support EGT-style dynamic symbolic execution. The closest work we see is SymMMU [24], which separates the dispatch mechanism for memory accesses in DSE from its handling policy. However, unlike KDAlloc, SymMMU is not directly concerned with determinism and stability.

[13] `https://doi.org/10.5281/zenodo.5771192`

Common memory-safety checkers, such as AddressSanitizer [28] or Valgrind Memcheck [29] have largely the same goals w.r.t. error detection via spatial and temporal distancing, but, while cross-run determinism is of some interest, they have no notion of multiple paths, which leads them to not consider cross-path determinism, stability, or a method for efficiently forking the allocator state. KDAlloc takes inspiration from these memory-safety checkers to detect faults with spatial and temporal distancing, while also considering our remaining goals, trading away performance and delegating error detection to the underlying DSE engine.

Similar questions and solutions arise when considering fault tolerance in addition to fault detection. In Rx [23], memory errors in a process are detected and as one potential mitigation strategy, allocations are moved to different locations on recovery to avoid subsequent crashes. The Windows Fault Tolerant Heap (FTH) [25] is automatically enabled when a program shows behaviour related to faults in dynamic memory management. It mitigates potential problems by utilising a quarantine and additional space between objects. Similarly, DieHard [3] randomises the heap layout over a large region to probabilistically increase both spatial and temporal distance between allocated objects.

8 Conclusion

In this paper, we show that the memory allocator can have a significant impact in dynamic symbolic execution (DSE). We first identify six key design principles – support for external calls, cross-run and cross-path determinism, spatially and temporally distanced allocations, and stability – and propose KDAlloc, a memory allocator specifically designed for DSE, whose design is guided by these principles.

We implemented KDAlloc in KLEE, a popular DSE engine, and show that it has a neutral or positive impact on memory consumption and performance, while improving use-after-free error detection and several DSE-based techniques such as MoKlee, an approach for saving DSE runs to disk and later (partially) restoring them, and SymLive, an approach for finding infinite-loop bugs.

References

1 Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *Proc. of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)*, May 2019.

2 Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, November 2000.

3 Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'06)*, June 2006.

4 Peter Boonstoppel, Cristian Cadar, and Dawson Engler. RWset: Attacking path explosion in constraint-based test generation. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.

5 Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Memory models in symbolic execution: key ideas and new thoughts. *Software Testing, Verification and Reliability*, 29(8), 2019. doi:10.1002/stvr.1722.

6 Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)*, September 2008.

7 Frank Busse, Martin Nowack, and Cristian Cadar. Running symbolic execution forever. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

8 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, December 2008.

9 Cristian Cadar and Dawson Engler. Execution generated test cases: How to make systems code crash itself. In *Proc. of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05)*, August 2005. doi:10.1007/11537328_2.

10 Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, October 2006. doi:10.1145/1455518.1455522.

11 Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)*, 56(2):82–90, 2013. doi:10.1145/2408776.2408795.

12 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, March-April 2008.

13 Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proc. of the 2006 BSDCan Conference (BSDCan'06)*, May 2006.

14 International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, December 1999.

15 Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)*, July 2007.

16 Timotej Kapus and Cristian Cadar. A segmented memory model for symbolic execution. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, August 2019.

17 Doug Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

18 Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'04)*, June 2004.

19 Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

20 Martin Nowack. Fine-grain memory object representation in symbolic execution. In *Proc. of the 34th IEEE International Conference on Automated Software Engineering (ASE'19)*, November 2019.

21 Martin Nowack, Katja Tietze, and Christof Fetzer. Parallel symbolic execution: Merging in-flight requests. In *Proc. of the Haifa Verification Conference (HVC'15)*, December 2015.

22 Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*, July 2013.

23 Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proc. of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, October 2005.

24 Anthony Romano and Dawson Engler. SymMMU: Symbolically executed runtime libraries for symbolic memory access. In *Proc. of the 29th IEEE International Conference on Automated Software Engineering (ASE'14)*, September 2014.

25 Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows®Internals, Part 2*. Microsoft Press, 6th edition, September 2012.

26 Daniel Schemmel, Julian Büning, Oscar Soria Dustmann, Thomas Noll, and Klaus Wehrle. Symbolic liveness analysis of real-world software. In *Proc. of the 30th International Conference on Computer-Aided Verification (CAV'18)*, July 2018.

27 Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. Symbolic partial-order execution for testing multi-threaded programs. In *Proc. of the 32nd International Conference on Computer-Aided Verification (CAV'20)*, July 2020.

28 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Address-Sanitizer: A fast address sanity checker. In *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, June 2012.

29 Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*, April 2005.

30 Matthias Springer and Hidehiko Masuhara. DynaSOAr: A parallel memory allocator for object-oriented programming on GPUs with efficient memory access. In *Proc. of the 33rd European Conference on Object-Oriented Programming (ECOOP'19)*, July 2019.

31 David Trabish, Shachar Itzhaky, and Noam Rinetzky. Address-aware query caching for symbolic execution. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'21)*, April 2021.

32 David Trabish and Noam Rinetzky. Relocatable addressing model for symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'20)*, July 2020.

33 Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)*, July 2012.