The 4th International KLEE Workshop on Symbolic Execution (15–16 April 2024)

Concretely Mapped Symbolic Memory Locations for Memory Error Detection

Haoxin Tu, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding (Singapore Management University)
He Jiang (Dalian University of Technology)

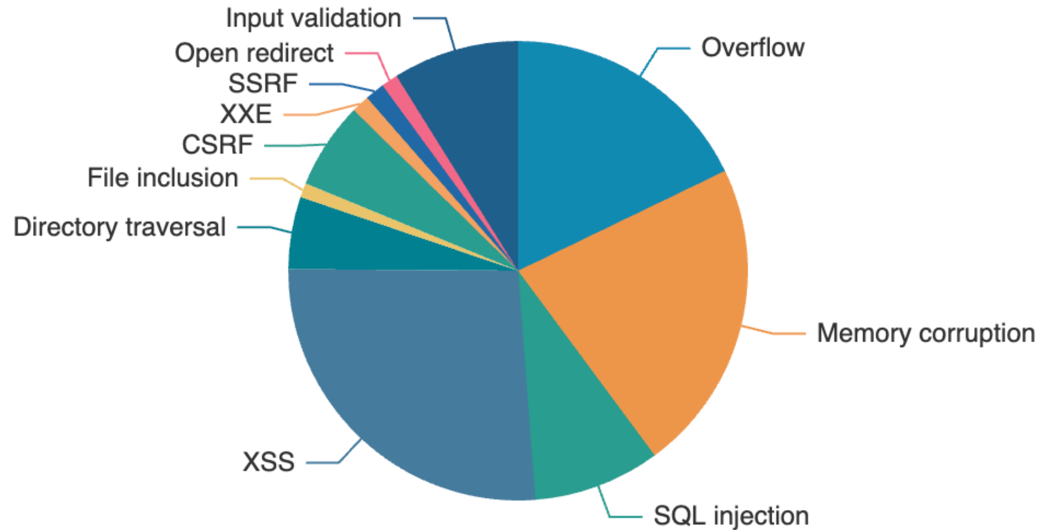
(Under Review of IEEE Transactions on Software Engineering)

16/04/2024, Lisbon

Background: annoying memory errors

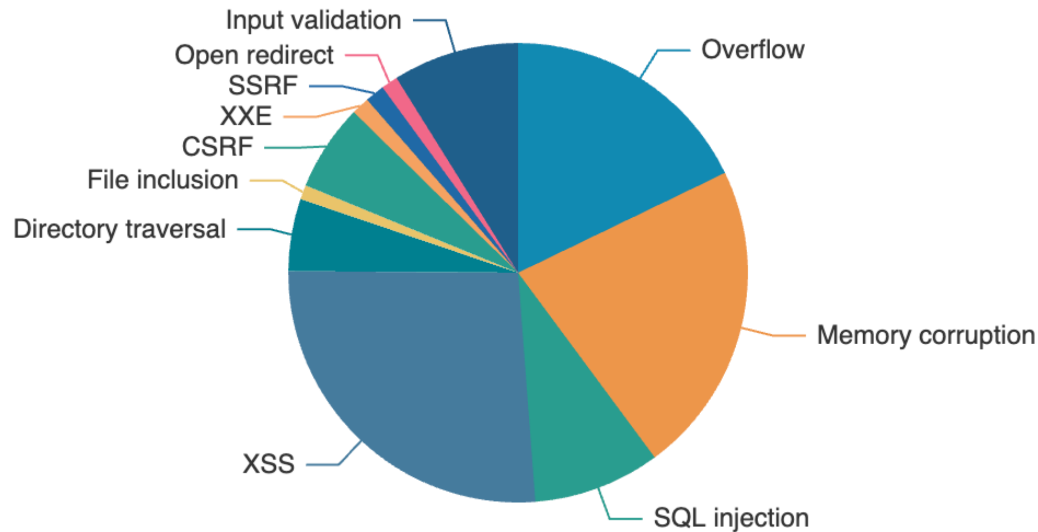
Background: annoying memory errors

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

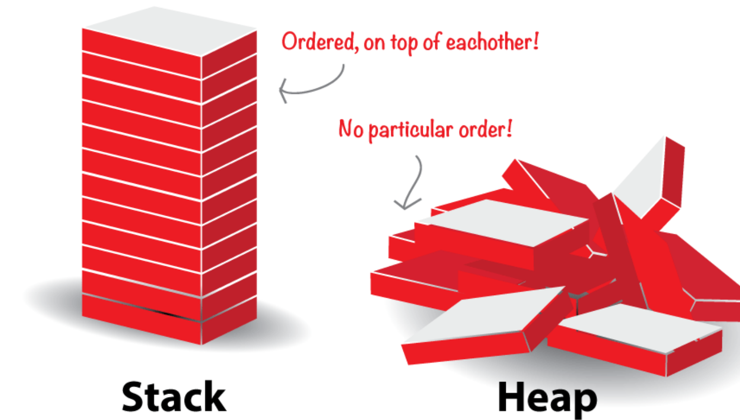


Vulnerability By Type (1999-2023)
(https://www.cvedetails.com/vulnerabilities-by-types.php)

Background: annoying memory errors



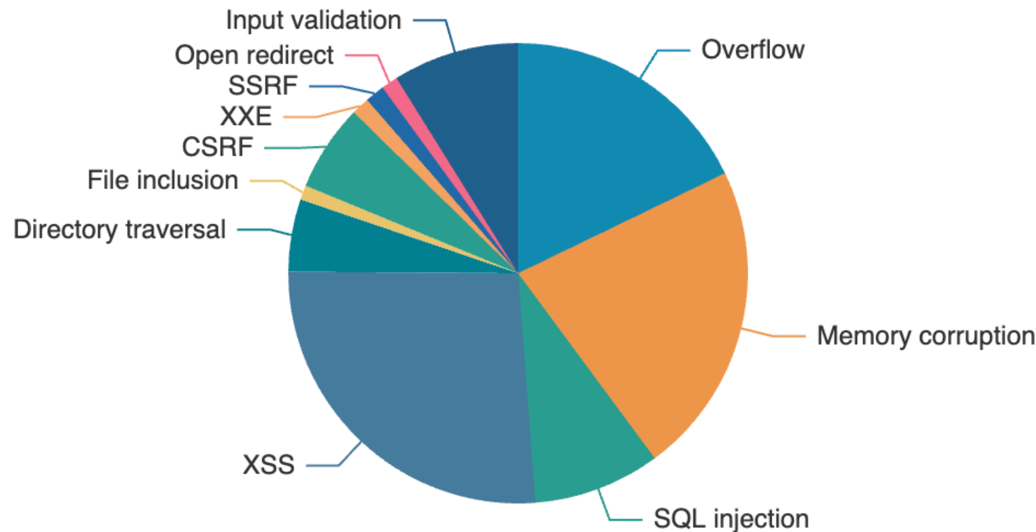
Vulnerability By Type (1999-2023)
(https://www.cvedetails.com/vulnerabilities-by-types.php)



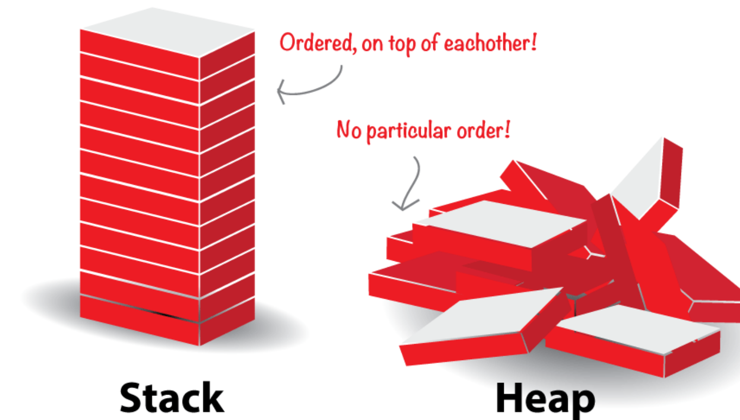
https://shivab.com/2018/09/10/memory-management-in-c/

Background: annoying memory errors

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems



Vulnerability By Type (1999-2023)
(https://www.cvedetails.com/vulnerabilities-by-types.php)



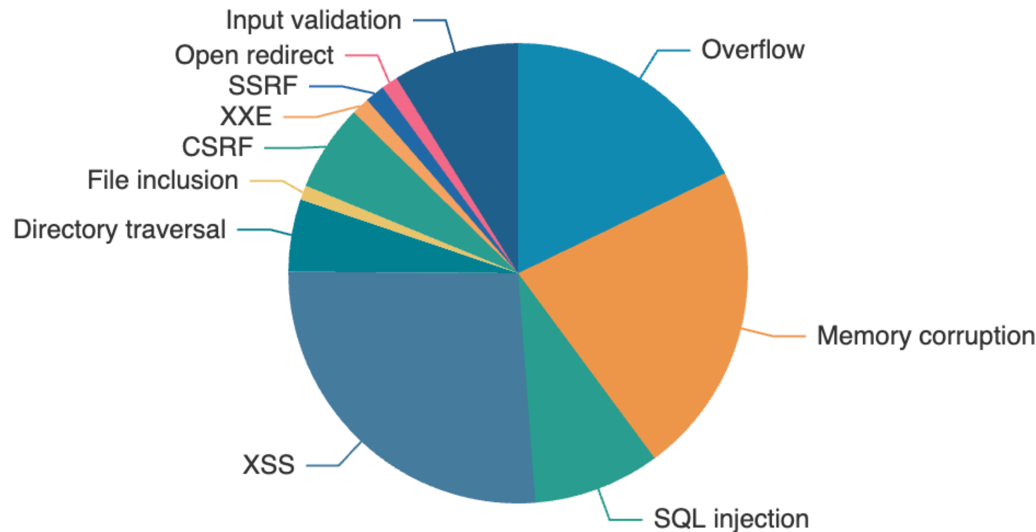
https://shivab.com/2018/09/10/memory-management-in-c/

- Microsoft: **70%** of all security bugs are memory safety issues for past 12 years [1]

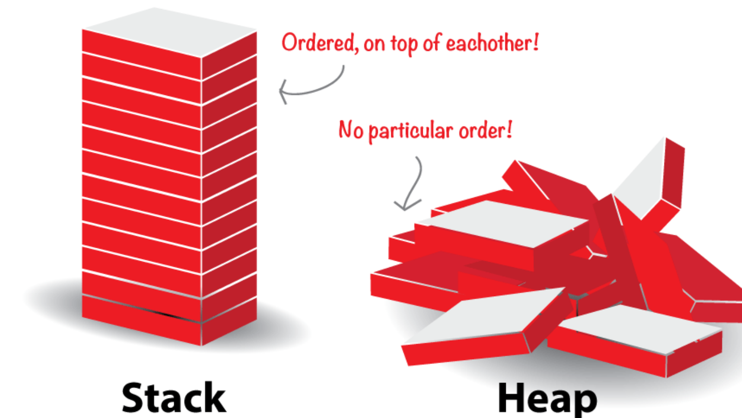
[1] https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Background: annoying memory errors

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems



Vulnerability By Type (1999-2023)
(https://www.cvedetails.com/vulnerabilities-by-types.php)



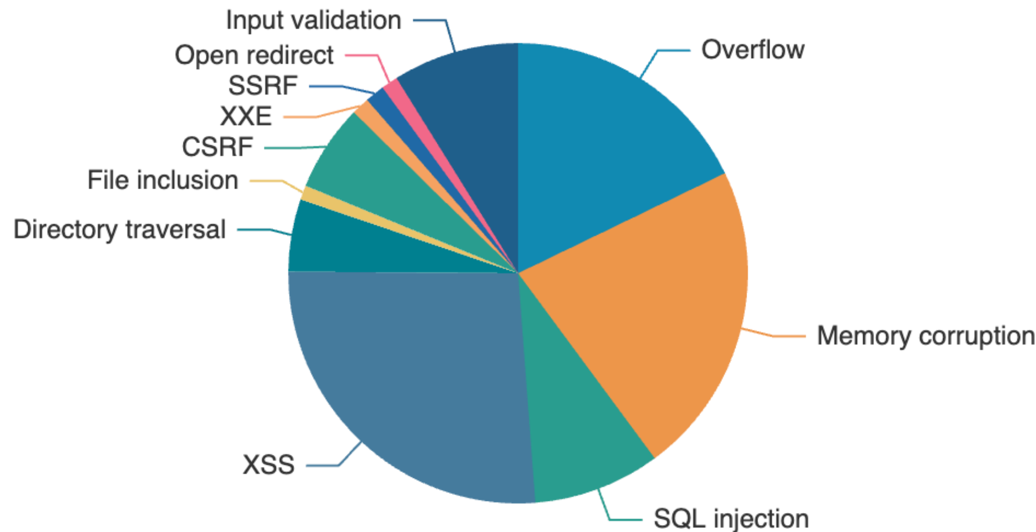
https://shivab.com/2018/09/10/memory-management-in-c/

- Microsoft**:** **70%** of all security bugs are memory safety issues for past 12 years [1]
  - Memory errors caused by ***dynamic memory allocation (i.e., from heap)*** continue to be the preferred bugs when attackers are developing exploits

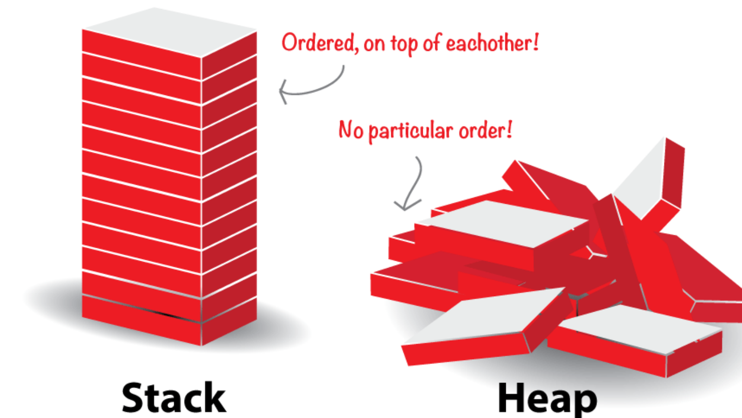
[1] https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Background: annoying memory errors

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems



Vulnerability By Type (1999-2023)
(https://www.cvedetails.com/vulnerabilities-by-types.php)



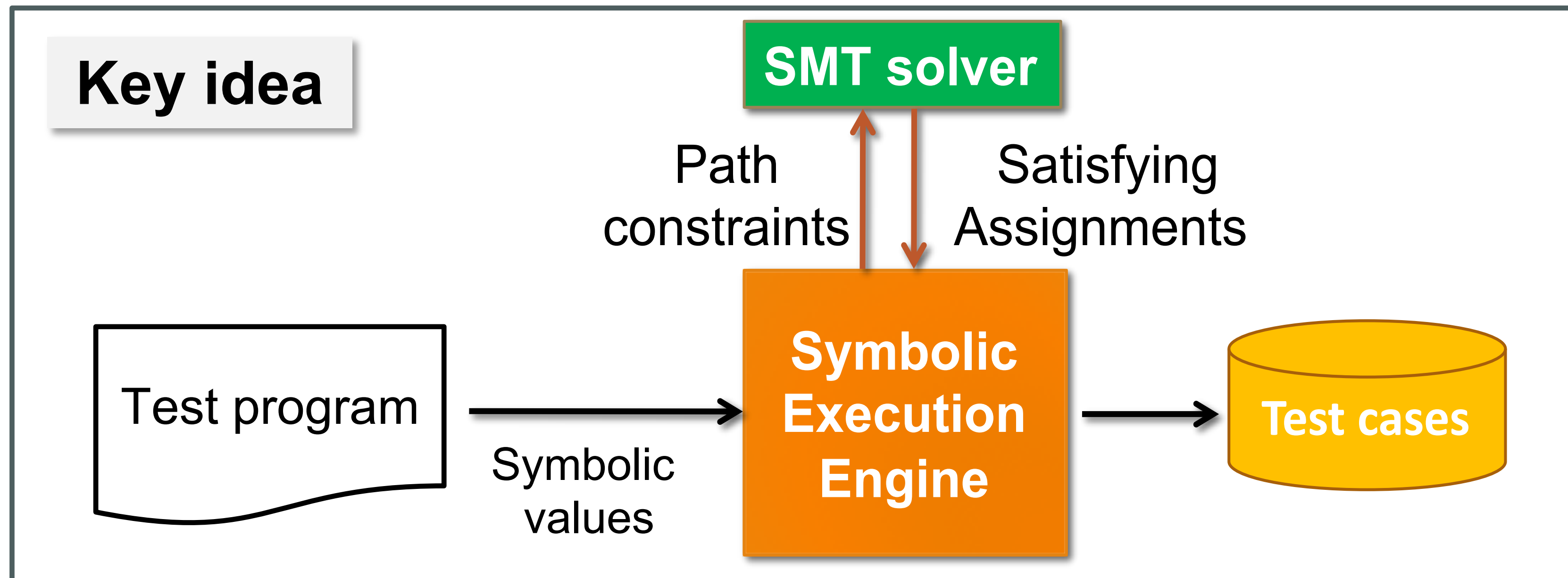
https://shivab.com/2018/09/10/memory-management-in-c/

- Microsoft**: 70%** of all security bugs are memory safety issues for past 12 years [1]
  - Memory errors caused by ***dynamic memory allocation (i.e., from heap)*** continue to be the preferred bugs when attackers are developing exploits
    - **Spatial**: out of bound access (e.g., buffer overflow or null pointer dereference)
    - **Temporal**: out of liveness access (e.g., use-after-free or double free)

[1] https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Background: symbolic execution

- **What is symbolic execution?**
  - Proposed in 1976 [1], one of the most popular program analysis techniques, which scales for many **software testing** and **computer security** applications



[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394.

Motivation (1/2)

Motivation (1/2)

- **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

Motivation (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0) {
     ... // Path-A
  }
  if (to > from) {  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to) {  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

Motivation (1/2)

SMU Singapore Management University | School of Computing and Information Systems

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
    ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

Motivation (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
     ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

Motivation (1/2)

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
     ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
   char * data = NULL;
   data = (char *) malloc (100);
   if (data == NULL) { abort(); }
   free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
   char * ret = dothing();
   printf("%s\n", ret); // use-after-free error
   return 0;
}
```

Example (b)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

Motivation (1/2)

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
    ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
   char * data = NULL;
   data = (char *) malloc (100);
   if (data == NULL) { abort(); }
   free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
   char * ret = dothing();
   printf("%s\n", ret); // use-after-free error
   return 0;
}
```

Example (b)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

Motivation (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
    ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
   char * data = NULL;
   data = (char *) malloc (100);
   if (data == NULL) { abort(); }
   free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
   char * ret = dothing();
   printf("%s\n", ret); // use-after-free error
   return 0;
}
```

Example (b)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

Motivation (1/2)

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
     ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
   char * data = NULL;
   data = (char *) malloc (100);
   if (data == NULL) { abort(); }
   free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
   char * ret = dothing();
   printf("%s\n", ret); // use-after-free error
   return 0;
}
```

Example (b)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

Motivation (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ **Problem: symbolic execution engines may have trouble comprehensively analyzing certain programs that involve dynamic memory allocations**

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
     ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
   char * data = NULL;
   data = (char *) malloc (100);
   if (data == NULL) { abort(); }
   free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
   char * ret = dothing();
   printf("%s\n", ret); // use-after-free error
   return 0;
}
```

Example (b)

➢ **Limited code coverage**

- Existing symbolic executors (e.g., KLEE) are only able to cover Path-B while missing Path-A and Path-C

➢ **Limited bug-finding capability**

- Miss the detection of *use-after-free* bug due to the fundamental design issue in KLEE (https://github.com/klee/klee/issues/1434)

Motivation (2/2)

Motivation (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- To **mitigate the above two limitations**, **three fundamental requirements are needed**

Motivation (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- To **mitigate the above two limitations**, **three fundamental requirements are needed**
  A. Symbolization of addresses and modeling them into path constraints

Motivation (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- To **mitigate the above two limitations**, **three fundamental requirements are needed**
  A. Symbolization of addresses and modeling them into path constraints
  B. Practical read/write operation from/to symbolic addresses

Motivation (2/2)

- To **mitigate the above two limitations**, **three fundamental requirements are needed**
  A. Symbolization of addresses and modeling them into path constraints
  B. Practical read/write operation from/to symbolic addresses
  C. Effectively tracking the uses of symbolic addresses

Motivation (2/2)

- To **mitigate the above two limitations**, **three fundamental requirements are needed**
  - A. Symbolization of addresses and modeling them into path constraints
  - B. Practical read/write operation from/to symbolic addresses
  - C. Effectively tracking the uses of symbolic addresses

- **Existing approaches are difficult to satisfy all the above requirements**
  - KLEE and Symsize (FSE'21): none of the requirements can be satisfied
  - RAM (ICSE'18): satisfies requirements #B and partially #A but not #C
  - Memsight (ASE'17): satisfies requirements #A and #B but not #C

Rethinking Pointer Reasoning in Symbolic Execution

Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu
Department of Computer, Control, and Management Engineering

Relocatable Addressing Model for Symbolic Execution

David Trabish, Tel Aviv University, Israel, davivtra@post.tau.ac.il
Noam Rinetzky, Tel Aviv University, Israel, maon@cs.tau.ac.il

A Bounded Symbolic-Size Model for Symbolic Execution

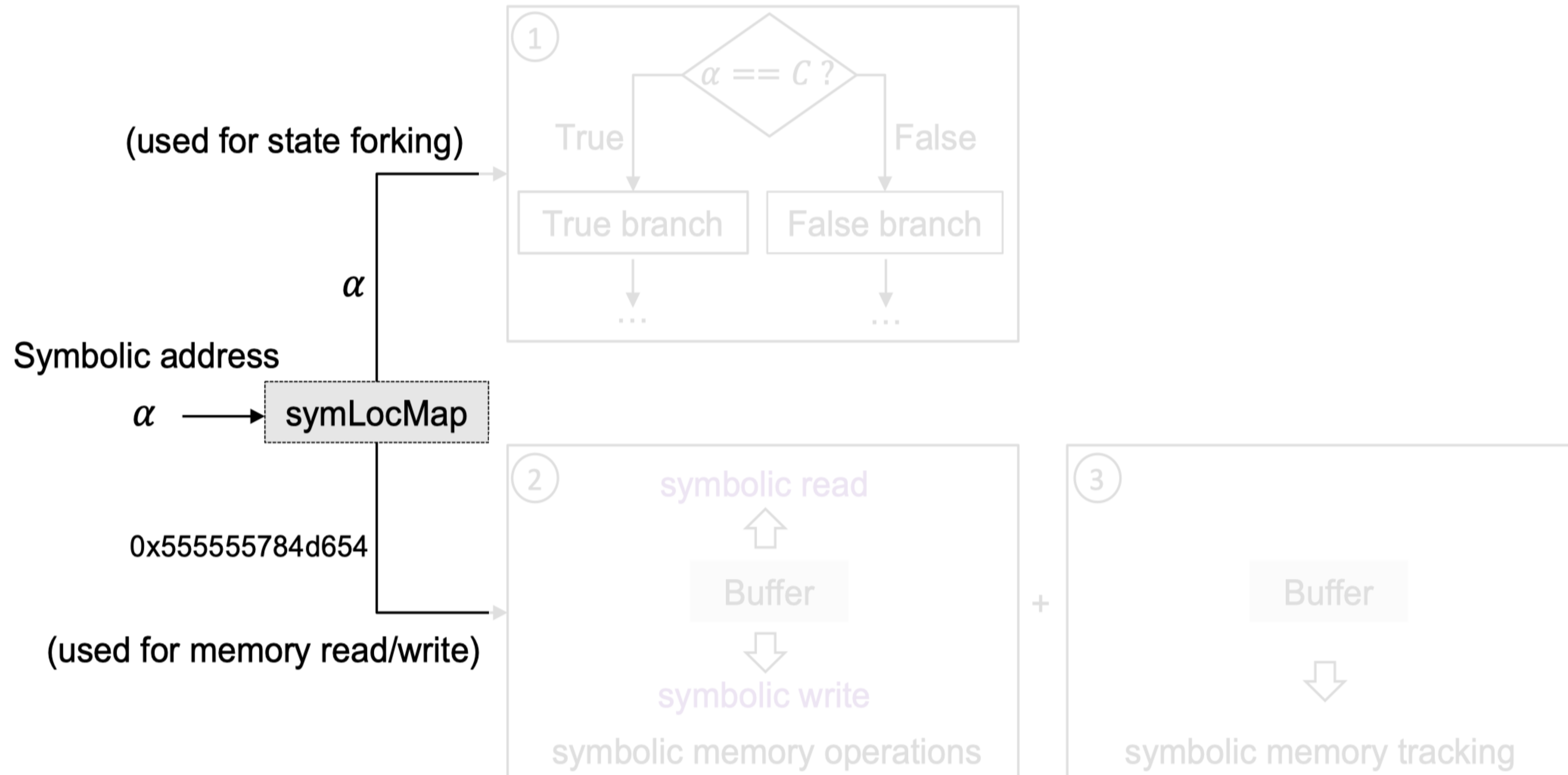
David Trabish, Tel Aviv University, Israel, davivtra@post.tau.ac.il
Shachar Itzhaky, Technion, Israel, shachari@cs.technion.ac.il
Noam Rinetzky, Tel Aviv University, Israel, maon@cs.tau.ac.il

Solution: SymLoc (1/3)

Solution: SymLoc (1/3)

SMU Singapore Management University | School of Computing and Information Systems

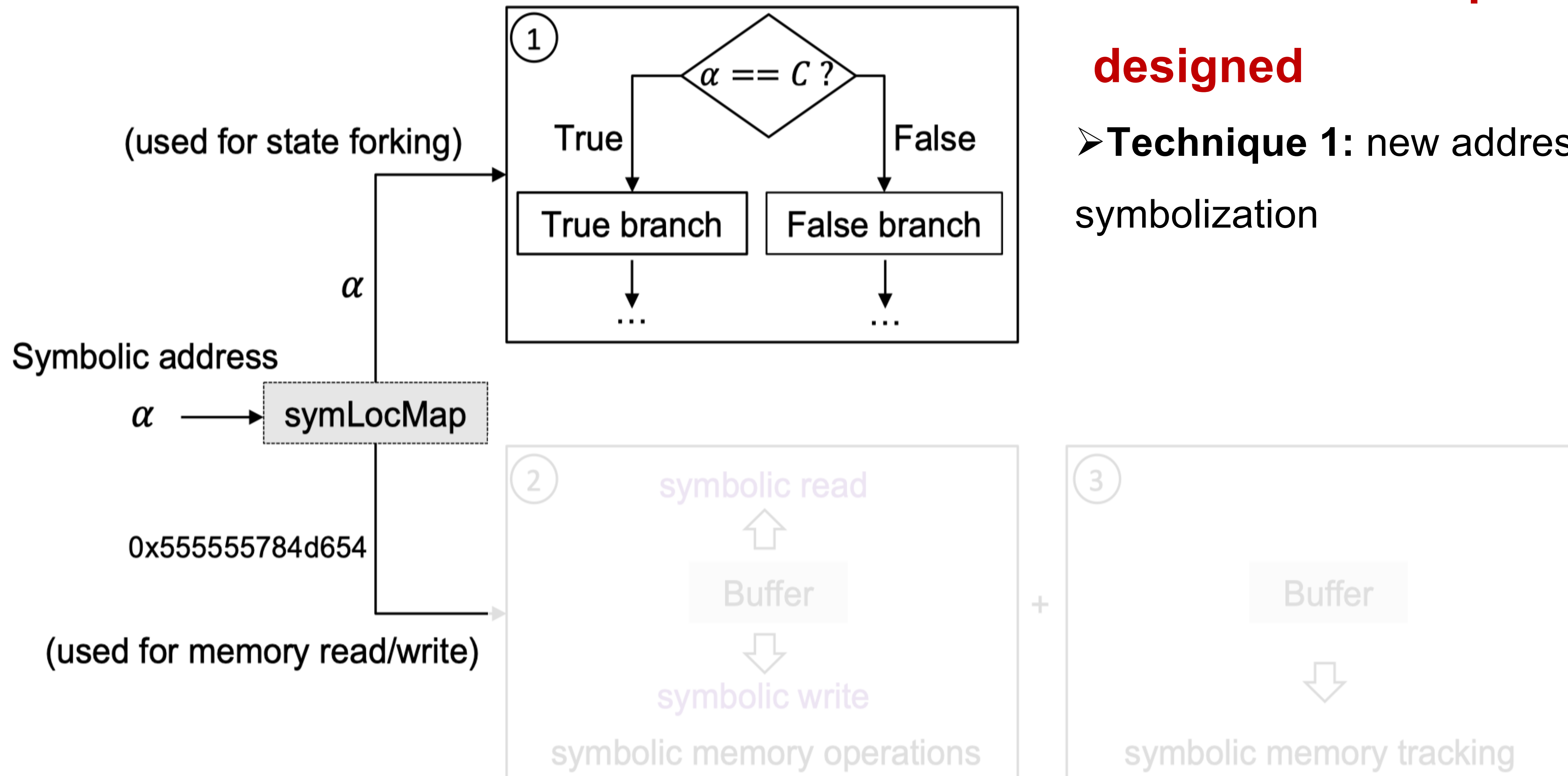
High-level Idea



Solution: SymLoc (1/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

High-level Idea



Two new techniques are designed

- **Technique 1:** new address symbolization

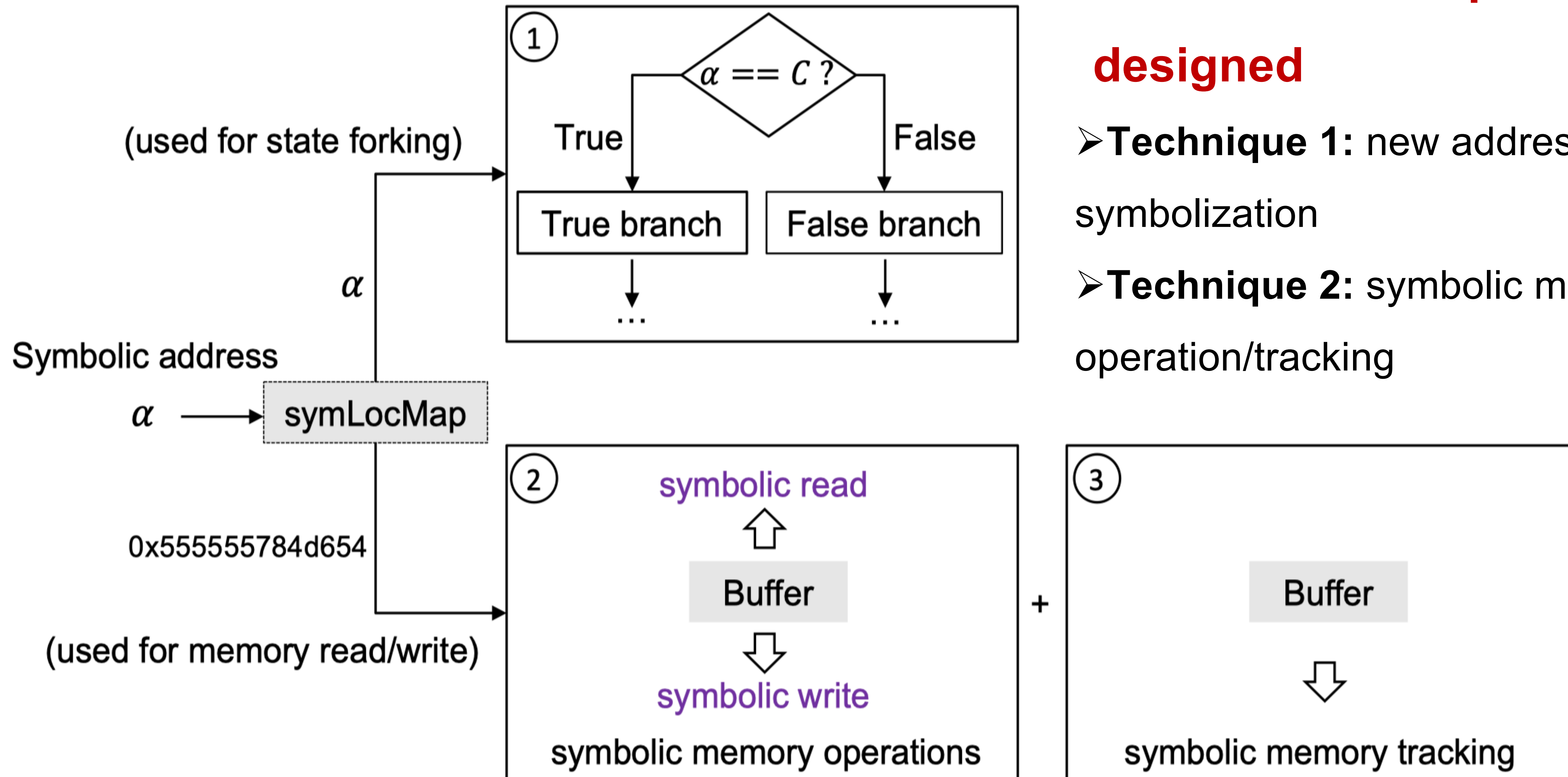
Solution: SymLoc (1/3)

SMU Singapore Management University | School of Computing and Information Systems

High-level Idea

Two new techniques are designed

- ➢**Technique 1:** new address symbolization
- ➢**Technique 2:** symbolic memory operation/tracking



Solution: SymLoc (2/3)

Solution: SymLoc (2/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- New address symbolization

Solution: SymLoc (2/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ New address symbolization

- **Symbolic addressing model**
  - Encoding the symbolic address into path constraints

Existing: $(addr, size, arry) \in N^{+} \times N^{+} \times A$

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

Solution: SymLoc (2/3)

SMU Singapore Management University | School of Computing and Information Systems

☐ New address symbolization

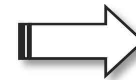
- **Symbolic addressing model**
  - Encoding the symbolic address into path constraints

Existing: $(addr, size, arry) \in N^+ \times N^+ \times A$

Ours: $(symAddr, size, arry) \in N^+ \times N^+ \times A$

- **Flexible symbolization strategy**
  - Based on the number of malloc functions used in the test program
  - fully, random, and selective (user-defined)

```
int *buffer;
klee_make_malloc_symbolic("symbolic_buffer");
buffer = (int*) malloc(100);  // example for illustrating Mode 3
```



- **Input**
  - A set of variables to return from malloc function
- **Output**
  - A symbolic-concrete memory map (symLocMap)
  - Will be used in the latter phase

Solution: SymLoc (3/3)

Solution: SymLoc (3/3)

SMU Singapore Management University | School of Computing and Information Systems

☐ Symbolic memory operation and tracking

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 $\longleftarrow \emptyset$ // initialize a concrete expression
2. FreeList $\longleftarrow \emptyset$ // initialize a list to store freed objects
3. **Function SymAddrRes** (*symLocMap*, *symExpr*, *func*):

   …

- **Input**
  - symLocMap, a symbolic expression, and a function
- **Output**
  - A concrete address or normal symbolic variable or a bug

Solution: SymLoc (3/3)

☐ Symbolic memory operation and tracking

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 $\longleftarrow \emptyset$ // initialize a concrete expression
2 FreeList $\longleftarrow \emptyset$ // initialize a list to store freed objects
3 **Function SymAddrRes** (*symLocMap*, *symExpr*, *func*):
...

- **Input**
  - ➢ symLocMap, a symbolic expression, and a function
- **Output**
  - ➢ A concrete address or normal symbolic variable or a bug

- **Two common situations**
  - ➢ Handle read/write for normal function
  - ➢ Handle free: maintain a **free list** and detect potential double-free and use-after-free bugs

Solution: SymLoc (3/3)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

☐ Symbolic memory operation and tracking

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):
  ...
```

- **Input**
  - symLocMap, a symbolic expression, and a function
- **Output**
  - A concrete address or normal symbolic variable or a bug

- **Two common situations**
  - Handle read/write for normal function
  - Handle free: maintain a **free list** and detect potential double-free and use-after-free bugs
- **Tracking example**
  - A memory address is symbolized as “sym_a”
  - If the freed object is “**sym_a**” or “**sym_a + 100**”
    - Indicating UAF bugs

Retrospection of motivating examples

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Retrospection of motivating examples


```
void *
memmove((void * from, const void * to, int n){
 if (from == to || n == 0){
    ... // Path-A
 }
 if (to > from){  / * copy in reverse * /
   ... // Path-B
 }
 if (from > to){  / * copy forwards * /
   ... // Path-C
 }
 return dest;
}
```

Example (a)

Retrospection of motivating examples

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void *
memmove((void * from, const void * to, int n){
 if (from == to || n == 0){
    ... // Path-A
 }
 if (to > from){  / * copy in reverse * /
   ... // Path-B
 }
 if (from > to){  / * copy forwards * /
   ... // Path-C
 }
 return dest;
}
```

Example (a)

➢ New symbolic addressing model

- Covers all paths Path-A,B, and C

Retrospection of motivating examples

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
     ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
    char * data = NULL;
    data = (char *) malloc (100);
    if (data == NULL) { abort(); }
    free(data);
   return data; // return freed memory
}

int main(int argc, char** argv){
    char * ret = dothing();
    printf("%s\n", ret); // use-after-free error
    return 0;
}
```

Example (b)

➢ New symbolic addressing model

- Covers all paths Path-A,B, and C

Retrospection of motivating examples

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

```
void *
memmove((void * from, const void * to, int n){
  if (from == to || n == 0){
    ... // Path-A
  }
  if (to > from){  / * copy in reverse * /
    ... // Path-B
  }
  if (from > to){  / * copy forwards * /
    ... // Path-C
  }
  return dest;
}
```

Example (a)

```
static char * dothing () {
    char * data = NULL;
    data = (char *) malloc (100);
    if (data == NULL) { abort(); }
    free(data);
    return data; // return freed memory
}

int main(int argc, char** argv){
    char * ret = dothing();
    printf("%s\n", ret); // use-after-free error
    return 0;
}
```

Example (b)

➢ New symbolic addressing model

- Covers all paths Path-A,B, and C

➢ Symbolic memory operation and tracking

- Reliably catches the UAF bug

Evaluation – for RQ1

Evaluation – for RQ1

■ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

Evaluation – for RQ1

Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

Evaluation Setup

- **Benchmark:** GNU Coreutils
- **Comparative approaches:** KLEE and Symsize
- **Metrics:** code coverage and the number of detected spatial memory errors

Evaluation – for RQ1

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

■ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

■ Evaluation Setup

- **Benchmark:** GNU Coreutils
- **Comparative approaches:** KLEE and Symsize
- **Metrics:** code coverage and the number of detected spatial memory errors

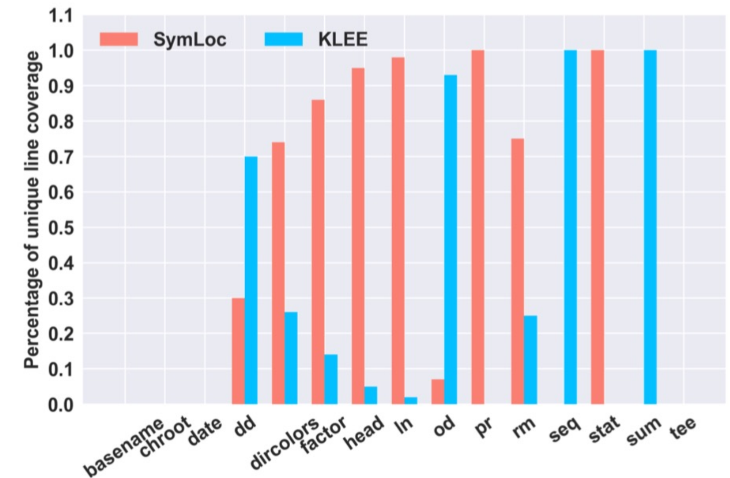


Fig. 7. Unique line coverage (measured by `gcov`): SYMLOC vs KLEE

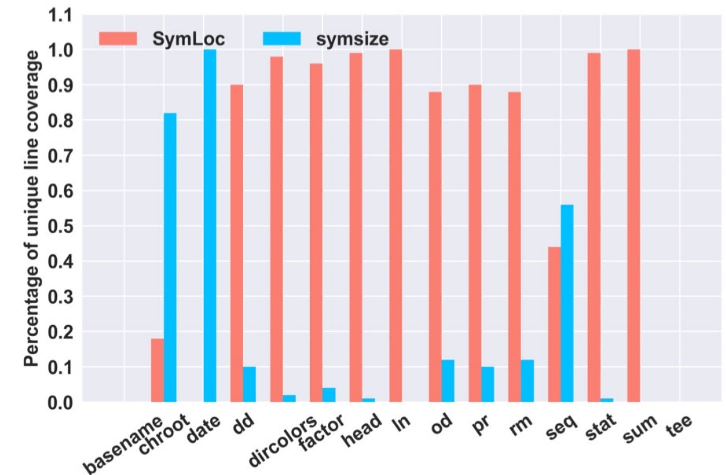


Fig. 8. Unique line coverage (measured by `gcov`): SYMLOC vs *symsize*

Evaluation – for RQ1

■ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

■ Evaluation Setup

- ➢ **Benchmark:** GNU Coreutils
- ➢ **Comparative approaches:** KLEE and Symsize
- ➢ **Metrics:** code coverage and the number of detected spatial memory errors

■ Summary

- ➢ SymLoc could cover **15%** and **48%** more unique lines of code on average than the two baseline approaches.

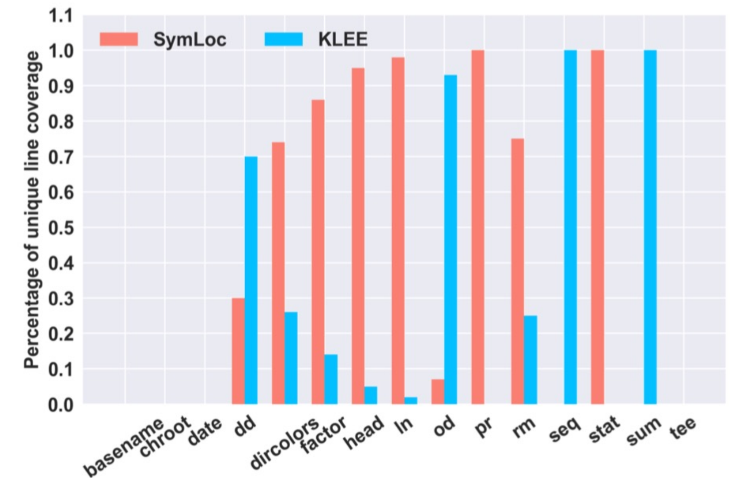


Fig. 7. Unique line coverage (measured by `gcov`): SYMLOC vs KLEE

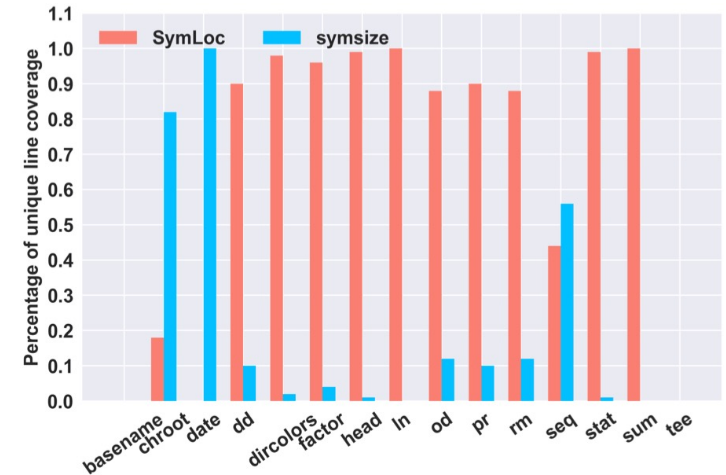


Fig. 8. Unique line coverage (measured by `gcov`): SYMLOC vs *symsize*

Evaluation – for RQ1

- **Research Question (1/2)**

How does SymLoc perform in detecting spatial memory errors?

Evaluation – for RQ1

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

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

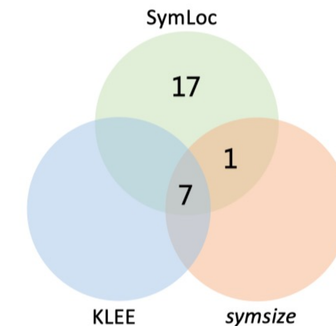


Fig. 6. Distribution of address-specific spatial memory errors detected by comparative approaches

Evaluation – for RQ1

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

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

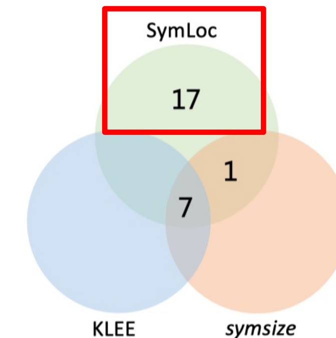


Fig. 6. Distribution of address-specific spatial memory errors detected by comparative approaches

Evaluation – for RQ1

➢ Research Question (1/2)

How does SymLoc perform in detecting spatial memory errors?

➢ Among 17 reported unique errors

- ➢ **8** happens when the address is allocated in kernel space
  - We provide a post-processing option to filter them out
- ➢ **9** happens when the address is allocated in user space
  - **2** null pointer dereference issues, confirmed by developers
  - **3** caused by compiler optimization issue [1]
    - make KLEE fail to fork at a branch that should be forked
  - **4** caused by an improper comparison between stack and heap pointers

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

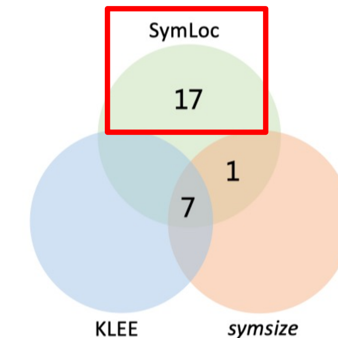


Fig. 6. Distribution of address-specific spatial memory errors detected by comparative approaches

[1] https://gist.github.com/haoxintu/183dda2923965d1e33f64ad59c7f5338

Evaluation – for RQ2

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Research Question (2/2)**

Evaluation – for RQ2

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

■ Research Question (2/2)

How does SymLoc perform in detecting temporal memory errors?

Evaluation – for RQ2

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Research Question (2/2)

How does SymLoc perform in detecting temporal memory errors?

Evaluation Setup

- **Benchmark:** JTS test suits
- **Comparative approaches**
  - Various symbolic/static/dynamic approaches
- **Metrics:** the number of detected temporal memory errors

Evaluation – for RQ2

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Research Question (2/2)

How does SymLoc perform in detecting temporal memory errors?

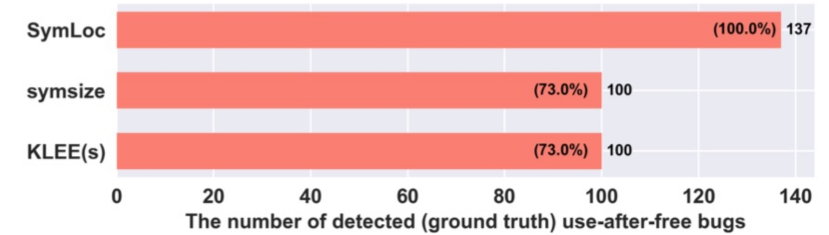


Fig. 12. Completeness of UAF error detection among symbolic execution-based approaches (137 in total)

Evaluation Setup

- **Benchmark:** JTS test suits
- **Comparative approaches**
  - Various symbolic/static/dynamic approaches
- **Metrics:** the number of detected temporal memory errors

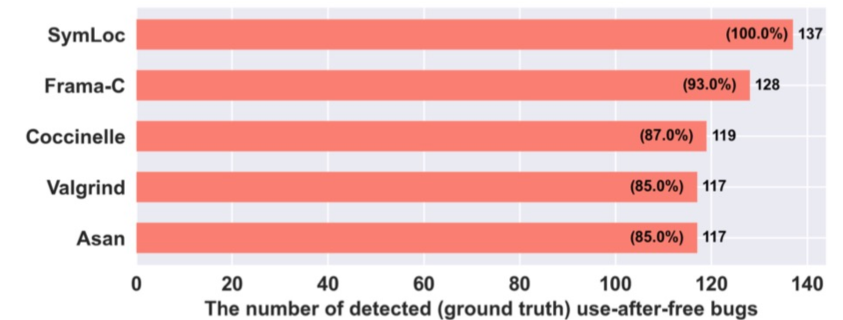


Fig. 10. Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total)

Evaluation – for RQ2

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

Research Question (2/2)

How does SymLoc perform in detecting temporal memory errors?

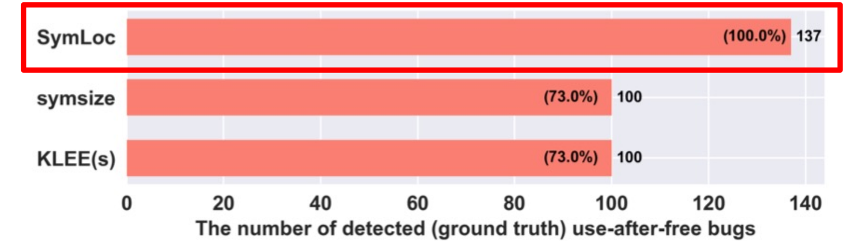


Fig. 12. Completeness of UAF error detection among symbolic execution-based approaches (137 in total)

Evaluation Setup

- **Benchmark:** JTS test suits
- **Comparative approaches**
  - Various symbolic/static/dynamic approaches
- **Metrics:** the number of detected temporal memory errors

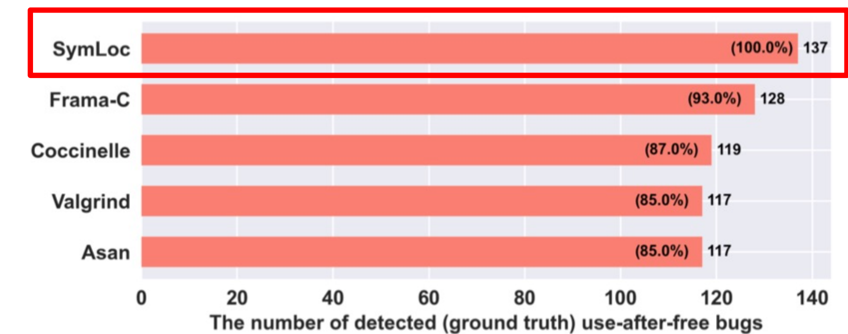


Fig. 10. Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total)

Summary

- SymLoc has better temporal memory error detection capability compared with various approaches.

Case Studies (1/2)

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

Case Studies (1/2)

SMU Singapore Management University | School of Computing and Information Systems

- **Case study 1: Consecutive NULL Pointer Returns**

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
      buf.size += need * 2;
      fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

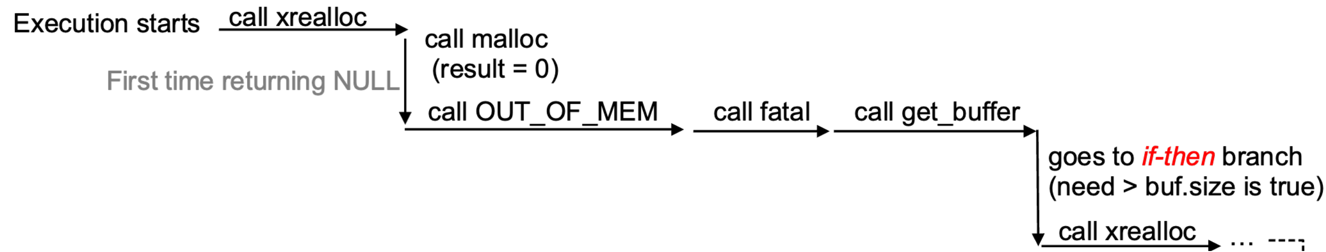


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
      buf.size += need * 2;
      fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 1: Consecutive NULL Pointer Returns**

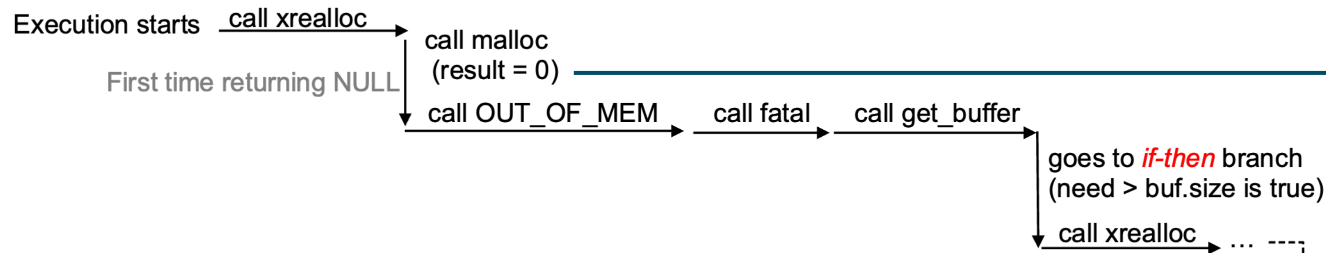


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
      buf.size += need * 2;
      fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

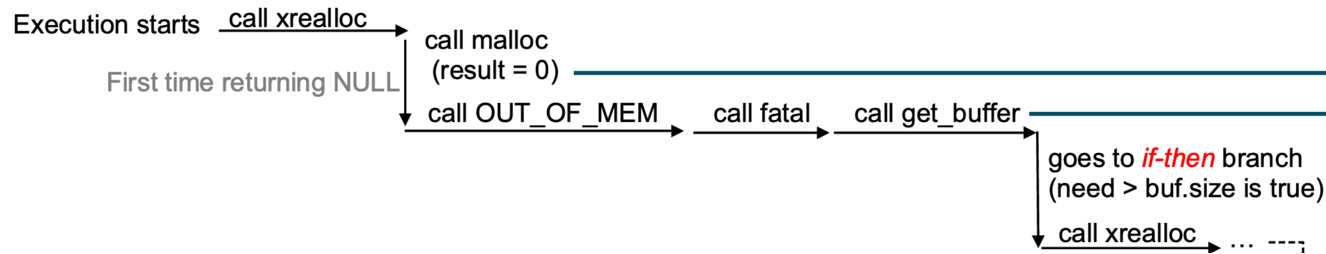


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
      buf.size += need * 2;
      fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 1: Consecutive NULL Pointer Returns**

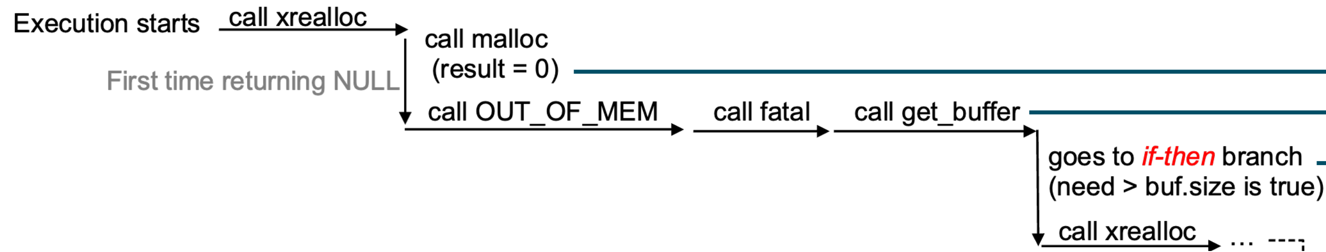


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

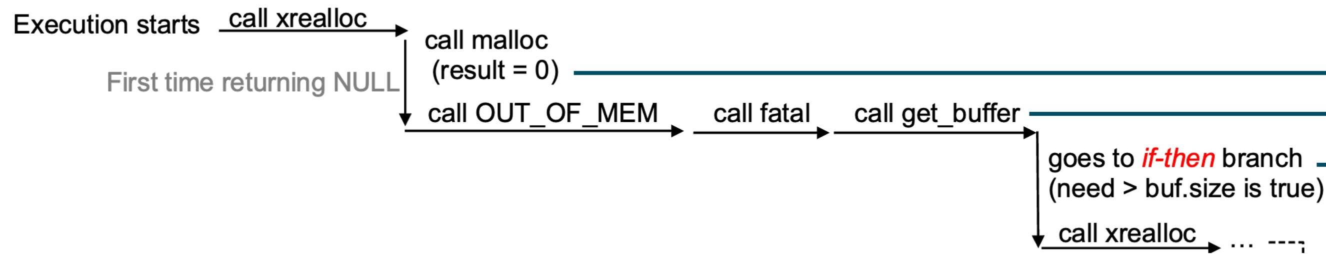


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

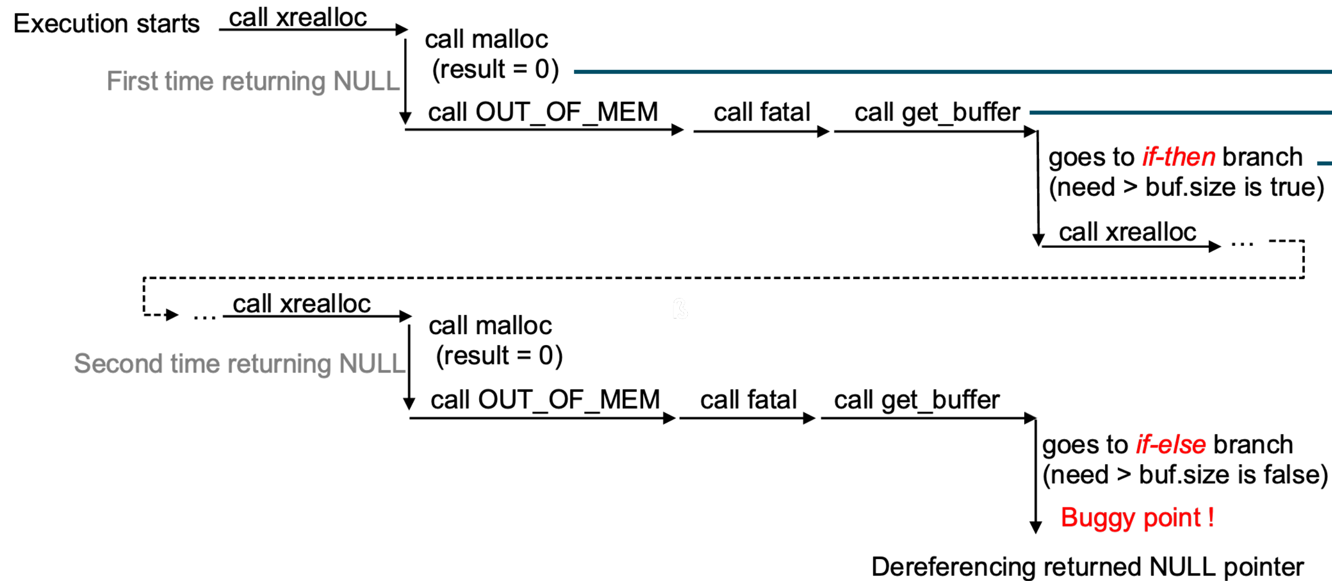


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 1: Consecutive NULL Pointer Returns**

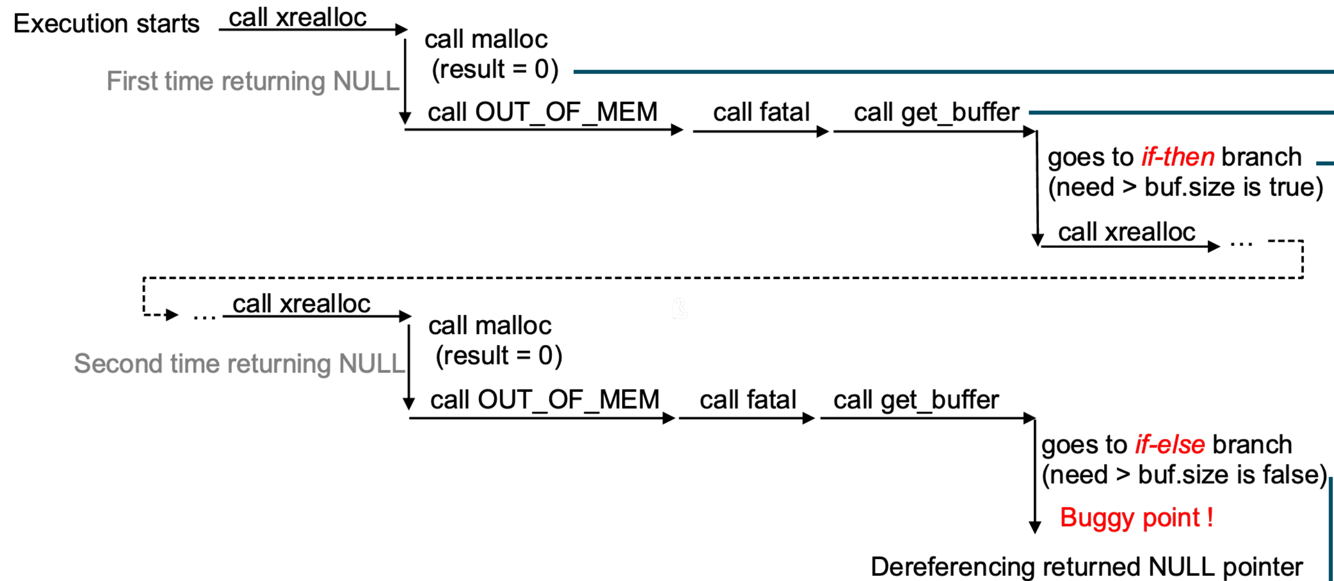


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

- **Case study 1: Consecutive NULL Pointer Returns**

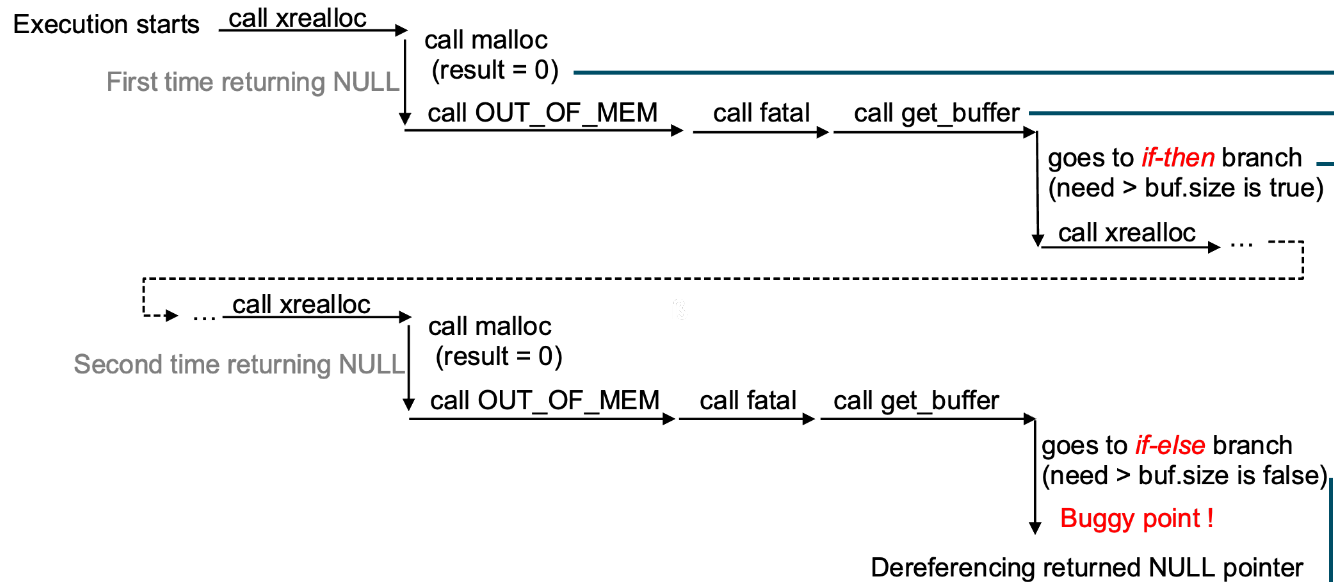


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

Case Studies (1/2)

SMU Singapore Management University — School of Computing and Information Systems

- **Case study 1: Consecutive NULL Pointer Returns**

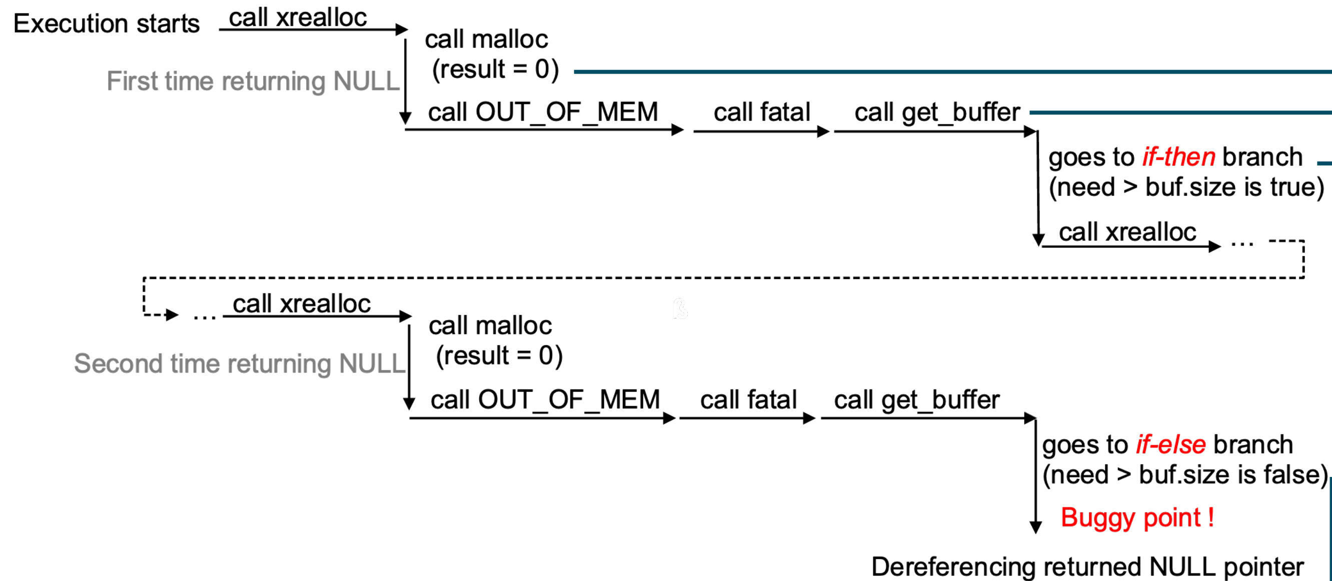


Fig. Execution flow of case 1

```
#define OUT_OF_MEM() O (fatal, NILF, _("info"))
#define O(_t,_a,_f)  _t((_a), 0, (_f))

void * xrealloc (void *ptr, unsigned int size) {
  void *result;
  result = ptr ? realloc (ptr, size) : malloc (size);
  if (result == 0)
    OUT_OF_MEM();
  return result;
}
void fatal (const floc *flocp, size_t len, ...) {
  len += ...;
  char * p = get_buffer(len);
  ...
  die (MAKE_FAILURE);
}
static struct fmtstring {char *buffer; size_t size;}
buf = {NULL, 0};
static char * get_buffer (size_t need) {
  if (need > buf.size) {
     buf.size += need * 2;
     fmtbuf.buffer = xrealloc (buf.buffer, buf.size);
  }
  buf.buffer[need-1] = '\0'; // out-of-bound
  return fmtbuf.buffer;
}
```

Code from Make-4.2

- **Dereferencing a NULL pointer**
  - Allocated via **malloc** (return 0 twice afterwards)
  - NULL pointer dereference after the second NULL returns

Case Studies (2/2)

Case Studies (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

Case Studies (2/2)

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Case Studies (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Case Studies (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Case Studies (2/2)

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Case Studies (2/2)

SMU Singapore Management University | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
  void *p = malloc(s);
  klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
  return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
 if (!p){
  printf("if branch in nonnull\n");
  xalloc_die ();
 } else
  printf("else branch in nonnull\n");
 return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Case Studies (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Expected behavior

```
$./run-opt-false.sh
KLEE: KLEE: WATCHDOG: watching 5001

else branch in nonnull
if branch in nonnull
memory exhausted

KLEE: done: total instructions = 20360
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Case Studies (2/2)

SMU Singapore Management University | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Expected behavior

```
$./run-opt-false.sh
KLEE: KLEE: WATCHDOG: watching 5001

else branch in nonnull
if branch in nonnull
memory exhausted

KLEE: done: total instructions = 20360
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Buggy behavior

```
$./run-opt-true.sh
KLEE: KLEE: WATCHDOG: watching 5010

KLEE: done: total instructions = 12516
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

Case Studies (2/2)

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **Case study 2: Missing Forking in KLEE**

```
void* imalloc (unsigned int s) {
    void *p = malloc(s);
    klee_make_symbolic(&p, sizeof(void *), "sym"); // inserted
    return s <= size_max ? p : _gl_alloc_nomem ();
}
static void* nonnull (void *p) {
  if (!p){
    printf("if branch in nonnull\n");
    xalloc_die ();
  } else
    printf("else branch in nonnull\n");
  return p;
}
void* ximalloc (unsigned int s) { return nonnull (imalloc (s)); }

int main () {
  char* p1 = (char *) ximalloc(1);
  free(p1);
  return 0;
}
```

Code from *dircolor*

Expected behavior

```
$./run-opt-false.sh
KLEE: KLEE: WATCHDOG: watching 5001

else branch in nonnull
if branch in nonnull
memory exhausted

KLEE: done: total instructions = 20360
KLEE: done: completed paths = 2
KLEE: done: generated tests = 2
```

Buggy behavior

```
$./run-opt-true.sh
KLEE: KLEE: WATCHDOG: watching 5010

KLEE: done: total instructions = 12516
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

- **Lessons we learned**
  - **Use the stable version of KLEE and LLVM to build KLEE!**
  - Call for more investigation on how compiler optimization affects symbolic execution

Conclusion

Conclusion

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

- **SymLoc: a boosted symbolic execution engine for memory error detection**

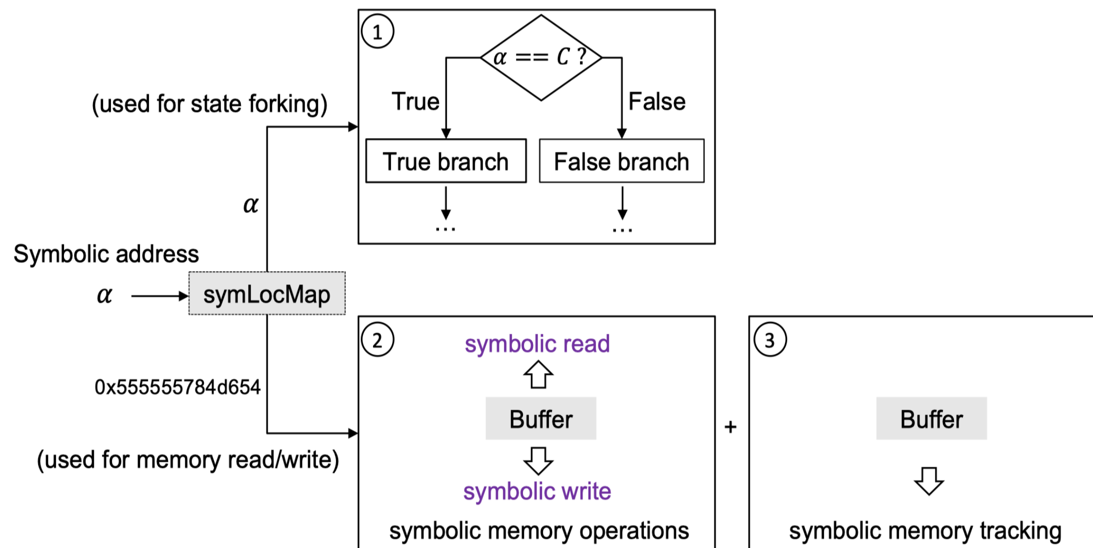
(new address symbolization + concretely mapped symbolic memory operation and tracking)

Conclusion

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➢ **SymLoc: a boosted symbolic execution engine for memory error detection**

(new address symbolization + concretely mapped symbolic memory operation and tracking)



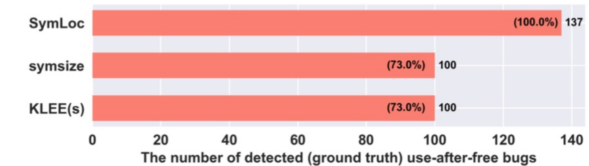


Fig. 12. Completeness of UAF error detection among symbolic execution-based approaches (137 in total)

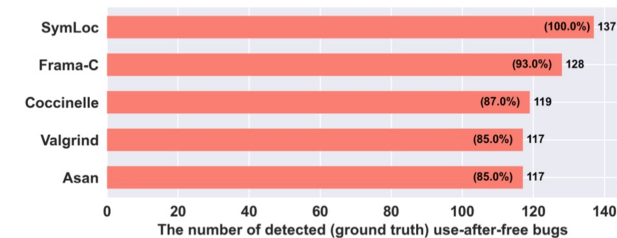


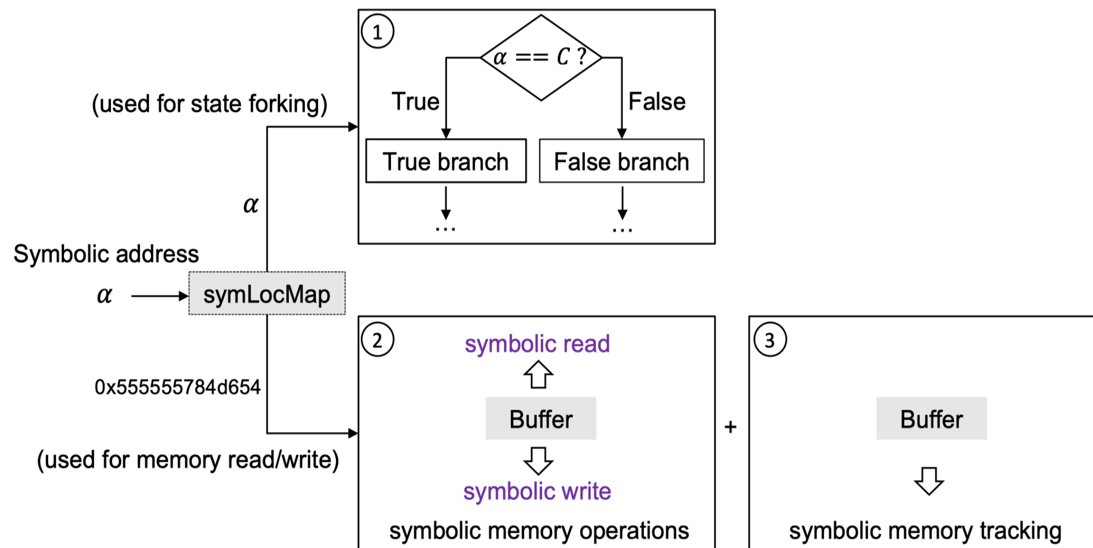
Fig. 10. Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total)

Conclusion

SMU SINGAPORE MANAGEMENT UNIVERSITY | School of Computing and Information Systems

➤ **SymLoc: a boosted symbolic execution engine for memory error detection**

(new address symbolization + concretely mapped symbolic memory operation and tracking)



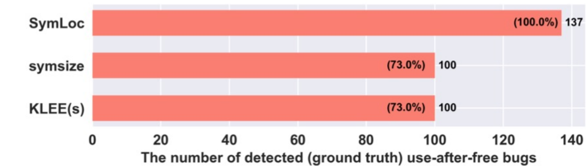


Fig. 12. Completeness of UAF error detection among symbolic execution-based approaches (137 in total)

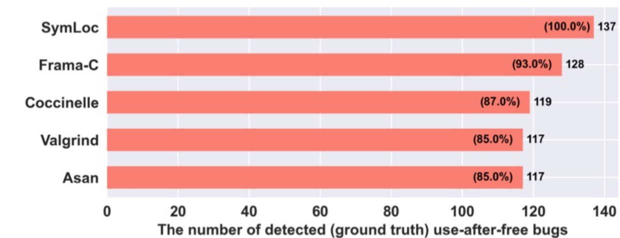


Fig. 10. Completeness of UAF error detection among static/dynamic analysis-based approaches (137 in total)

- **Future work**
  - More complete modeling of memory
    - Integrate the modeling of size/offset/address togeter
  - Apply our approach to memory mangament systems
    - Heap allocators

Code

Thank you & Questions?

Concretely Mapped Symbolic Memory Locations for Memory Error Detection

Haoxin Tu, Lingxiao Jiang, Jiaqi Hong, Xuhua Ding (Singapore Management University)
He Jiang (Dalian University of Technology)

16/04/2024, Lisbon