Automatic Exploitable Heap Layout Generation for Heap Overflows Through Manipulation Distance-Guided Fuzzing

Bin Zhang Jiongyi Chen Runhao Li

Chao Feng Ruilin Li Chaojing Tang

National University of Defense Technology

Introduction

Heap-based buffer overflows (heap overflows) are becoming one of the most prevailing threats to software.

Heap Layout Manipulation
SHRIKE/GOLLUM/MAZE

Works on Interpreters/Kernel

Previous HLM work depend on explicit, powerful and easy-to-trigger heap primitives.

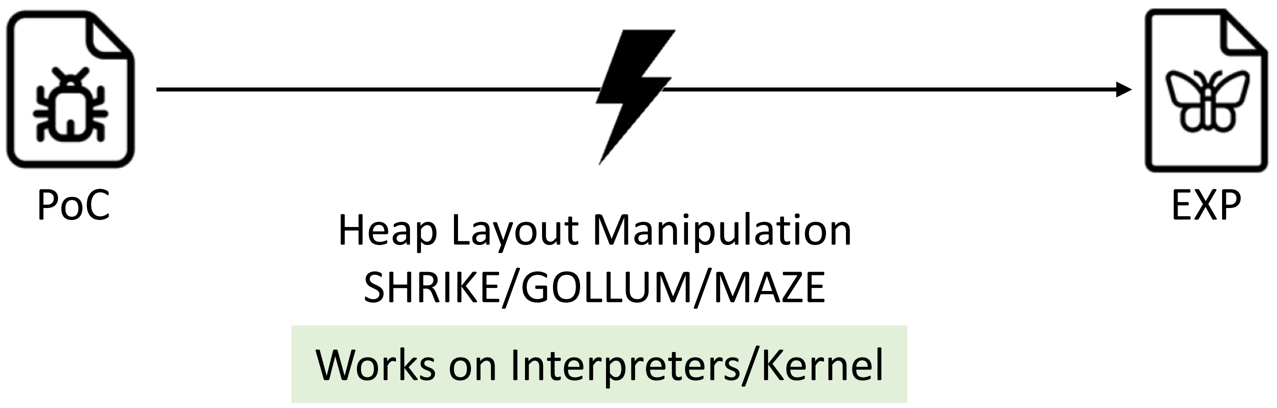
And the input generation for exploit code is simple.

```
$var = str_repeat("STR", x)  ⟶  p = malloc(x)
$var = 0                     ⟶  free(p)
```

Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. CCS 2019.

Introduction

Heap-based buffer overflows (heap overflows) are becoming one of the most prevailing threats to software.



Previous HLM work depend on explicit, powerful and easy-to-trigger heap primitives

General purpose programs (such as image parsers, executable parsers, word processors) are lack of explicit, powerful and easy-to-trigger heap primitives, and the side effect introduced by the heap managers makes HLM more complicated.

Sean Heelan, Tom Melham, and Daniel Kroening. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. CCS 2019.

Motivating Example

```
void main(int argc, char *argv[]) {
  char *data = read_file(argv[1]);
  uint32_t offset, home_num = 0;
  // process optional entry information block if needed
  if (data[0] == 'E' && data[1] == 'N')
    offset += process_home_entries(data+2,&home_num);
  char *file_magic = (char*)malloc(0x2); // m2
  memcpy(file_magic, data+offset, 2);
  FI *c_file = NULL;
  // process optional file information block if needed
  if (file_magic[0] == 'F' && file_magic[1] == 'I') {
    offset += 2;
    c_file = (FI*)malloc(sizeof(FI)); // m3
    for (int i = 0; i < home_num; ++i) { // loop l3
      uint8_t name_size = *(uint8_t*) (data + offset);
      c_file->name[i] = (char*)malloc(name_size); // m4
      memcpy(c_file->name[i], data+offset+1, name_size);
      offset += (name_size+1);
    }
  } else
    free(x); // f2
  uint32_t data_len  = *(uint32_t*)(data+offset);
  char *pure_data = (char*)malloc(0xF0); // m5
  memcpy(pure_data, data+offset+4, data_len);   //overflow
  ...
}
```

```
int process_home_entries(char *data,uint32_t* home_num) {
  char *all_homes[3]; int read = 0;
  for (int i = 0; i < 3; ++i) { // loop l1
    char *user_home_raw = data + read;
    int home_len = strlen(user_home_raw);
    if (home_len == 0xF0 || home_len == 0x20) {
      all_homes[i] = strdup(user_home_raw); // m1
      *home_num += 1;
    }
    ... // update read
  }
  ... // process all home entries
  for (int i = 0; i < 3; ++i)  // loop l2
    if (all_home[i]) free(all_home[i]); // f1
  return read;
}
```

PoC

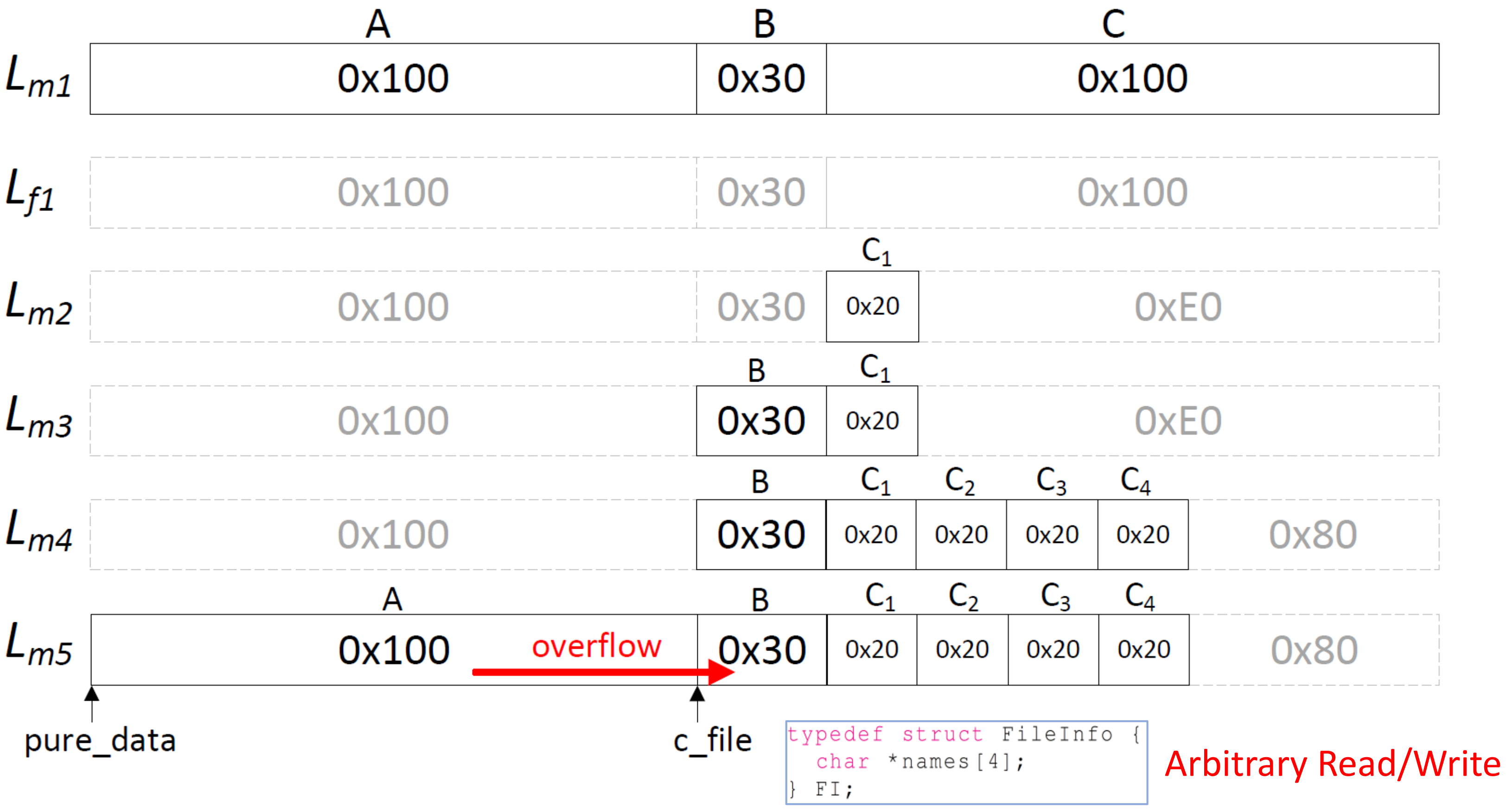
\x00\x01\x00\x00T···T

Motivating Example

PoC: `\x00\x01\x00\x00T···T`

	home entry	file information	pure data
ePoC:	`ENa···a\x00b···b\x00c···c\x00`	`FI\x02dd\x02dd\x02dd`	`\x00\x01\x00\x00T···T`

0xF0 0x20 0xF0

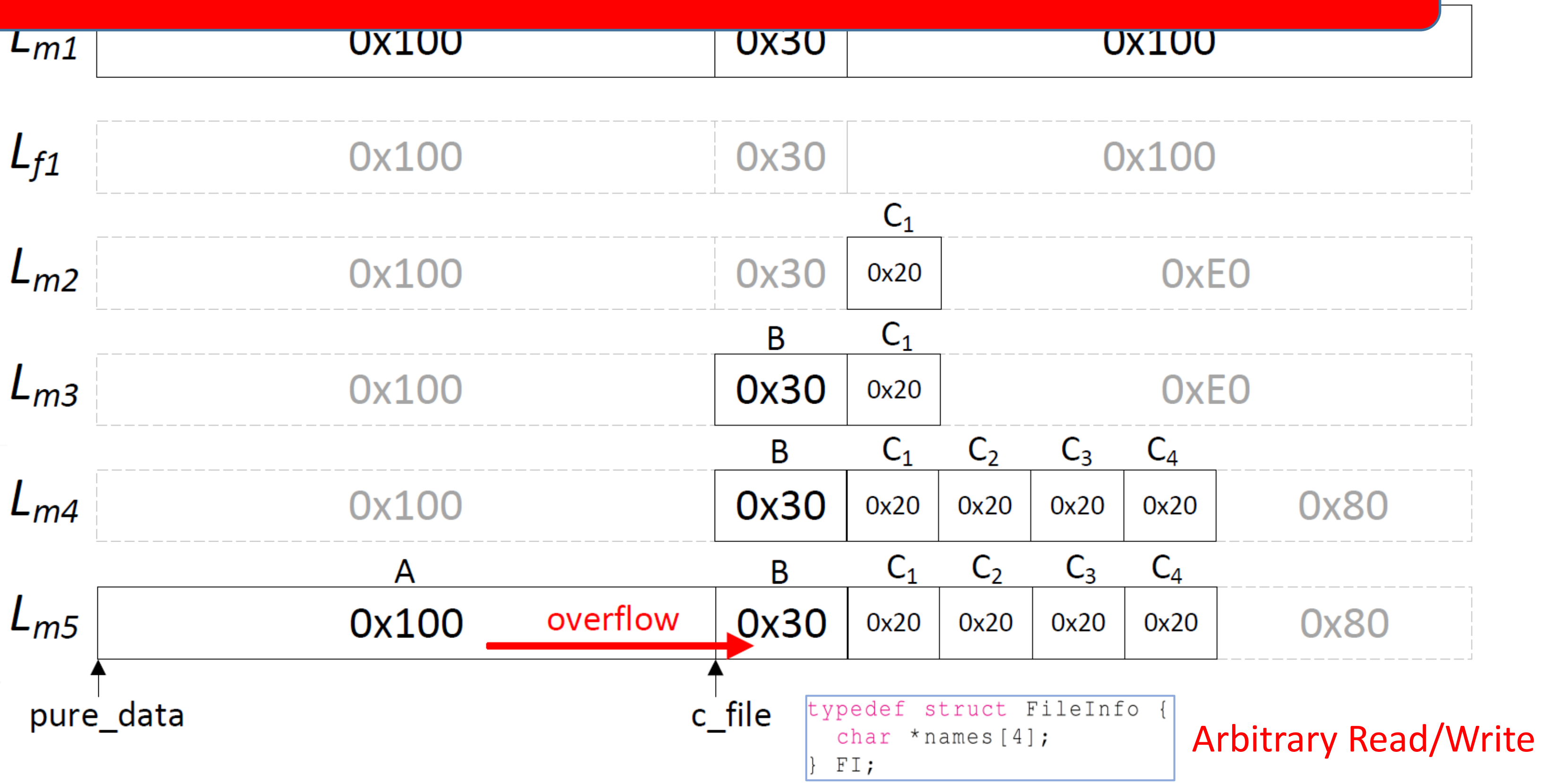


Motivating Example

PoC: `\x00\x01\x00\x00T···T`

It is difficult to extract precise heap layout primitives!

```c
void main(int argc, char *argv[]) {
  char *data = read_file(argv[1]);
  uint32_t offset, home_num = 0;
  // process optional entry information block if needed
  if (data[0] == 'E' && data[1] == 'N')
    offset += process_home_entries(data+2,&home_num);
  char *file_magic = (char*)malloc(0x2); // m2
  memcpy(file_magic, data+offset, 2);
  FI *c_file = NULL;
  // process optional file information block if needed
  if (file_magic[0] == 'F' && file_magic[1] == 'I') {
    offset += 2;
    c_file = (FI*)malloc(sizeof(FI)); // m3
    for (int i = 0; i < home_num; ++i) { // loop l3
      uint8_t name_size = *(uint8_t*) (data + offset);
      c_file->name[i] = (char*)malloc(name_size); // m4
      memcpy(c_file->name[i], data+offset+1, name_size);
      offset += (name_size+1);
    }
  } else
    free(x); // f2
  uint32_t data_len  = *(uint32_t*)(data+offset);
  char *pure_data = (char*)malloc(0xF0); // m5
  memcpy(pure_data, data+offset+4, data_len);  //overflow
  ...
}
```

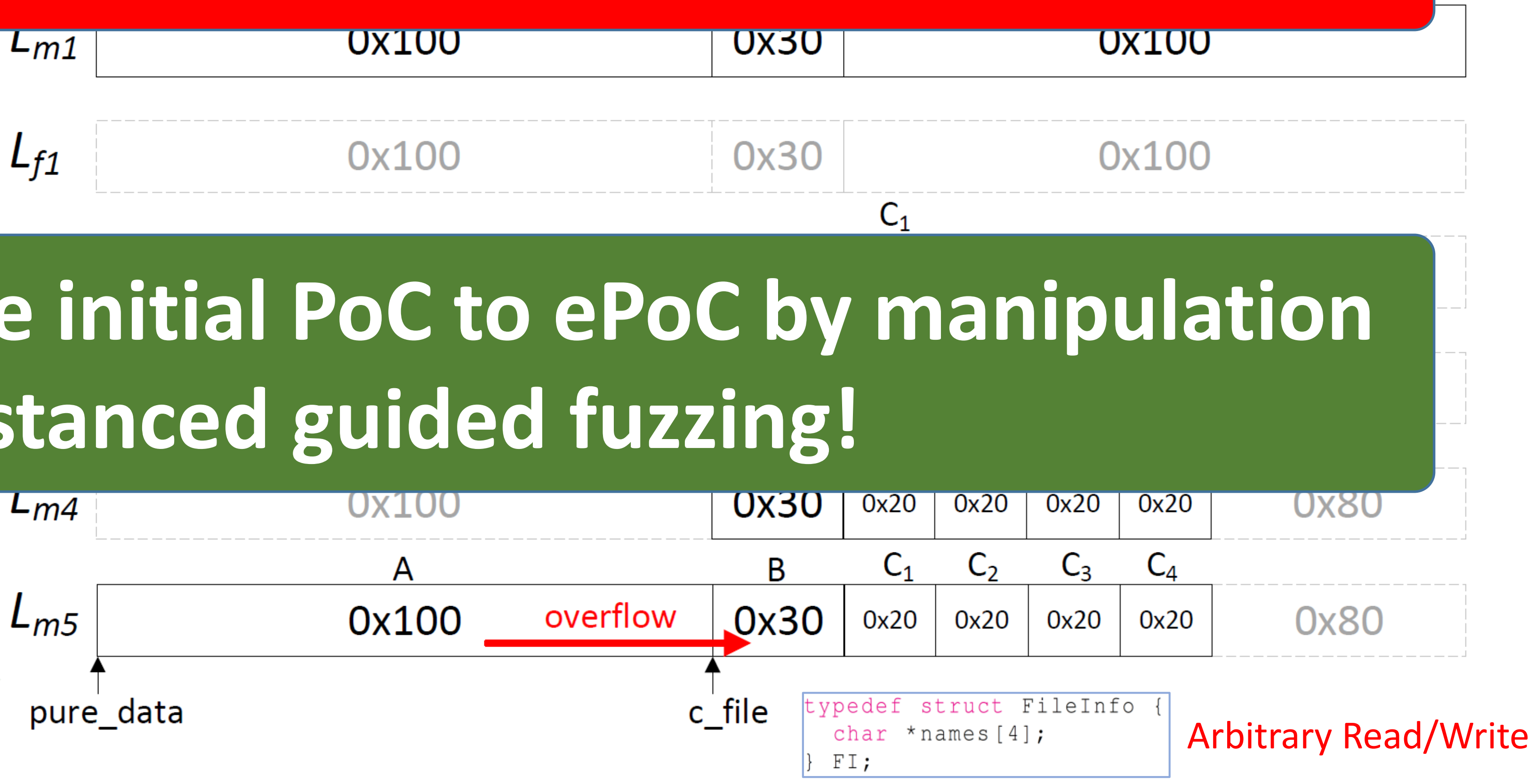


Motivating Example

PoC: \x00\x01\x00\x00T···T

It is difficult to extract precise heap layout primitives!

```
void main(int argc, char *argv[]) {
  char *data = read_file(argv[1]);
  uint32_t offset, home_num = 0;
  // process optional entry information block if needed
  if (data[0] == 'E' && data[1] == 'N')
    offset += process_home_entries(data+2, &home_num);
  char *f
  memcpy(
  FI *c_f
  // proc
  if (fil
    offse
    c_fil
    for (
      uin
      c_f
      memcpy(c_file->name[i], data+offset+1, name_size);
      offset += (name_size+1);
    }
  } else
    free(x); // f2
  uint32_t data_len  = *(uint32_t*)(data+offset);
  char *pure_data = (char*)malloc(0xF0); // m5
  memcpy(pure_data, data+offset+4, data_len);  //overflow
  ...
}
```



Transforming the initial PoC to ePoC by manipulation distanced guided fuzzing!

Technical Challenges

Challenge 1: How to specify a desired exploitable layout?

- Gollum & Maze flexibly specify a victim object or a desired as their input.
- It is difficult to specify desired layouts for general-purpose programs without using powerful primitives, because the creation of victim objects is highly dependent on the program’s execution logic.

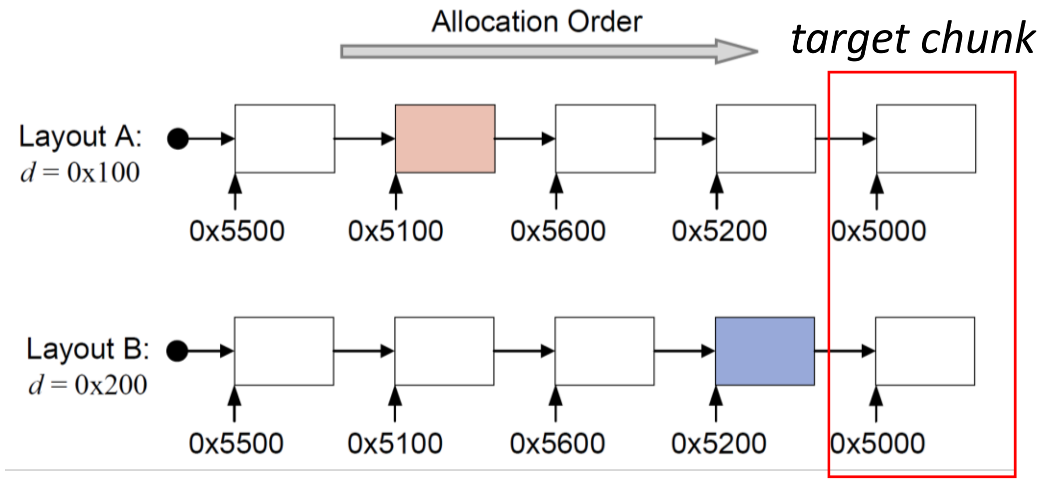
```c
void main(int argc, char *argv[]) {
  char *data = read_file(argv[1]);
  uint32_t offset, home_num = 0;
  // process optional entry information block if needed
  if (data[0] == 'E' && data[1] == 'N')
    offset += process_home_entries(data+2,&home_num);
  char *file_magic = (char*)malloc(0x2); // m2
  memcpy(file_magic, data+offset, 2);
  FI *c_file = NULL;
  // process optional file information block if needed
  if (file_magic[0] == 'F' && file_magic[1] == 'I') {
    offset += 2;
    c_file = (FI*)malloc(sizeof(FI)); // m3
    for (int i = 0; i < home_num; ++i) { // loop l3
```

c_file is created only when the input contains the optional file information block.

Technical Challenges

Challenge 2: How to improve the efficiency of the fuzzing-based approach?

- The mutation in the program input is to eventually control the heap operations.
- The metrics used in prior fuzzing-based approaches are coarse grained, as they measure the manipulation objective using the distance in the memory space.



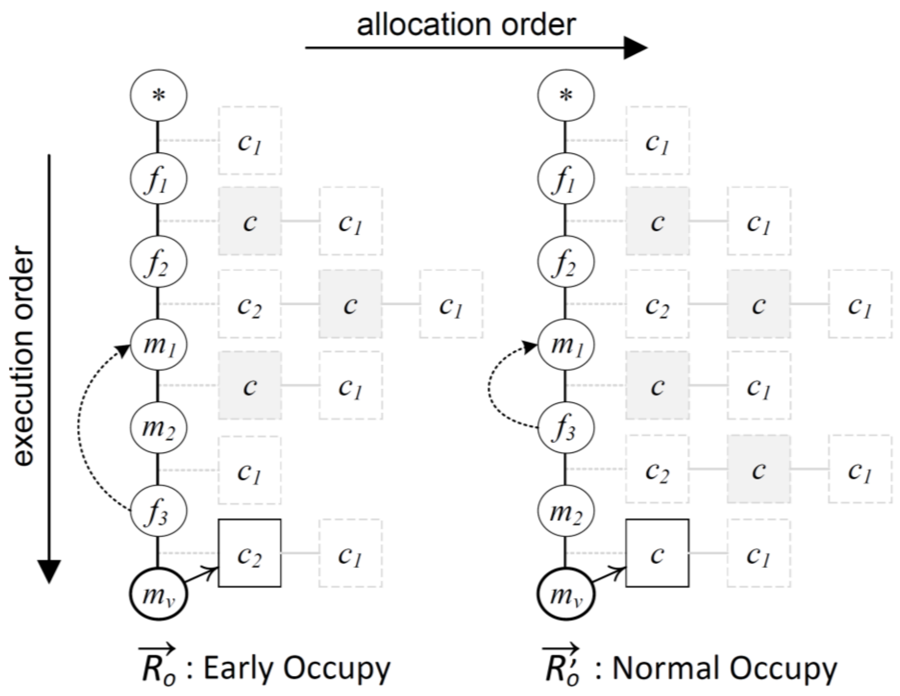
It seems that layout A is better than layout B since $d_A < d_B$. However, layout B is supposed to be better because it only needs one more allocation but A needs three more allocations to occupy the target chunk.

Technical Challenges

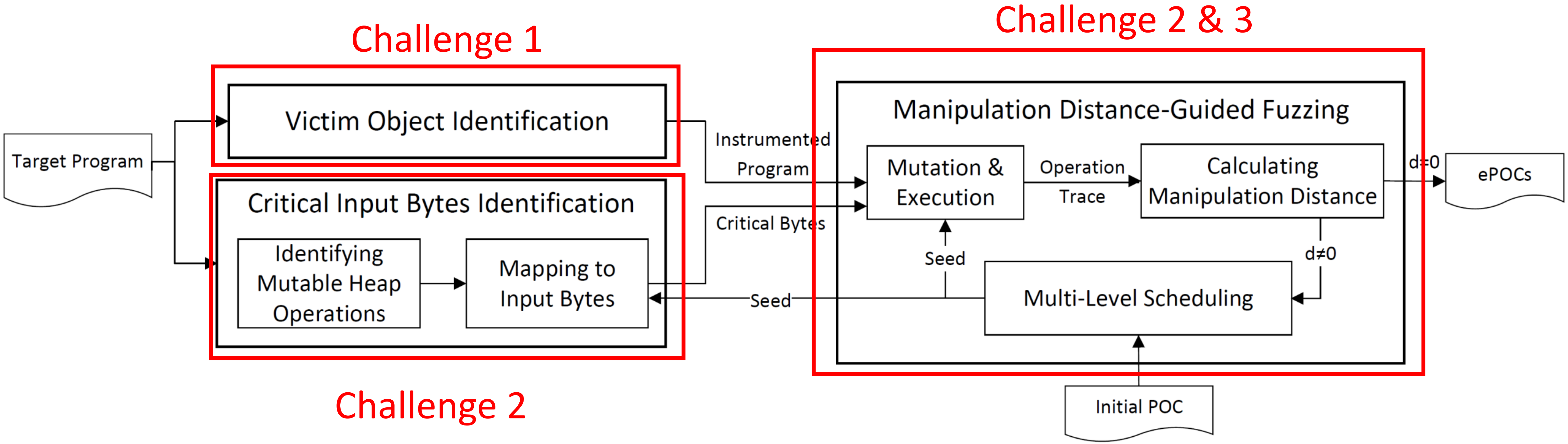
Challenge 3: How to model the side effects brought by complex heap behaviors so as to precisely control the manipulation?

- Chunk split/merge mechanism.
- Early Occupation Problem.

e.g., Target chunk c is allocated by operation m_1 but then is never freed before operation m_v which allocates memory for vulnerable object. In this case, c is early occupied and leads the manipulation to fail.



Overview



SCATTER

Tech 1: Victim Objects Identification

SCATTER focuses on the following 3 types of sensitive structures:

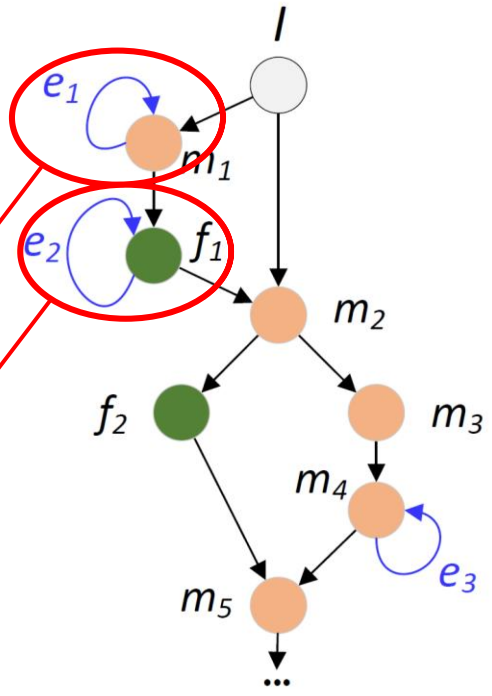
- A structure that contains pointers.
- A structure that has no pointers but contains a member that can affect a buffer's access.
- A *union* structure that contains previous two types of structures and is accessed as its structure type.

SCATTER hooks all *bitcast* instruction on LLVM IR to identify victim object and collects a victim object o_s information as:

$$\langle \underbrace{index}_{\text{allocation index in the execution trace}}, \underbrace{type\_id}_{\text{static sensitive structure ID}}, \underbrace{addr}_{\text{heap address of this object}} \rangle$$

Tech 2: Pinpointing Critical Input Bytes

- Identifying Mutable Operations
  - A mutable heap operation is an operation whose parameter(s) as well as execution times can be affected by input.
  - We constructs and leverages the Layout Dependence Graph (LDG) built from heap-operation-guided fuzzer to identify mutable operations.
    - Each vertex is represent as $v = \langle o, s_c \rangle$, where *o* is the operation type, and s_c is the call stack.
    - Whether the parameters are mutable is determined by dynamic taint analysis.
    - Whether the execution times are mutable is determined by checking LDG's back edges' hit times.

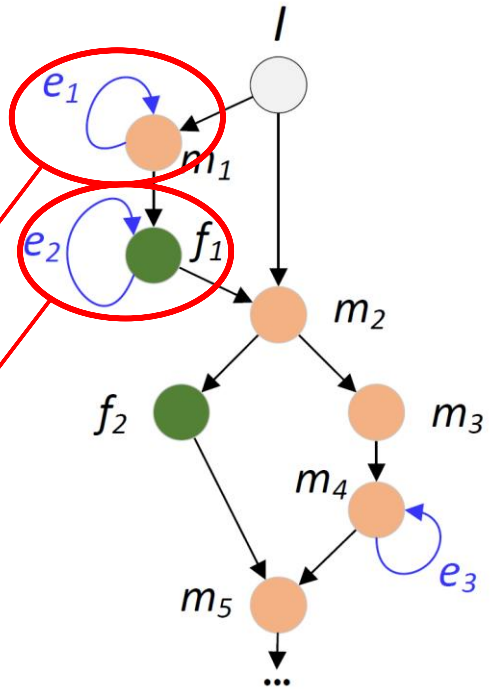


both parameter and execution times of m_1 are mutable

execution times are mutable of f_1 is mutable

Tech 2: Pinpointing Critical Input Bytes

- Identifying Mutable Operations
  - A mutable heap operation is an operation whose parameter(s) as well as execution times can be affected by input.
  - We constructs and leverages the Layout Dependence Graph (LDG) built from heap-operation-guided fuzzer to identify mutable operations.
    - Each vertex is represent as $v = \langle o, s_c \rangle$, where *o* is the operation type, and s_c is the call stack.
    - Whether the parameters are mutable is determined by dynamic taint analysis.
    - Whether the execution times are mutable is determined by checking LDG's back edges' hit times.



Use a lightweight "mutate-check" strategy to locate input bytes that can affect mutable heap operations.

Tech 3: Manipulation Distance

- Basic Manipulation Distance Definition
  - For each victim object identified at runtime, we traverse the trace of heap operations $\overrightarrow{R}$ to locate all *suitable free chunks* for placing the vulnerable object o_v, and calculate the manipulation distances.
  - Given a suitable free chunk *c*, and its position index δ in its free list, we define the manipulation distance τ_d to occupy *c* with o_v as :

- freed before allocating o_v
- has the same size with o_v
- can overflow to o_s

o_v : vulnerable object

o_s : victim object

$\overrightarrow{R}$: heap operation trace

$$\tau_d = |\delta + \zeta_{\{0,1\}} \cdot n_F - n_A - 1|$$

Remove all the free chunks whose position index is before chunk c in c's free list.

- n_A and n_F denote the number of allocation and free operations with the same size of *c* in $\overrightarrow{R}$.
- $\zeta_{\{0,1\}}$ = 0 if free list behaves FIFO, $\zeta_{\{0,1\}}$ = 1 for FILO.

Tech 3: Manipulation Distance

- Handling Split-Merge Mechanism
  - The side effect caused by the split-merge mechanism affects the accurate calculation
  - We update of manipulation distance (i.e., n_A and n_F) according to different behaviors.

e.g., target chunk locates in free list L_x, for a free operation free(c) where c's size is y:

Size Condition	Merge Behavior	Distance Updating
$y \neq x$	act as an allocation operation merges with one chunk in L_x	$\boxed{n_A = n_A + 1}$ if the merged chunk's allocation order is before the target free chunk c's
$y \neq x$	merges with another chunk and the result chunk's size is *x*.	$n_F = n_F + 1$
$y \neq x$	merges with another chunk and the result chunk's size is not *x*.	*N/A*
$y = x$	merges with one chunk in L_x	$n_F = n_F + 1$ $n_A = n_A + 1$ if the merged chunk's allocation order is before chunk c's
$y = x$	merges with chunk in another free list	$n_F = n_F - 1$

Tech 3: Manipulation Distance

- Handling Early Occupation Problem

$$\tau_d = |\delta + \zeta_{\{0,1\}} \cdot n_F - n_A - 1| \longrightarrow \tau_d = |1 + 1 \cdot 2 - 2 - 1| = 0 \quad \otimes$$

Chunk c is early occupied by m_2 and leads the manipulation to fail.

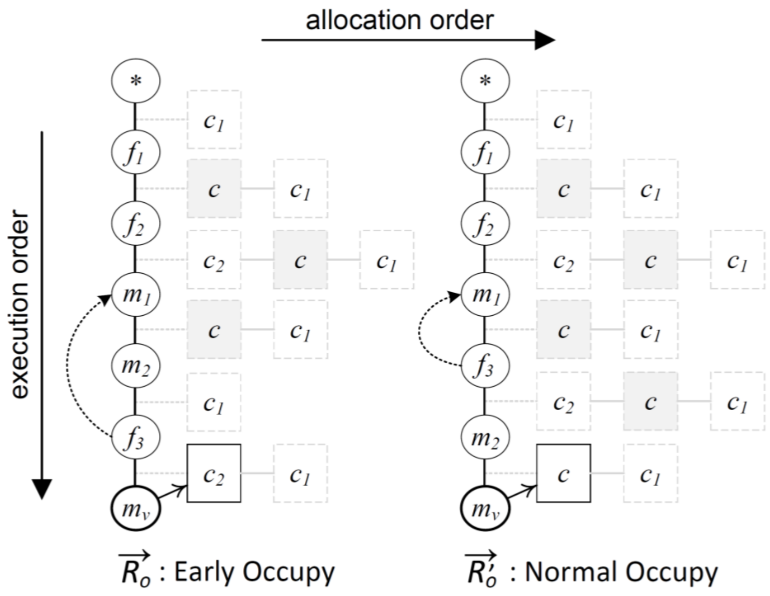
- We introduced *overload factor* to describe the overall changes. For $\vec{R}_o = \langle \sigma_1, \sigma_2, \cdots, \sigma_N \rangle$,

  Let $\sum_{i=1}^{\theta} e_i$ denotes accumulated changes of L_c *until* σ_θ,

  After executing σ_θ, the position index of chunk c is updated to:

  $$\delta + \sum_{i=1}^{\theta} e_i - 1$$ negative means chunk c is already occupied!

  $$\Delta_{\vec{R}_o} = \max_{1 \le \theta \le N} \left(\max\left(1 - \delta - \sum_{i=1}^{\theta} e_i, 0\right)\right)$$



- The final extended manipulation distance is:

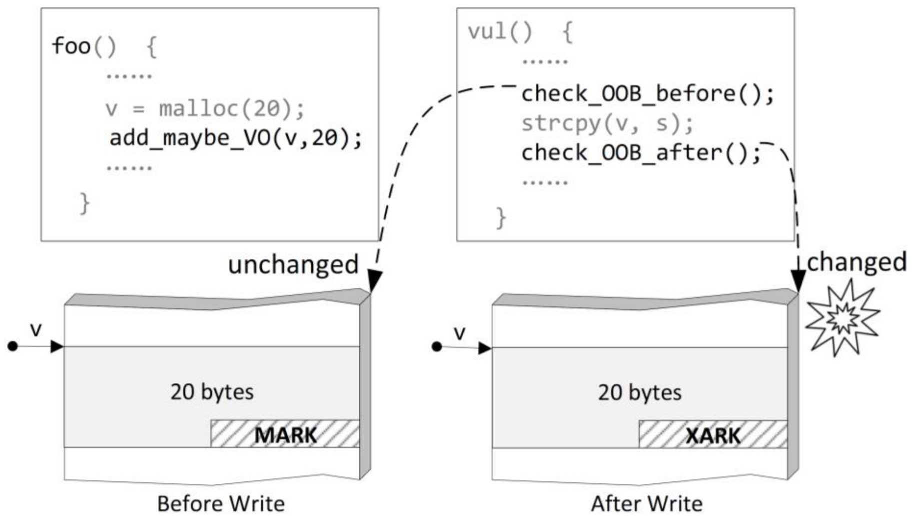
$$d = \tau_d + \Delta_{\vec{R}_o}$$

Tech 4: Distance Guided Fuzzing

- How to determine a mutated PoC triggers the same vulnerability as the initial PoC does?

Address Sanitizer? NO! It changes target's heap layout!!!

- SCATTER disables ASAN and implements the following three instrumentation functions, to determine whether the PoCs trigger the same bug.



Advantages:

- No affect to heap layout
- Less overhead since check happens only when overflow writing occurs after v_o is created

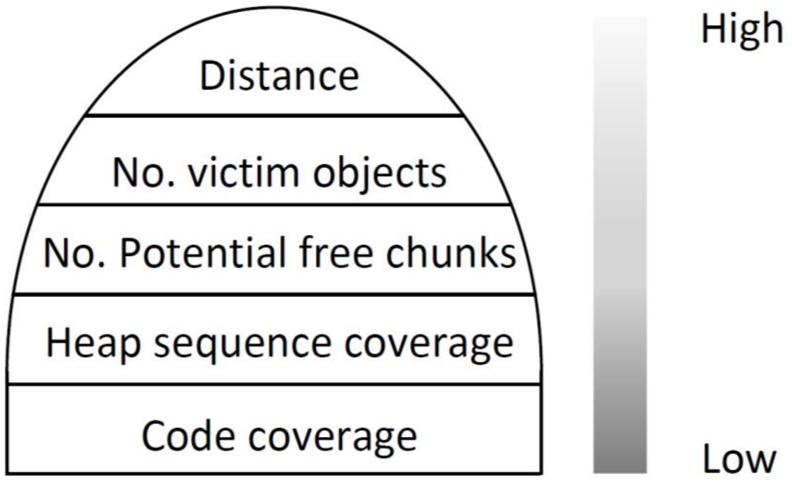
Short backs:

- Cannot detect discrete overflow writing

Tech 4: Distance Guided Fuzzing

- Which PoCs deserve higher priorities to fuzz and how much mutation energy should be assigned?
  - An interesting PoC that should be preserved for further fuzzing if it has:
    - ✓ *Shorter distance*
    - ✓ *More victim objects*
    - ✓ *More free chunks*
    - ✓ *Diverse heap operation sequences*
    - ✓ *New code coverage*
  - SCATTER adopts a greedy seed schedule strategy.
  - For each scheduled test case, SCATTER generates an expansion factor ε to adjust the mutation energy.

$$\varepsilon = a \cdot \frac{1}{d} + b \cdot \frac{n_s}{N_s} + c \cdot \frac{n_c}{N_c}$$



Evaluation

Benchmark Selection Rules:

- The vulnerabilities cause heap OOB-write overflows and their PoCs are public.
- The programs are open sourced general-purpose programs.
- The programs do not implement their customized heap managers.

We select 27 heap overflow vulnerabilities in 10 real-world general-purpose programs as our benchmark. The input types include:

- executable files
- command line arguments
- images
- raw text files

General-purpose programs

✓ Ubuntu 18.04 LTS server (with default Glibc version 2.27) running with 128G RAM and Intel(R) Xeon(R) Gold 6254 CPU @3.10GHz*70.

✓ Each case in our benchmark is fuzzed for 10 times, and each fuzzing campaign lasts for 24 hours.

Evaluation

- ePoC generation result.
  - ✓ successful cases: 18
  - ✓ total ePoCs: 126

- Failure reasons:
  - Limited number of victim objects
  - Limited heap operations
  - Limited explored paths
  - Running failure

  e.g., CVE-2018-15209 consumes 102 seconds to trigger the final crash, which impedes fuzzing from running.

Program	Vulnerability ID	Length of Overflow	# of Overflowed Victims in PoC[2]	# of Victims in PoC[3]	# of Sensitive Struct. /Identified Victims	# of Mutable Cycles/Ops.	# of Unique ePoCs
readelf	CVE-2019-9077	16[1]	0	3	246 / 89	681 / 273	3
	CVE-2017-6965	16	0	1	273 / 88	652 / 264	0
sudo	CVE-2021-3156	unlimited	2	99	165 / 88	496 / 647	4
exiv2	Issue-456	16	0	13	474 / 129	83 / 90	15
	CVE-2018-17230	16	0	13	474 / 129	88 / 86	13
	CVE-2018-11531	151	0	120	486 / 145	82 / 80	14
	CVE-2017-12955	74	0	2	301 / 112	88 / 86	2
	CVE-2017-11339	16	0	2	302 / 112	85 / 84	2
	CVE-2017-1000127	16	0	34	301 / 112	90 / 81	7
	CVE-2018-17229	16	0	5	481 / 115	88 / 86	4
opj_decompress (openjpeg suite)	CVE-2020-6851	16	0	104	72 / 89	88 / 86	3
	CVE-2014-7903	45	0	27	58 / 28	94 / 76	3
opj_compress (openjpeg suite)	CVE-2017-14039	16	1	27	71 / 84	79 / 83	10
	CVE-2017-14164	16	0	25	71 / 84	71 / 37	10
vim	CVE-2021-4019	1023	0	13	285 / 20	397 / 106	0
	Issue-2466	12092	0	15	272 / 27	433 / 138	5
	CVE-2022-0359	800	0	1	355 / 18	413 / 118	0
	CVE-2022-0392	16	0	1	273 / 23	422 / 126	0
gpac	Issue-1703	138	0	52	1243 / 191	253 / 128	2
	Issue-1317	16	0	11	1243 / 234	237 / 116	22
	CVE-2019-20162	16	0	5	1150 / 234	253 / 128	2
	CVE-2022-26967	16	0	57	1470 / 191	261 / 122	5
ffjpeg	CVE-2019-16352	16	0	2	5 / 7	10 / 5	0
tiff2pdf (libtiff suite)	CVE-2018-15209	16	0	3	50 / 18	130 / 290	✗[4]
	CVE-2018-16335	16	0	1	50 / 18	130 / 290	✗
ngiflib	CVE-2019-16346	16	0	1	5 / 12	6 / 4	0
	CVE-2019-16347	48	0	1	5 / 12	6 / 4	✗

Evaluation

- Comparison with State-of-the-Art

  ✓ SCATTER generated the highest number of ePoCs among all the tools. Compared to the other three tools, the number of ePoCs generated by SCATTER is increased by 133.3%, 38.6%, 31.3% and 6.8%.

  ✓ Since the distance of Gollum less accurate, the number of ePoCs found by SCATTER_G decreases 30 when compared with SCATTER.

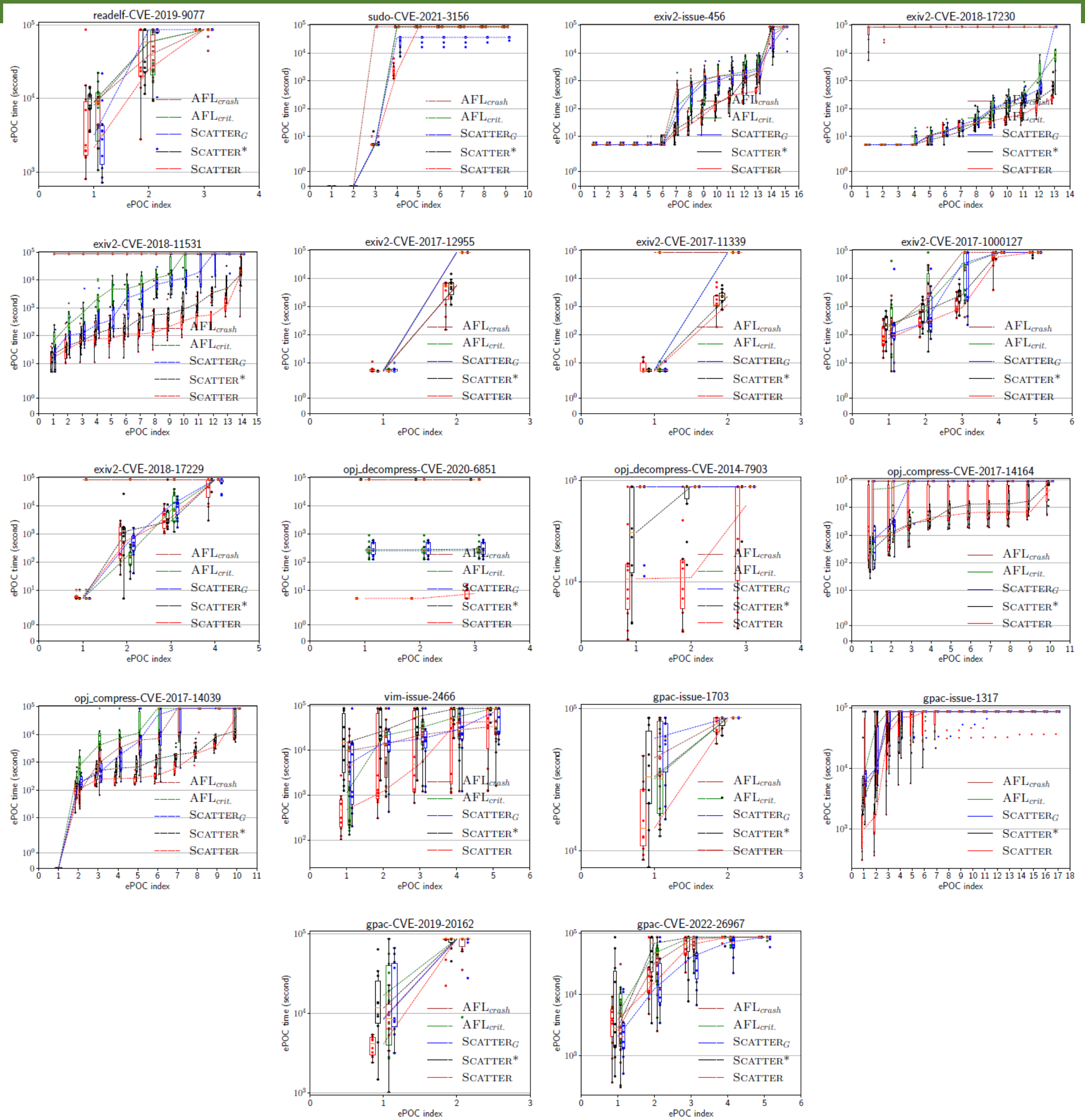
  ✓ After introducing the critical input bytes, $\text{AFL}_{crit.}$ successfully uncovers ePoCs in 18 cases. Since $\text{AFL}_{crit.}$ schedules seeds based on code coverage, it ignores the seeds that identify new heap operation sequences. (*The queue size of SCATTER is 7.3x than* $\text{AFL}_{crit.}$.)

Vulnerability	AFL_{crash}	$\text{AFL}_{crit.}$	SCA.^1_G	SCA.*	SCA.
CVE-2019-9077	2	3	2	3	3
CVE-2017-6965	0	0	0	0	0
CVE-2021-3156	2	3	3	8	4
Issue-456	6	15	15	15	15
CVE-2018-17230	2	13	12	13	13
CVE-2018-11531	4	13	14	14	14
CVE-2017-12955	1	1	1	2	2
CVE-2017-11339	0	1	1	2	2
CVE-2017-1000127	3	3	3	7	7
CVE-2018-17229	3	3	4	4	4
CVE-2020-6851	0	3	3	3	3
CVE-2014-7903	0	3	2	3	3
CVE-2017-14039	8	7	7	10	10
CVE-2017-14164	2	3	3	10	10
CVE-2021-4019	0	0	0	0	0
Issue-2466	5	5	5	5	5
CVE-2022-0359	0	0	0	0	0
CVE-2022-0392	0	0	0	0	0
Issue-1703	1	1	1	2	2
Issue-1317	8	7	11	10	22
CVE-2019-20162	2	2	2	2	2
CVE-2022-26967	5	5	7	5	5
CVE-2019-16352	0	0	0	0	0
CVE-2019-16346	0	0	0	0	0
Total	**54**	**91**	**96**	**118**	**126**

Evaluation

- Time Consumption

Evaluation



- ✓ The average time to generate an ePoC for SCATTER is around 1 hour.
- ✓ The total time consumed to generate all ePoCs for SCATTER is decreased by 59.3%, 41.5%, 32.2%, 21.1%, when compared with AFL$_{crash}$, AFL$_{crit.}$, SCATTER$_G$, and SCATTER*.
- ✓ SCATTER also shows a more stable performance (the time to generate ePoCs is less accidental).

Discussion

SCATTER generates exploitable heap layouts for heap overflows of general-purpose programs, working in a primitive-free manner by adopting a fuzzing-based method.

- Automatically identifies potential victim objects at runtime by instrumentation.
- Defined a more accurate distance to measure heap layout manipulation result.
- Handled the side effects that popularly exist in heap managers.

Limitations:

- Implemented only for glibc (ptmalloc).
- Customized heap managers.
- Multi-threads programs.

Thanks / Questions?

b.zhang@nudt.edu.cn