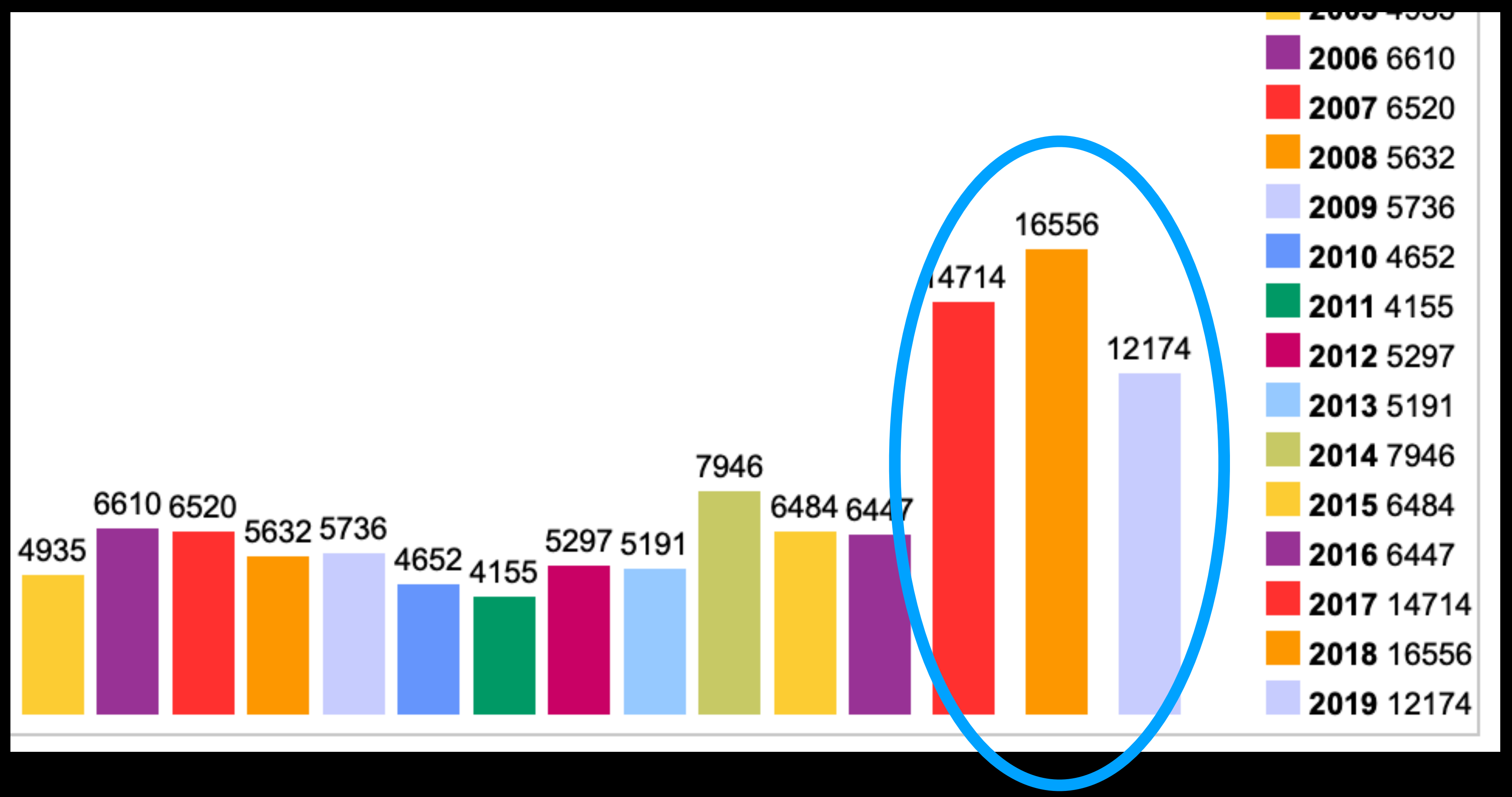
# MAZE: Towards Automated Heap Feng Shui

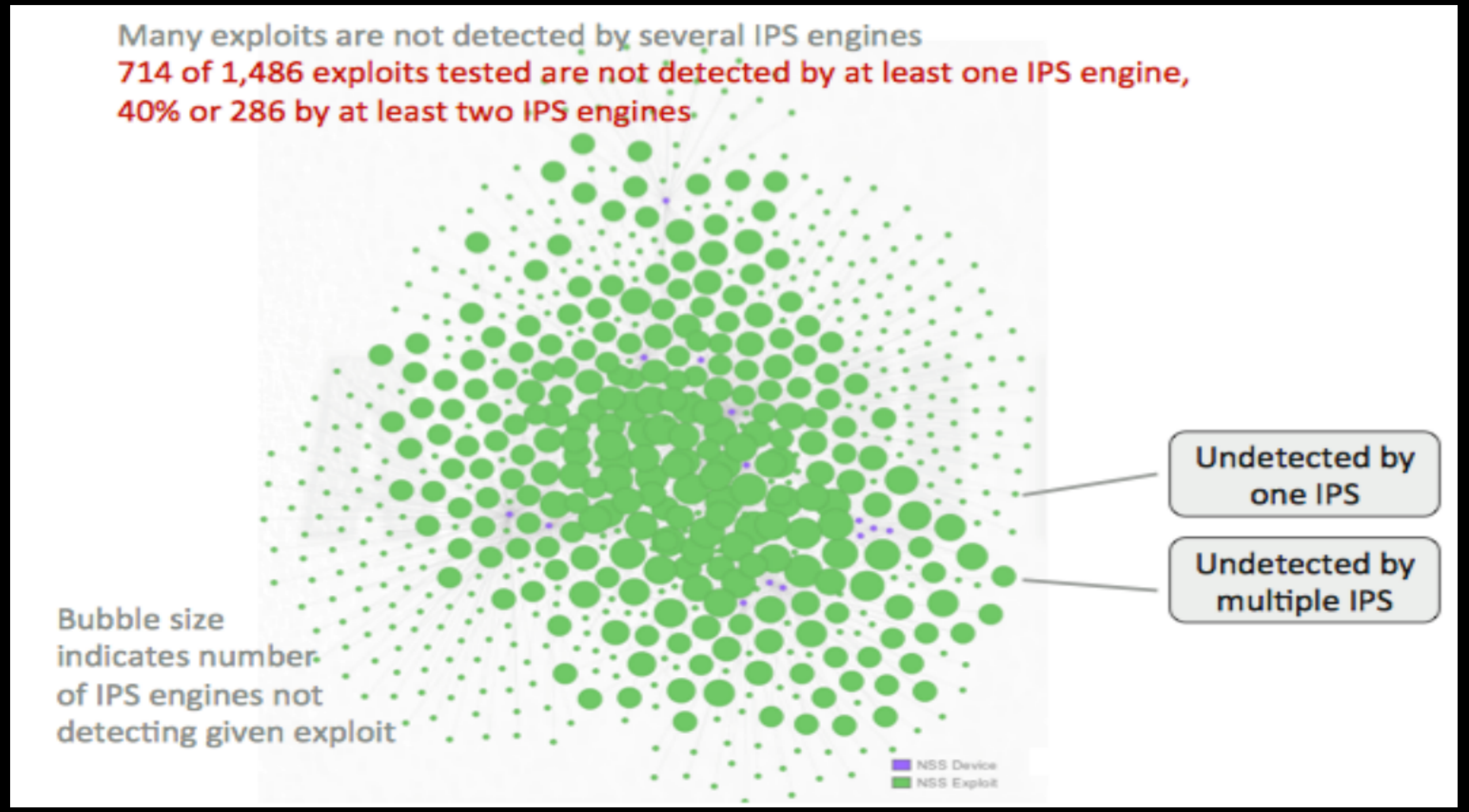
Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, Wei Zou
wangy0129@gmail.com

# 0x01 Motivation

# AEG is important

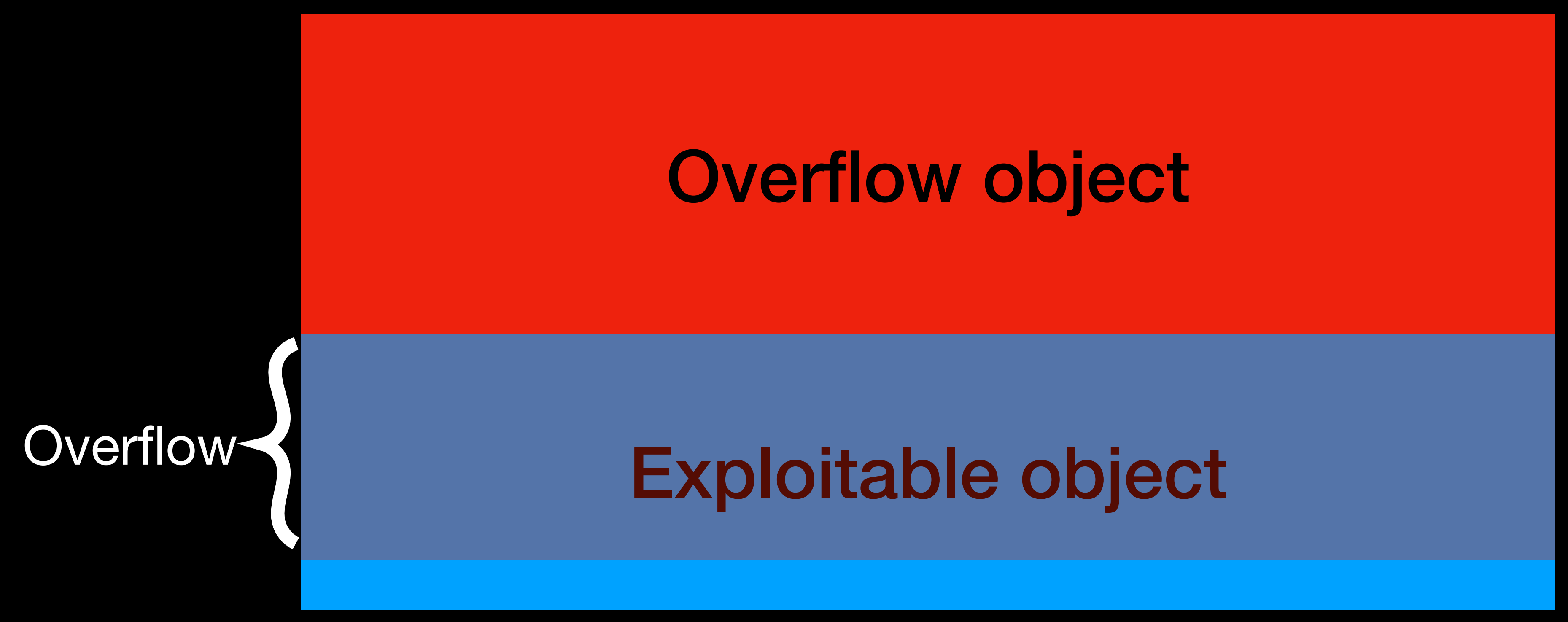
- The number of vulnerabilities is growing explosively.
- Software vendors need to quickly evaluate the severity of security vulnerabilities and allocate appropriate resources to fix critical ones.
- Defenders could learn from synthetic exploits to generate IDS (Intrusion Detection System) rules and block potential attacks.



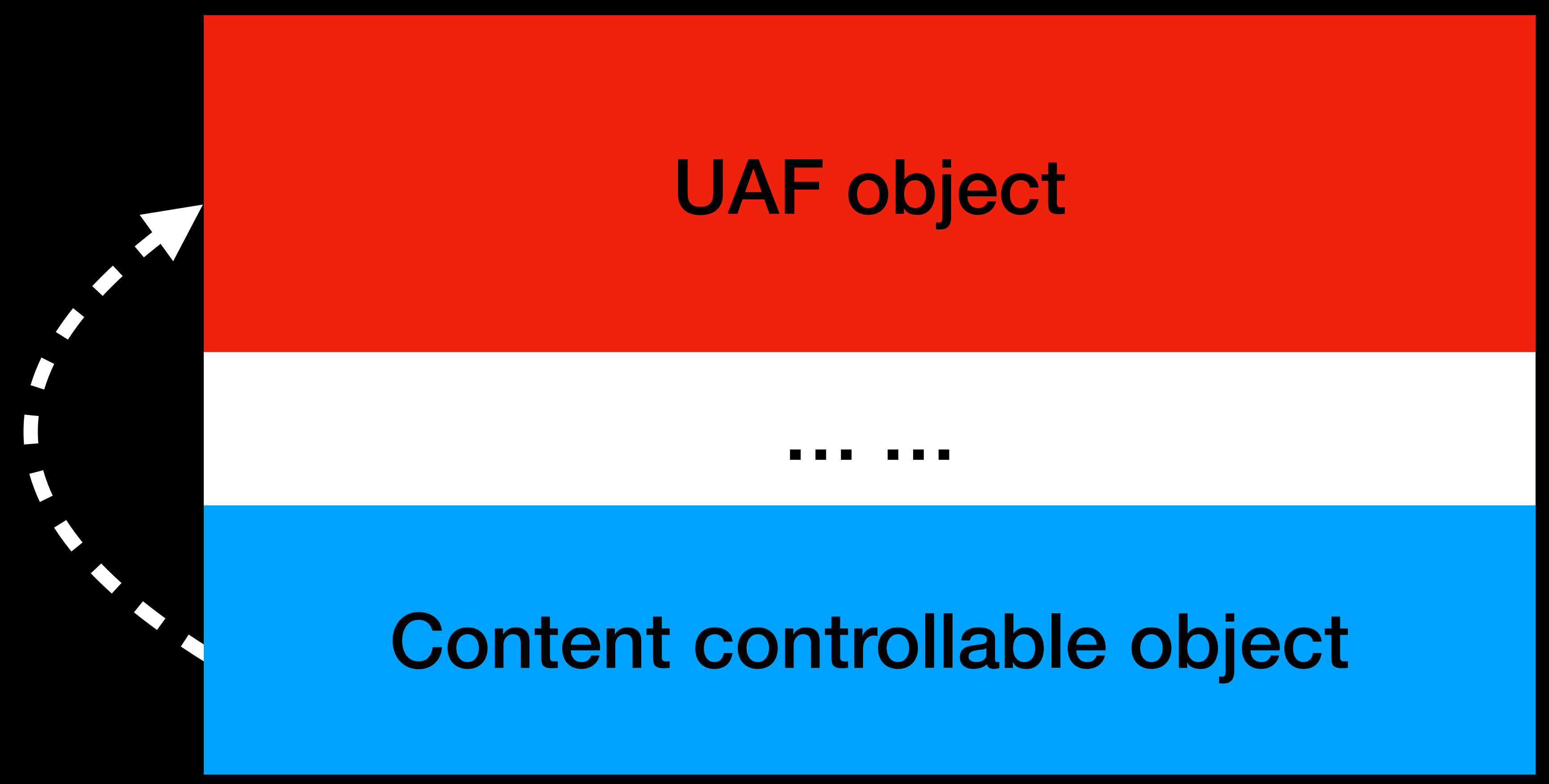



# Automated Heap Feng Shui is demanded

- Lots of vulnerabilities, e.g., heap overflow and UAF, could only be exploited in specific heap layouts via techniques like heap feng shui.
  - Heap overflow: An exploitable object should be placed at a position which is next to the overflowed object.
  - UAF: A content controllable object should be placed at the freed object's position.
  - Complex exploit techniques require complicated heap layout, e.g., unsafe unlink attack requires two chunks to be allocated before and after the overflowed chunk.



Heap overflw



Use After Free

# Problem Scope

Maze

## Expected Memory Layout Generation

Vulnerability analysis
—> ASAN, Valgrind(Memcheck)

Expected Memory Layout
—>Heuristic, e.g. in this example, a controllable object (e.g. switch->name) should take the freed exceptional object's position

## Memory Layout Manipulation

Heap Layout Primitives Analysis

Heap Layout Primitives Assembly

## Full Chain Exploit Composition

Exploit primitive composition
—> Automated multi-hop exploit generation

Security mechanisms bypass
—> bypass ASLR, NX, CFG

# 0x02 Introduction

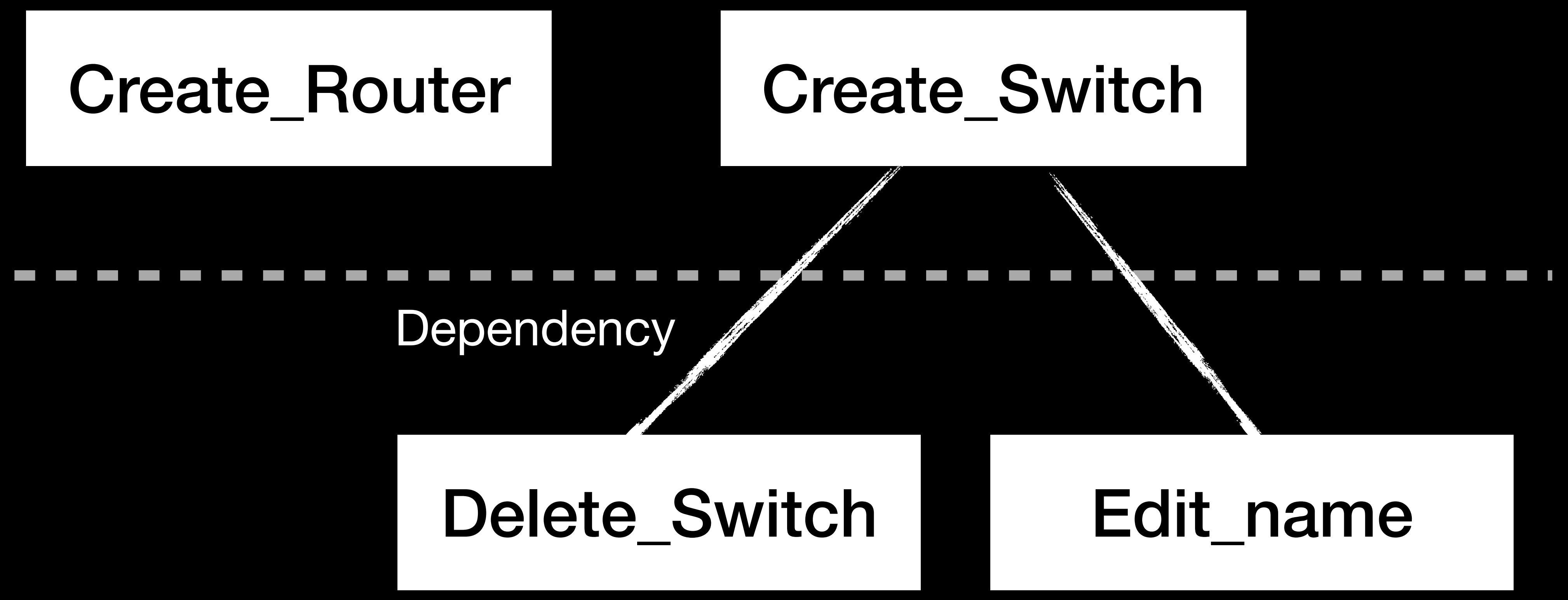
# Example: A UAF vulnerability

```
void main(void){
  while(1){ switch(c){               //function dispa
     case 1: Create_Router();        //primitive 1
     case 2: Create_Switch();        //primitive 2
     case 3: Delete_Switch();        //primitive 3
     case 4: Edit_name(); }          //           }}
Router Create_Router(){...
     Router *router    = malloc(0x160);
     router->protocol = malloc(0x160);
     router->r_table  = malloc(0x160); ...}
Switch Create_Switch(){...
     Switch *switch    = malloc(0x160);
     switch->name      = malloc(0x160);
     glist[count++]    = switch;      ...}
void Delete_Switch(int index){...
    if (glist[index]!=Null) {..
     free(glist[index]);
     free(glist[index]->name); }..    ...}
void Edit_name(int index){...
     Switch *s  = glist[index];
     read(0, s->name, 0x60)           ...}
```

- Program: An interactive program that selects the corresponding function based on user input
- UAF: Delete_Switch() function does not clear the pointer to this object after deleting an object
- Expected layout: a controllable object (i.e. switch->name) should take the freed object's position (i.e. switch), to hijack the sensitive pointer s->name, yielding arbitrary memory writes.
- Layout primitive noise: There is at least one noise (de)allocation in one primitive.

# Primitives Analysis in Example

```
Router Create_Router(){...
     Router *router   = malloc(0x160);
     router->protocol = malloc(0x160);
     router->r_table  = malloc(0x160); ...}
Switch Create_Switch(){...
     Switch *switch   = malloc(0x160);
     switch->name     = malloc(0x160);
     glist[count++]   = switch;       ...}
void Delete_Switch(int index){...
   if (glist[index]!=Null) {..
     free(glist[index]);
     free(glist[index]->name); }..    ...}
void Edit_name(int index){...
     Switch *s  = glist[index];
     read(0, s->name, 0x60)           ...}
```




- Primitives Extraction
  - Reentrant code snippets: exist in function dispatchers that are enclosed in loops.
  - Maze utilizes the code structure characteristic to recognize candidate heap layout primitives, via static analysis.
  - Example: Create_Router, Create_Switch, Delete_Switch
- Primitives Dependency Analysis
  - Some reentrant code snippets may depend on other snippets.
  - Maze analyzes the pre-condition and post-condition of each snippet to recognize such dependencies and merge them into one primitive
  - Example: Delete_Switch <—> Create_Switch
- Primitives Semantics Analysis
  - It's necessary to understand the semantics of each primitive, especially the size of (de)allocated  objects
  - Taint analysis and symbolic execution.
  - Example: Create_Router: 3 malloc, size=0x160; Create_Switch: 2 malloc. Size = 0x160; Delete_Switch: 2 free, target from Create_Switch

# Challenge —How to assemble primitives ?

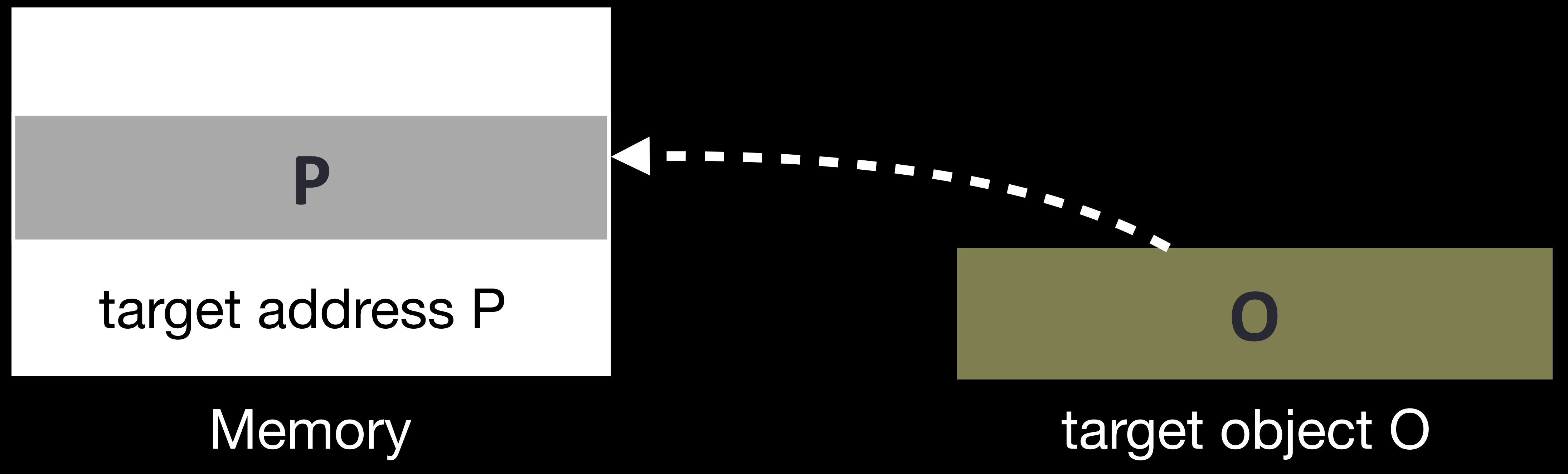
- Random Search? (SHRIKE) Genetic Algorithm ? (Gollum)
  - Path space explosion
  - Unnecessary time consumption
  - Low success rate
- Why? *Noise*: unwanted (de)allocations in heap primitives.

| Allocator | Noise | % Overall Solved | % Natural Solved | % Reversed Solved |
|---|---|---|---|---|
| avrlibc-r2537 | 0 | 100 | 100 | 99 |
| dlmalloc-2.8.6 | 0 | 99 | 100 | 98 |
| tcmalloc-2.6.1 | 0 | 72 | 75 | 69 |
| avrlibc-r2537 | 1 | 51 | 50 | 52 |
| dlmalloc-2.8.6 | 1 | 46 | 60 | 31 |
| tcmalloc-2.6.1 | 1 | 52 | 58 | 47 |
| avrlibc-r2537 | 4 | 41 | 44 | 38 |
| dlmalloc-2.8.6 | 4 | 33 | 49 | 17 |
| tcmalloc-2.6.1 | 4 | 37 | 51 | 24 |

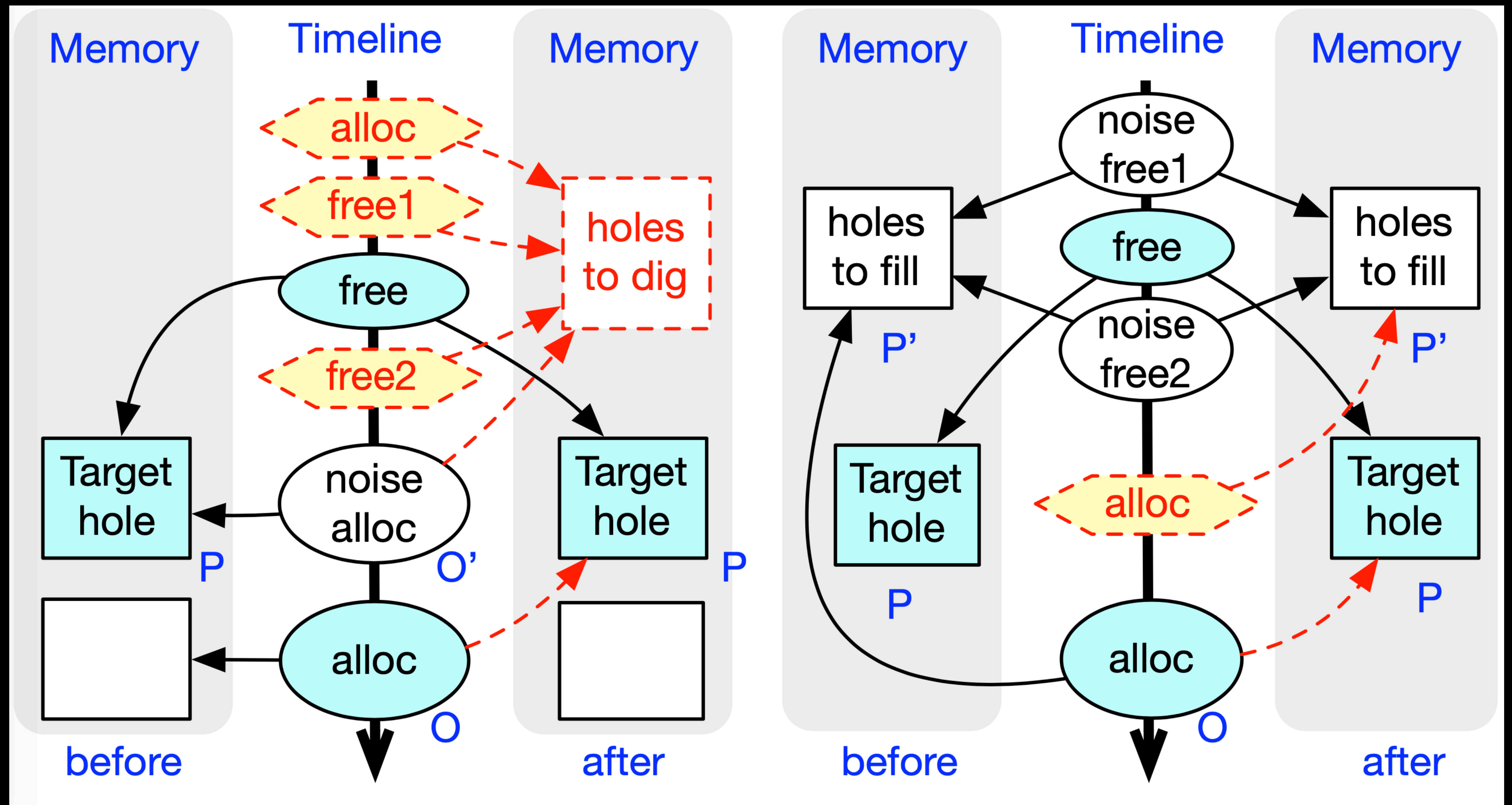
The success rate drops dramatically when the number of noises grows!

# Dig & Fill—A novel algorithm regardless of *Noise*

- Redefine Problem
  - One-object constraint layout: placing one object O into one target address P.
  - Multi-objects constraint layout: placing multi objects into multi target addresses.




- Dig
  - At the time of allocating the target object O, the target address P is taken by object O'.
  - We need to dig memory holes before allocating O', by adding primitives that could free objects of proper sizes, to accommodate O'.
- Fill
  - At the time of allocating the target object O, the target address P could be empty, but O still falls into other holes.
  - We need to fill (multiple) holes before allocating O, by adding primitives that could allocate objects.




(a) Dig operation (b) Fill operation

# Counteract *Noise* using *Diophantine Equations*

- Linear Diophantine Equation Setup

$$\begin{cases} \Delta d_1 x_1 + \Delta d_2 x_2 + \Delta d_3 x_3 + \ldots + \Delta d_n x_n + d = 0 \\ x_1, x_2, x_3 \ldots x_n \geq 0 \end{cases} \tag{1}$$

➡ Measurement Unit——Standard fill (or dig) operation

- 1) Contains only one allocation (or deallocation) and 2)the size equals to the size of O (or P)

➡ d: Target Distance (PoC) Measurement

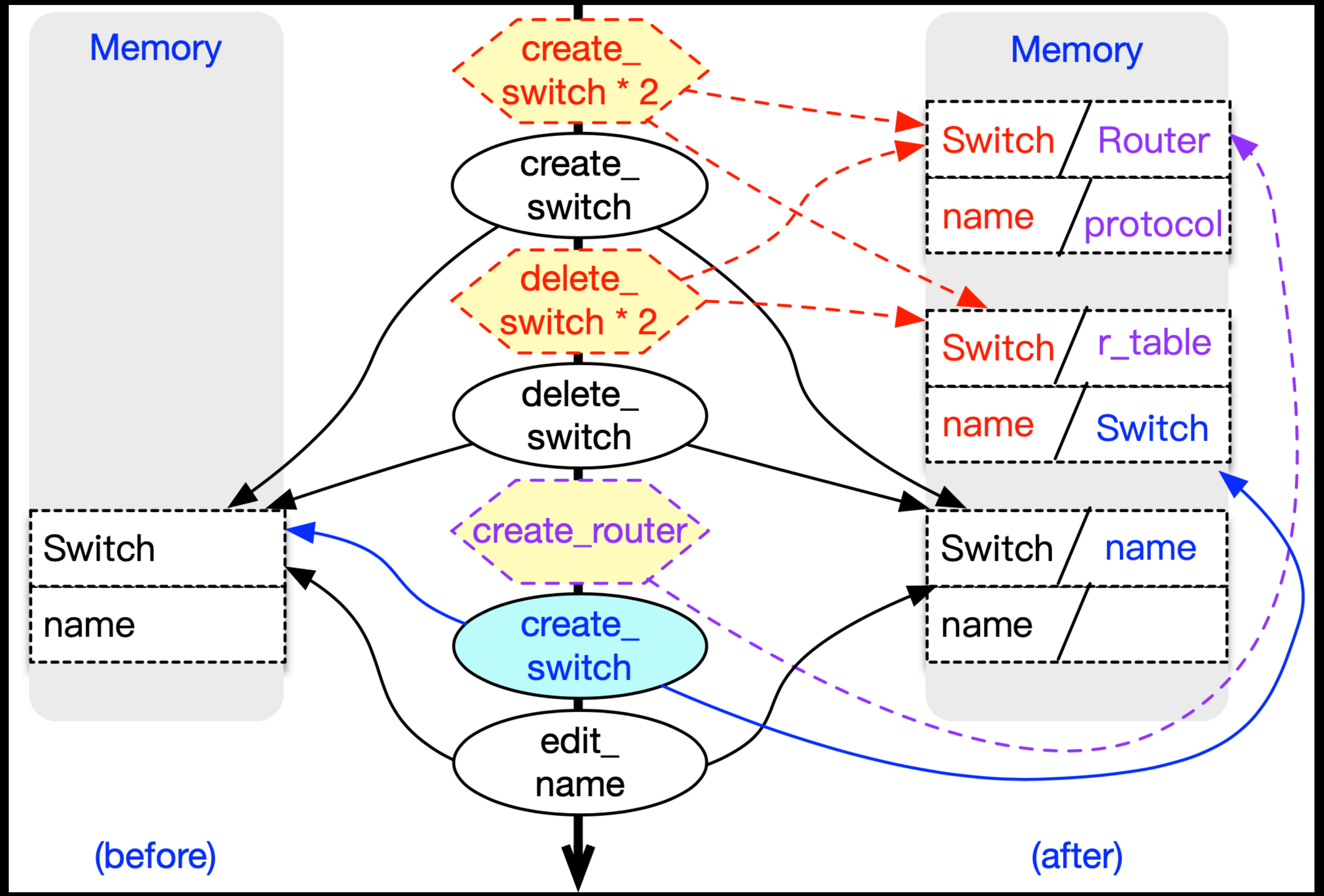
- Add standard fill (or dig) operations into the program execution trace of PoC, until the target object O is placed into the target address P.
- If d standard dig operations are required, Target Distance is +d. If d standard fill operations are required, Target Distance is -d.

➡ Δd: Delta Distance (Layout Primitives) Measurement

- Target Distance before and after inserting a primitive are d1 and d2, then the Delta Distance (Δd) of this primitive is d2-d1.

# Primitive Assembly in Example

```
Router Create_Router(){...
      Router *router    = malloc(0x160);
      router->protocol = malloc(0x160);
      router->r_table  = malloc(0x160);
Switch Create_Switch(){...
      Switch *switch    = malloc(0x160);
      switch->name      = malloc(0x160);
      glist[count++]    = switch;
void Delete_Switch(int index){...
    if (glist[index]!=Null) {..
      free(glist[index]);
      free(glist[index]->name); }..
void Edit_name(int index){...
      Switch *s   = glist[index];
      read(0, s->name, 0x60)
```




- The Target Distance (d) of POC is +1. (one standard dig operation is needed so that switch->name can be placed at the target position.
- The Δd of Create_Switch, Create_Router and Delete_Switch (combining with its dependant Create_Switch) are +2, +3 and -2
- A Linear Diophantine Equation can be build:

$$\begin{cases} 2x_1 + 3x_2 - 2x_3 + 1 = 0 \\ x_1, x_2, x_3 \geq 0 \end{cases} \Rightarrow x_1 = 0, x_2 = 1, x_3 = 2$$

- One Create_Router and two Delete_Switch primitives are needed.

# Overview of Maze

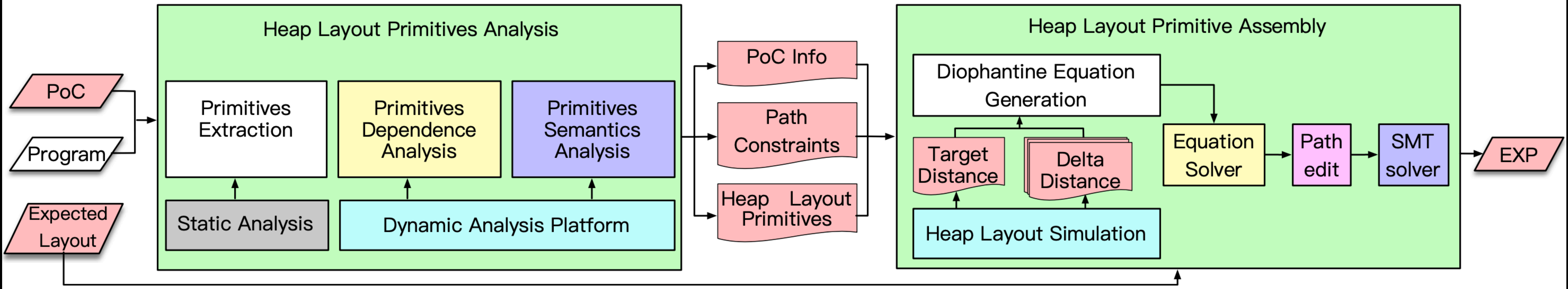



Figure 2: Overview of Maze

- Heap Layout Primitives Analysis
  - Taking the program and POC as inputs, Maze will extract primitives in them. ( Heap layout primitives (e.g., Create_Switch) are the building blocks for heap layout manipulation.)
- Heap Layout Primitives Assembly
  - The inputs of this part are heap primitives, POC info, path constraints and expected layout.
  - Maze will utilize heap primitives to manipulate POC's layout (infered from the POC info) to the expected layout and generate an exploit using a constraint solver.

# 0x03 Evaluation

# CTF benchmark

Table 1: CTF programs successfully processed by MAZE.

| Name | CTF | Vul Type | Final State |
|---|---|---|---|
| sword | PicoCTF ’18 | UAF | EIP hijack |
| hacknote | Pwnable.tw | UAF | EIP hijack |
| fheap | HCTF ’16 | UAF | EIP hijack |
| main | RHme3 CTF ’17 | UAF | Memory write |
| cat | ASIS Qual ’18 | Double free | Memory write |
| asvdb | ASIS Final ’18 | Double free | Memory leak |
| note3 | ZCTF ’16 | Heap bof | Unlink attack |
| stkof | HITCON ’14 | Heap bof | Unlink attack |
| Secure-Key-Manager | SECCON ’17 | Heap bof | Unlink attack |
| RNote2 | RCTF ’17 | Heap bof | Unlink attack |
| babyheap | RCTF ’18 | Off-by-one | Unlink attack |
| secret-of-my-heart | Pwnable.tw | Off-by-one | Unlink attack |
| Mem0 | ASIS Final ’18 | Off-by-one | Unlink attack |
| quotes_list | FireShell ’19 | Off-by-one | Unlink attack |
| freenote | 0CTF ’15 | Double free | Unlink attack |
| databank | Bsides Delhi | UAF | fastbin attack |

- MAZE can hijack control flow for 5.
- Leak arbitrary memory address information for 1.
- MAZE outputs exploitable layout without generating exploits for 10, extra techniques (e.g., unlink attack) are required.

Table 3: Heap layout primitives results on CTF programs.

| Program | Paths | Symbolized Paths | Independent Primitives | Dependent Primitives | Time(s) |
|---|---|---|---|---|---|
| sword | 118 | 11 | 5 | 5 | 500 |
| hacknote | 8 | 5 | 3 | 1 | 71 |
| fheap | 55 | 5 | 4 | 1 | 370 |
| main | 182 | 8 | 4 | 4 | 398 |
| cat | 44 | 10 | 4 | 5 | 1064 |
| asvdb | 7440 | 10 | 6 | 3 | 1156 |
| note3 | 198 | 6 | 4 | 2 | 942 |
| stkof | 30 | 11 | 1 | 3 | 267 |
| babyheap | 18 | 6 | 3 | 2 | 163 |
| secret... | 12 | 4 | 2 | 2 | 186 |
| Mem0 | 183 | 11 | 8 | 3 | 1099 |
| Secure... | 1332 | 55 | 5 | 3 | 445 |
| quotes... | 98 | 5 | 2 | 3 | 149 |
| freenote | 1068 | 7 | 3 | 4 | 1643 |
| RNote2 | 62 | 6 | 3 | 3 | 359 |
| databank | 100 | 11 | 9 | 2 | 192 |

- Path simplification: 15 programs’ paths are reduced to about 10 symbolized paths, the average rate of is 98.4%.
- Dependency Analysis: Column 5 shows the number of primitives that depend on others and can be analyzed by MAZE.

# Real world Program Benchmark

Table 5: Evaluation results of different solutions on PHP.

| Solution | Solve time(s) | Succ | POC analysis time(s) |
|---|---|---|---|
| Maze | 100% in 68s | 100% | 922s |
| Shrike | 25% in 300s, 60% in 3000+s | 60% | Not Supported |
| Gollum | 75% in 300s, 85% in 2500+s | 85% | Not Supported |

- Efficiency: MAZE is much faster than Shike and Gollum. MAZE: 100% in 68s. Shrikes: 25% in 300s. Gollum: 75% in 300s
- Effectiveness: MAZE can solve all the benchmarks. Shrike can only solve 60% of them, and Gollum solved 85%.
- Maze doesn't need a template to guide the heap layout manipulation process.

Table 6: Evaluation results on Python and Perl.

| Target | Vulnerabilities | Average time(s) |
|---|---|---|
| Python | CVE-2007-4965, 2014-1912, Issue24105, 24095, 24094 | 100% in 118s |
| Perl | Issue132544, 130703, 130321, 129024, 129012 | 100% in 141s |

- Compared with others, Maze broadly extends the application scope. (supports both Python and Perl)
- Maze can generate expected heap layouts for all of them, and is much faster.

# Synthetic Benchmarks

- Influence of heap layout noise

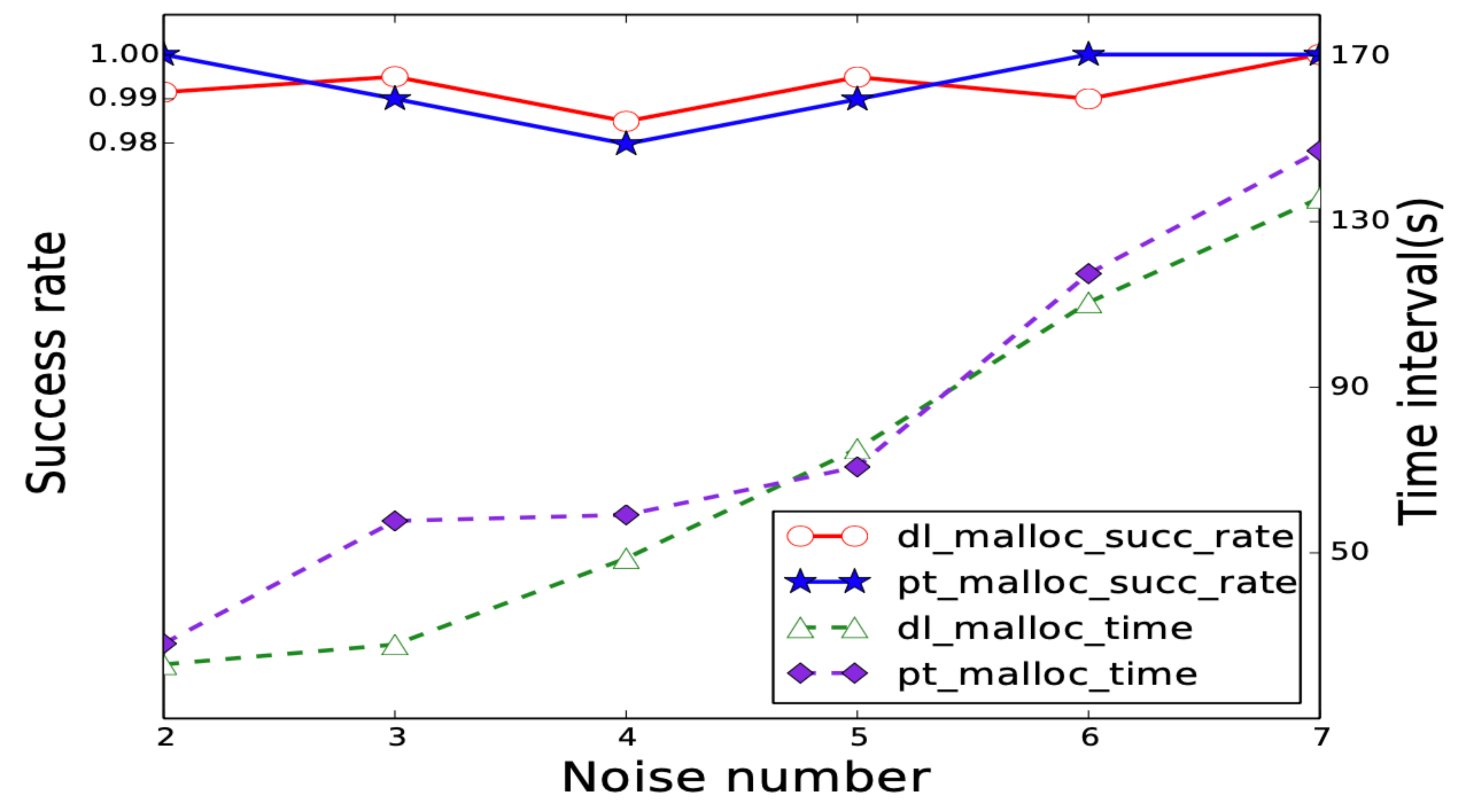



Figure 6: Influences of different number of noises.

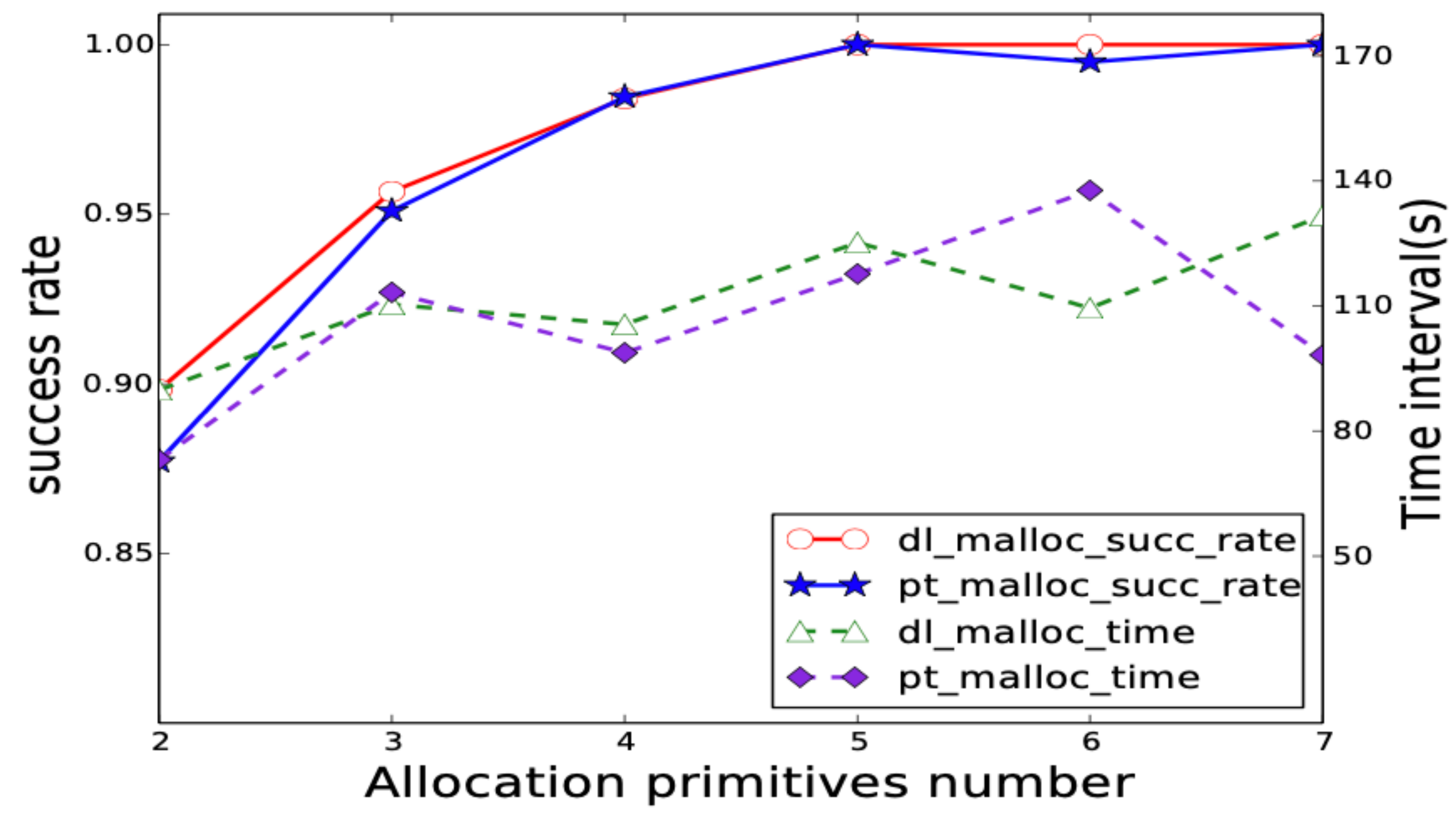



Figure 7: Influences of different number of primitives.

- The success rate keeps between 98% and 100%, showing that the number of noises does not influence the success rate of Dig & Fill.
- The time cost increases along with the number of noises, since noises will make the heap layout more complicated and cost more time to solve them.
- The number of primitives increases, the success rate also increases. This proves that the diversity of primitives influences the success rate. (still >= 87.7%)
- The time spent by MAZE to solve the problem does not grow along with the number of primitives.

# Synthetic Benchmarks

- Multi-object Position Constraint

Table 8: Results of multi-object layout constraint evaluation.

| Target | Object count | Time (s) | Success rate | Nature | Reversed |
|---|---|---|---|---|---|
| PT | 2 | 73.1 | 98.0% | 72.1% | 27.9% |
| PT | 3 | 95.2 | 97.0% | 55.1% | 44.9% |
| PT | 4 | 145.6 | 96.4% | 52.2% | 47.8% |
| PT | 5 | 238.8 | 95.6% | 50.4% | 49.6% |

- Setup: a) noise is 3 ; b) 3 allocation primitive and 4 deallocation primitive; c) 100 random heap layouts for each constraint
- While the number of objects increases (from 2 to 5), the success rate decreased (still > 95%) and the time interval increased: With more object layout constraints, MAZE has to generate more Diophantine Equations to solve.
- The order of allocation relative to memory corruption direction doesn't influenced the success rate: For 5 object constraints, the Nature ratio is even 50%, but the success rate can still be 95.6%. (Nature means an earlier allocation takes the lower memory address but a later allocation takes the higher address)

# 0x04 Take-away

# Conclusion

- MAZE can transform POC samples' heap layouts into expected layouts and automatically generate working exploits when possible.
- MAZE extends heap layout primitives to reentrant code snippets in event loop driven applications, and could efficiently recognize and analyze them.
- MAZE adopts a novel Dig & Fill algorithm to assemble primitives to generate expected layout, by deterministically solving a Linear Diophantine Equation.
- Maze is very efficient and effective and can even support multi-object constraints and many heap allocators.

# Other Challenges of AEG[1]

- Exploit Specification problem (A, H)
- Input generation problems (B, C, D, E)
- Exploit Primitive composition problem (F)
- Environment determination (I, J, K)
- State space representation (G)
- …

Maze (problem I:Heap likelihood inference)

[1] J.Vanegue, "The automated exploitation grand challenge," in *presented at H2HC Conference*, 2013.

# Thank you!

**Wang Yan**
**wangy0129@gmail.com**