KDAlloc: The KLEE Deterministic Allocator*

Deterministic Memory Allocation during Symbolic Execution and Test Case Replay

Daniel Schemmel
d.schemmel@imperial.ac.uk
Imperial College London
United Kingdom

Julian Büning
buening@comsys.rwth-aachen.de
RWTH Aachen University
Germany

Frank Busse
f.busse@imperial.ac.uk
Imperial College London
United Kingdom

Martin Nowack
m.nowack@imperial.ac.uk
Imperial College London
United Kingdom

Cristian Cadar
c.cadar@imperial.ac.uk
Imperial College London
United Kingdom

ABSTRACT

The memory allocator can have an important impact in symbolic execution. Taking a user-centric view, this tool demonstration paper discusses some of the main benefits provided by KLEE's new allocator KDAlloc in terms of improved deterministic execution and bug-finding capabilities. We then introduce a new replay tool for KLEE which enables the native execution to integrate KDAlloc and receive the same heap addresses as during symbolic execution.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Symbolic Execution, test case replay, memory allocation

ACM Reference Format:
Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2023. KDAlloc: The KLEE Deterministic Allocator: Deterministic Memory Allocation during Symbolic Execution and Test Case Replay. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604921

1 INTRODUCTION

Symbolic execution [2, 6, 7] is a technique that systematically explores possible execution paths of a software under test (SUT) and automatically generates test inputs for some of them, e.g. paths that reached new coverage or exposed a program error. A variant of this technique is dynamic symbolic execution [3], where the non-symbolic operations in the SUT are executed normally and all side effects, such as memory allocations and file writes, are (typically) observable on the host system.

KLEE [1] is a well-known dynamic symbolic execution engine which, at a high level, works as a symbolic interpreter for SUTs compiled to LLVM [9] bitcode. KLEE explores paths incrementally, and usually has several execution paths under exploration at the same time. In this work, we see a *path* as characterised by the sequence of branch decisions that depend on symbolic input.

Memory allocation in symbolic execution can have an important impact in terms of execution determinism and bug-finding capabilities. Regarding the former, we note that the execution of some programs depends on the addresses returned by the allocator. Such programs are quite common—for instance, addresses are often used as keys in hash tables and other data structures.

The default memory allocator in KLEE works in such a way that all execution paths, together with the KLEE engine itself, share the same address space, served by the same general-purpose allocator.

This design has several consequences. First, multiple similar runs of KLEE could behave differently, depending on the addresses returned by the allocator—for example, different addresses may render the previous sequence of symbolic branch decisions for a path infeasible, if a branch decision depends on such an address.

Second, the execution of a particular path may be influenced by the execution of other paths, e.g. if a path A is executed *before* a path B, it would receive different addresses from the allocator than if it is executed *after* path B, which, as explained before, may change the feasibility of previous symbolic branch decisions on path A. This also means that changing the search heuristic may change the set of explored paths, even if all feasible paths are explored.

The resulting non-determinism is undesirable. In particular, it makes debugging difficult, as a path running in isolation might behave differently than in the presence of other paths. Non-determinism can also lead similar paths to behave differently, as common objects may be allocated at different addresses.

Memory allocation can also have an important impact on the bug-finding capabilities of a symbolic execution tool. This is because general-purpose allocators are not designed to enforce properties desirable for dynamic symbolic execution, such as spatial and temporal distancing to find certain classes of memory errors.

To address these issues, we have recently designed KDAlloc [12], which allocates a large amount of virtual memory and distributes allocations throughout that area. Importantly, each *state* (the representation of a path in KLEE) is given its separate state virtual address space [12], enabling KLEE to isolate states from one another

*Partially funded by the European Research Council (grant agreement no. 819141).

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3604921

and behave deterministically across runs. In addition, KDAlloc spatially distances allocations to ensure large inter-object redzones, and temporally distances allocations using a tunable quarantine. These features increase the chance of finding buffer overflows and respectively use-after-free errors. Further details on KDAlloc's design can be found in the paper that introduced it [12].

While the original KDAlloc paper focuses on the symbolic execution stage, similar problems related to allocator non-determinism are encountered when *replaying* the inputs generated by KLEE on a native version of the SUT. For each explored path, KLEE can generate a test input which is guaranteed to follow that path, *under the assumption that the same addresses are returned by the allocator.*

In this paper, we show how KDAlloc can be effectively integrated into the replay process. This allows replayed executions to receive the same heap addresses[1] as during symbolic execution, improving bug reproducibility and making debugging easier. The improved replay support is a unique contribution of this paper.

This tool paper is focused on the practical use of KDAlloc, targeting KLEE users interested in using KDAlloc during symbolic execution or replay. We are actively merging the contributions of this paper into mainline KLEE [8]. Until that point, we maintain a fork at https://github.com/srg-imperial/kdalloc-issta-2023.

2 DETERMINISTIC SYMBOLIC EXECUTION

We will illustrate some of the benefits of using KDAlloc with the example in Figure 1. Its code implements a randomised treap [4]. Each node has associated a key (in our case the C string being inserted into the treap, line 3) and a priority (in our case a 64-bit integer, computed by hashing the address of the node, lines 6 to 11). The keys form a binary search tree, while the priorities form a min-heap—priorities are used to ensure that the height of the tree is logarithmic in the number of nodes with a high probability.

On every insertion, both structures of the treap are maintained: Following a binary-search-tree insert of the key (comparison in line 18, recursive calls in line 20 and line 28), the heap criterion is fixed up using tree rotations (lines 21 to 26 and lines 29 to 34).

The main program starts by inserting three strings into the treap: `"1"`, `"2"`, and `"3"` (lines 40 to 42). Suppose the developer mistakenly assumes that the tree's root should always be the node with the key `"1"`, and thus adds the `assert` on line 43. In practice, this situation could easily result from the developer not understanding a data structure as deeply as they thought.

This program has non-deterministic behaviour: The bug is triggered depending on the addresses of the allocated nodes. Due to address space layout randomisation (ASLR), both native execution and execution with KLEE are non-deterministic with the respective default allocators. This makes it difficult to understand, replay and debug the issue. KDAlloc addresses this issue by eliminating this source of non-determinism in repeated runs and during replay.[2]

At its most basic, using KLEE with KDAlloc just requires passing the `--kdalloc` option:

```
$ klee --kdalloc treap.bc
...
```

[1] Divergences can also be caused by non-heap addresses, but these are less common.
[2] We note that a user might want to run KLEE multiple times, under different allocations, to maximise their chances of finding the bug. KDAlloc could readily facilitate such runs, by using it to run KLEE deterministically with different base addresses.

```
1  struct node {
2    struct node *lhs, *rhs;
3    char const *key;
4  }* root = NULL;
5
6  static uint64_t priority(void *p) {
7    uint64_t h = 0xcbf29ce484222325;
8    for (size_t i = 0; i < sizeof(p); ++i)
9      h = (h ^ ((unsigned char*)&p)[i]) * 0x100000001b3;
10   return h;
11 }
12
13 void insert(struct node **n, char const *str) {
14   if (!*n) {
15     *n = calloc(1, sizeof(struct node));
16     (*n)->key = str;
17   } else {
18     int cmp = strcmp(str, (*n)->key);
19     if (cmp < 0) {
20       insert(&(*n)->lhs, str);
21       if (priority((*n)->lhs) < priority(*n)) {
22         struct node *lhs = (*n)->lhs;
23         (*n)->lhs = lhs->rhs;
24         lhs->rhs = (*n);
25         *n = lhs;
26       }
27     } else if (cmp > 0) {
28       insert(&(*n)->rhs, str);
29       if (priority((*n)->rhs) < priority(*n)) {
30         struct node *rhs = (*n)->rhs;
31         (*n)->rhs = rhs->lhs;
32         rhs->lhs = (*n);
33         *n = rhs;
34       }
35     }
36   }
37 }
38
39 int main() {
40   insert(&root, strdup("1"));
41   insert(&root, strdup("2"));
42   insert(&root, strdup("3"));
43   assert(strcmp(root->key, "1") == 0);
44 }
```

Figure 1: Treap program used to illustrate the benefits of KDAlloc. For conciseness, include statements are omitted.

```
KLEE: Deterministic allocator: globals
     (start-address=0x7f6c6ce00000 size=10 GiB)
KLEE: Deterministic allocator: constants
     (start-address=0x7f69ece00000 size=10 GiB)
KLEE: Deterministic allocator: heap
     (start-address=0x7e69ece00000 size=1024 GiB)
KLEE: Deterministic allocator: stack
     (start-address=0x7e49ece00000 size=128 GiB)
...
```

This will run KLEE with the KDAlloc deterministic allocator, but the base addresses used for the different memory sections (heap, stack, etc.) are *not* deterministic from run to run (*cross-run*). Instead, the base addresses are decided by the operating system and the allocation is deterministic from that point onwards. We chose this default, as any fixed start address may at times conflict with the memory layout of the KLEE process, as decided by the OS.

Consequently, the program in Figure 1 will encounter the bug in some runs and run to completion in others. To achieve full cross-run determinism, the base addresses for the different memory sections need to be specified. The easiest way to pick these values is to perform one run without setting the addresses explicitly and to reuse the values chosen by the OS for the following runs. A more robust, but slightly more complicated technique is to instead choose values that are not likely to be blocked by other mappings after ASLR. For the default KDAlloc sizes, on 64-bit Linux, using addresses in the `0x610000000000` to `0x640000000000` range seems fairly robust:

```
$ klee --kdalloc                                    \
  --kdalloc-constants-start-address=0x610000000000 \
  --kdalloc-globals-start-address=0x620000000000   \
  --kdalloc-heap-start-address=0x640000000000      \
  --kdalloc-stack-start-address=0x630000000000     \
  treap.bc
```

The default sizes used by the allocators may seem exorbitantly large at 10 GiB for the globals and constants to 1 TiB for the heap, but KDAlloc only maps virtual memory *without* any physical backing. With roughly 2^{48-1} B = 128 TiB virtual address space available on current x86_64 CPUs for userspace processes, we have found that, if desired, the heap size can usually be increased to around 80 TiB. KLEE only actually uses a part of this address range for external function calls [12], and periodically releases the physical pages again. As this process can become somewhat expensive when executed often, it is only initiated once at least 1024 superfluous pages (tunable via `--kdalloc-external-page-threshold`) are backed with physical pages during an external function call and the number of backed pages is at least twice the average number of recently used pages. The whole process can also be disabled with `--kdalloc-mark-as-unneeded=false`, if the program under test does not consume large quantities of addresses including due to the quarantine (KDAlloc's quarantine is described in §4).

3 DETERMINISTIC REPLAY

If KLEE reports an issue, ideally the root cause is immediately obvious. However, developers often find it useful to debug the issue by replaying the test input generated by KLEE with a native version of the SUT. For programs whose execution depends on the memory layout, in order to be useful, replay must be deterministic and use the same addresses as during symbolic execution.

Returning to our running example of Figure 1, suppose that instead of inserting the strings `"2"` and `"3"` at lines 41 to 42, it instead inserts two bytes read from a symbolic `stdin` (KLEE provides support for a symbolic `stdin` stream). Given this modified program, KLEE (with LLVM 14) explores 14 paths through the code, depending on the values of the two bytes (in the next command, `--libc` tells KLEE to load a C standard library, `--posix-runtime` to enable support for a symbolic `stdin`, among other features, `--emit-all-errors` to create tests for all errors, even if they are at the same line of code, and `--sym-stdin 2` to return two symbolic bytes from `stdin`):

```
$ klee --libc=uclibc --posix-runtime --emit-all-errors \
  treap-sym.bc --sym-stdin 2
...
KLEE: done: generated tests = 14
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *x = malloc(sizeof(int));
  *x = 1;
  free(x);  // Bug: object freed too early
  int *y = malloc(sizeof(int));
  *y = 2;
  printf("%p: %d\n%p: %d\n", x, *x, y, *y);
  free(y);
}
```

Figure 2: Program with a simple use-after-free bug which is reliably found by KDAlloc configured with a quarantine.

Because execution depends on the memory addresses returned by the allocator, KLEE may or may not find the bug. However, suppose it does and the user tries to replay the input generated by KLEE to debug the issue. The existing replay tool provided by KLEE, called `klee-replay`, uses the default allocator, which is likely to return different addresses than during symbolic execution. This in turn may cause the replayed execution not to hit the bug anymore.

To prevent such issues, we have enhanced `klee-replay` to incorporate KDAlloc and use the same addresses as during symbolic execution. To make this work, we modified KLEE to record the base addresses used by KDAlloc during symbolic execution in a file called `klee.kconfig`. The replay tool then reads the recorded addresses and integrates KDAlloc before replaying the inputs.

Assuming the error-triggering input generated by KLEE is in file `klee-last/test000001.ktest`, we can simply run the following command to reproduce the error:

```
$ klee-replay klee-last/test000001.ktest ./treap
```

In order to provide the same addresses as during symbolic execution, `klee-replay` uses `LD_PRELOAD` to inject `libKDAlloc.so` into the target process. The injected library contains a `malloc` interceptor similar to those enabling the use of alternative memory allocators, such as jemalloc [5]. Our enhanced version of `klee-replay` searches for the `klee.kconfig` file in the directory of the test case by default, so we do not need to explicitly specify its location here.

4 USE-AFTER-FREE ERROR DETECTION

Quarantines delay the reuse of freed addresses to increase the chance of finding use-after-free errors [10, 13]. That is, instead of freeing an object so that the allocator can reuse its memory, it is instead placed into a quarantine. Only when the quarantine fills up, does memory start to be reused in a first-in first-out fashion.

Consider the program in Figure 2, which shows a simple use-after-free bug. Many general-purpose allocators will prefer to reuse freed addresses as soon as possible, to optimise cache locality. This immediate address reuse can hide the bug during analysis with KLEE, if run with the default allocator, as no unallocated address is ever accessed:

```
$ klee uaf.bc
...
0x55d87e308208: 2
0x55d87e308208: 2
...
```

Even if memory does not get reused, without a quarantine KLEE would report such a use-after-free bug as a buffer overflow, which can be confusing to users. By contrast, KLEE configured with quarantine-enabled KDAlloc can reliably find the bug in Figure 2 and report it properly as a use-after-free error:

```
$ klee --kdalloc uaf.bc
...
KLEE: Deterministic allocator: Using quarantine queue size 8
...
KLEE: ERROR: uaf.c:10: memory error: use after free
...
```

Unless specified otherwise, KDAlloc runs with a quarantine of eight objects per allocation bin. Allocation bins store objects of certain sizes (e.g., one bin stores 1 B objects, another bin objects between 2 B and 4 B, and so on in powers of two up to objects of 2048 B), with a *large object bin* managing objects larger than 2048 B.

For example, the program `quarantine.bc` allocates and immediately deallocates a one-byte object ten times, and prints all addresses it receives:

```
$ klee --kdalloc quarantine.bc
...
Allocated addresses: 0x7ee436400000 0x7ee036400000
     0x7ee836400000 0x7ede36400000 0x7eea36400000
     0x7ee236400000 0x7ee636400000 0x7edd36400000
     0x7eeb36400000 0x7ee436400000
...
```

As can be seen, KDAlloc's quarantine of eight 1 B objects guarantees that the program will receive different addresses (blue) for the first eight 1 B allocations. After the eighth allocation, the quarantine has filled up and addresses start to be reused (red).

If temporal distancing is not a concern, memory usage can be reduced by disabling the quarantine:

```
$ klee --kdalloc --kdalloc-quarantine=0 quarantine.bc
...
Allocated addresses: 0x7e7f5d600000 0x7e7f5d600000
     0x7e7f5d600000 0x7e7f5d600000 0x7e7f5d600000
     0x7e7f5d600000 0x7e7f5d600000 0x7e7f5d600000
     0x7e7f5d600000 0x7e7f5d600000
...
```

On the other hand, if sufficient memory is available, the quarantine size can be increased. If users find it difficult to determine the right quarantine size, instead of setting a very high size, it can be faster and more memory efficient to set the quarantine size to unlimited (-1). Using an unlimited quarantine engages additional optimisations that reduce memory usage compared to a large but finite quarantine size, as no addresses can ever be reused:

```
$ klee --kdalloc --kdalloc-quarantine=-1 quarantine.bc
...
Allocated addresses: 0x7e5d7a800000 0x7e597a800000
     0x7e617a800000 0x7e577a800000 0x7e637a800000
     0x7e5b7a800000 0x7e5f7a800000 0x7e567a800000
     0x7e647a800000 0x7e587a800000
...
```

5 RELATED WORK

KLEE previously included a deterministic allocator that similarly allocated a memory range with a specifiable base address [8]. Among many other differences [12], that allocator never freed memory, making it unusable for many programs.

The quarantine is an established concept for memory allocators, and has been used in several prior projects [10, 11, 13].

More generally, memory allocators have been well-studied in research and practice. We refer the reader to our research paper on KDAlloc [12] for a more extensive discussion of related work.

6 CONCLUSION

KDAlloc, the new memory allocator for KLEE, is a tunable, deterministic memory allocator purposely designed to support symbolic execution. Taking a user-centric view, we have discussed some of the main benefits of KDAlloc, and introduced a new replay functionality that can replay KLEE-generated inputs using the same heap layout. We believe that KDAlloc is a valuable tool for anyone using KLEE, and plan to make it the default choice soon.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI'08* (San Diego, CA, USA).

[2] Cristian Cadar and Dawson Engler. 2005. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *SPIN'05* (San Francisco, CA, USA). https://doi.org/10.1007/11537328_2

[3] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *CACM* 56, 2 (2013), 82–90. https://doi.org/10.1145/2408776.2408795

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. 2001. *Introduction to Algorithms* (second ed.). MIT Press/McGraw Hill.

[5] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *BSDCan'06* (Ottawa, Canada).

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *PLDI'05* (Chicago, IL, USA).

[7] James C. King. 1976. Symbolic Execution and Program Testing. *CACM* 19, 7 (1976), 385–394.

[8] KLEE website [n. d.]. KLEE website. http://klee.github.io/.

[9] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO'04* (Palo Alto, CA, USA). https://doi.org/10.1109/CGO.2004.1281665

[10] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003).

[11] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. 2012. *Windows® Internals, Part 2* (6th ed.). Microsoft Press.

[12] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2022. A Deterministic Memory Allocator for Dynamic Symbolic Execution. In *ECOOP'22* (Berlin, Germany). https://doi.org/10.4230/LIPIcs.ECOOP.2022.9

[13] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC'12* (Boston, MA, USA).

Received 2023-05-18; accepted 2023-06-08