Concretely Mapped Symbolic Memory Locations for Memory Error Detection

Haoxin Tu, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding, and He Jiang*

Abstract—Memory allocation is a fundamental operation for managing memory objects in many programming languages. Misusing allocated memory objects (e.g., buffer overflow and use-after-free) can have catastrophic consequences. Symbolic execution-based approaches have been used to detect such memory errors, benefiting from their capabilities in automatic path exploration and test case generation. However, existing symbolic execution engines still suffer from fundamental limitations in modeling dynamic memory layouts; they either represent the locations of memory objects as concrete addresses and thus limit their analyses only to specific address layouts and miss errors that may only occur when the objects are located at special addresses, or represent the locations as simple symbolic variables without sufficient constraints and thus suffer from memory state explosion when they execute read/write operations involving symbolic addresses. Such limitations hinder the existing symbolic execution engines from effectively detecting certain memory errors. In this study, we propose SYMLOC, a symbolic execution-based approach that uses concretely mapped symbolic memory locations to alleviate the limitations mentioned above. Specifically, a new integration of three techniques is designed in SYMLOC: (1) the symbolization of addresses and encoding of symbolic addresses into path constraints, (2) the symbolic memory read/write operations using a symbolic-concrete memory map, and (3) the automatic tracking of the uses of symbolic memory locations. We build SYMLOC on top of the well-known symbolic execution engine KLEE and demonstrate its benefits in terms of memory error detection and code coverage capabilities. Our evaluation results show that: for address-specific spatial memory errors, SYMLOC can detect 23 more errors in `GNU Coreutils`, `Make`, and `m4` programs that are difficult for other approaches to detect, and cover 15% and 48% more unique lines of code in the programs than two baseline approaches; for temporal memory errors, SYMLOC can detect 8%-64% more errors in the Juliet Test Suite than various existing state-of-the-art memory error detectors. We also present two case studies to show sample memory errors detected by SYMLOC along with their root causes and implications.

Index Terms—Software Reliability, Software Security, Memory Errors, Program Analysis, Symbolic Execution

1 INTRODUCTION

MEMORY allocation functions (e.g., `malloc` and `free`) are fundamental operations for managing memory objects in programs written in C/C++ programming languages [1], [2]. However, previous studies [3], [4], [5], [6], [7] show that memory errors such as *buffer overflow* and *use-after-free* caused by misuse of such allocation functions are common and can have catastrophic consequences [8].

To facilitate the detection of memory errors, many techniques have been proposed, including static analysis-based approaches such as Frama-C [9] and Coccinelle [10], dynamic analysis-based approaches such as Valgrind [11] and Asan [12], and symbolic execution-based approaches such as KLEE [13] and *symsize* [14].

Symbolic execution could be one of the most automated approaches to detect memory errors since it can automatically explore different paths in programs and generate actionable test cases to assist in validating a reported error [15]. In contrast, static analyses usually do not excel at generating test cases to validate an error reported for a program and dynamic analyses have trouble covering different paths in the program. Benefiting from its capabilities in automatic path exploration and test case generation, symbolic execution has become a popular technique broadly adopted in many domains, e.g., test case generation [13], [16], [17], bug-detection [13], [18], [19], debugging and repairing [20], [21], [22], [23], [24], cross-checking [25], [26], side-channel analysis [27], [28], [29], and exploit generation [30], [31], [32].

Despite the success of symbolic execution, existing symbolic execution engines still suffer from fundamental limitations in modeling memory addresses related to memory allocation functions. Specifically, most existing symbolic execution engines, such as KLEE [13] and *symsize* [14], usually model the locations of dynamically allocated memory objects (e.g., `malloc` in C/C++) with concrete values, where each object is located at a fixed address. However, those address values should be *non-deterministic* during actual executions as they are dynamically allocated based on different run-time environments. The limitation may cause existing symbolic execution engines to miss detecting certain kinds of memory errors. First, since the engines do not encode memory addresses into path constraints, they may not be able to cover certain lines of code whose executions are dependent on the memory addresses and miss certain address-specific spatial memory errors such as *buffer overflow*. Second, without maintaining and tracking the con-

* *He Jiang is the corresponding author.*

- *Haoxin Tu is with the School of Computing and Information Systems, Singapore Management University, Singapore. Haoxin Tu is also with the School of Software, Dalian University of Technology, Dalian, China. E-mail: haoxintu.2020@phdcs.smu.edu.sg*
- *Lingxiao Jiang, Jiaqi Hong, and Xuhua Ding are with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mails: lxjiang@smu.edu.sg, jqhong@smu.edu.sg, xhding@smu.edu.sg*
- *He Jiang is with the School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. He Jiang is also with DUT Artificial Intelligence, Dalian, China. E-mail: jianghe@dlut.edu.cn*

straints among the addresses of different memory objects, they may fail to reliably check for unsafe uses of memory objects and miss certain temporal memory errors such as *use-after-free* or *double-free*. Even though the engines allow symbolization of the memory addresses, without proper handling of read/write operations involving symbolic addresses, the engines can face memory state explosion and cannot explore the programs effectively. Therefore, existing symbolic execution engines need better ways to model the addresses of dynamically allocated memory objects for more effective detection of memory errors in programs.

Since the locations of dynamically allocated memory objects can be nearly arbitrary, a more complete (i.e., treat those locations as symbolic and make greater use of those symbols) modeling of those locations can help explore more execution paths and detect more memory errors in the test program. Hence, we believe that a symbolic execution engine should satisfy at least the following three fundamental capabilities to comply with the more complete modeling requirements: 1) symbolization of addresses and modeling them into path constraints, 2) practical read/write operation from/to symbolic addresses, and 3) effectively tracking the uses of symbolic addresses. Unfortunately, no existing symbolic execution engine fulfills all the above requirements. For example, KLEE [13] or *symsize* [14] satisfies none of those requirements. They treat memory locations as concrete; even though one can force an address to be symbolic, the subsequent symbolic memory operations would fail as the engines usually concretize the symbolic addresses to invalid ones, thus producing false alarms of errors. A recent work, RAM [33], satisfies requirements #2 and partially #1 but not #3. It simply assumes that the memory locations do not affect the behavior or execution paths so it does not encode them into path constraints. Unfortunately, such an assumption is not always valid in the real world (see more details in the code example in Fig. 1).

This paper proposes SYMLOC to integrate three techniques, i.e., address symbolization, a symbolic-concrete memory map, and symbolic memory tracking, to meet all the above three fundamental requirements. With the new integration of the techniques, SYMLOC can more effectively detect memory errors: (1) by encoding symbolic addresses into path constraints and equipped with the symbolic-concrete memory map, SYMLOC is able to cover more code and detect more address-specific spatial memory errors; (2) by enabling the automatic propagation of symbolic addresses in symbolic execution, SYMLOC is able to detect temporal memory errors reliably.

We have built SYMLOC on top of KLEE and evaluated its effectiveness. Our empirical evaluation results show that: (1) SYMLOC is able to detect 23 more address-specific spatial memory errors and cover 15% and 48% more unique lines of code than two symbolic execution engines (i.e., KLEE [13] and *symsize* [14]) over `GNU Coreutils`, `Make`, and `m4` programs; (2) when compared against various state-of-the-art memory error detectors [3], including symbolic executors (KLEE and *symsize*), static detectors (Frama-C [9] and Coccinelle [10]), and dynamic detectors (Vargrind [11] and Asan [12]), over the C/C++ programs in Juliet Test Suite (JTS) datasets [34], SYMLOC can reliably detect 8%-64% more errors than comparative memory error detectors.

In short, this paper makes the following contributions.

- We propose a novel approach named SYMLOC that integrates three techniques to satisfy the three fundamental capability requirements for symbolic execution involving memory allocation functions, which facilitates more effective detection of memory errors in C/C++ programs.
- We empirically demonstrate the effectiveness of SYMLOC against many other state-of-the-art memory error detectors over various real-world benchmark programs, and showcase two sample memory errors undetected by existing approaches and discuss their implications.
- We release a replication package for SYMLOC[1] to facilitate future work on more optimized symbolic execution engines and more effective memory error detection.

Organization. Section 2 gives the background and motivating examples. Section 3 describes the design and implementation of SYMLOC. Section 4 presents our evaluation results. Section 6 discusses additional findings and threats to the validity of SYMLOC. Section 5 presents two case studies showing sample memory errors detected by SYMLOC along with their root causes. Section 6 discusses the pros and cons of SYMLOC in relation to various other techniques with the same purpose, limitations, as well as threats to validity. Sections 7 and 8 describe more related work and conclude with future work. The Appendix discusses two extra case studies indicating the usefulness of SYMLOC.

2 MOTIVATION

This section describes common memory errors briefly, and shows two examples to illustrate the limitations of existing approaches and highlight the advantages of our approach.

2.1 Common Memory Errors

Memory errors can broadly be divided into two categories: *spatial* and *temporal* errors [5], [35], [36], [37].

Spatial memory errors. Such errors are usually in two forms: (1) *Out-Of-Bound (OOB)* or *buffer overflow* – assessing (i.e., read or write) to an address that is out of bound of a valid memory area (a.k.a. a *buffer*); (2) *NULL Pointer Error* — dereferencing *null* pointers or uninitialized wild pointers. Here, we refer to them as *address-specific* errors as they violate proper usage of the specific allocated address of a memory object.

Temporal memory errors. Such errors broadly fall into the following three types [5] involving a sequence of more than one memory operation: (1) *Use-After-Free* (UAF) – accessing a memory area via a pointer after the memory area pointed to by the pointer has been deallocated; (2) *Double-Free* (DoF) – deallocating a memory area again via a pointer after the memory area pointed to by the pointer has been deallocated, which can be considered a special case of UAF; (3) *Invalid-Free* (InF) – deallocating a memory area via a pointer while the pointer does not point to the beginning of a valid allocated memory area (i.e., freeing a pointer whose value was not returned by a heap allocator).

Keeping memory *spatial* and *temporal* safety is critical for assuring the quality of software systems [3], [5] because memory errors can have catastrophic security risks, e.g.,

1. https://github.com/haoxintu/SymLoc

```
1  void *memmove((void *dest, const void *src, size_t n){
2      unsigned long int to = (long int) dest;
3      unsigned long int from = (long int) src;
4      if (from == to || n == 0) {
5              .. // Path-A
6      }
7      if (to > from) {   /* copy in reverse */
8          if (to - from >= (int) n)     {
9              int i; // Path-B
10             for(i=n-1; i>=0; i--)
11                 to[i] = from[i]; // symbolic memory read/write
12             return dest;
13         } else {
14             ... // Path-C
15         }
16     }
17     if (from > to) {   /* copy forwards */
18         if (from - to >= (int) n)) {
19             ... // Path-D
20         } else {
21             dest[n + 100] = 0; // address-specific memory error
22             ... // Path-E
23         }
24     }
25     return dest;
26 }
27 int main(){
28     char *buf1 = malloc (100 * sizeof(char));
29     char *buf2 = malloc (50 * sizeof(char));
30     if (buf1 == NULL) { abort(); }
31     if (buf2 == NULL) { abort(); }
32     memmove(buf1, buf2, 10);
33     free(buf1);free(buf2); return 0;
34 } /* Example 1 (E1) */
```

Fig. 1. Existing symbolic executors (e.g., KLEE) are only able to cover `Path-B` while missing `Path-A`, `C`, `D`, and the memory error in `Path-E`

being exploited by attackers [3], [5], [38]. Memory errors still rank among the most dangerous software errors in recent CVE announcements [7] and a recent analysis of the Chromium project shows that more than 50% of serious memory safety errors are temporal memory errors [39].

Approaches for detecting memory errors. According to the way to analyze a program to detect certain memory errors, existing detection approaches can be broadly classified into three categories: static analysis-based, dynamic analysis-based, and symbolic execution-based approaches [3]. Static analysis-based approaches detect errors by analyzing the source or machine code of the program without executing it, while dynamic analysis-based approaches report errors by executing test programs. In contrast, symbolic execution-based approaches are in-between, and they search for memory errors by simulating the executions. In this paper, we focus on improving the capabilities of symbolic execution-based approaches.

2.2 Motivating Examples

The first example shown in Fig. 1 originated from the widely used implementation of the function `memmove()` [40], [41], and the second example shown in Fig. 2 is adapted from the JTS benchmarks [34] that were used in our experimental evaluation. In the following, we first explain the execution flows of each program and then point out the limitations of existing approaches in detecting certain memory errors. Finally, we present the main advantages of our approach.

2.2.1 *E1: Missing Spatial Memory Errors*

Execution flows. The functionality of the `memmove()` function shown in Fig. 1 is to copy n bytes from memory area *src* to memory area *dest* and finally return the pointer *dest* to users. Note that the copying memory areas are allowed to be overlapped. The typical implementation logic of this function is (1) casting two pointer values *dest* and *src* to integer values *to* and *from* to avoid undefined behavior[2] and (2) handling different scenarios according to the different locations of two input memory objects by comparing the casted values of pointers [40], [41]. There will be five scenarios in total: the values of the pointers *from* and *to* are the same or the copying size is zero, leading to `Path-A` in Line 5; the value of the pointer *to* is larger than *from* wo/w overlap copying, leading to `Path-B` (Line 9)/`Path-C` (Line 11); the value of the pointer *to* is smaller than *from* wo/w overlap copying, leading to `Path-D` (Line 16)/`Path-E` (Line 18), where an address-specific spatial memory error, i.e., *buffer overflow*, is hidden in `Path-E` in Line 21.

Limitations of existing approaches. As aforementioned in Section 1, static analysis-based approaches can detect the error. Still, they are hard to produce a useful test case to reproduce the error. At the same time, dynamic analysis-based approaches also encounter difficulties as they usually produce redundant test cases for the shadow area of the test program or suffer from limited analysis algorithms to catch certain errors. For symbolic execution-based approaches, although simple, existing symbolic execution engines miss the majority of (80%, 4 out of 5) paths due to the fundamental design flaw in those engines. Specifically, since existing engines linearly manage memory objects that are consecutively allocated, i.e., the value returned from the second `malloc` function (Line 29) is always larger than the value returned from the first one (Line 28), meaning the value of *to* is always larger than the value of *from*. Therefore, only `Path-B` (Line 9) will be covered, while missing the covering of the rest of the four paths, i.e., `Path-A`, `C`, `D`, and `E`, thus missing detecting the memory error in Line 21. Worse still, existing engines (e.g., KLEE) struggle to handle symbolic memory read/write operations as shown in Line 11. It might be true for some variants of KLEE that do not always concretize the symbolic address. However, for the main base version of KLEE, as far as we know, when KLEE encounters a symbolic address, no matter whether the base address or the offset is a symbolic value, KLEE concretizes it (i.e., KLEE leverages a constraint solver to decide which concrete memory addresses could be given to the symbolic address) before performing read/write to a symbolic address. KLEE sacrifices the accuracy of memory modeling to continue the execution. In this case, the main version of KLEE generates false alarms of memory errors mainly because KLEE simply tries all possible concrete addresses from the solver (usually from "0") instead of keeping track of all valid addresses during execution and gives out the tracked valid addresses. Since "0" is less likely to be a valid memory address, any read/write upon this address will cause a memory error. To avoid such errors, KLEE has to either add the same implementations as SymLoc, i.e., maintaining a memory map to store the valid base address of each symbolic address to ensure the normal memory read/write operations or rely on other advanced techniques to handle symbolic memory

2. Direct comparison of two pointers from different memory objects is undefined behavior according to C standard (ISO/IEC 9899:201x)

```
1  static  char * dothing (int magic) {
2      if  (magic == 0x1234ABCD) {
3          wchar_t * data1 = NULL;
4          data1 = (wchar_t *) malloc (100 * sizeof(wchar_t));
5          if  (data1 == NULL) { abort(); }
6          wmemset(data1, L'A', 100-1);
7          data1[100-1] = L'\0';
8          free(data1);
9          wprintf(L"%s\n", data1); // use-after-free error 1
10         return NULL;
11     } else {
12         char * data2 = NULL;
13         data2 = (char *)  malloc (100 * sizeof(char));
14         if  (data2 == NULL) { abort(); }
15         memset(data2, 'A', 100-1);
16         data2[100-1] = '\0';
17         free(data2);
18         return data2; //   use-after-free error 2
19     }
20 }
21 int  main(int argc, char** argv){
22     int  a;
23     read(0,  &a, sizeof(int));
24     char * ret  = dothing(a);
25     if  (ret  != NULL)
26         printf("%s\n", ret);
27     return 0;
28 }   /*  Example 2 (E2) */
```

Fig. 2. Missing the detection of UAF errors

read/write.

Note that to ensure the program quality, the problem of finding all faults in a program for any meaningful program is essentially unsolvable [42]. To improve the quality of the program, a common criterion is to use code path coverage to comprehensively test the program when we don't know beforehand where the faults are [43]. The idea is that if a large portion of code is covered with no faults, the program can be more reliable and contain fewer faults. Therefore, it is necessary to cover all the branches in the code example.

Advantages of SymLoc. Instead of modeling memory addresses as concrete ones, SymLoc first symbolizes those address values as symbolic ones and then encodes symbolic addresses into path constraints. Furthermore, a symbolic-concrete memory map is designed in SymLoc for efficient symbolic memory operations, and a relaxed overlapping property is supported to have better modeling of dynamic memory allocation behavior. Thus, SymLoc can smoothly cover all the branches in Fig. 1 and detect the memory error in Line 21. Importantly, SymLoc could provide a useful test case (i.e., in what conditions the error will be triggered) that helps developers quickly locate and further fix the error.

2.2.2 *E2: Missing Temporal Memory Errors*

Execution flows. The program first reads a value from the user, passes it to the function `dothing`, and finally, prints out the content of the string pointed to by the return value *ret* if *ret* is not *NULL* (Lines 23-26). Inside the function `dothing`, it compares the argument `magic` number against a specific value (e.g., 0x1234ABCD). If the corresponding value is provided, the program will go through the subsequent branches and exercise potential *abort* failures or UAF errors. In the *if-branch* starting from Line 2, a 100-size buffer is allocated by invoking the `malloc` function and two *if-branch*es (Line 5 and 14) are used to check whether the allocated address (the value of *data1* or *data2*) equals *NULL*. Then, the whole allocated buffer is initialized by calling the `wmemset` function in Line 6 or the `memset` function in Line 15. Note that there are two UAF errors inside this function. The first UAF error lies in Line 9, where the object *data1* is freed (Line 8) but it is used as an argument in the `wprintf` function later in Line 9. The function raises the second UAF error since a value *data2* is freed in 17 but later returned in Line 18, where the object *data2* is further used as an argument in the `printf` function in Line 26.

Limitations of existing approaches. Existing state-of-the-art memory error detectors are struggling to detect both two UAF errors. Static/dynamic memory error detectors have at least suffered from two major issues. Apart from the limitations of the test case (i.e., required values to satisfy the *if* conditions) generation as we mentioned, they are also difficult to detect the UAF errors due to their limited capability in inter-procedure analysis or handling certain C library functions. Symbolic executors are easy to get the concrete input to go through the two branches but still fail to catch both two UAF errors. For example, the well-known executor KLEE can not detect UAF error 2 due to its fundamental design issue in their memory management [44]. Specifically, there are some memory relocation operations inside of the `printf` function (code is not shown); during KLEE's execution, one of the addresses relocated inside of the `printf` function in Line 26 is the same as the address of the freed object in Line 17. Thus, KLEE will treat the freed address as "valid" which leads to the missing UAF error. More importantly, both static/dynamic or symbolic execution-based approaches need to take extra effort to track, relate, and check the uses of the potentially enormous number of memory objects with concrete addresses.

Advantages of SymLoc. Instead of extra analysis to track, relate, and check the uses of memory objects with concrete addresses, SymLoc effectively validates the safety of using symbolic memories by checking the propagated symbolic expressions during symbolic execution. Therefore, SymLoc is able to reliably detect the two UAF errors in Fig. 2. Note that reliable detection in this paper has two meanings: (1) effectively catches temporal memory errors and (2) precisely reports the root cause information of those errors such as "*memory error: a use after free is detected*" when a freed memory object is used rather than imprecise information (e.g., "*memory error: out of bound error*" reported by KLEE [13]).

3 Design of SymLoc

This section describes the overview of SymLoc and then details the design of SymLoc.

Overview. Fig. 3 presents the high-level design of SymLoc. The main insight is that, to satisfy the three fundamental requirements as aforementioned in Section 1, we symbolize the return address from dynamically allocated memory and maintain a concretely mapped map (i.e., *SymLocMap*) to enable the symbolic memory operation as well as symbolic memory tracking. To this end, as shown on the right side of Fig. 3, we integrate three techniques in SymLoc. In ①, the path constraint encoded address symbolization technique first treats dynamically allocated addresses as symbolic values and propagates them into path constraints to enable the comprehensive exploration of execution paths that depend

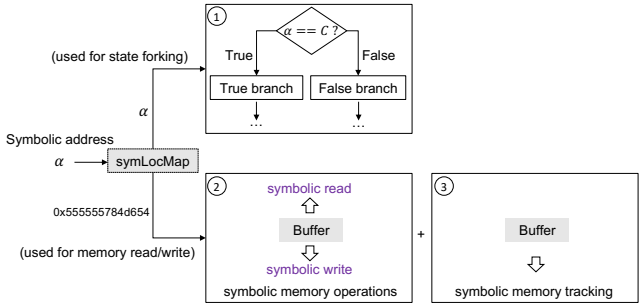


Fig. 3. High-level design of SYMLOC

on memory locations, thus enabling the detection of more spatial memory errors. In ②, the symbolic-concrete memory mapped memory operations technique utilizes the map, where each symbolic address holds its underlying concrete buffer, to support performing symbolic memory operations. In ③, to make greater use of newly defined symbols, a new error detection technique tracks the use of symbolic addresses to assist SYMLOC in performing reliable temporal error detection. Next, we detail these techniques separately.

3.1 Address Symbolization

3.1.1 Definition of Symbolic Addressing Model

Existing memory addressing models. Memory addressing models in most symbolic execution engines are designed to represent the allocated memory objects with as much concrete information as possible during executions [13]. For example, in KLEE, a memory object *mo* is presented as a tuple: $(addr, size, arry) \in N^+ \times N^+ \times A$, where *addr* is a *concrete* base address of the *mo*, *size* is a *concrete* size of the `mo`, and *arry* is a solver array that tracks the concrete or symbolic values written to the *mo*. N^+ means all the natural numbers corresponding to the indices for all memory objects. A is the set of all possible solver arrays.

Such memory addressing models often hold the *non-overlapping* property, i.e., every memory object in the address space is within its unique address range which does not intersect with other memory objects' address ranges. This *non-overlapping* property is useful for identifying a memory object via an address, i.e., a concrete address *a* is associated with at most one memory object that can be determined by checking if the following condition is true for a *mo*: $mo.addr \leq a \leq mo.addr + mo.size$.

Our memory addressing model. A memory object in our model is presented as follows:

$$(symAddr, size, arry) \in S_{N^+} \times N^+ \times A$$

where *symAddr* is a symbolic rather than a concrete value and S_{N^+} represents the set of symbolic variables that maintain a concrete address in the range of N^+. Note that we focus on making those addresses that are dynamically allocated symbolic, i.e., the addresses determined at runtime by invoking the dynamic memory allocation function (e.g., `malloc` in C/C++). Such allocations can return different addresses based on the runtime environment or even some specific addresses desired by hackers when exploiting the errors [45], [46], [47], [48].

The major benefit of our memory model is that we enable path exploration in the paths that require a condition involving address arithmetic and comparison, which is different from existing symbolic execution engines such as KLEE [13] and *symsize* [14]. To exemplify how SYMLOC is capable of covering more lines of code, take again the code snippets shown in Fig. 1 as an example. In SYMLOC, the returned address of the `malloc` function is represented as a symbolic variable, say α for the pointer *buf1* in Line 28 and β to the pointer *buf2* in Line 29. Then, during path exploration, every path where the condition relies on the comparison of pointers will be forked if the current path constraint is solvable. Later, a constraint solver (e.g., STP [49] or Z3 [50]) will be used to determine the feasibility of those forked branches, and the satisfied path will be further explored as the normal symbolic execution does. For example, the `Path-C` in Line 11 in Fig. 1 will be encoded as:

$$(\alpha > \beta) \ \& \ (!(\alpha - \beta >= n))$$

A capable solver can check the satisfiability of the constraint, and this constraint can be true and solved to a value that could help explore the corresponding path `Path-C`. Empowered by such a capability, SYMLOC is able to explore all five scenarios in the `memmove` function, enabling the covering all five paths from `Path-A` to `Path-E` as well as the error handling code in Lines 30-31, importantly, the address-specific spatial memory error in Line 21.

It is worth noting that existing symbolic execution engines can easily make the addresses symbolic by invoking the supported API (e.g., `klee_make_symbolic` in KLEE). However, such a default solution brings an intractable problem since they do not handle symbolic memory operations (see more details in Section 3.2).

3.1.2 Relaxed Non-overlapping Property

In SYMLOC, we no longer assume the *non-overlapping* property that is held in many other symbolic execution engines. We opt for such a design mainly because the address can be dynamically allocated nearly anywhere for each memory object, and we aim to *over-approximate* those addresses to investigate how different memory locations could reflect program behaviors under test. Many system memory managers are capable of allocating the same memory for different objects for efficiency purposes, which is much different from the memory allocation supported in existing symbolic execution engines. For instance, considering the following code snippet, the addresses of *buff1* and *buff2* are often the same in a native-run (violating the *non-overlapping* assumption) because the memory manager reuses the memory allocated for *buf1* after the `free`. Existing symbolic execution engines always assume the address of *buff2* is higher than *buff1*, while SYMLOC does not have this restriction and provides more freedom to explore more execution paths.

```
1 int *buff1 = (int*) malloc(100 * sizeof(char));
2 free(buff1);
3 int *buff2 = (int*) malloc(100 * sizeof(char));
```

It is worth noting that there is no need to be free of the first allocated buffer between two allocation functions to make the memory chunks that *buff1* and *buff2* point to overlapping. Considering two sightly complicated code

```
1  int main(){
2    int *p1 = malloc(8); // start manipulating heap allocator
3    free(p1);
4    ...  // other implementation code
5    free(p1); // end of manipulating heap allocator
6
7    void *p2 = malloc(8);
8    void *p3 = malloc(8);
9    if ((long)p2 == (long)p3) BUG(); // bug is triggered
10   return 0;
11 }
```

Fig. 4. Overlapping allocation example 1: overlapping at the beginning

```
1  int main(int argc , char* argv[]){
2    long long *p1,*p2,*p3,*p4;
3    p1 = malloc(0x500 − 8);
4    p2 = malloc(0x500 − 8);
5    p3 = malloc(0x80 − 8);
6
7    free(p2); // start manipulating heap allocator
8    int designed_chunk_size = 0x581;
9    int designed_region_size = 0x580 − 8;
10   *(p2−1) = designed_chunk_size; //end of manipulating heap allocator
11
12   p4 = malloc(designed_region_size);
13   if ((long) p4 + 0x500 == (long) p3) BUG(); // bug is triggered
14   return 0;
15 }
```

Fig. 5. Overlapping allocation example 2: overlapping in the middle

examples presented in Fig. 4 and Fig. 5, a programmer can still make it happen by manipulating the heap allocator. In the first example[3], a programmer may manipulate the heap allocator to make the chunk *p3* pointing to is overlapped at the beginning of *p2*, meaning the addresses of *p2* and *p3* are the same and the bug in Line 9 will be triggered. The mechanism behind it is that by using the tcache (a thread local caching strategy in heap management) free lists, after allocating a chunk and mistakenly freeing it twice, the allocator can return a desired memory location by writing into the duplicated chunks. Similarly, in the second example[4], after manipulating the heap allocator, i.e., exploiting the overwrite of a freed chunk size in the unsorted bin (i.e., stores small and large freed chunks, which acts as a cache layer to speed up allocation and deallocation requests) in order to make a new allocation overlap with an existing chunk, the chunk *p4* is overlapped in the middle of the chunk *p3*, making the condition in Line 13 true and the bug triggering. In this case, the offset "0x500" might be changed based on different manipulating strategies.

It is also worth noting that we do not claim the cases in Fig. 4 and Fig. 5 will always happen. Instead, malicious users (e.g., malicious programmers who have a deep understanding of heap allocation) may write the same code which could lead to certain errors. If that happens, SYMLOC could help client analyzers pinpoint the root cause of the error, which contributes to the efficient debugging process.

Applicable scenarios. We recognize that the relaxed non-overlapping property is not good for all cases. However, we support it in SymLoc mainly because we aim to have closer modeling of the heap allocator. As one of the best practices when managing memory, it is recommended that "*it is important to free up memory as soon as you are done using it*" [51]. Therefore, frequently allocating and deallocating memory can be widely used in programs (We also confirmed the benchmarks used in this study follow the above best practice). In such situations, the addresses from afterward allocations are more likely to be overlapped with previous freed ones. In addition, heap allocators may contain vulnerabilities that may be exploited to allocate overlapping memory ranges [52]. Therefore, the relaxed non-overlapping property should be a better choice for modeling the heap allocation behavior in practice. In contrast, existing symbolic execution engines (e.g., KLEE) apply strict non-overlapping property that all the memory objects cannot be overlapped, thus missing code coverage (e.g., `Paths A, C, D`, and `E` in Fig. 1) and the detection of important categories of memory errors (e.g., the one shown in Fig. 14). The relaxed non-overlapping property can have a limitation in the situation that the program frees memory objects only once before the execution terminates. Note that we provide a post-processing option (see Section 3.3 for more details) to analyze the results by adding extra constraints for each symbolic address, although SYMLOC does not add more constraints at the first time mainly due to the performance issue. Therefore, when an error is reported by SYMLOC, we run SYMLOC again to perform post-processing to filter out potential false positives (e.g., the ones that require the address in the kernel space).

3.2 Symbolic Memory Operations and Tracking

3.2.1 *Symbolic Memory Operations*

Existing symbolic memory read/write operations. Even though the base addresses and sizes are concrete in this addressing model, symbolic addresses can still be introduced indirectly via symbolic offsets or other symbolic values stored in memory objects used for indirect addressing. Once a symbolic address is used during symbolic execution, a major concern is how to perform memory operations on the memory location(s) addressed by the symbolic address.

Modern symbolic execution engines (e.g., KLEE [13] or *symsize* [14]) often rely on constraint solving to resolve the symbolic addresses and then determine how to handle read/write. For example, if a symbolic pointer `p` is resolved to a single memory object *mo*, then a read via `p` can be represented as `select(mo.addr, e)`, where *e* indicates the offset of *p* in *mo* and can be calculated as (*p* - *mo.addr*); a write can be represented as `store(mo.addr, e, v)` where *v* is the value to be written into the e^{th} offset of *mo*. If the symbolic pointer `p` can be resolved to more than one value, the symbolic execution engine can choose to either fork the execution states during exploration so that *p* is resolved to a single value in each state for easier analysis [13], [17], or use disjunctive conditions to constrain *p* to all the resolved objects [19], [53], [54] and rely on capable constraint solvers to analyze different combinations of the `select` and `store` operations via pointer *p*.

By equipping with the address symbolization technique, an intractable problem that needed to be resolved in SYMLOC is the symbolic memory read/write operations during execution. This problem is difficult to solve as the symbolic pointer (i.e., address) modeling and resolution is an

3. This allocation can be successful before the commit d081d of `glibc`.
4. This allocation can be successful before the commit 88cd0 of `glibc`.

unsolved and challenging problem in symbolic execution, although it has been actively studied in the literature [30], [31], [55], [56], [57], [58]. The difficulties mainly come from the possible state explosion due to the huge numbers of addresses for symbolic values and the heavy constraint resolution overheads. To bound the state search space and simplify constraint solving, existing symbolic execution engines (e.g., KLEE [13]) usually concretize symbolic addresses into possible concrete values or confine their possible values to certain memory segments based on current path constraints during state exploration. However, such strategies always lead to inaccurate/limited modeling of memory states and program behavior because only a small number of concrete values or approximated segments are taken into consideration, thus obstructing the effectiveness of the analysis.

In this study, a symbolic-concrete memory map-based read/write mechanism is designed for accessing symbolic memory objects. Note that we do not claim to solve the challenging problem of symbolic read/write, as we only focus on enabling symbolic read/write operations by decoupling the operations from heavy constraint solving and avoiding incorrect concretization upon symbolic addresses. Our key idea is that, for each symbolized address returned from a memory allocation function, via a memory map, we still associate the symbolic address with a concrete memory buffer that would be allocated by actually invoking the allocation function. When encountering a symbolic memory read/write operation introduced via memory allocations, the associated concrete buffer address is provided to perform the read/write operations, while the address itself remains symbolic and can be encoded into path constraints when they are used in branch conditions. As such, our symbolic memory operations could decouple the symbolic memory read/write operations from heavy constraint resolution and guarantee the correctness of the execution. For example, considering the memory allocation code "char *p = malloc(100)", SymLoc first declares a symbolic variable for the variable *p* and passes it to the path exploration (as shown in ① in Fig. 3). In the meantime, SymLoc maintains a concrete value (derived from KLEE) of the symbol. Such a concrete value will be used to construct a pair "<*sym_name*, <*mo, obj*»" which is stored in the *SymLocMap*, where *sym_name* is a unique name for each memory object and *mo/obj* are memory object and object state maintained by KLEE. Then, the map will be used to further assist symbolic memory operation and tracking (as shown in ② and ③ in Fig. 3).

It is also worth noting that the idea of using a memory map to support the symbolic memory operations is straightforward but practically useful for alleviating the path explosion problem in existing symbolic execution engines. In particular, it enables the engine to continue path exploration even when a symbolic address is unresolved. Considering the following example code, when KLEE is applied to explore the *if-else* branches and use the KLEE's API `klee_make_symbolic` to make the pointer *buff* symbolic, KLEE can fork two execution states. However, its exploration of the *if* and *else* branches will fail (due to invalid concretized addresses) or take a long time (because it tries to explore all possible concrete values for *buff*) when it encounters the write to the symbolic *buff[1]* (Line 6) or read from the symbolic address (Line 8). In contrast, SymLoc enables the read/write supported by the underlying concrete buffer associated with the symbolic `buff` and thus can smoothly explore those branches.

```
1  int main() {
2      char temp;
3      char *buff = (char *) malloc(100 * sizeof(char));
4      klee_make_symbolic(&buff, sizeof(char*), "buff");
5      if ((long long) buff > (long long) some_address) {
6          buff[1] = '9'; // write to a symbolic address
7          ...;   // code to be explored further
8      } else {
9          temp = buff[1]; // read from a symbolic address
10         ...;    // code to be explored further
11     }
12     return 0;
13 }
```

3.2.2 *Symbolic Memory Tracking*

As aforementioned in Section 1, existing symbolic execution engines may struggle to reliably detect certain temporal memory errors due to their fundamental design downside. One solution to mitigate the problem may be tracking the use of concrete values in symbolic execution engines and checking the possible errors based on tracking results. However, such a strategy is extremely hard and even practically impossible, for there will usually be millions of internal memory objects to maintain during execution. Therefore, maintaining the checking of those internal memory objects could be time-consuming, which aggravates the scalability and performance issues in symbolic execution.

In SymLoc, the potential of symbolic memory locations is further activated to perform symbolic memory tracking for reliably detecting temporal memory errors. In general, SymLoc reuses the capability in symbolic execution, i.e., automatically propagating and tracking the use of symbolic addresses during execution, and adds our designed checking strategy to reliably detect temporal memory errors. The designed checking strategy is straightforward but effective. Note that the reliability of error detection designed in SymLoc has two important forms. First, SymLoc is able to effectively catch those errors. Second, SymLoc could precisely report the root cause information to end-users when a potential error is detected. For example, existing symbolic execution engines (e.g., KLEE [13]) usually emit bogus "*memory error: out of bound error*" to end-users, which may mislead the developers in the debugging process and significantly affect the development progress. In contrast, SymLoc could precisely report "*memory error: a use after free is detected*" which could quickly help developers locate the root cause of the error. It is also worth noting that existing symbolic execution engines can detect the errors in the form of out-of-bound memory accesses, invalid pointers, etc.; however, it would be troublesome for them to reliably identify the exact *types* of errors as they would need to implement extra analysis to track, relate, and check the uses of the potentially enormous number of memory objects with concrete addresses.

Algorithm 1 presents the details of symbolic memory operations and tracking designed in SymLoc. The function `SymAddrRes` takes *symLocMap*, a symbolic address *SymExpr*, and a parameter *func* as inputs and returns a temporal memory error, a concrete address, or the original symbolic variable (when unresolved). Since a symbolic address can

Algorithm 1: Symbolic memory operations and tracking

```
   Input: the map symLocMap, a symbolic expression symExpr,
          and a function func being executed
   Output: a concrete or symbolic expression, or an error
1  conExpr ⟵ ∅ // initialize a concrete expression
2  FreeList ⟵ ∅ // initialize a list to store freed objects
3  Function SymAddrRes (symLocMap, symExpr, func):
4      std::string fname = "free";
       // Situation 1: handle read/write for normal functions
5      if (fname.compare(func->getName()) != 0) then
6          if symLocMap.find(symExpr) then
7              detectUAF(symExpr, FreeList)
8              conExpr = getAddr(symLocMap, symExpr)
9              return conExpr
10         else
11             return symExpr
       // Situation 2: handle free function
12     if (fname.compare(func->getName()) == 0) then
13         if symLocMap.find(symExpr) then
14             detectDoFOrInF(symExpr, FreeList)
15             FreeList.add(symAddr)
16             conExpr = getAddr(symLocMap, symExp)
17             return conExpr
18         else
19             return symExpr
```

be used in different situations, e.g., normal read/write, parameters for calls to external functions that cannot be symbolically executed, and a memory object to be freed, we check for two situations when encountering a symbolic address based on their differences. First, they handle different usages of symbolic expressions that involve the symbols defined by SymLoc. In terms of implementation, the first situation (with the function call name not equals to "free") is implemented in functions `executeMemoryOperation` and `callExternalFunction` while the second situation (with the function call name does not equal to "free") is used for a special function handler in the function `handleFree`. Second, they indicate the locations for different temporal error detection: SymLoc implements two different functions detectUAF and detectDoFOrInF to detect UAF, DoF, or InF errors. Before handling those situations, the algorithm first initializes a concrete address *conAddr* to be returned (if any) and a list *FreeList* to record freed objects (Lines 1-2). Then, inside these situations, if the symbolic address *symAdddr* is in the map *symLocMap* (the *if* condition in Line 6 or 13 is true), each situation first checks the use of symbolic addresses aiming to detect potential temporal memory errors. In the following, if there are no such errors, the associated concrete address *conAddr* is returned (Lines 9 and 17) by the calling function *getAddr* (Lines 8 and 16); Otherwise, the *SymExpr* is simply returned (Lines 11 and 19) to be handled by existing resolution strategies in symbolic execution engines.

For more complex scenarios, we replace the symbol with a stored concrete address (i.e., base), and other portions (e.g., offset or scale) are kept the same in KLEE. For example, if the engine got a symbolic expression "α + 100" when writing over an object, where α is the symbolic address symbolized by SYMLOC and 100 is the offset, SYMLOC will leverage Algorithm 1 to replace the symbolic α to a mapped concrete value in the *SymLocMap*, say, 0x555555784d654. Then, the symbolic expression "α + 100" will be written back to a constant expression "0x555555784d654 + 100" and the write operation will be conducted over the constant expression to avoid the potential state explosion due to forking on α and thus enabling further execution. When ITE (If-Then-Else) involves different addresses, SYMLOC inherits the state-maintaining strategy when forking new states: if the forked ITEs are an *EqualExpression* and a *NotEqualExpression*, the first state will use the concrete value that complies with the constraints and the second will use the original symbolic value with the concretely mapped address when memory operations are involved; if the ITE is not an *Equal* or *NotEqualExpression* (e.g., *GreaterExpression*), SYMLOC uses the symbolic variable with the same concrete mapped address in both two forked states. Note that in the latter case, we assume the heap allocator could allocate a memory object under some conditions, and the concretely mapped addresses are only used for assisting further path exploration without trapping by the path explosion due to the forking from the symbolic address. Furthermore, If the writes happen on the same path, then the later write would overwrite the previous written values. This operation is inherited from KLEE's design for state forking.

During the symbolic execution process, SYMLOC records all the freed memory objects into a *FreeList* in Line 15, and such a recording will reliably guide the detection of temporal memory errors. Due to the different root causes of three main kinds of errors, SYMLOC performs the corresponding checking on them by invoking `detectUAF` (in Line 7) or `detectDoFOrInF` (in Line 14) function. Those two functions take the symbolic address *symAddr* and the list *FreeList* as inputs and report potential UAF, DoF, or InF errors. For detecting UAF and DoF errors, SYMLOC directly checks whether the symbolic variable under handling is in *FreeList*. For detecting InF errors, SYMLOC checks the type of symbolic expression to decide whether the pointer pointers to the beginning of a heap object or does not point to any heap object.

With the above capabilities, SYMLOC is able to perform efficient symbolic memory operations and could reliably detect the UAF error in Fig. 2 when the freed variable *buf* in Line 4 is used in the external function call `printf` in Line 8, where the error is reported in Line 11 in Algorithm 1. It is worth noting that there are static/dynamic detectors (e.g., Frama-C [9], Coccinelle [10], Valgrind [11], and Asan [12]) were designed to detect such temporal memory errors but they need extra analysis algorithms to identify and track the uses of memory objects to detect errors following certain patterns. SYMLOC essentially enables the tracking of memory locations by conveniently utilizing the capability of symbolic execution: it shares the same core idea as other program analysis-based checkers, enabling more error-detection capability in symbolic execution engines, but without the need for the troubles of building custom analysis tracking algorithms.

3.3 Implementation of SYMLOC

We implemented SYMLOC on top of KLEE (version 2.1) with LLVM 9.0.0 and STP 2.3.3. We modified KLEE's allocation

API to return symbolic addresses instead of concrete ones and maintained a memory map to support practical symbolic memory operations. Our memory model allows the allocation of memory objects using either concrete or symbolic addresses for different usages (i.e., three options detailed in Section 3.1). We also modified the APIs involving memory read/write, external function calls, and `free` in KLEE to support the normal uses of symbolic addresses and the reliable detection of temporal memory errors (see Algorithm 1 in Section 3.2). Besides, for each test case generated by SYMLOC, a text file that records essential information (e.g., unique names of symbolic addresses, invoking locations, the forking point, and the free point) is provided to help facilitate error verification, debugging, or exploit generation.

SYMLOC does not add extra constraints for each symbolic address when detecting new errors mainly due to the performance issue: more complex constraints for each symbolic hardness the constraint solving and affect bug detection capabilities. However, to reduce potential false positives (e.g., the required address is not in user space) reported by SYMLOC, SYMLOC supports a post-processing option to filter out these cases, i.e., SYMLOC adds extra constraints to validate the validity of each allocated address. To be specific, for each symbolized address, we augmented its constraints to be within the address range of user space (i.e., 0x0-0x7fffffffffff) and re-run the program to filter such error/false positive reports out. We believe such post-processing of reported errors can help users analyze the root causes of the error reports more effectively. For more complex modeling of relationships of different memory objects, we leave it as our future work.

Symbolization strategies for users. Although SYMLOC enables the exploration of more execution paths, introduced symbolic addresses may lead to additional execution states forking and more complex path constraints. To alleviate this problem, our implementation provides three options to end-users, i.e., the full, random, and selective symbolization of addresses. The first option is fully symbolization, which symbolizes every memory location (i.e., returned addresses of all the memory allocations). The second option is random symbolization, which randomly symbolizes memory locations, which can save some computing resources than the first option. The final option is selective symbolization. Such a selection allows users to specify the memory locations to be symbolized. We provide an API (i.e., `klee_make_malloc_symbolic` for this purpose, where the memory location after inserting the API will be symbolized. Users can have their own choices to select a more appropriate way to perform the address symbolization. For example, the first option takes more computing resources and may be used for programs with a small number of memory allocations; the second option can be used when the number of allocated buffers is large and it is unknown yet which allocated addresses may affect the program behaviors during the initial testing of the subject programs; the last option is preferred when users know what are the interesting allocation points in a program to be analyzed.

4 EVALUATION

This section presents our experimental settings and results. We aim to answer the following research questions (**RQs**):

RQ1: How does SYMLOC perform in detecting spatial memory errors?

RQ2: How does SYMLOC perform in detecting temporal memory errors?

4.1 Answers to RQ1

We measure the performance in terms of the number of spatial memory errors detected and code coverage achieved.

4.1.1 *Experimental Settings*

Baseline approaches. We focus on a comparison with the widely used symbolic execution engine KLEE first since we implement SYMLOC on it. We also use a recent symbolic execution engine (*symsize* [14]) that models allocation sizes during symbolic execution, to see the effects of buffer sizes on error detection and code coverage. We did not compare SYMLOC with other static/dynamic memory detectors in RQ1 because SymLoc is built on a symbolic executor KLEE and we mainly aim to investigate the multi-path exploration capability (that is only applicable to symbolic executors) among comparative approaches, and it should be fairer to compare with them. To this end, we used a multi-path exploration mode of comparative symbolic executors and compared them in terms of code coverage and the number of errors detected.

Benchmark programs. We use 15 programs in GNU `Coreutils` (version 9.0) and two large programs (GNU `Make` [59] and `m4` [60]) (cf. Table 1) for the evaluation, as they are commonly used in evaluating various symbolic execution techniques [13], [14], [33], [61], [62]. We excluded some Coreutils programs that: (1) do not invoke dynamic memory allocation through the `malloc` function or (2) may cause non-deterministic behaviors (e.g., `kill`, `ptx`, and `yes`), following existing studies [61], [62].

Running settings. We followed prior work [13], [62] to set symbolic inputs for GNU `Coreutils` programs; we configured the symbolic options based on their input formats and prior work [33] for the two large programs. We use *Breadth First Search (BFS)* to deterministically guide the path exploration of all the comparative approaches. For *symsize*, we run it under *Merging* mode with optimizations. We run the benchmarks with a timeout of one hour per test program, following the same setting as existing studies [13], [14], [33], [61], [62]. Then, we measure the code coverage achieved and the errors detected. Besides, we use the *full* symbolization option to run the small-size benchmarks and the *random* symbolization option to run the large-size benchmarks. We run our experiments on a Linux PC with Intel(R) Xeon(R) W-2133 CPU @ 3.60GHz x 12 processors and 64GB RAM running Ubuntu 18.04 operating system.

4.1.2 *Results*

1) Spatial memory error detection capability. Table 2 presents the summarized results of detected errors by the three approaches. The first column represents the error types, and columns 2-4 record the number of detected errors by each approach. The row of ***Spatial Memory Errors*** represents

TABLE 1
The benchmarks used in the evaluation, with their version, size, and the source lines of code (SLOC)

Benchmark	Version	Size	SLOC
basename	9.0	small-size	1.0K
chroot	9.0	small-size	1.2K
date	9.0	small-size	2.8K
dd	9.0	small-size	2.1K
dircolors	9.0	small-size	1.1K
factor	9.0	small-size	2.2K
head	9.0	small-size	1.4K
ln	9.0	small-size	2.3K
od	9.0	small-size	1.8K
pr	9.0	small-size	2.6K
rm	9.0	small-size	2.6K
seq	9.0	small-size	1.2K
stat	9.0	small-size	2.8K
sum	9.0	small-size	1.6K
tee	9.0	small-size	1.0K
Make	4.2	large-size	20K
m4	1.4.18	large-size	80K
Juliet Test Suite	1.3	-	various

TABLE 2
Results of the overall number of detected errors

Error Types	KLEE	*symsize*	SYMLOC
Spatial Memory Errors	7	8	25
Others	5	4	10
Total	12	12	35

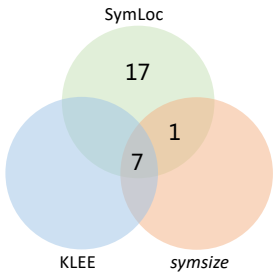


Fig. 6. Distribution of address-specific spatial memory errors detected by comparative approaches

the number of spatial memory errors detected, and the row ***Others*** indicates other errors such as unsupported modeling of certain program states (e.g., symbolic size), unsupported interpreting inline assembly code, and failed external function calls due to symbolic arguments. We show the relation among the errors detected by the three approaches as a Venn diagram in Fig. 6.

We observe that SYMLOC significantly outperforms the others: all the errors reported by KLEE and *symsize* can be detected by SYMLOC, and 17 unique errors are address-specific spatial memory errors that can only be detected by SYMLOC; SYMLOC improves the error-detection capability of KLEE and *symsize* by 169% and 218%, respectively.

To further inspect the causes of the memory errors, we manually analyze the 17 unique errors detected by SYMLOC. We categorize the errors mainly based on the concrete values solved by the constraint solver, and we can check whether a concrete address can happen in a user-space (0x0-0x7fffffffffff) or kernel-space (0xffff800000000000-0xffffffffffffffff) memory region:

Type 1: User space errors. A large portion (i.e., nine) of the errors are of this type, i.e., they can be potentially reproduced in user space.

Type 2: Kernel space errors. Five errors happen when the memory locations are in kernel space.

Type 3: Mixed-space errors. Three errors are under this type. The locations of their memory objects are a combination of user-space and kernel-space, which may involve complex interactions between the user-space program and the kernel's execution to reproduce them.

Since the memory errors in Type 2 and Type 3 are less likely to happen in user-space programs. Therefore, as mentioned in Section 3.3, we use the post-processing option supported by SYMLOC to filter them.

It is worth noting that the remaining errors (i.e., nine) whose addresses are inside user space can have important implications. We carefully checked every error and sorted them into the following categories based on their root causes: (1) two *NULL* pointer dereference issues; (2) four caused by an improper comparison between stack and heap pointers; (3) three caused by a KLEE's optimization issue[5]. For every category of error, we have communicated with developers to seek their confirmation and suggestions. We present more details of two NULL pointer dereference issues in category (1) as case studies in Section 5. For the remaining three issues, we discuss another two representative case studies in categories (2) and (3) in the Appendix.

2) Improved code coverage. Besides the errors detected, code coverage metrics are often used to measure the effectiveness of a software testing tool. We use the tool `klee-stat` in KLEE to collect branch coverage and the tool `gcov` [63] to compute the line coverage information. `klee-stat` measures the "internal coverage" of the program under test, where the coverage is measured at the level at which the symbolic execution engine operates—LLVM *bitcode* instructions [64]. In contrast, `gcov` computes the "external coverage" of the test program. Note that internal coverage such as branch coverage reported by KLEE is more correlated with code coverage and even error-detection capability [65]. For line coverage, we measure the code covered in both the program itself and the libraries used by the program (src+lib). Further, we measure how SYMLOC covers code that is *not* covered by other approaches. Note that we re-run the test cases generated by SYMLOC and measure the code coverage of test programs under test. Therefore, all lines measured are feasible lines covered by the test cases produced by SYMLOC.

Unique line coverage. To calculate the unique line coverage of each approach (say A) with respect to another approach (say B), we first obtain the intersection of the lines covered by both approaches, i.e., $I(A,B) = A \cap B$. Then, the unique line coverage achieved by A is measured as

$$A_u = \frac{N(A - I(A,B))}{N(A - I(A,B)) + N(B - I(A,B))}$$

5. https://gist.github.com/haoxintu/183dda2923965d1e33f64ad59c7f5338

TABLE 3
Results of branch coverage (measured by `klee-stats`) and line coverage (measured by `gcov`)

Benchmarks	Branch Coverage (%)			Line Coverage (src+lib) (%)		
	KLEE	symsize	SymLoc	KLEE	symsize	SymLoc
basename	**29.17**	29.02	**29.17**	**42.20**	38.90	38.90
chroot	33.24	33.75	**33.93**	30.90	**32.40**	30.90
date	17.66	**26.41**	17.74	23.80	**35.80**	23.80
dd	35.96	29.42	**36.77**	**39.60**	32.90	39.40
dircolors	35.39	34.92	**35.60**	44.80	42.40	**46.40**
factor	26.84	22.76	**27.93**	24.00	21.30	**24.70**
head	34.58	31.44	**35.96**	37.70	17.60	**41.70**
ln	27.47	24.93	**27.80**	26.00	19.70	**31.40**
od	**28.43**	23.85	28.39	**42.10**	23.20	40.20
pr	17.99	17.39	**18.24**	34.50	33.70	**34.60**
rm	26.02	24.95	**28.05**	27.70	25.40	**31.80**
seq	34.71	**34.94**	34.71	**34.80**	**34.80**	**34.80**
stat	21.10	19.82	**21.56**	24.20	20.70	**24.80**
sum	36.20	34.72	**36.44**	**30.10**	17.80	30.00
tee	27.75	27.64	**28.19**	**44.10**	**44.10**	**44.10**
m4	4.58	**4.66**	4.47	**9.50**	**9.50**	8.50
make	20.12	15.90	**21.28**	22.10	22.10	**23.80**
Num. of Best	2	3	13	7	5	10

where $N(A - I(A, B))$ represents the number of lines covered by A minus the number of intersected lines. Fig. 7 and Fig. 8 show the results of unique coverage achieved by three approaches on GNU Coreutils. The labels under the *x-axis* indicate the names of the programs, and the values on the *y-axis* represent the unique line coverage of each approach. We can see that SYMLOC achieves higher unique line coverage in most of the benchmark programs. On average, SYMLOC is able to cover 15% and 48% more unique lines than KLEE and *symsize*, respectively. The result is expected as SYMLOC has a unique capability to cover code blocks where the condition depends on the addresses allocated by the heap allocator. For example, existing symbolic execution engines (either KLEE [13] or *symsize* [14]) always assume that the address of "*buf2*" is larger than "*buf1*" in Fig. 1, leading to limited code coverage. However, such an assumption is not held in SYMLOC. Specifically, SYMLOC could generate test cases that exercise unique lines of code, e.g., `Path-C` in Fig. 1 where the address of "*buf2*" is smaller than "*buf1*".

From Fig. 7 and Fig. 8, we can see that some lines of code are uniquely covered by KLEE or *symsize*. The main reason for this is that SymLoc takes relatively more time in constraint solving than the two comparative approaches and in exploring program branches that would not be explored otherwise, thus may have less time in exploring some branches that would be explored sooner by other tools. To support our claim, we use the tool `klee-stats` to get how much time is spent on constraint solving for each approach, i.e., `TSolver` reported by KLEE, where the solving time indicates the percentage of the relative time spent in the solver over the whole program execution time. As a result, for the four benchmarks (i.e., `dd`, `od`, `seq`, and `sum`) that cover more unique lines of code than SymLoc, the results show KLEE spends 76.89%, 35.78%, 3.27%, and 49.83% time over the whole execution time (i.e., 1 hour) on constraint solving, whereas SymLoc takes 79.83%, 41.81%, 19.26%, and 65.41% time on solving constraints, respectively. The results show the same trends when comparing *symsize* with SymLoc. *symsize* contributes 16.54%, 40.40%, and 45.71% constraint solving time, while SymLoc uses 48.71%, 82.23%, and 48.45% constraint solving time over the three benchmarks basename, date, and seq, respectively.

Branch and line coverage. As presented in Table 3, the first column shows the names of benchmark programs, and the rest of the columns record the branch or line coverage over each program under comparative approaches. Columns 3-8 are divided into two groups; each group represents a different coverage metric, i.e., branch coverage or line coverage. Apart from the numbers in the last row, each coverage number is calculated as $\frac{N_{covered}}{N_{total}} \times 100$, where the $N_{covered}$ represents the covered number of branches/lines and N_{total} corresponds to the total number of branches/-lines. The coverage on `m4` and `Make` is the *median* value of five repeated runs. The last *row* counts the total number of the best coverage achieved by each approach for the programs. In terms of branch coverage, we observe that SYMLOC outperforms KLEE and *symsize* overall and dominates 72% (13 out of 18) over all the benchmarks. Specifically, SYMLOC improves at best by 18% (in `factor`) and 25% (in `dd`) than KLEE and *symsize*, respectively.

The line coverage results are shown in the last three columns in Table 3. We observe that SYMLOC is able to cover more code than KLEE and *symsize*; it improves the line coverage by up to 21% (in `ln`) and 137% (in `head`) than KLEE and *symsize*, respectively.

Impact of different symbolization modes. We run random and full modes in one of the large-scale benchmarks (i.e., `Make-4.2`) and compare branch coverage and the memory error detection capability of SYMLOC. Fig. 9 shows the detailed results. As shown in the left side of the box plot in Fig. 9(a), where the *x-axis* represents two different symbolization modes and the *y-axis* describes the branch coverage `Bcov` or solving time `TSolver` reported by KLEE. We can see the branch coverage in the *Full* mode is lower than the one under *Random* mode. This is reasonable because *Full*

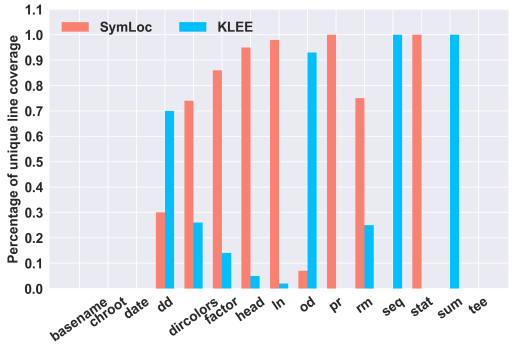


Fig. 7. Unique line coverage (measured by `gcov`): SYMLOC vs KLEE

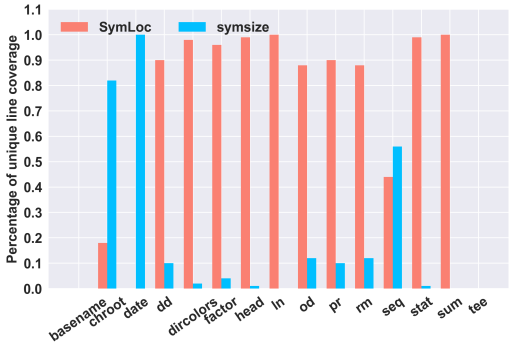
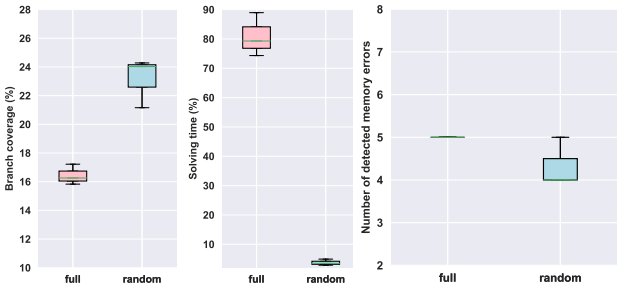


Fig. 8. Unique line coverage (measured by `gcov`): SYMLOC vs *symsize*



(a) Branch coverage and constraint solving time over `Make-4.2` (b) Number of detected errors over `Make-4.2`

Fig. 9. Comparison of different symbolization modes in SYMLOC (*Full* symbolization mode VS *Random* symbolization mode)

symbolization means more complex constraints during the symbolic execution, which leads to more time for constraint solving. As shown on the right side in Fig. 9(a), we can observe that the constraint-solving time in *Full* mode is significantly larger than the one in *Random* mode, which supports our claim. For the error detection capability, the *Full* mode is better than the *Random* mode. This is because the full symbolization mode has a better chance to explore more execution paths, thus making a larger number of errors detected. We did not report the *Selective* mode because this is designed for users who have a certain knowledge of the target program and know how to select a better object to be symbolized. We have manually tested this mode on selected benchmarks and the results show it works as expected.

SUMMARY: SYMLOC is able to detect 169% and 218% more spatial memory errors as well as cover 15% and 48% more unique lines of code than the two baseline approaches.

4.2 Answers to RQ2

We measure the performance in terms of the number of temporal memory errors detected and detection time costs.

4.2.1 *Experimental Settings*

Baseline approaches. We evaluate SYMLOC against different kinds of state-of-the-art memory error detectors. We use two static (Frama-C [9] and Coccinelle [10]) and two dynamics (Valgrind [11] and Asan [12]) detectors studied in [3], as they are shown to be the top approaches in the categories of static and dynamic memory error detectors. For symbolic execution-based approaches, we opt for *symsize* [14] and several variants of KLEE for error detection, including, KLEE(DEF), KLEE(DET), KLEE(OPT), and KLEE(DET+OPT): KLEE(DEF) is the default setting of KLEE; KLEE(DET) enables "*–allocate-determ*" option for deterministic allocation on top of the default KLEE; KLEE(OPT) compiles the source code with a higher compiler optimization (i.e., "*-O1*"); KLEE(DET+OPT) turns on the "*–allocate-determ*" and "*-O1*" together. The choice of w/wo the deterministic allocation or higher optimization is justified by the fact that different configurations of KLEE can have different effects [66] and the two selected options could affect the temporal memory error-detection capability of KLEE as confirmed by KLEE's authors [44].

Benchmark programs. We use the C/C++ programs in Juliet Test Suite (JTS) [34] (cf. the last row in Table 1) that use the `malloc` function in this subsection. It includes 137 programs in CWE416 that have known UAF and 283 programs in CWE415 that have known DoF. Note that since the benchmarks used in RQ1 rarely have temporal memory errors (We run Coccinelle on the benchmarks used in RQ1 and the results show no single temporal memory error is detected), we then used the wide-used benchmark JTS for a fairer comparison. So, we compared SymLoc with both symbolic executors running single-path mode and static/-dynamic tools over the JTS benchmarks in terms of the number of memory errors detected.

Running settings. We use the Frama-C (version Phosphorus-20170501) with "*-val*" and Coccinelle (version 1.0.4) with the UAF patterns specified with its official UAF (*osdi_kfree.cocci*) and DoF (*frees.cocci*) scripts to static analyses each test program. Valgrind (version 3.13) is used after compiling with "*gcc-7.5 -O2*". Asan (the built-in version in LLVM-10) is run with "*clang-10 -fsanitize=address -O2*", following the existing setting [3]. We applied the same setting to symbolic execution-based approaches on the same testing machine as RQ1.

4.2.2 *Results*

1) Comparison with static/dynamic memory detectors. Fig. 10 and Fig. 11 show the experimental results, where the labels under the *x-axis* correspond to the number of errors, and the values on the *y-axis* represent the total number of errors (the number on the right side of the bar) or the completeness of detecting all the errors (the percentage point inside the bar) detected by each detector. We can see that SYMLOC performs the best, detecting all (100%) the UAF and DoF errors in the JTS benchmark.

Reasons for detection failures. Static detectors (i.e., Frama-C [9] and Coccinelle [10]) can not detect certain errors due to two reasons. First, Frama-C is limited in its inter-procedural analysis and Coccinelle is limited in the comprehensiveness of its error detection patterns (which usually needs expert knowledge and is time-consuming to craft), which are insufficient to catch all possible UAF or DoF errors, especially inter-procedural ones [3]. Second, both the Frama-C and Coccinelle are limited in analyzing C++ language so they miss errors written in C++. Dynamic detectors (i.e., Valgrind [11] and Asan [12]) miss errors mainly due to their capability of handling certain C library functions, e.g., `wmemset` and `wprintf` presented in Fig. 2. Adding support for those functions is practically feasible, but it can amply much overhead. More specifically, they all use *shadow memory*, i.e., one-level or multi-level lookup tables, to store the state of all memory objects. When more functions are supported, the number of stored objects can be numerous which may significantly enlarge the search space. Furthermore, Asan can only detect 44.5% DoF errors, as its memory error detection is significantly affected by aggressive compiler optimization levels due to the source-code level instrumentation [67].

For DoF errors, although SYMLOC and Valgrind are able to detect all the DoF errors in JTS datasets, SYMLOC can still have advantages in detecting more errors in general. For example, consider the example code in Fig. 14 (adapted from

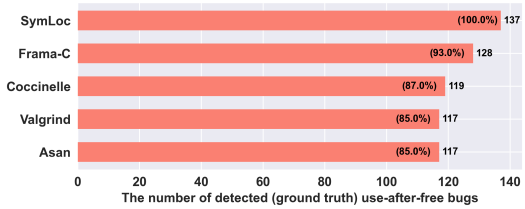


Fig. 10. Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total)

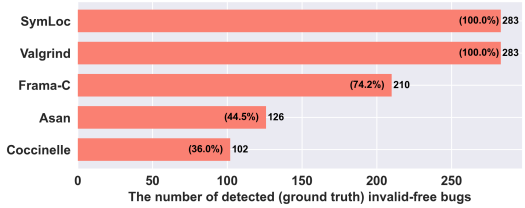


Fig. 11. Completeness of DoF error detection among static/dynamic analysis-based approaches (283 in total)

a GitHub issue[6]) beyond the JTS dataset; Valgrind cannot detect this DoF sometimes due to the same reason why they miss UAF errors: the allocated buffer in Line 3 can have the same concrete address as the allocated buffer in Line 1 after *buff1* is freed. In contrast, SYMLOC has no such problem, and it attributes the freed object to the free list so that it will report errors if some of the objects in the list are used later. It is worth noting that some dynamic analysis tool fails to detect such errors as well. For Valgrind, it cannot detect this bug mainly due to the limited modeling of dynamically allocated memory objects as well. Valgrind maintains a "shadow memory", which is a mirror of the actual memory used by the program. For every byte of memory in the application, there is a corresponding byte (or bytes) in the shadow memory that records whether the application's byte is defined, undefined, or addressable. However, such modeling of the heap still treats the address of x and y differently, causing the miss detection of the double-free error. Note that in a native execution of the program, we run this example 10000 times, and the executable always treats the address of x and y the same and terminates the execution with an error with "*free(): double free detected in tcache 2, Aborted (core dumped)*". Therefore, based on the above explanation, we believe this is an error that has a high possibility to occur in real-world execution.

2) Comparison with variants of KLEE. For UAF error detection, Fig. 12 shows the overall experimental results. Since KLEE and most of its variants detect the same number of errors, we use "KLEE(s)" to refer to each of those detectors (either KLEE(DEF), KLEE(DET), KLEE(OPT), or KLEE(DET+OPT)) and do not repeat showing them. From Fig. 12, we can see that SYMLOC performs the best and detects all (100%) the UAF and DoF errors, while other detectors miss 27% of UAF memory errors and some variants (i.e., KLEE(OPT) and KLEE(DET+OPT)) miss nearly half (i.e., 54.4%) of DoF memory errors in the used benchmarks.

6. https://github.com/staticafi/symbiotic/issues/89

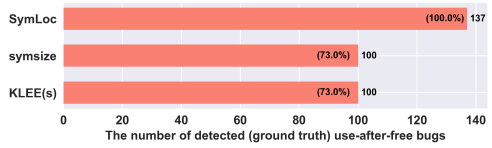


Fig. 12. Completeness of UAF error detection among symbolic execution-based approaches (137 in total)

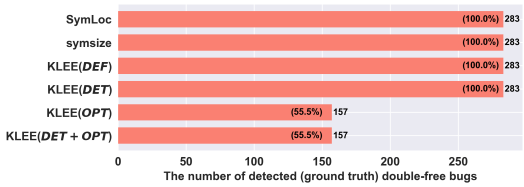


Fig. 13. Completeness of DoF error detection among symbolic execution-based approaches (283 in total)

```
1 int main() {
2     void *x = malloc(100);
3     free(x);
4     void *y = malloc(100);
5     if (x == y)
6         free(y);
7     free(y); // a double free error
8 }
```

Fig. 14. A simple DoF error missed by KLEE [13] and Valgrind [11]

Reasons for detection failures. KLEE and its variants miss some certain UAF and DoF memory errors mainly due to the fundamental issue mentioned in Section 2.2.2, where the addresses of freed memory objects may overlap with addresses returned in subsequent allocations and thus the freed objects are mistakenly treated as "valid", where the objects should be marked as invalid instead. Besides, enabling "*-allocate-determ*" or with higher optimization "*-O1*" does not help alleviate the fundamental issue in KLEE's internal design. Worse still, the higher optimization may even aggregate the problem due to the fact that compiler optimization can be too aggressive [68], [69], so this is the reason why KLEE(OPT) and KLEE(DET+OPT) miss nearly half of Dof memory errors. Note that although the KLEE(DEF) and KLEE(DET) detect the same errors as SYMLOC, our approach can still have advantages in detecting more memory errors in general. For example, consider the example code in Fig. 14 again, which is also confirmed by experts [70]; KLEE and its variants cannot detect this DoF sometimes due to the same reason for missing UAF errors.

It is worth noting that the error information reported by SYMLOC is more precise as aforementioned in Section 3.2.2. Specifically, SYMLOC could directly report the root cause of the error such as "*use after free*", or "*double free*", while KLEE only yields unclear ones such as "*out of bound error*" or "*invalid free*". There is no doubt that more precise information could assist developers in quickly debugging and fixing potential errors.

Speed comparison. We also measured the time of the comparative approaches. For symbolic execution-based approaches, at first, we count the time spent on testing all the programs. The results show that SYM-

LOC, *symsize*, KLEE(DEF), KLEE(DET), KLEE(OPT), and KLEE(DET+OPT) take 150.5, 150.3, 152.8, 152.9, 152.9, and 155.1 seconds on running 137 programs that contain UAF errors, respectively. For DoF errors, those approaches spend 323.0, 321.2, 332.3, 333.9, 333.3, and 335.2 seconds to finish 283 programs that include DoF errors, respectively. Second, for the four approaches to detect the same number of DoF errors as presented in Fig. 11, we compared the time spent in detecting each of the DoF errors. The results show that SYMLOC, *symsize*, KLEE(DEF), and KLEE(DET) spent 1.14, 1.12, 1.18, and 1.18 seconds to detect each error, respectively, indicating SYMLOC has a relatively lower overhead in detecting temporal memory errors. The results are reasonable as existing symbolic execution engines (e.g., KLEE [13]) usually look up the freed memory object from its self-maintained huge memory management pool, which may take time. *symsize* takes less time than SYMLOC as it has a strategy to reduce the time in executing loops when the number of iterations depends on the buffer size; e.g., in the programs, KLEE and SYMLOC currently only use 100 as the default buffer size, while *symsize* tries small ones such as 1 first. In contrast, SYMLOC catches those errors by directly searching the memory objects in *FreeList* mentioned in Algorithm 1, resulting in a better performance.

For the other memory detectors, Coccinelle, Frama-C, Valgrind, and Asan spent 12.1, 60.5, 83.2, and 40.7 seconds to finish all 137 test programs that contain UAF errors, while 31.8, 150.4, 180.3, and 88.9 seconds to run over all 283 test programs that contain DoF errors. The results are reasonable as they are very different approaches from symbolic execution and are expected to be faster than symbolic executors as they do not need to interpret/simulate program executions.

SUMMARY: SYMLOC has an overall better temporal memory error detection capability for detecting UAF and DoF errors than other state-of-the-art static, dynamic, and symbolic execution-based approaches.

5 CASE STDUIES

This section showcases two errors detected by SYMLOC but missed by other tools and discusses their implications.

5.1 Case 1: Single *NULL* Pointer Dereference in `rm`

NULL pointer dereferencing is a dangerous operation in memory assessing [7]. SYMLOC can detect certain *NULL* pointer dereferencing caused by improper checking on allocated pointers in complex programs.

Error details. Fig. 15 presents a code example showing an error that happens when programmers insufficiently check a possible *NULL* pointer that is passed across multiple functions in multiple source files. The functionality of the `rm` function in Line 2 is to first find the target files/directories and then remove found targets if any. In the `rm` function, it opens files from one of the argument *file* in Line 4 and starts to read the file inside a *while-loop* in Line 6 through the function `fts_read`. Inside the `fts_read` function, several *if* checkings are performed before it sets up the directory environment via function `setup_dir` in Line 19. Inside function `setup_dir`, the object **fts->fts_cycle.state** is allocated through function `malloc` in Line 36, and a *NULL* pointer

```
1  // From the ./src/remove.c file:
2  enum RM_status rm (char *const *file, struct rm_options const *x) {
3    ...
4    FTS *fts = xfts_open (file, bit_flags, NULL);
5    while (true) {
6      FTSENT *ent = fts_read (fts); ...
7    }
8  }
9  // From the ./lib/fts.c file :
10 FTSENT * fts_read (register FTS *sp) {
11   ...
12   if (sp->fts_cur == NULL || ISSET(FTS_STOP)) return (NULL);
13   p = sp->fts_cur;
14   ...
15   if ((p = p->fts_link) != NULL) { ...
16     if (p->fts_level == FTS_ROOTLEVEL) {
17       if (restore_initial_cwd(sp)) { ... }
18       ...
19       setup_dir(sp) ;
20       goto check_for_dir;
21     }
22   }
23   ...
24 check_for_dir:
25   ...
26   if (p->fts_info == FTS_D) {
27     if (! enter_dir (sp, p)) { ... }
28   }
29   ...
30 }
31 // From the ./lib/fts-cycle.c file :
32 static bool setup_dir (FTS *fts) {
33   if (fts->fts_options & (FTS_TIGHT_CYCLE_CHECK |
         FTS_LOGICAL)) {
34     ...
35   } else {
36     fts->fts_cycle.state = malloc (sizeof *fts->fts_cycle.state);
37     if (! fts->fts_cycle.state)
38       return false;
39     cycle_check_init (fts->fts_cycle.state);
40   }
41   return true;
42 }
43 // From the ./lib/cycle-check.c file:
44 void cycle_check_init (struct cycle_check_state *state) {
45   state->chdir_counter = 0;
46   state->magic = CC_MAGIC;
47 }
48 // From the ./lib/fts-cycle.c file :
49 static bool enter_dir (FTS *fts, FTSENT *ent) {
50   ...
51   if (cycle_check (fts->fts_cycle.state, ent->fts_statp)) ...
52 }
53 // From the ./lib/cycle-check.c file:
54 bool cycle_check (struct cycle_check_state *state, struct stat const *) {
55   assure (state->magic == CC_MAGIC); // out-of-bound error
56   ...
57 }
```

Fig. 15. Case 1: memory error at `cycle-check.c:60` in `rm`

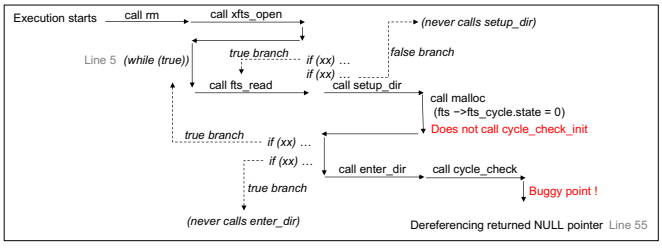


Fig. 16. Execution flow of `NULL-pointer dereference` in Case 1

checking is applied in Line 37. If the allocation returns *NULL*, the `setup_dir` function returns *false* in Line 38. Subsequently, the `fts_read` function continues to check for certain conditions and calls the `enter_dir` function.

The *NULL* pointer dereference error occurs when the allocation in function `setup_dir` returns *NULL* to the object

```
1  #define OUT_OF_MEM() O (fatal, NILF, _("info"))
2  #define O(_t,_a,_f)   _t(( _a),  0,  (_f))
3
4  void * xrealloc (void *ptr, unsigned int size) {
5    void *result;
6    result = ptr ? realloc (ptr, size) : malloc (size);
7    if (result == 0)
8      OUT_OF_MEM();
9    return result;
10 }
11 void fatal (const floc *flocp, size_t len, ...)  {
12   len += (strlen (fmt) + strlen(program) + (flocp && flocp->filenm ?
           strlen(flocp->filenm):0)+INTSTR_LENGTH+8+strlen(stop)+1);
13   char * p = get_buffer (len);
14   ...
15   die (MAKE_FAILURE);
16 }
17 static struct fmtstring {char *buffer; size_t size;} fmtbuf = {NULL, 0};
18 static char * get_buffer (size_t need) {
19   if (need > fmtbuf.size) {
20       fmtbuf.size += need * 2;
21       fmtbuf.buffer = xrealloc (fmtbuf.buffer, fmtbuf.size);
22   }
23   fmtbuf.buffer[need-1] = '\0'; // out-of-bound error
24   return fmtbuf.buffer;
25 }
```

Fig. 17. Case 2 : memory error at `output.c:605` in `Make-4.2`

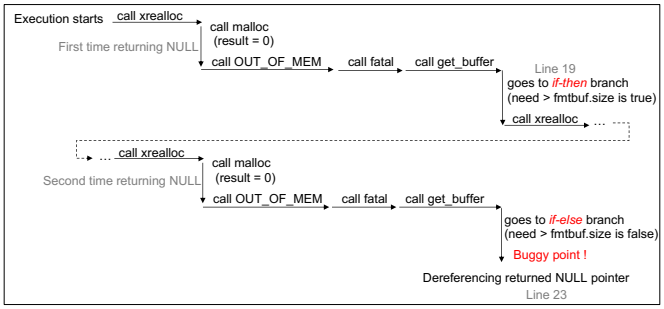


Fig. 18. Execution flow of `NULL-pointer dereference` in Case 2

fts->fts_cycle.state in Line 36 and the subsequent invocation of `enter_dir` in Line 27 in the function `fts_read` uses the **fts->fts_cycle.state** object. To be specific, the object **fts->fts_cycle.state** with *NULL* value is used via function `cycle_check` in Line 51 and the *NULL* pointer is finally dereferenced in Line 55 (reported as "out-of-bound" error). Fig. 16 shows the execution flow of Case 1, where the arrows with the solid line represent the calling flow and arrows with dashed lines refer to the control flow for the error triggering. We can see that the error happens if the object **fts->fts_cycle.state** is assigned with *NULL* value but is used across many functions in multiple files. The developers have fixed the bug by adding an `if` to check if `setup_dir(sp)` returns true[7] in the latest versions of `rm`.

5.2 Case 2: Consecutive *NULL* Pointer Returns in `Make`

Error details. Fig. 17 shows a memory error that occurs when programmers lack the handling of consecutive *NULL* pointer returns. Normally, a *NULL* pointer checking will be conducted after the allocation of the memory. For example, in the function `xrealloc` in Line 4, the return value *results* of the allocation (either by invoking `realloc` or `malloc`) was checked in Line 7. If this value equals 0, the program will be terminated using the function `OUT_OF_MEM` in Line 8 as expected. The implementation of `OUT_OF_MEM` (Line 1) is a macro definition of the function `O` (Line 2) which finally invokes function `_t` (i.e., function `fatal` presented in Line 11). However, a memory error can happen when the `xrealloc` function is called *recursively* through the `OUT_OF_MEM` macro and its returned *NULL* pointer is dereferenced via *fmtbuf.buffer*.

Fig. 18 shows the execution flow of Case 2. To be specific, the function `fatal` (Line 11) is invoked by the macro `OUT_OF_MEM`. The `fatal` function first allocates a buffer via function `get_buffer` (Line 13) based on the needed length *len* and terminates the execution via function `die`. The function `get_buffer` first checks whether the needed size *need* is larger than the buffer size *fmtbuf.size* in Line 19. The *need* argument is passed through function `fatal` and updated in Line 12 (holds a non-zero value) and *fmtbuf.size* is initially 0 (set in Line 17), so the if-then-branch in Lines 20-21 will be executed when the `get_buffer` is invoked for the first time, and the buffer *fmtbuf.buffer* will be updated through another call to `xrealloc`. The key point is that the function `OUT_OF_MEM` (invoked when the *result* gets a 0 when invoking `realloc` or `malloc` inside `xrealloc`) does not immediately terminate the program execution, and `OUT_OF_MEM` will continue to allocate another buffer via a second call to `xrealloc` for printing purposes. Then, the memory error will happen when the `malloc` buffer allocation function in the second call to `xrealloc` returns 0 and triggers `OUT_OF_MEM` again: since the *fmtbuf.size* was defined in the global scope and was set to double the needed size in Line 20, the condition of *if-statement* in Line 19 will be *false* when `get_buffer` is called again while the *fmtbuf.buffer* will still keep the original *NULL*. Then, the dereferencing of the pointer *fmtbuf.buffer[need-1]* will lead to the *NULL-pointer dereference* memory error (reported as "*out-of-bound*" error) in Line 23.

Note that we reported both of the errors in Case 1 and Case 2 to the developers and they have confirmed the issues as true bugs [71], [72]. The developer also acknowledged the quality of the report. For example, during the discussion of Case 2 with developers, one developer mentioned:

> *"There's a nice catch there - where, in that recursive failure, the writing of that terminator overflows a buffer that wasn't actually reallocated yet."*

5.3 Can existing tools detect the errors?

Although many approaches are proposed to detect such errors, it is still challenging to capture cases involving complex control or data flows in the test program. Static analysis approaches should be able to detect them in theory. However, after we ran three well-known static analysis-based memory detectors, including cppcheck [73], clang static analyze [74], and OCLint [75], on the same benchmarks[8], the results showed that none of these tools could detect the two errors.

We note that conservative static analysis techniques should have the capability to find the errors in theory by over-approximating all possible execution flows through *loops* and *if* checks across many functions and multiple

7. https://git.savannah.gnu.org/cgit/gnulib.git/commit/?id=f17d397771164c1b0f77fea8fb0abdc99cf4a3e1

8. We tried to run Frama-C and Coccinelle but they failed to detect any of them due to the lack of supports for certain language features in Frama-C or desired error detection patterns in Coccinelle.

source files. However, to reduce false positives and make such techniques practical, they usually make certain trade-offs between soundness and completeness in their implementations and have limited supports for path-, flow-, context-, object-, and field-sensitive modeling of the code [76], [77], [78], leading to the missed errors in Case 1 and Case 2. For example, to detect the *NULL* pointer deference in Case 1 in Line 55, many branching conditions need to be analyzed (e.g., if checks at Lines 12, 15, 16, 17, 26, 33, and 37). Also note that the code example in Fig. 15 is a simplified version of the original code in order to illustrate the bug-triggering flow more clearly; the actual code involves many more if-checks and function calls across a few thousand lines of code in multiple files (e.g., `fts.c`, `fts-cycle.c`, and `cycle-check.c`); it can be computationally very expensive to track the inter-procedural flows of the FTS pointer across multiple source files in a path-/flow-/context-/object-/field-sensitive way, and the static analysis tools may have certain trade-offs in their implementations, limiting their accuracy in modeling the program semantics and thus missing the detection of such errors in practice.

Dynamic analysis-based tools are less likely to catch the error as well due to the lack of concrete inputs to run the test programs and the low possibility that the function `malloc` returns *NULL* in a runtime environment.

For other symbolic execution-based approaches, as aforementioned in Section 1, they are limited by the modeling of dynamically allocated memory objects, thus missing the detection of two errors.

How does SYMLOC detect the errors? Since SYMLOC supports path constraints encoded with symbolized addresses with concretely mapped symbolic memory operations, SYMLOC can explore deeper execution paths and fork more states involving `malloc` at Line 36 in Fig. 15 and at Line 6 in Fig. 17. Afterward, two execution states (one holds the value of 0 and another keeps the symbol for the allocated objects for further path exploration) are maintained during symbolic execution. Thanks to the nature of symbolic execution, the specific states where the symbolic memory object contains the *NULL* value can easily run through all the code and reach Line 55 in Fig. 15 or Line 23 in Fig. 17, making the successful detection of two *NULL* pointer dereference issues. In summary, SYMLOC is an automatic approach specialized to detect memory errors related to allocation operations, complementary to existing approaches, and a step further to detect more complex memory errors caused by improper pointer operations involving allocated objects.

6 DISCUSSION

6.1 Comparison with Other Existing Approaches

Many other static memory detectors can be facilitated to detect memory errors, but it may not be possible to evaluate them all. We compare and discuss a few more detectors that are most related to SYMLOC.

Comparison with RAM [33]. RAM proposes a relocatable memory addressing model that supports symbolic addresses to facilitate more flexible memory merging and splitting and make constraint solving more efficient in symbolic execution. It is possible to extend RAM for memory error detection purposes. However, the extension will encounter certain difficulties to make RAM achieve the same goal as SYMLOC. First, RAM assumes that the actual address of a memory object should not affect the behavior of the program and does not encode address-related constraints into path constraints, meaning RAM is not able to explore more paths than standard KLEE. However, such an assumption is not always held in real-world programs. For example, the case studies presented in Section 5 show the locations of a memory object matter a lot. Therefore, some important errors that rely on symbolic addresses might be missed due to the limited handling of symbolic memory addresses. Second, RAM relies on a constraint solver to resolve symbolic memory addresses, which can be time-consuming. Therefore, even though the errors do not involve a symbolic address, RAM will spend much time detecting certain errors. To demonstrate the advantages of SYMLOC in the second point in terms of performance, we ran SYMLOC and RAM on the following small piece of code shown in Fig. 19 to confirm whether SYMLOC is prior in terms of performance or not.

The code example simply allocates a buffer *buff* that iteratively writes a value and reads the value from 1 to 100000. We ran this code to compare the efficiency of SYMLOC with RAM. RAM spent 55 seconds finishing the operations while SYMLOC only took 13 seconds. When the situation becomes complicated, e.g., more complex path constraints involving dynamically allocated addresses are involved, the performance downside may be amplified. Overall, SYMLOC could be a complementary approach to RAM and other symbolic execution techniques, where RAM leverages more flexible symbolic addresses to support faster constraint solving, and SYMLOC aims to facilitate a more comprehensive exploration of pointer-related paths. Therefore, SYMLOC can complement RAM to facilitate a more comprehensive exploration of pointer-related paths.

Comparison with MEMSIGHT [57]. MEMSIGHT is an approach that models symbolic memory addresses that reduce the need for concretization. There are two major differences in the memory model design between MEMSIGHT and SYMLOC. First, MEMSIGHT and SYMLOC target to address different problems. MEMSIGHT addresses the problem of how to write or read a memory object that the pointer points to is symbolic. We recognize that this problem is challenging to resolve and MEMSIGHT provides a new memory modeling solution to this end. In contrast, SYMLOC focuses on memory error detection, and our concretely mapped symbolic memory model is for avoiding unnecessary state forking when a pointer is a symbol and continues the execution to facilitate path exploration and error detection. Second, from the implementation perspective, although the newly designed memory model in MEMSIGHT can have better capabilities in theory than ours, MEMSIGHT-KLEE fails to integrate its all features into KLEE due to the fundamental design of KLEE: KLEE utilizes Array Bit Vector (ABV) and MEMSIGHT leverages Bit Vector (BV) while transferring ABV to BV is practically impossible[9]. Limited

9. The authors of MEMSIGHT have explored several implementation solutions but finally failed to get it done mainly due to the difficulties in complex intervening inside several KLEE's internals.

```
1 int main() {
2     int *buff = (int *) malloc (100000 * sizeof(int));
3     for (int i = 0; i < 100000; i++) {
4         buff[i] = i; // memory write via a symbolic address
5         printf(buff[i]); // read via a symbolic address
6     }
7     return 0;
8 }
```

Fig. 19. Example compared with RAM [33]

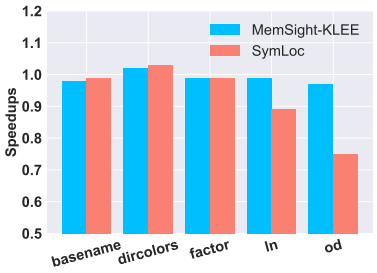


Fig. 20. Speedups comparison between MEMSIGHT-KLEE and SYMLOC

by the above facts, MEMSIGHT-KLEE implements a variant of KLEE that does not alter the address modeling (i.e., it will not explore additional execution paths even though the symbolic addresses are used under conditions) and only optimizes the performance of KLEE's execution. In other words, MEMSIGHT-KLEE inherits one of the drawbacks of the memory model adopted by KLEE: whenever a memory accesses a symbolic object, MEMSIGHT-KLEE will fork a new execution state. To this end, since MEMSIGHT-KLEE can not explore additional paths when compared with the standard KLEE [57], MEMSIGHT-KLEE uses the speedup to evaluate the effect on how MEMSIGHT-KLEE can improve KLEE in terms of performance. Therefore, in this study, SYMLOC and MEMSIGHT-KLEE are comparable only in terms of performance.

For a fair comparison, we use the reported speedup numbers in the paper MEMSIGHT-KLEE and run SYMLOC with the same setting. In particular, we used a fixed number of target instructions to be executed in both KLEE and SYMLOC and counted the speedups based on the common benchmarks used in MEMSIGHT-KLEE and SYMLOC. Fig. 20 presents the comparison results. We can see SYMLOC holds comparable speedups in the first three benchmarks. In the last two benchmarks, SYMLOC performs slower symbolic execution than MEMSIGHT-KLEE mainly because SYMLOC spent more time on constraint solving due to the complex path conditions involved.

Curious readers may still be concerned about the fundamental limitations that prevent us from employing this symbolic memory address in MEMSIGHT to reason about the memory errors and detect similar errors as can be identified by SYMLOC. Unfortunately, even though users choose to run the version of MEMSIGHT-Angr (i.e., the one combining MEMSIGHT with Angr [54]), MEMSIGHT-Angr still cannot detect the same errors reported by SYMLOC. This is because MEMSIGHT-Angr does not support symbolic modeling of constraints involving pointers from the heap due to the lack of implementation and thus loses the opportunities to explore more execution paths when the branch conditions involve heap pointers. In other words, users need to add the same kind of implementation code as SYMLOC into MEMSIGHT-Angr to support heap address modeling, such as the symbolization of heap addresses, and circumvent the challenges of symbolic read/write, for more comprehensive path exploration. It is worth noting that such an implementation requires considerable engineering efforts as it involves a deep understanding of the architecture and complicated details (e.g., the heap and memory management) of Angr. Furthermore, Angr itself does not support bug detection capabilities, leaving it to the expert users to design and implement bug-checking conditions[10],[11]. Thus, even if the support of constraints involving symbolic heap pointers were added, we still need to add extra implementations like SYMLOC to reason about and detect the same kinds of memory errors. We leave such an engineering-intensive extension of MEMSIGHT for future work.

We would like to emphasize the essential differences in the design goal/contributions between MEMSIGHT and SYMLOC. MEMSIGHT's principal contribution is a new memory model for symbolic execution engines such as KLEE and Angr to improve path exploration (involving stack symbolic pointers), but it has no optimizations to existing symbolic execution engines in terms of other important capabilities, e.g., read/write via symbolic heap pointers, or reasoning about memory errors or bug detection. In contrast, the new memory model is only a part of the contributions of SYMLOC, and SYMLOC is also a new memory error detection system that detects many memory errors, including the ones that can be detected by existing engines (e.g., buffer overflow errors) or others that are hard for existing engines to detect (e.g., use-after-free errors). Due to the different design goals and implementations, neither MEMSIGHT-KLEE nor MEMSIGHT-Angr can detect the same errors reported by SYMLOC, unless users add the same implementation as SYMLOC.

Comparison with CRED [76]. CRED is a pointer analysis-based static UAF detector that aims to address the challenge of reasoning about the exponential number of program paths to find bugs at a low false positive rate. However, as a static analyzer, as mentioned in Section 5.5 of its paper [76], CRED suffers from both false negatives and false positives. False negatives are due to limitations on handling loops, linked lists, and array access aliases. False positives are due to its imprecise path reduction and imprecise points-to information for out-of-budget points-to queries. Its evaluation on 10 real-world programs reported 85 bugs, but 47 of them were false positives, with a false positive rate of more than 30%. However, existing studies [79], [80] show that false positives do matter and a common industrial requirement of false positive rate is lower than 30%. If the false positive rate of an approach is higher than 30%, true bugs would be lost among false ones, and developers will discard the approach for industrial uses. In contrast, SYMLOC inherits the advantages of symbolic execution techniques and can be more accurate. In addition, SYMLOC can also provide more precise inputs to help trigger the detected errors, which can

10. https://docs.angr.io/en/latest/faq.html#how-do-i-find-bugs-using-angr
11. https://github.com/angr/angr/issues/1536#issuecomment-487047190

facilitate the debugging and fixing of such errors.

Since CRED is not open-sourced, we have contacted the authors of CRED multiple times and we have not received any response yet. Therefore, we did not run CRED natively to conduct further experimental comparison. We have conducted experiments using a couple of existing static analyzers (i.e., Frama-C or Coccinelle) and the JTS benchmarks, and the results presented in Fig. 10 and Fig. 11 show that they miss the detection of certain UAF and DoF bugs due to limited inter-procedural analysis or error detection patterns. For CRED, the approach only targeted UAF bug detection, it cannot detect other types of bugs (e.g., buffer overflows) that can be detected by SYMLOC. Even for UAF bug detection, as mentioned by CRED authors, CRED suffers from both false positives (due to imprecise path reduction and imprecise points-to information for out-of-budget points-to queries) and false negatives (due to unsound modeling of loops and limited modeling of pointer objects and their fields and contexts during analysis). These limitations may be due to imprecise modeling of various aspects of program semantics (e.g., paths, flows, contexts, objects, fields). Taking the case shown in Fig. 15 as an example, detecting the UAF bug may need more path-, context-, object-, and field-sensitive program analysis, but CRED and other static analyzers have a high possibility of missing the bug inside the function call `fts_read` that is invoked with many *if* checks and other function calls across multiple files. In contrast, SYMLOC, as a dynamic symbolic execution tool, excels at providing more precise information (e.g., the concrete test inputs) to help debug the error once it was detected, with the result aligning the common industrial requirement of 30% false positive rates [79], [80].

We have also tried to fix the LLVM runtime errors, but the code implementation required for fixing the errors is intricate and demanding. For example, there is a list of unsupported LLVM instructions maintained by KLEE[12], and we have encountered the unsupported instruction "*llvm.x86.sse2.packuswb.128*" when running `ghostscript` package. KLEE's developers could not support it for more than three years due to implementation difficulties; supporting it requires a good amount of engineering work based on developers' feedback[13]; it is even harder for us to change the LLVM implementation in a few months. We also acknowledge that symbolic execution techniques have their limitations such as path exploration, which requires future improvements. We are actively working on combining normal and heap address symbolization to alleviate the problem in future work.

6.2 Threats to Validity

One threat lies in the address symbolization strategy designed in SYMLOC. Ideally, a test program may invoke multiple `malloc` functions. When a user selects the full symbolization option, it may produce complex constraints that could significantly slow down the execution. Although we did not design an extra constraint reduction solution to such a situation, we have designed a selective option with a new API (i.e., `klee_make_malloc_symbolic`) which allows users to identify interesting allocation (e.g., those addresses are extensively used in the comparison within a *if* condition) first. Such a strategy could potentially help release some pressure on the solver side. Another threat comes from the test programs. We used selected utilities in GNU `Coreutils`, two larger benchmarks, and JTS. Although they have been widely used for evaluating symbolic execution [13], [14], [33], [61], [62], [81] and temporal memory error detection [3], these programs may not be representative enough for various software systems. We are considering expanding the program sets in our future work.

12. https://github.com/klee/klee/issues/678#issue-235902374
13. https://github.com/klee/klee/issues/1154#issuecomment-531295026

6.3 Limitations of SYMLOC

SYMLOC has an implementation limitation. Different allocation functions (e.g., `malloc`, `calloc`, and `realloc`) are supported in C/C++ programming languages, and SYMLOC so far only supports the symbolization of addresses returned by `malloc`. However, the `malloc` is the basic function used for dynamic memory allocation in the program. We are considering adding support for other allocation functions into the extended version of SYMLOC. Although SYMLOC provides more complete modeling of dynamically allocated memory objects, due to the sophisticated mechanism of heap management, SYMLOC still has some modeling limitations. First, SYMLOC does not support symbolic offset and symbolic size. Symbolic offset is designed in Mayhem [31] while the symbolic size is supported in *symsize* [14]. We plan to integrate them into SYMLOC. Second, when ITE involves two different addresses, SYMLOC is not able to provide the required addresses at run-time. We also recognize that SYMLOC has a limitation on allocating heap buffers to the required concrete addresses that satisfy different path constraints during symbolic execution. That means if SYMLOC forks two states that contain two different addresses (e.g., 0 and 0xffff543400) separately for a symbolic pointer, SYMLOC does not provide a new heap buffer that is located at the concrete address 0 or 0xffff543400 for the pointer in each execution state during symbolic execution. Instead, SYMLOC simply uses the buffer previously allocated for the pointer (if any) maintains the constraints for the symbolic address, and continues the exploration of the paths. We plan to add a heap simulator [82] in SYMLOC to simulate the run-time behavior of the heap allocator in future work to overcome such a limitation.

6.4 Integration with Other Techniques

It can be interesting to integrate various addressing models of RAM [33], *symsize* [14], and SYMLOC together into symbolic execution. RAM's relocatable addressing model improves the ability of symbolic pointer resolution and reduces the cost of solving array theory constraints with big arrays. *symsize* leverages a bounded symbolic-size model that symbolizes the size of allocated objects. SYMLOC considers program behaviors and execution paths that may be affected by memory addresses. Integrating the above three addressing models could yield a more *complete* and *efficient* model for symbolic execution.

7 Related work

This section describes the most related work for detecting memory errors, including static analysis, dynamic analysis, and symbolic execution-based approaches.

Static analysis for memory error detection. Static tools typically leverage pattern matching and abstract interpretation techniques to facilitate the detection of memory errors. Coccinelle [10] employs a given pattern to analyze and certify memory errors, mainly in C programs. Cppcheck [83] checks non-standard code to detect potential memory errors. TscanCode [84] extends CppCheck and detects temporal memory errors using specific syntactic patterns. Regarding abstract interpretation-based approaches, Frama-C [9] implements program safety verification and uses value flow analysis to detect memory errors. Clang static analyzer [74] performs path-sensitive analysis to detect general memory errors.

Dynamic analysis for memory error detection. Typical dynamic memory detectors can be classified into two types based on their strategy of instrumentation. On run-time-instrumentation, Valgrind [11] uses disassembly and resynthesizing technology to detect memory errors. Purify [85] utilizes object code insertion technology to instrument object files with additional instructions. Dr. Memory [4] applies a copy-and-annotate technique to copy the incoming instructions verbatim. On compile-time instrumentation, Mudflap [86] statically inserts the predicate validity assertion at the pointer deference site (i.e., the pointer's use site) to decide whether the pointer accesses valid memory. ASan [12] leverages a direct shadow mapping scheme with compile-time instrumentation to detect memory errors.

Symbolic execution for memory error detection. KLEE [13] is the leading symbolic execution that can detect many memory errors by using a concrete memory model. Recently, David *et al.* propose *symsize* [14] which adopts a bound memory model to model the size of memory allocation. David *et al.* [33] propose a novel symbolic memory model RAM that accesses objects with symbolic offsets which are handled using array theory to facilitate constraint solving in symbolic execution. Matin [87] introduces an enhanced, fine-grain, and efficient representation of memory that mimics the allocations of tested applications. Coppa *et al.* [57] model symbolic memory as a set of tuples, where each tuple associates an address expression to a timestamp-based and a condition-based value expression. Angr [54] and Mayhem [31] share the same index-based memory model that allows a symbolic read under certain conditions.

Unlike existing approaches, our goal is to leverage a symbolic execution-based approach SymLoc to facilitate the detection of memory errors. Unlike existing static/dynamic analysis-based memory detectors, SymLoc not only can detect more memory errors but could also provide useful test cases for those detected errors to facilitate the debugging and fixing processes during software development. Compared with existing symbolic execution engines, we design a new memory model with concretely mapped symbolic memory locations and a new error-detecting mechanism to assist in the detection of address-specific spatial memory errors. Furthermore, SymLoc could reliably detect temporal memory errors with the help of the newly designed symbolic memory tracing technique.

8 Conclusion with Future Work

This paper presents SymLoc using concretely mapped symbolic memory locations to facilitate the detection of memory errors. A new integration of three techniques is designed in SymLoc: (1) the symbolization of addresses and encoding of the symbolic addresses into path constraints, (2) the symbolic memory read/write operations using a symbolic-concrete memory map, and (3) the automatic tracking of the use of symbolic memory locations. Our evaluation results show that SymLoc outperforms various state-of-the-art memory detectors in terms of detecting memory errors involving allocated memory addresses. We also discuss some interesting errors detected by SymLoc but missed by other tools and their implications. We also provide a replication package (https://github.com/haoxintu/SymLoc) of SymLoc to facilitate further research in this area. As part of future work, we are actively pursuing to integrate different addressing models to support more complete, accurate, and efficient memory modeling of program semantics for symbolic execution.

Acknowledgments

The authors would like to thank anonymous reviewers for their insightful comments and all developers who participated in this work. This article is partially supported by the National Research Foundation (NRF) Singapore and the National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) award number NSOE-TSS2019-04.

References

[1] I. A. Astrakhantseva, R. G. Astrakhantsev, and A. V. Mitin, "Randomized C/C++ dynamic memory allocator," *Journal of Physics: Conference Series*, vol. 2001, no. 1, pp. 1–6, 2021.

[2] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "HeapHopper: Bringing bounded model checking to heap implementation security," in *USENIX Security*, 2018, pp. 99–116.

[3] B. Gui, W. Song, H. Xiong, and J. Huang, "Automated use-after-free detection and exploit mitigation: How far have we gone," *IEEE Transactions on Software Engineering*, 2021.

[4] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *CGO*, 2011, pp. 213–223.

[5] R. M. farkhani, M. Ahmadi, and L. Lu, "PTAuth: Temporal memory safety via robust points-to authentication," in *USENIX Security*, 2021, pp. 1037–1054.

[6] M. T. Aga and T. Austin, "Smokestack: Thwarting dop attacks with runtime stack layout randomization," in *CGO*, 2019, pp. 26–36.

[7] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on S & P*. IEEE, 2013, pp. 48–62.

[8] CVE-2022-0667, "Assertion failure on delayed ds lookup," 2022. [Online]. Available: https://kb.isc.org/docs/cve-2022-0667

[9] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *SEFM*, 2012, pp. 233–247.

[10] M. C. Olesen, R. R. Hansen, J. L. Lawall, and N. Palix, "Coccinelle: tool support for automated cert c secure coding standard certification," *Science of Computer Programming*, vol. 91, pp. 141–160, 2014.

[11] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," in *PLDI*, 2007, pp. 89–100.

[12] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC*, 2012, pp. 1–28.

[13] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[14] D. Trabish, S. Itzhaky, and N. Rinetzky, "A bounded symbolic-size model for symbolic execution," in *ESEC/FSE*, 2021, pp. 1190–1201.

[15] J. Cohen, "Contemporary automatic program analysis," 2022. [Online]. Available: https://www.blackhat.com/docs/us-14/materials/us-14-Cohen-Comtemporary-Automatic-Program-Analysis.pdf

[16] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[17] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, pp. 391–425, 2013.

[18] R. Rutledge and A. Orso, "Pg-klee: Trading soundness for coverage," in *ICSE*, 2020, pp. 65–68.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[20] W. Jin and A. Orso, "Bugredux: Reproducing field failures for inhouse debugging," in *ICSE*, 2012, pp. 474–484.

[21] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *EuroCCS*, 2010, pp. 321–334.

[22] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016, pp. 691–701.

[23] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *ICSE*, 2018, pp. 129–139.

[24] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[25] P. Collingbourne, C. Cadar, and P. H. Kelly, "Symbolic crosschecking of floating-point and simd code," in *CCS*, 2011, pp. 315–328.

[26] T. Kapus, O. Ish-Shalom, S. Itzhaky, N. Rinetzky, and C. Cadar, "Computing summaries of string loops in c for better testing and refactoring," in *PLDI*, 2019, pp. 874–888.

[27] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu, "Symbolic path cost analysis for side-channel detection," in *ISSTA*, 2018, pp. 27–37.

[28] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, "Casym: Cache aware symbolic execution for side channel detection and mitigation," *IEEE SP*, pp. 505–521, 2018.

[29] S. Guo, Y. Chen, J. Yu, M. Wu, Z. Zuo, P. Li, Y. Cheng, and H. Wang, "Exposing cache timing side-channel leaks through out-of-order symbolic execution," vol. 4, 2020.

[30] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[31] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE SP*, 2012, pp. 380–394.

[32] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *CCS*, 2019, pp. 1689–1706.

[33] D. Trabish and N. Rinetzky, "Relocatable addressing model for symbolic execution," in *ISSTA*, 2020, pp. 51–62.

[34] N. C. for Assured Software, "Juliet test suite 1.3," 2017. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php/

[35] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *PLDI*, 2009, pp. 245–258.

[36] N. S. Z. J. M. M. M.K. and S. Zdancewic, "CETS: compiler enforced temporal safety for c," in *ISMM*, 2010, p. 31–40.

[37] Z. Chen, C. Wang, J. Yan, Y. Sui, and J. Xue, "Runtime detection of memory errors with smart status," in *ISSTA*, 2021, pp. 296–308.

[38] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities," in *USENIX Security*, 2018, pp. 781–797.

[39] M. safety in the Chromium project, 2020. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[40] "Implementation of memmove in Apple Open Source," 2022. [Online]. Available: https://opensource.apple.com/source/network_cmds/network_cmds-481.20.1/unbound/compat/memmove.c.auto.html

[41] "Implementation of memmove in Glibc," 2022. [Online]. Available: https://github.com/lattera/glibc/blob/master/string/memmove.c

[42] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *ICSE*, 2014, pp. 72–82.

[43] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[44] B. Dolan-Gavitt, "KLEE may miss use-after-free in call to Libc function," 2022. [Online]. Available: https://github.com/klee/klee/issues/1434

[45] A. Sotirov, "Heap feng shui in javascript," 2007. [Online]. Available: https://llvm.org/docs/LangRef.html#introduction

[46] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards automated heap feng shui," in *USENIX Security*, 2021, pp. 1647–1664.

[47] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *USENIX Security*, 2018, pp. 763–779.

[48] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *CCS*, 2019, pp. 1707–1722.

[49] STP, "Simple theorem prover, an efficient smt solver for bitvectors," 2022. [Online]. Available: https://github.com/stp/stp

[50] Z3, "A theorem prover from microsoft research," 2022. [Online]. Available: https://github.com/z3prover/z3

[51] Apple, "Tips for Allocating Memory," 2023. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MemoryAlloc.html#//apple_ref/doc/uid/20001881-CJBCFDGA

[52] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *USENIX Security*, 2020, pp. 1111–1128.

[53] T. Kapus and C. Cadar, "A segmented memory model for symbolic execution," in *ESEC/FSE*, 2019, pp. 774–784.

[54] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE SP*, 2016, pp. 138–157.

[55] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[56] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, no. 3, pp. 265–278, 2011.

[57] E. Coppa, D. C. D'Elia, and C. Demetrescu, "Rethinking pointer reasoning in symbolic execution," in *ASE*, 2017, pp. 613–618.

[58] B. Farinier, R. David, S. Bardin, and M. Lemerre, "Arrays made simpler: An efficient, scalable and thorough preprocessing," in *LPAR*, vol. 57, 2018, pp. 363–380.

[59] G. make, "A building automation tool," 2022. [Online]. Available: https://www.gnu.org/software/make

[60] G. m4, "A traditional unix macro processor," 2022. [Online]. Available: https://www.gnu.org/software/m4

[61] J. He, G. Sivanrupan, P. Tsankov, and M. Vechev, "Learning to explore paths for symbolic execution," in *CCS*, 2021, pp. 2526–2540.

[62] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," *SIGPLAN Not.*, vol. 48, no. 10, pp. 19–32, 2013.

[63] GCOV, "A test coverage program in gnu gcc tool-chain," 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[64] L. IR, "A powerful intermediate representation for efficient compiler transformations and analysis," 2022. [Online]. Available: https://llvm.org/docs/LangRef.html#introduction

[65] C. Cadar and T. Kapus, "Measuring the coverage achieved by symbolic execution," 2022. [Online]. Available: http://ccadar.blogspot.com/2020/07/measuring-coverage-achieved-by-symbolic.html

[66] S. Cha, M. Lee, S. Lee, and H. Oh, "Symtuner: Maximizing the power of symbolic execution by adaptively tuning external parameters," in *ICSE*, 2022, p. 2068–2079.

[67] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE SP*, 2015, pp. 73–87.

[68] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011, pp. 283—294.

[69] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *IEEE S & P Workshops*, 2015, pp. 73–87.

[70] K. Dudka, "Missing a double free when heap pointers are compared," 2022. [Online]. Available: https://github.com/staticafi/symbiotic/issues/89

[71] G. Developer. (2024) Confirmation of the bug in Case 1. [Online]. Available: https://debbugs.gnu.org/cgi/bugreport.cgi?%20msg=10;bug=65269#10
[72] D. of make package. (2024) Confirmation of the bug in Case 2. [Online]. Available: https://github.com/haoxintu/SymLoc/blob/main/experiments/confirmation-case2.pdf
[73] Cppcheck, "A tool for static c/c++ code analysis." 2023. [Online]. Available: http://cppcheck.sourceforge.net/
[74] C. S. Analyzer, "A source code analysis tool that finds bugs in c, c++, and objective-c program," 2023. [Online]. Available: http://clang-analyzer.llvm.org
[75] Oclint, "A static code analysis tool for improving quality and reducing defects," 2023. [Online]. Available: https://oclint.org
[76] H. Yan, Y. Sui, S. Chen, and J. Xue, "Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities," in *ICSE*. IEEE, 2018, pp. 327–337.
[77] W. Griswold, D. Atkinson, and C. McCurdy, "Fast, flexible syntactic pattern matching and processing," in *WPC '96. 4th Workshop on Program Comprehension*, 1996, pp. 144–153.
[78] G. J. Badros and D. Notkin, "A framework for preprocessor-aware c source code analyses," *Software: Practice and Experience*, vol. 30, no. 8, pp. 907–924, 2000.
[79] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, p. 66–75, feb 2010. [Online]. Available: https://doi.org/10.1145/1646353.1646374
[80] S. McPeak, C.-H. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, p. 554–564.
[81] A. Pandey, P. R. G. Kotcharlakota, and S. Roy, *Deferred Concretization in Symbolic Execution via Fuzzing*, 2019, pp. 228–238.
[82] R. Li, B. Zhang, J. Chen, W. Lin, C. Feng, and C. Tang, "Towards automatic and precise heap layout manipulation for general-purpose programs." in *NDSS*, 2023, pp. 1–15.
[83] D. Marjamäki, "Cppcheck: a tool for static C/C++ code analysis," 2023. [Online]. Available: https://cppcheck.sourceforge.io/
[84] Tencent, "A fast and accurate static analysis solution for C/C++, C#, Lua codes," 2023. [Online]. Available: https://github.com/Tencent/TscanCode
[85] R. Hastings, "Purify: Fast detection of memory leaks and access errors," in *Proc. Winter USENIX Conference*, 1992, pp. 125–136.
[86] F. C. Eigler, "Mudflap: Pointer use checking for c/c+," *Proceedings of the First Annual GCC Developers' Summit*, pp. 57–70, 2003.
[87] M. Nowack, "Fine-grain memory object representation in symbolic execution," in *ASE*, 2019, pp. 912–923.
[88] "Building secure software: better than protecting bad software," *IEEE Software*, vol. 19, no. 6, pp. 57–58, 2002.
[89] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004.
[90] J. Viega and G. R. McGraw, *Building secure software: how to avoid security problems the right way*. Pearson Education, 2001.
[91] S. Developer. (2024) SQL Injection Prevention Cheat Sheet. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
[92] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98, 1998, pp. 63–78.
[93] T. Kapus and C. Cadar, "Automatic testing of symbolic execution engines via program generation and differential testing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 590–600.

APPENDIX A
SUPPLEMENTARY CASES STUDIES

A.1 Case 3: Return Malloc Address from Stack in `Make`

Address-specific memory errors, i.e., those that happen when the memory objects were allocated at a specific location (not *NULL*), are rarely considered in the community. The following is a case involving potentially improper pointer comparison between two different types of objects, i.e., one from the stack and another from the heap.

Error details. Fig. 21 illustrates a code example showing the memory error in the address from dynamically allocated memory returned from a stack object. Since the execution flow of this error is quite complicated, to clearly explain the root cause, we first describe the functionality of the key functions involved in the error triggering and then articulate the execution flow to trigger the error.

The function `construct_include_path` (Line 1), which is directly invoked by the `main` function in `Make-4.2` program, constructs the list of included directories from the arguments and the default list. The function `do_variable_definition` (Line 9), called in the *for-loop*, defines a variable by giving a variable *name*, a *value*, and a *flavor*. The function `lookup_variable` (Line 23) aims to lookup a variable whose name matches the *name* and the *len*, and returns the address of the "struct variable" or 0 if no such variable is defined. The function `hash_find_item` (Line 32) finds an item based on the provided *key*. The function `define_variable_in_set` (Line 36) defines a variable with the *name* and *value* in a global variable set: it finds the variable in the hash table first in Line 40 and allocates a new one via `xmalloc` in Line 44 and inserts the object into the hash table via the function `hash_insert_at` in Line 45. The function `hash_find_slot` (Line 49) is the place where the root cause lies. Inside this function, it first gets the hash value *hash_1* by looking up the provided *key*, and then a *for-loop* is applied to search the slot that matches the value of *hash_1*. The error happens when the value of *key* equals the value of **slot* and the object associated with the address of the *key* is dereferenced later in Line 17.

Fig. 22 further describes the detailed execution flow of the memory error triggering. The error occurs in the second iteration in the *while-loop* in Line 4. In the first iteration, since the variable to be looked up is not defined, the value of *v* returned by function `lookup_variable` in Line 12 is 0 and the new variable is stored into the hash table through function `define_variable_in_set`. When the second iteration starts, if the address of the variable *key* equals the allocated address *v* in the first iteration, the function `hash_find_slot` returns the address value of **slot*, where the value equals the value of the *key*. Note that although the address of the dereferenced pointer equals the address as allocated before, the memory region after the pointer *v->value* points to can be modified through the following allocation operation (e.g., the allocation via the function `xstrdup` in Line 45). Once this happens, every memory cell, possibly including the hash table storing the value of *v*, can be overwritten. Finally, the address *v* can be an invalid memory address and dereferencing *v->recursive* leading to the memory error in Line 17.

Note that the address stored in the hash cell (e.g., **slot* at Line 57) is a heap address (as it was allocated from the heap via `malloc`) function in most of the cases when the heap allocator behaves well. However, many existing studies [2], [46] shows that the heap allocator can contain defects that make the heap allocator behave abnormally. Therefore, if the heap allocator contains any errors (this is possible as detecting all bugs in heap allocators is undecidable), it is likely that the address stored in the hash cell (e.g., **slot* at line 57) is a stack address instead of a heap address. Second, we also understand that Line 57 is simply a shortcut to accelerate the hash key comparison during execution, but

```
1  void construct_include_path (const char **arg_dirs){
2    ...
3    dirs[idx] = 0;   //  idx = 3
4    for (cpp = dirs; *cpp != 0; ++cpp)  //add dirs to .INCLUDE_DIRS
5      do_variable_definition (NILF, ".INCLUDE_DIRS", *cpp,
6                              o_default, f_append, 0);
7    ...
8  }
9  struct variable * do_variable_definition (const floc *flocp,
10                          const char *varname, const char *value, ...) {
11   ...
12   v = lookup_variable (varname, strlen (varname));
13   if (v == 0) {  //  first  iteration
14     ...
15   } else {   // Paste the old and new values together in VALUE
16       ...
17       if (v->recursive) //   buggy point (out of bound error)
18       ...
19     }
20   v = define_variable_in_set (varname, strlen (varname), p, ...) ;
21   ...
22 }
23 struct variable * lookup_variable (const char *name, int len) {
24   struct variable var_key;
25   var_key.name = (char *) name; var_key.length = len;
26   ...
27   const struct variable_set *set = current_variable_set_list->set;
28   v = (struct variable *) hash_find_item (&set->table, &var_key);
29   ...
30   return v;
31 }
32 void * hash_find_item (struct hash_table *ht, const void *key){
33   void **slot = hash_find_slot (ht, key);
34   return (( HASH_VACANT (*slot)) ? 0 : *slot);
35 }
36 struct variable * define_variable_in_set (const char *name,
37                          unsigned int length, const char *value, ...) {
38   struct variable *v, **var_slot, var_key;
39   var_key.name = (char *) name; var_key.length = length;
40   var_slot = (variable **) hash_find_slot (&set->table, &var_key);
41   v = *var_slot;
42   ...
43   if (! HASH_VACANT(v)) { ... ; return v;}
44   v = xmalloc (sizeof (struct variable)); //  symbolic object
45   v->value = xstrdup (value);
46   hash_insert_at (&set->table, v, var_slot);  // also assign v to var_slot
47   ...
48 }
49 void ** hash_find_slot (struct hash_table *ht, const void *key){
50   void **slot;  unsigned int hash_1 = (*ht->ht_hash_1) (key);
51   for (;;) {
52     hash_1 &= (ht->ht_size - 1);
53     slot = &ht->ht_vec[hash_1]; // find target address mapping to 'key'
54     if (*slot == 0)
55       return (deleted_slot ? deleted_slot : slot); //  first  iteration
56     else {
57       if (key == *slot) //  constraints leading to the error
58         return slot;  //  problematic return
59       if ((* ht->ht_compare) (key, *slot) == 0)
60         return slot;
61     }
62     ....
63   }
64 }
```

Fig. 21. Case 3: memory error at `variable.c:1244` in `Make-4.2`

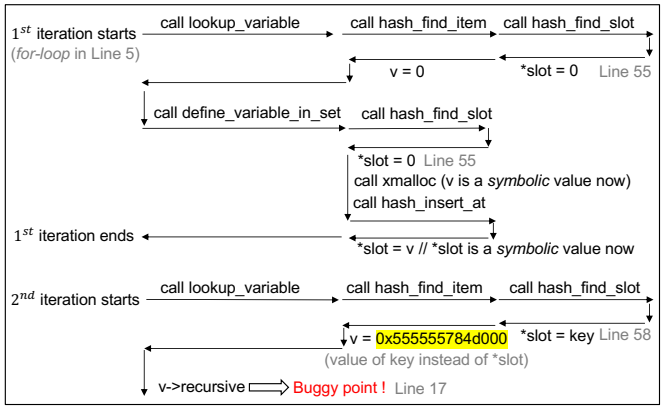


Fig. 22. Execution flow of the `out-of-bound` error in Case 3

the comparison expression in Line 57 only compares the actual value between two pointers and it is hard to conclude that the pointers must point to the heap. Again, if heap allocators make mistakes, although not common (but they do [46], [47]), the "*key*" could be from the stack and Line 57 could be true in an abnormal run without programmers realizing it.

Can existing tools detect it? To the best of our knowledge, SYMLOC may be the first in the community to detect the potentially improper pointer comparison between an address from the stack and another address from the heap. Due to incomplete modeling of dynamically allocated memory objects in their implementations, existing tools failed to detect the error in our experiments although they could theoretically detect it too with more conservative modeling.

How does SYMLOC detect it? Similar to the two cases presented in Section 5, SYMLOC models the address of `v` as a symbol in Line 44 and it is assigned the *var_slot* in Line 46. When encountering the if-branch in Line 57, SYMLOC forks the states and there will be different execution states: one *key* holds the value of **slot* and the other *key* does not hold the value of **slot*. The value of *key* in the former state should be equal to the original value stored in **slot* (i.e., from the heap), but in SymLoc's subsequent execution, it does not (i.e., it changed to a stack address after forking). Therefore, Line 58 will be taken. Since the value of *key* does not equal the value of the original value stored in the **slot*, the returned address of *v* is an invalid memory address, leading to the error when dereferenced by *v->recursive*.

Implications. Returning a dynamically allocated memory address to be in the stack can be a disaster. It may cause severe problems such as follows: (1) data corruption: the program will overwrite the region whatever is on the stack at that position; (2) security vulnerabilities: overwriting the stack can introduce vulnerabilities. For example, attackers can exploit stack-based buffer overflows to run arbitrary code; (3) unpredictable behavior: because of the tight coupling between the function call mechanism and the stack (return addresses, local variables, etc.), corrupting the stack can lead to unpredictable behavior, including unexpected function calls or wrong return values; (4) memory leaks: if users mistakenly think they are dealing with heap memory and try to free a pointer that's actually on the stack, the behavior is undefined. This could lead to further corruption of the heap's metadata structures; (5) difficult debugging: Such memory issues can be incredibly tricky to debug, as the malfunction might manifest long after the actual corruption in a completely unrelated part of the code.

In the literature, there are two approaches for manipulating the heap allocator to help users get the desired heap layout for the error triggering (1): heap fengshui [45], [46], [47]; (2) heap exploitation techniques. It is challenging to manipulate the heap allocator using heap fengshui [46], [47]. For heap exploitation techniques, as reported and collected in an online repository[14], the successful manipulation of the heap allocators can happen in the off-the-shelf heap allocator in widely used Linux systems (e.g., Ubuntu 18.04).

We also seek the developer's suggestion of `Make` on whether the issue reported in Fig. 21 is a potential issue or not: the following is the feedback from the developer[15]:

> *"I don't suspect GNU Make is used in that general way, as an application language for the writing program which is itself trustworthy and trusted, but falls victim to malicious data which, through that program, attacks GNU Make."*

Based on the feedback from developers, it is not considered as a bug in `Make`, but might be misused if users write a client program to trigger the issue reported in Fig. 21. We note that it may not be clear-cut who (users or developers of `Make` or the heap allocator used) should take responsibility for the potential misbehavior in this case. When users write their client programs using `Make` along with the heap allocator, their assumptions of the heap allocator behaviors can be different from and even in conflict with those of the developers of `Make`. One may argue that users should take responsibility as they break the underlying assumptions of `Make` or the heap allocator, while one may also argue that the developers of `Make` or the heap allocator should be responsible for providing a protection mechanism or extra checking to avoid any misuse by users that may violate their implicit assumptions.

Regarding responsibility for preventing potential attacks, as known in the literature [88], [89], [90], there are mainly two directions, namely application security and software security. It is worth noting that the two protect software differently by either filtering its input in a post facto way (application security) or by designing software to withstand attacks in the first place (software security) are of similar importance [88]. Users of the software may take responsibility for ensuring application security, while the developers of the software may take responsibility for maintaining software security.

Taking the well-known SQL injection attacks and C stack overflow bugs as examples, from the application security perspective, the SQL query engines and C compilers are *not* directly responsible for the vulnerable programs that use the engines and compilers. But from the software security perspective, the developers of SQL engine and C compiler *can* take some responsibility for preventing such attacks. For example, SQL engines support and encourage users to adopt prepared statements with parameterized queries [91] that automatically handle user inputs safely, which has been instrumental in preventing SQL injection attacks;

14. https://github.com/shellphish/how2heap
15. https://lists.gnu.org/archive/html/help-make/2023-08/msg00001.html

```
1  void* imalloc (unsigned int s) {
2      void *p2 = malloc(s);
3      klee_make_symbolic(&p2, sizeof(void *), "sym_test");  // inserted
4      return s <= size_max ? p2 : _gl_alloc_nomem ();
5  }
6  static  void* nonnull (void *p) {
7    if  (! p){
8      printf("if_branch_in_nonnull\n");
9      xalloc_die () ;
10   } else
11     printf("else_branch_in_nonnull\n");
12   return p;
13 }
14 void* ximalloc (unsigned int s) {
15   return nonnull (imalloc (s));
16 }
17 int  main () {
18   char* p1 = (char *)  ximalloc(1);
19   free(p1);
20   return 0;
21 }
```

Fig. 23. Example code in `dircolors` package triggering different behavior of KLEE when the "–optimize=true/false" option enabled

StackGuard [92] was introduced as a compiler extension (i.e., GCC[16] and LLVM[17]) to significantly mitigate the risk of buffer overflow attacks against malicious C programs.

While reporting the error with the assumption that user client programs could manipulate the heap allocator would lead to more false positives (which may be considered as a disadvantage of SYMLOC), this case illustrates the capability of SYMLOC to consider the relations among client programs, the subject software, and the underlying heap allocators more comprehensive, pointing to the need of more precise modeling and analysis of the assumptions and semantics of the software in future work.

A.2 Case 4: Missing Forking Issue in KLEE

In this case, we discuss an issue in LLVM/KLEE that was found by our analysis based on our experimental results. Since KLEE is a notable and widely-used symbolic execution tool (as of April 2024, the KLEE [13] paper has been cited more than 4,000 times and KLEE's repository has more than 2,400 stars), we believe the quality of KLEE itself matters a lot when it is used for many applications.

Error details. To help make the behavior simple and easy to understand, we make minor changes to the source code of several source files of the `dircolors` package[18] and present the minimized code example in Fig. 23. The overall logic of this issue is straightforward. The `main` function in Line 17 invokes an allocation function `ximalloc` (a packed function of `malloc`) to allocate a memory buffer *p1* and free the buffer later. Inside the function `ximalloc`, a buffer is returned by calling function `nonnull`. The main functionality of the function `nonnull` is to ensure the returned value by dynamically allocated memory objects is not *NULL*, so it either returns a non-null value from *if-then* branch in Line 12 or directly exits from *if-then* branch in Line 9. The parameter of the function is a memory object returned by function `imalloc`, where a *p2* is pointing to an object allocated by `malloc` function and the *p2* is returned if the allocation size is less than the maximum size. Ideally, the pointer value of *p1* should never be *NULL* as the dynamically allocated buffer is carefully checked via function `nonnull`.

A user/developer may want to utilize a symbolic pointer to analyze the function `ximalloc` and they may assume the symbolic execution engine used should be reliable. However, using some version of KLEE to analyze this function produces wrong results, i.e., missing a path that should be explored, and the constraint of the pointer is not collected. The buggy behavior is that KLEE fails to fork at a branch that should be forked with the option "–optimize" option enabled (i.e., "–optimize=true"). While the "–optimize" option is disabled (i.e., "–optimize=false"), the branch can be successfully forked as usual[19]. We also checked the internal constraint queries (i.e., `.kquery` files) produced by KLEE, and the result shows the symbolic variables are correctly stored. However, KLEE does not fork when the option "–optimize=true" and no symbolic variables remain in the `.kquery` files[20]. After the communication and confirmation with KLEE's developers, they confirmed this is an odd behavior in KLEE and suggested that this specific issue probably stems from an LLVM bug[21].

How does SYMLOC detect it? SYMLOC exposed this error of LLVM/KLEE mainly because of its uses of address symbolization along with their encoding into path constraints and the symbolic-concrete memory map. In this case, SYMLOC would symbolize the dynamically allocated memory object (*p2* at Line 2 in Fig. 23) and encode it to path constraints to have better modeling of the program semantics, which would require appropriate supports of address symbolization and practical symbolic memory operations in the underlying symbolic execution. However, the engine produced the wrong results, i.e., forking at a branch that should not be forked due to the issue of the missing constraint of the symbolized address, which led us to identify the issue in LLVM/KLEE when manually analyzing the results.

Implications. Through this case, we also learned two lessons. First, when using software symbolic execution and testing tools (e.g., KLEE), it would be better to use stable versions and the recommended building toolchains (e.g., compilers) to have more reliable results. Second, although a variety of work is dedicated to improving the capabilities of such tools for bug detection, only a few focus on assuring the quality of such tools themselves [93]. SYMLOC may indirectly help to improve the capabilities and quality of symbolic execution engines as it requires more comprehensive support of address symbolization and encoding. Thus, we call for more future research work on improving the capabilities and quality of symbolic execution engines, e.g., by studying how compiler optimizations affect the correctness of symbolic execution.

16. https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html
17. https://llvm.org/docs/LangRef.html-llvm-stackprotector-intrinsic
18. For more modified file details, please refer to https://gist.github.com/haoxintu/183dda2923965d1e33f64ad59c7f5338#prepare-the-code.
19. For more detailed execution results, please refer to https://gist.github.com/haoxintu/183dda2923965d1e33f64ad59c7f5338#run-klee.
20. We also checked other benchmarks used in the evaluation, and the results show only `dircolors` invokes the function `ximalloc`. Therefore, the results of running other benchmarks are not affected by this optimization issue.
21. https://www.mail-archive.com/klee-dev@imperial.ac.uk/msg03276.html