BLUEHAT
SHANGHAI 2019

Exploring The Maze Of Memory Layouts Towards Exploits

Yan Wang

PHD, IIE, CAS

Chao Zhang

Associate Professor, Tsinghua University

2019-5

关于我们

张超，清华大学 副教授

Experience

- PKU，UC Berkeley，THU

Hack for fun

- 2012，Microsoft BlueHat Prize Contest
- 2014-2016, DARPA CGC
- 2015-2017，DEFCON CTF

Awards/Honors

- 2018 MIT TR35 China

王琰，信工所 博士

Experience

- 绿盟，信工所（导师：邹维）

Focus

- 网络设备漏洞分析，漏洞利用自动化生成

Hack for fun

- NeSE 战队队长
- XCTF 2016个人年度积分第11名
- XCTF Final 2017
- RHG 2017

Motivation

漏洞利用 & 内存布局

- 特定的内存布局是漏洞利用成功的必要条件

➔Before executing toJSON

evil_object	1	1	1	1	1	1	1

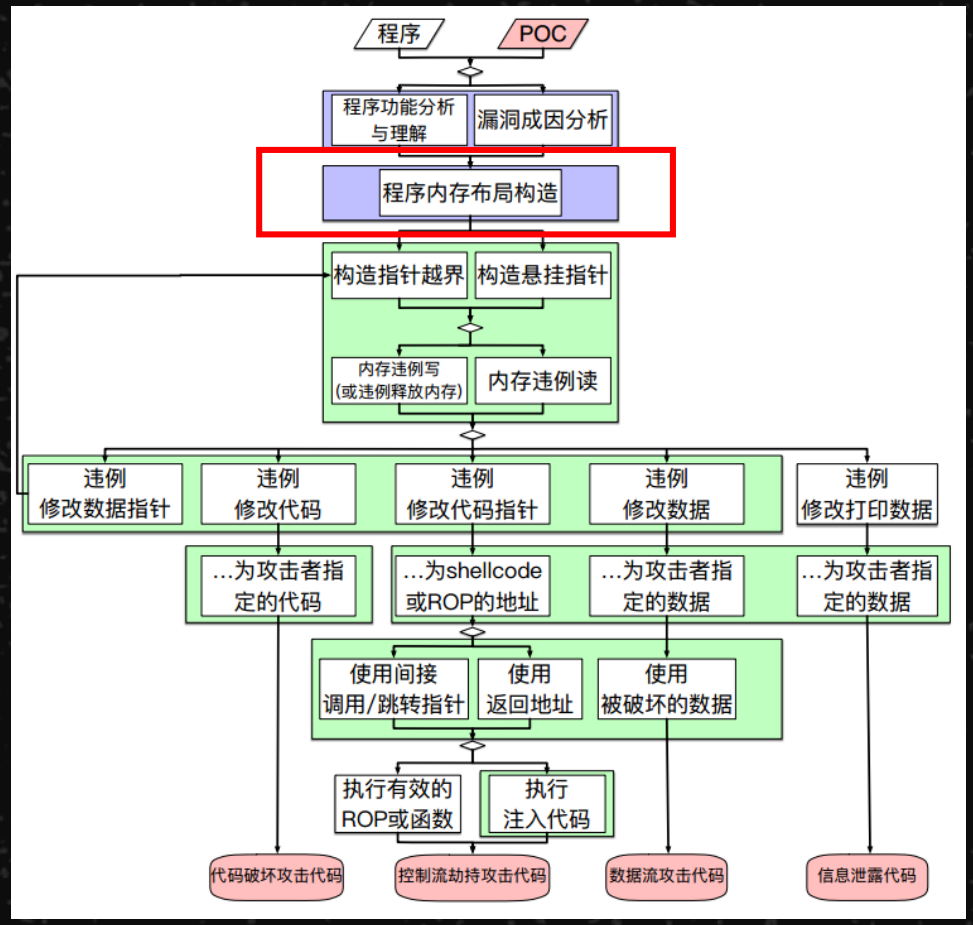
➔After executing toJSON, set R = random value

evil_object	R	map	properties	elements	byteLength	BackingStore	R

➔SerializeElement(BackingStore) BackingStore is even, leak the the point

```
0:024> dd 11110000
11110000  562853c4 063b75c0 00000000 00010005
11110010  00000033 00000000 11110024 11110024
11110020  05a25ae0 00000000 00000033 00000033
11110030  00000000 0c0c0c0c 0c0c0c0c 0c0c0c0c
11110040  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
11110050  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
11110060  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
11110070  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c
```

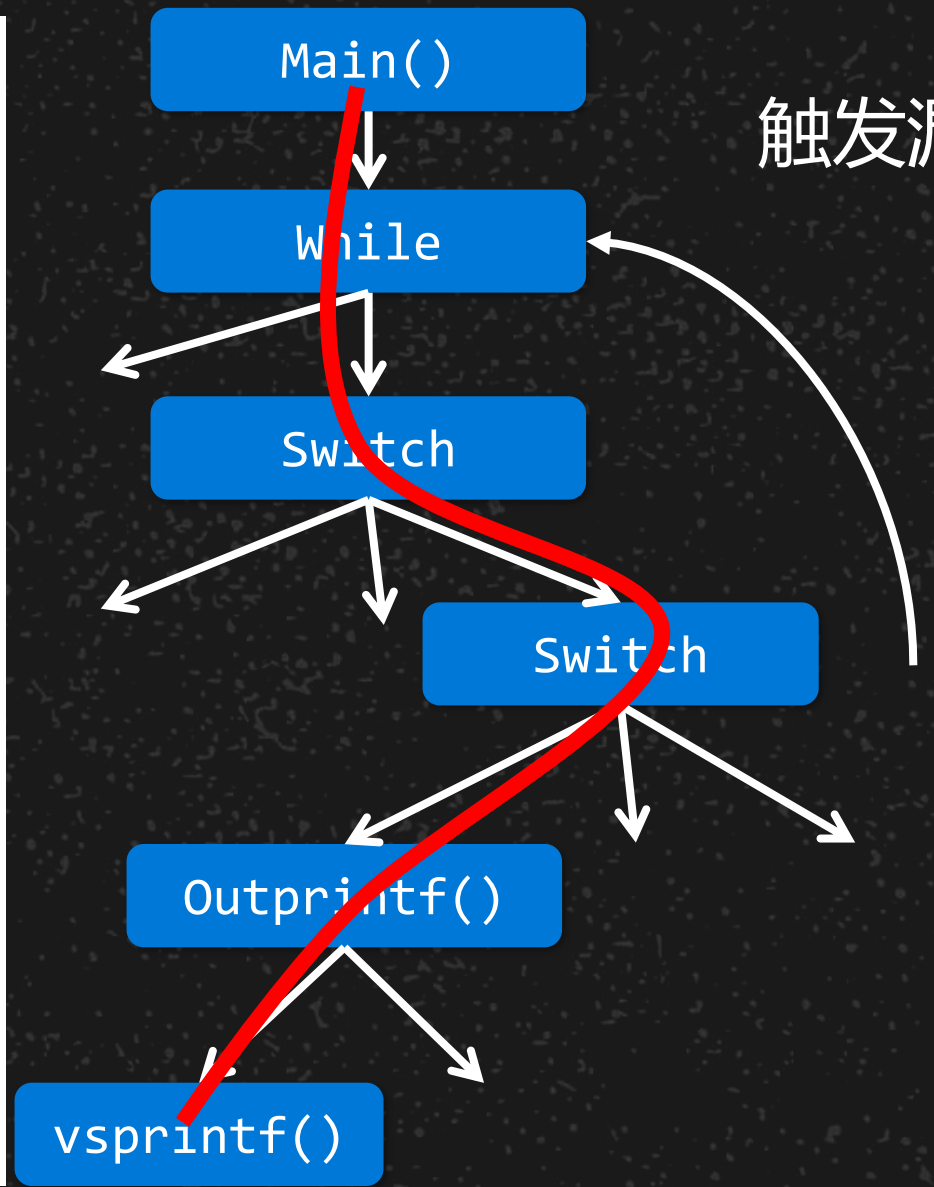
- 构造可利用的内存布局是漏洞利用流程中必不可少的一部分。



漏洞示例：CVE-2009-4270

```
int outprintf( const char *fmt, … )
{
  int count; char buf[1024]; va_list args;
  va_start( args, fmt );
  count = vsprintf( buf, fmt, args );
  outwrite( buf, count ); // print out
}

int main( int argc, char* argv[] )
{
  const char *arg;
  while( (arg = *argv++) != 0 ) {
    switch ( arg[0] ) {
    case '-': {
      switch ( arg[1] ) {
      case 0:
      …
      default:
        outprintf( "unknown switch %s\n", arg[1] );
      }
    }
    default: …
    }
  …
```



触发漏洞的要素：

- 路径约束
- 漏洞约束

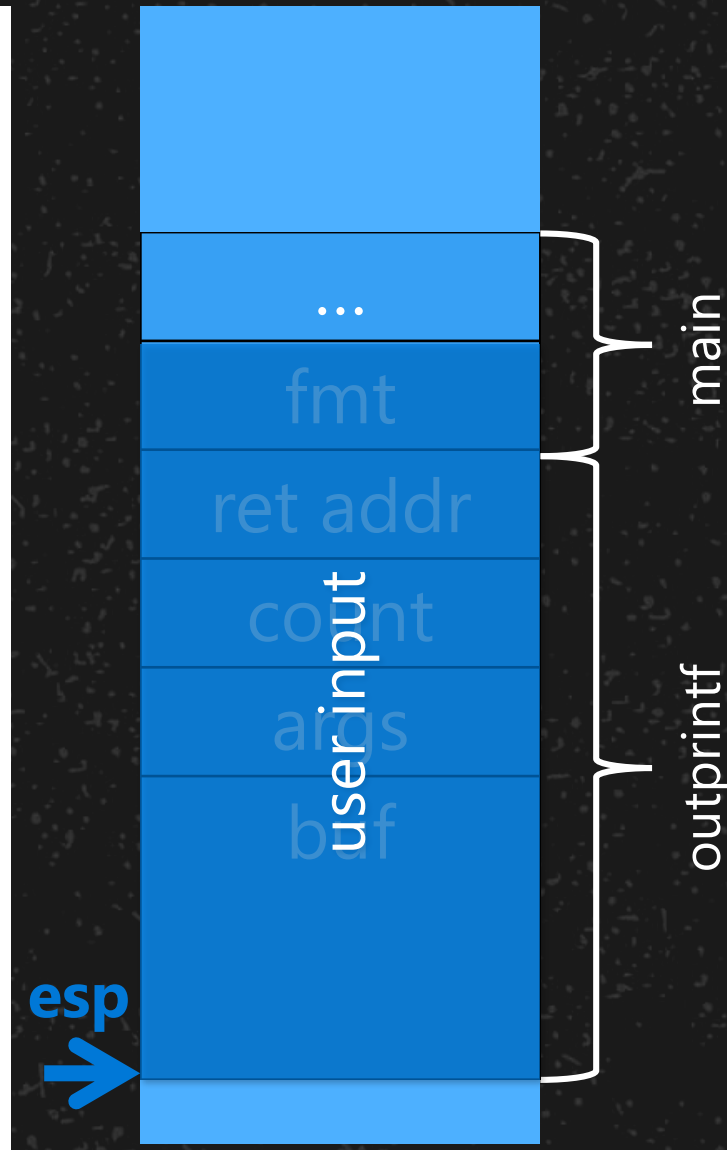
解决方案：

- 符号执行
- 模糊测试

漏洞利用

```
int outprintf( const char *fmt, … )
{
  int count; char buf[1024]; va_list args;
  va_start( args, fmt );
  count = vsprintf( buf, fmt, args );
  outwrite( buf, count ); // print out
}

int main( int argc, char* argv[] )
{
  const char *arg;
  while( (arg = *argv++) != 0 ) {
    switch ( arg[0] ) {
    case '-': {
      switch ( arg[1] ) {
      case 0:
      …
      default:
        outprintf( "unknown switch %s\n", arg[1] );
      }
    }
    default: …
    }
  …
```



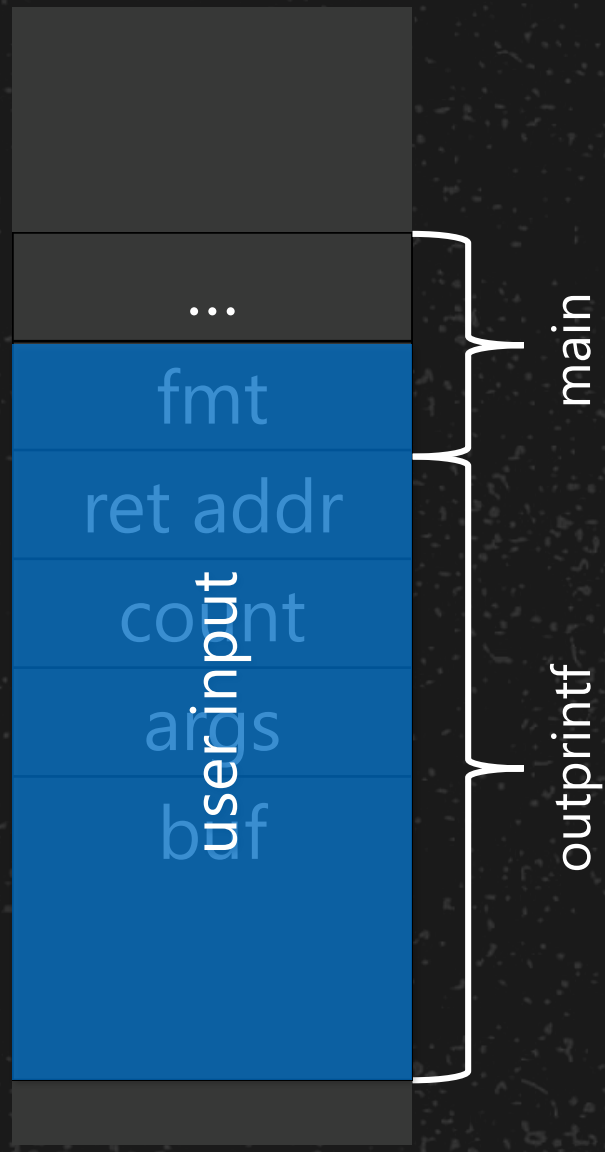
漏洞利用的要素：

- 触发漏洞
  - 路径约束
  - 漏洞约束
- 利用漏洞
  - Shellcode约束
  - EIP约束
  - 内存布局约束
  - 防御绕过约束

解决方案：

- 符号执行
- 模糊测试

AEG (Automated Exploit Generation)



动态分析

PoC

漏洞定位

崩溃现场分析

EIP可控性

Shellcode 填充

搜索ROP gadgets

符号分析

路径约束

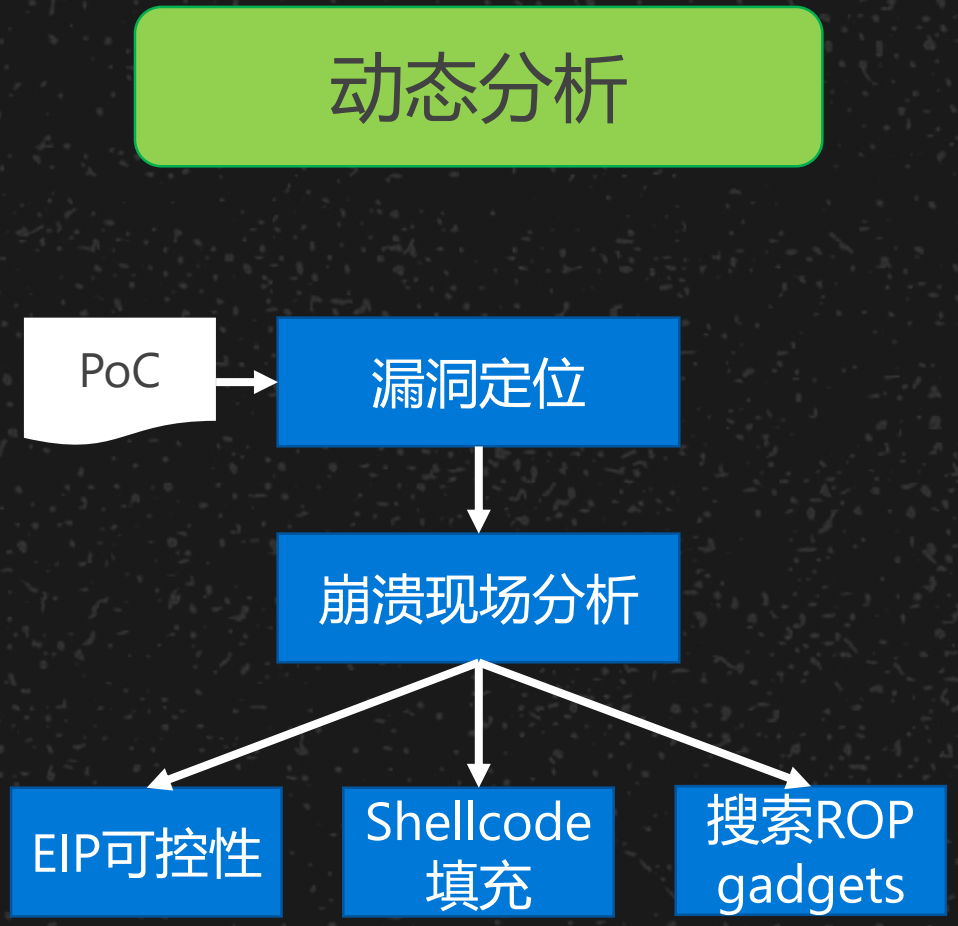
漏洞约束

EIP约束 (shellcode/ROP)

shellcode约束

SMT solver

AEG 挑战：无法构造堆上的内存布局



漏洞利用生成的成功率较低

只能当PoC的内存布局“恰好”可利用时，才能成功生成漏洞利用。

无法适用于堆相关漏洞

UAF

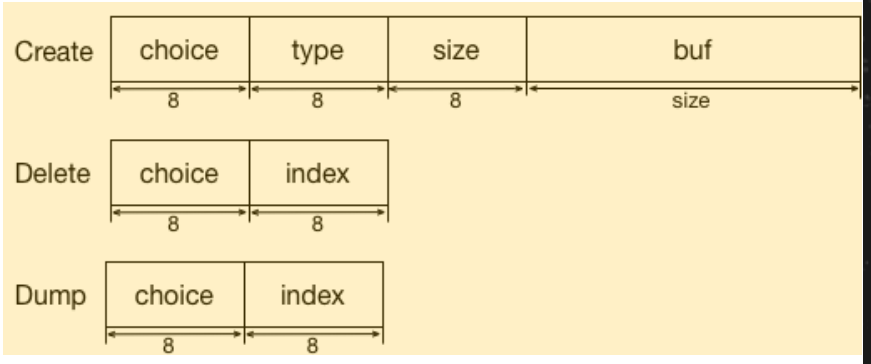
堆溢出...

Null byte offset

Example

示例代码

```
1  Router glist[32];
2  int count = 0;
3  typedef struct {
4     int (*func_ptr)();
5     Router *connect;
6     char *name;
7  } Router;
8
9    void main(void){
10       while(1){                    //功能调度
11          int c = read_num();
12          switch(c){
13          case 1: Create_Switch();     //功能单元
14          case 2: Create_Router();     //功能单元
15          case 3: Create_Terminal();   //功能单元
16          case 4: Delete();          //功能单元
17          case 5: Dump();            //功能单元
18          }
19       }
20   }
```

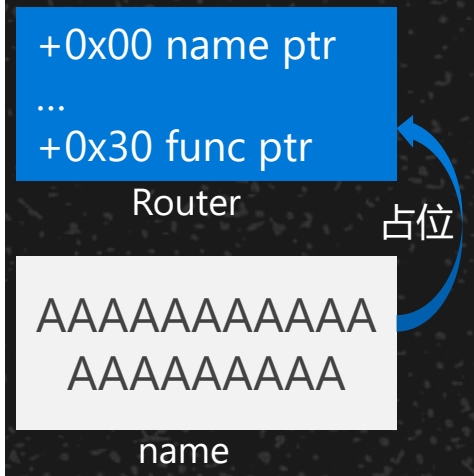


程序特征

- 交互式程序，根据用户的输入选择对应的功能。
- 存在一个UAF漏洞：Delete() 功能删除一个对象后没有清除指向这个对象的指针
- 程序存在一个输入协议。

How to Exploit ?

```
00  Router Create(){
01     int size = read_num();
02     ...
03     Router *router = malloc(0x18);
04     router->name = malloc(size);
05     ...
06     router->func_ptr=show;
07     glist[count++]=router;
08     ...
09  }
10
11  void Delete(){
12     ...
13     free(router->name);
14     free(router);
15     ...
16  }
17
18  void Dump(){
19     ...
20     int index = read_num();
21     glist[index]->func_ptr(); //劫持控制流
22     ...
23  }
```



利用过程：

- 通过分析确定发生UAF的是Router结构，且Router结构中含有函数指针。
- 使用数据可控的router->name占位释放后的Router结构。并通过可控数据设置函数指针。

自动化构造可利用的内存布局

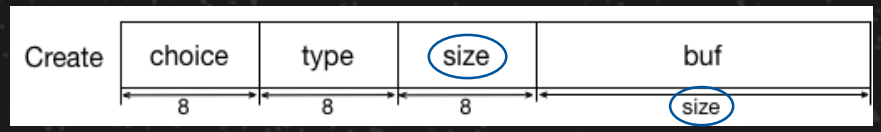
- 通过Dump功能调用函数指针，跳转到攻击者指定的位置执行代码。

可利用内存布局的自动化构造

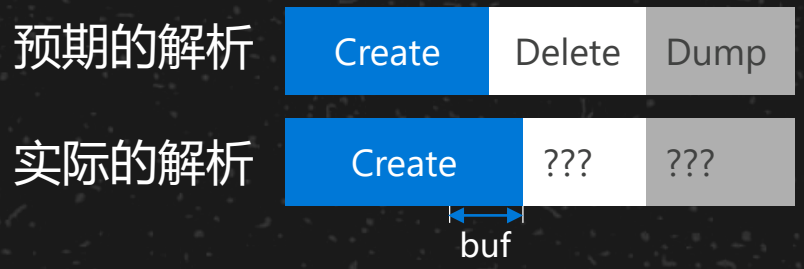
● 内存布局的自动化操控

➢ 通过程序输入间接操控内存布局？

□程序输入的自动化解析难题



- Create 功能对应的输入单元中，读入的buf长度由size指定。
- 如果没有输入协议，则会造成解析“歧义”：



buf长度和size字段不符合，如len (buf)<size时，会导致Delete中的choic字段被归入到buf中，导致Delete解析失败，并导致后续的功能解析失败。

内存布局的自动化转换

➢随机遍历（Random Search）？

□路径空间爆炸

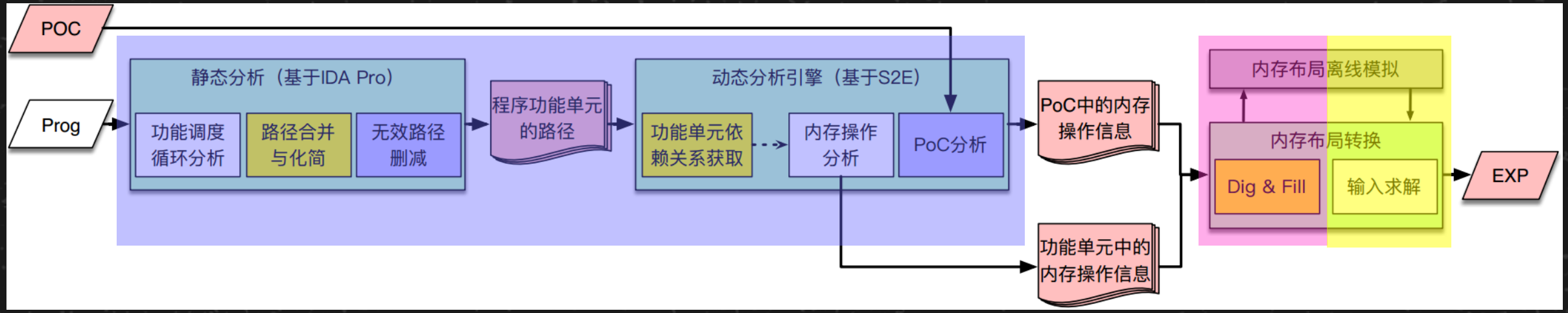
□大量的无效耗时

➢生成一条完整的内存布局转换路径？

□ 程序路径中的内存布局操作“噪音”问题

Our Solution——Maze

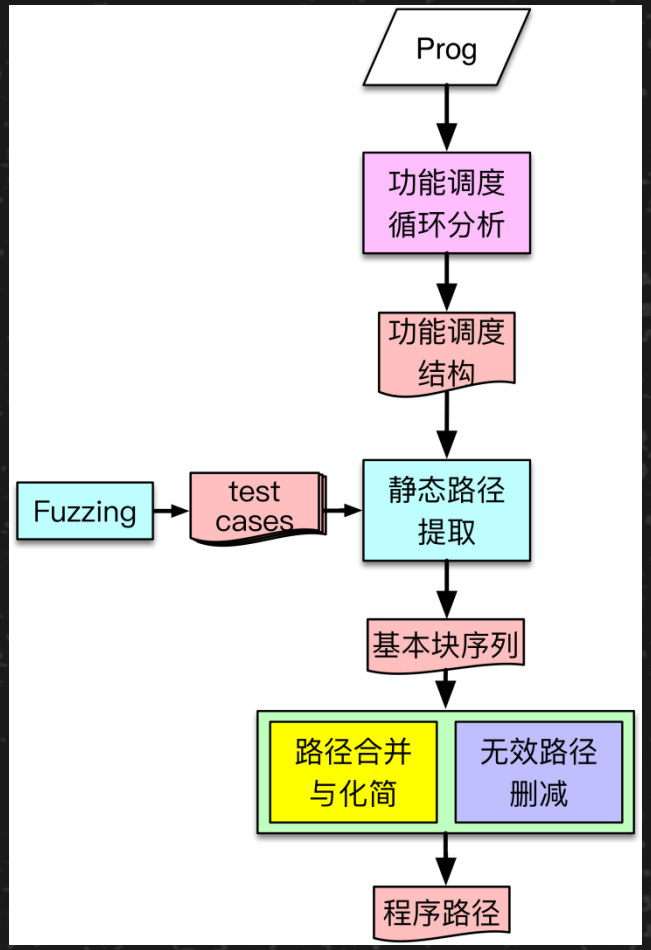
Overview of Maze



- 分析程序路径的内存操作（包括side effects）
  - 静态分析、动态分析
- 组合程序路径，实现目标内存布局
  - 通过Dig & Fill技术定向生成内存布局转换路径
- 求解输入，满足目标程序路径组合
  - 符号执行、约束求解

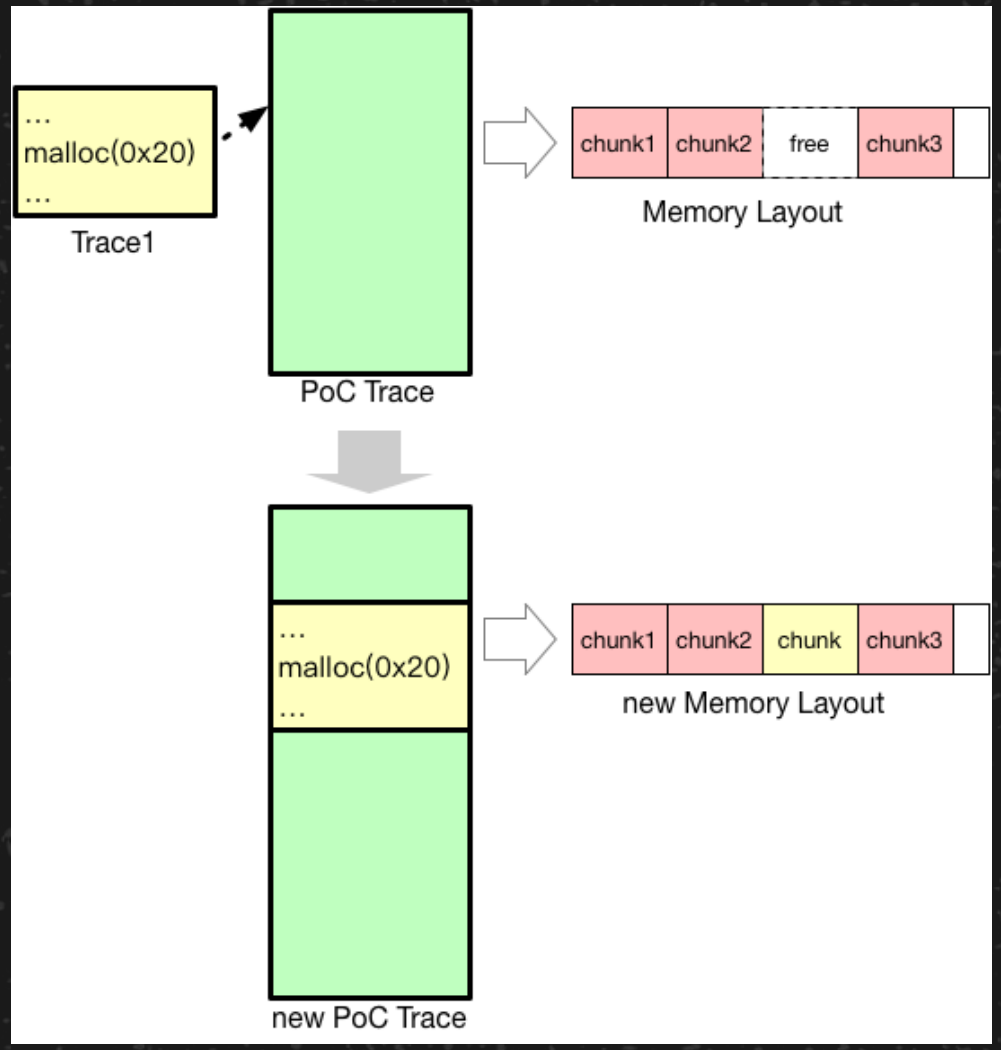
程序路径的提取（静态分析）

- Maze通过静态分析提取程序中的功能单元路径。Maze通过组合这些路径操控程序内存布局。



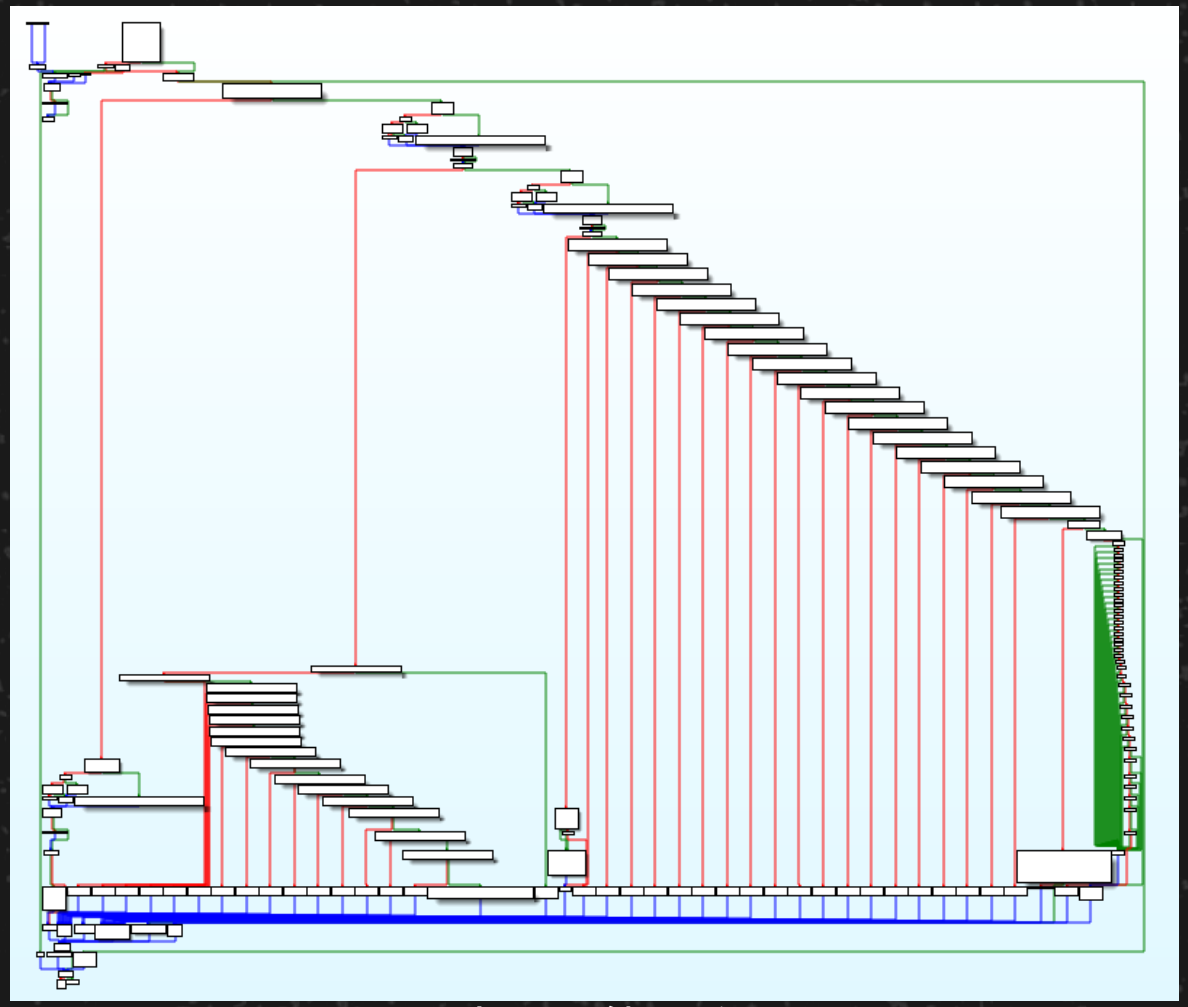
- 通过分析程序的CFG图，定位程序中的功能调度结构。
- 通过分析Fuzzing产生的测试用例和分析CFG图获取功能调度结构中的程序路径。
- 删除用于错误处理的无效路径分支。
- 符号化路径中的程序分支和循环。

通过修改程序路径构造新的内存布局



- 找到一条含有内存布局操作原语(malloc)的路径Trace1。
- 把这条路径插入到原PoC路径中，构造一条新的程序路径。
- 新构造的程序路径产生了新的内存布局。

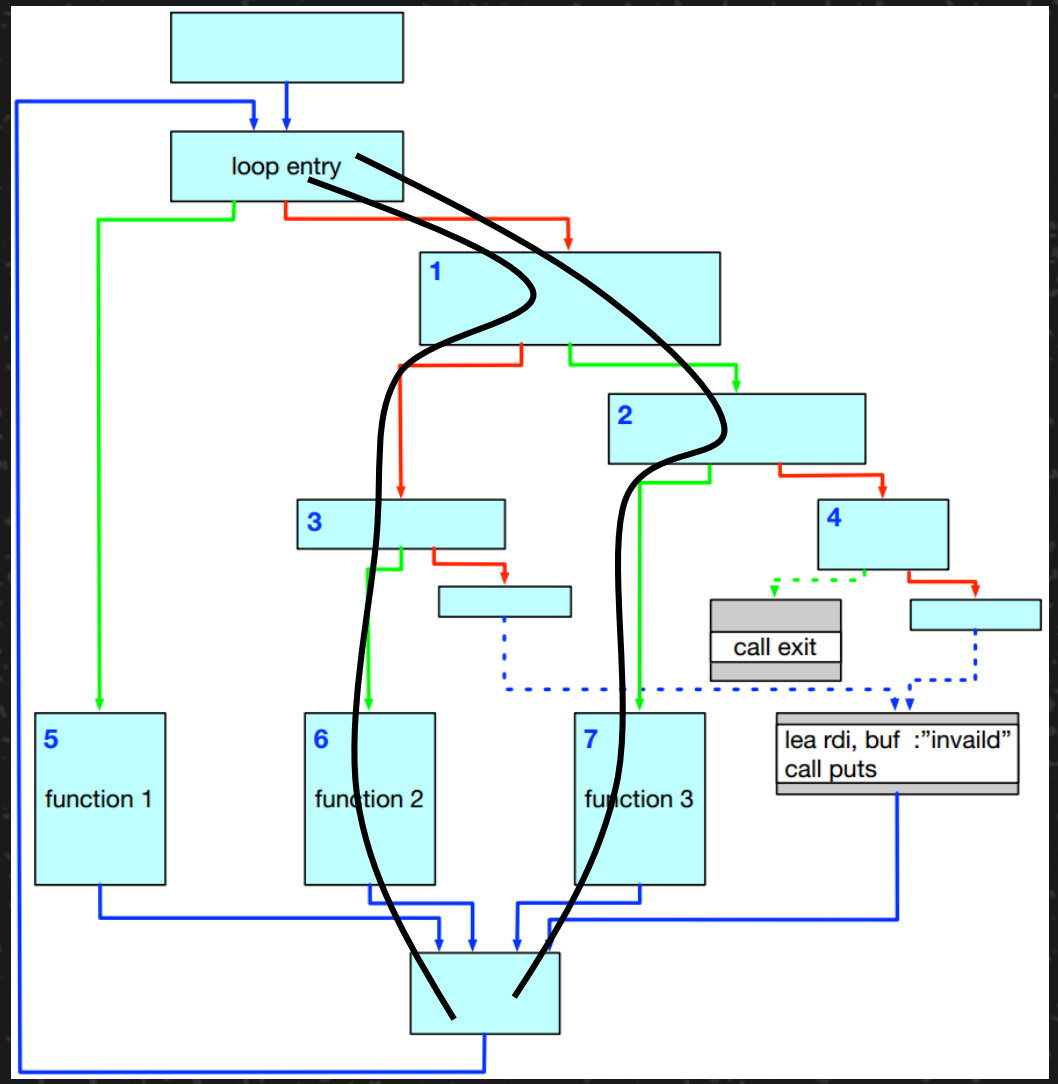
程序功能调度结构(Function Dispatcher)



一个解释器中的功能调度结构

- 功能调度结构(Function Dispatcher)普遍存在于程序中，如浏览器（解释执行js脚本）、网络服务类程序（解析用户输入选择对应功能）、甚至恶意代码（command & control）。
- 功能调度一般存在于一个循环中，根据用户的输入选择具体的功能单元。功能单元相互之间独立性较高。
- Maze在PoC中定位功能调度结构，并通过插入、替换同一个功能调度中的功能单元来操控内存布局。

程序功能调度结构的识别



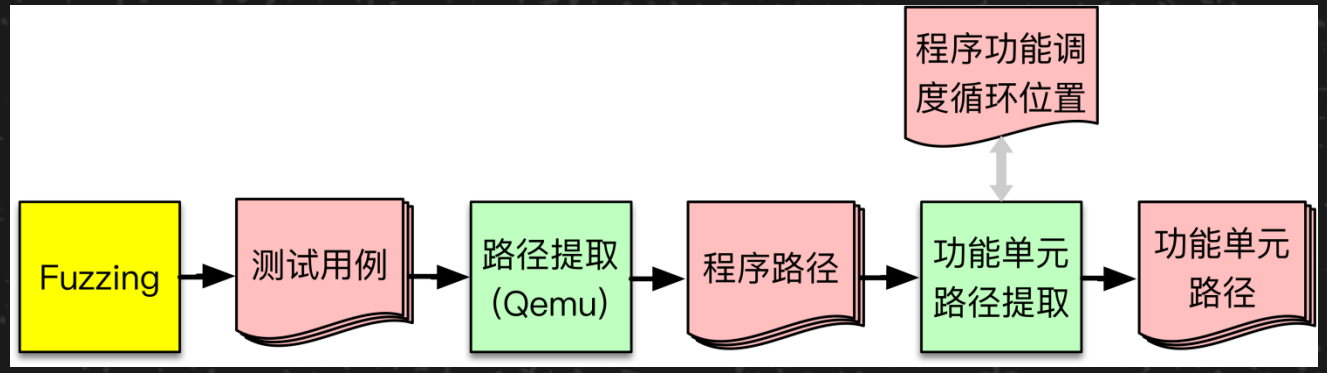
- **定位程序中的循环**
- **确定循环是否为功能调度结构：**
  - 循环中的基本块总数大于300
  - 循环中至少含有3条含有不同库函数调用序列的路径。
  - 循环中至少含有1条含有内存操作（malloc、realloc、free）的路径。
- **循环嵌套的处理：**
  - 识别不同循环之间的嵌套关系
  - 按照先内后外的顺序确定循环是否为功能调度结构

程序路径的获取

Maze通过Fuzzing和静态分析两种方法获取程序中的功能单元路径

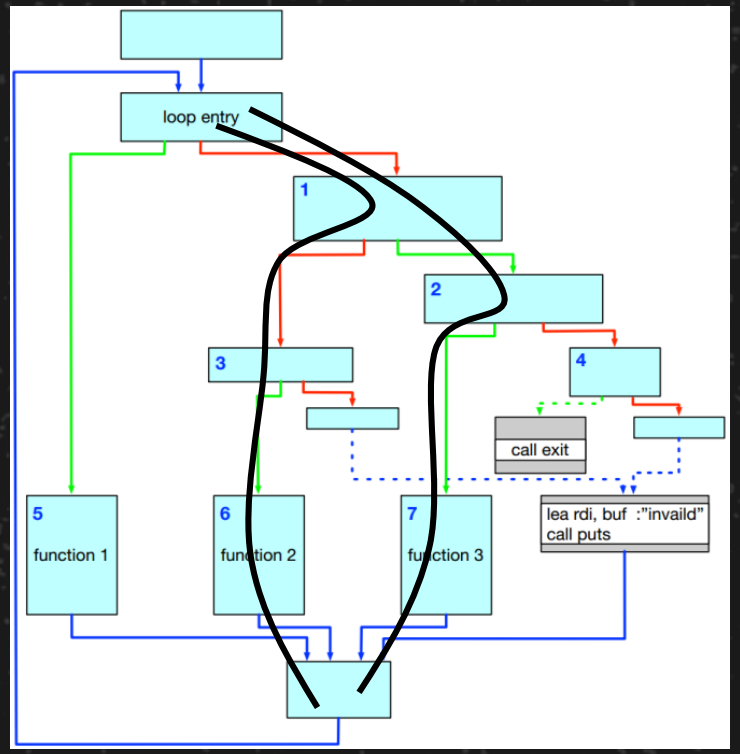
- 通过Fuzzing获取：

➢ 动态获取test case对应的程序路径，提取在功能调度循环中的功能单元路径。

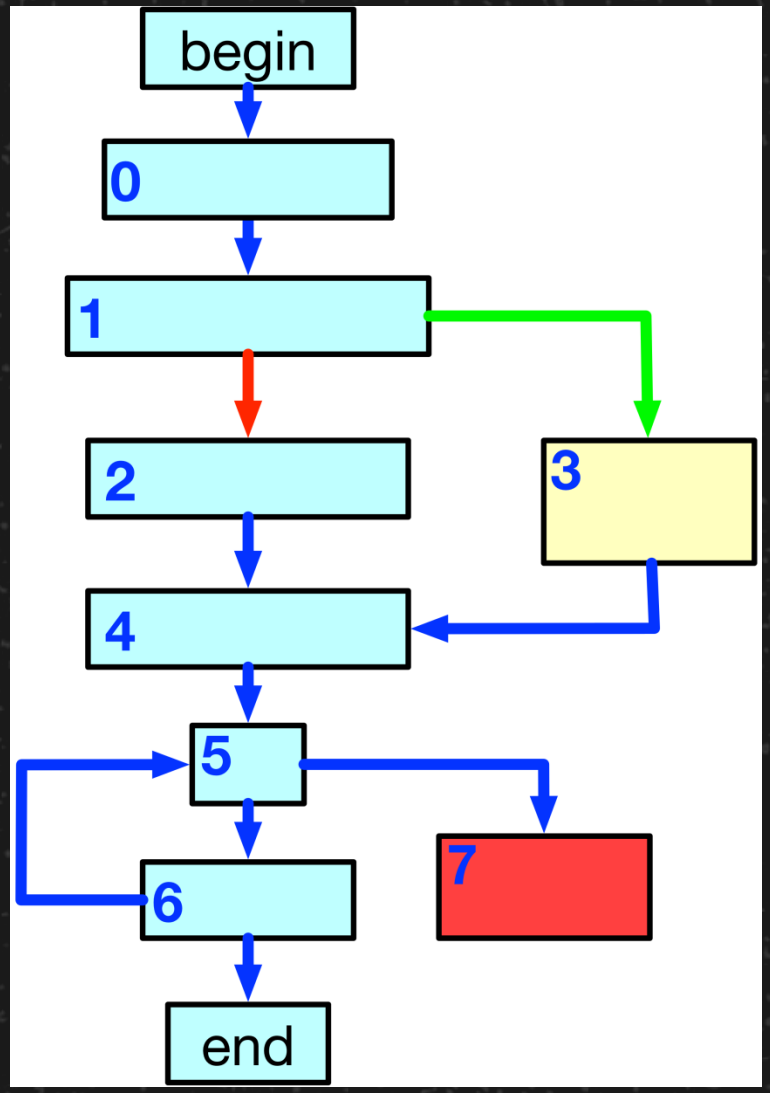


- 通过静态分析获取：

➢ 获取CFG图，基于图论的方法获取两点之间的路径。



路径的删减与合并(静态分析)



对于含有内存布局操作原语的路径，Maze会进一步删减及合并相似路径

- 路径删减：

基本块7为错误处理分支，因此Maze删除掉所有含有7的程序路径。

- 分支合并：

0->1->2->4->5->6
0->1->3->4->5->6
Concrete path

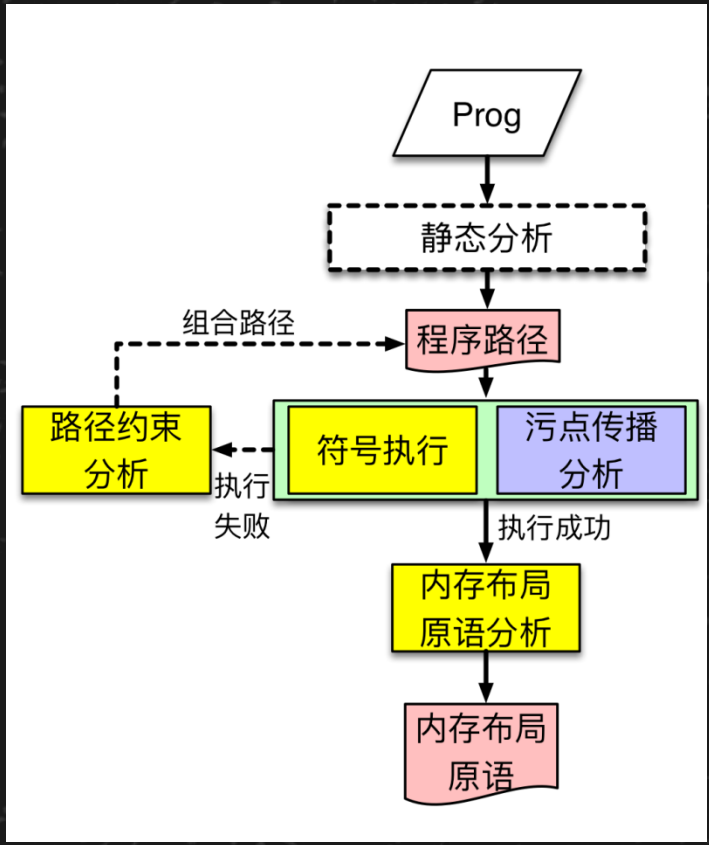
0->1->B->4->5->6
Path Symbolization

- 循环合并：

0->1->B->4->5->6 ==> 0->1->B->4->L

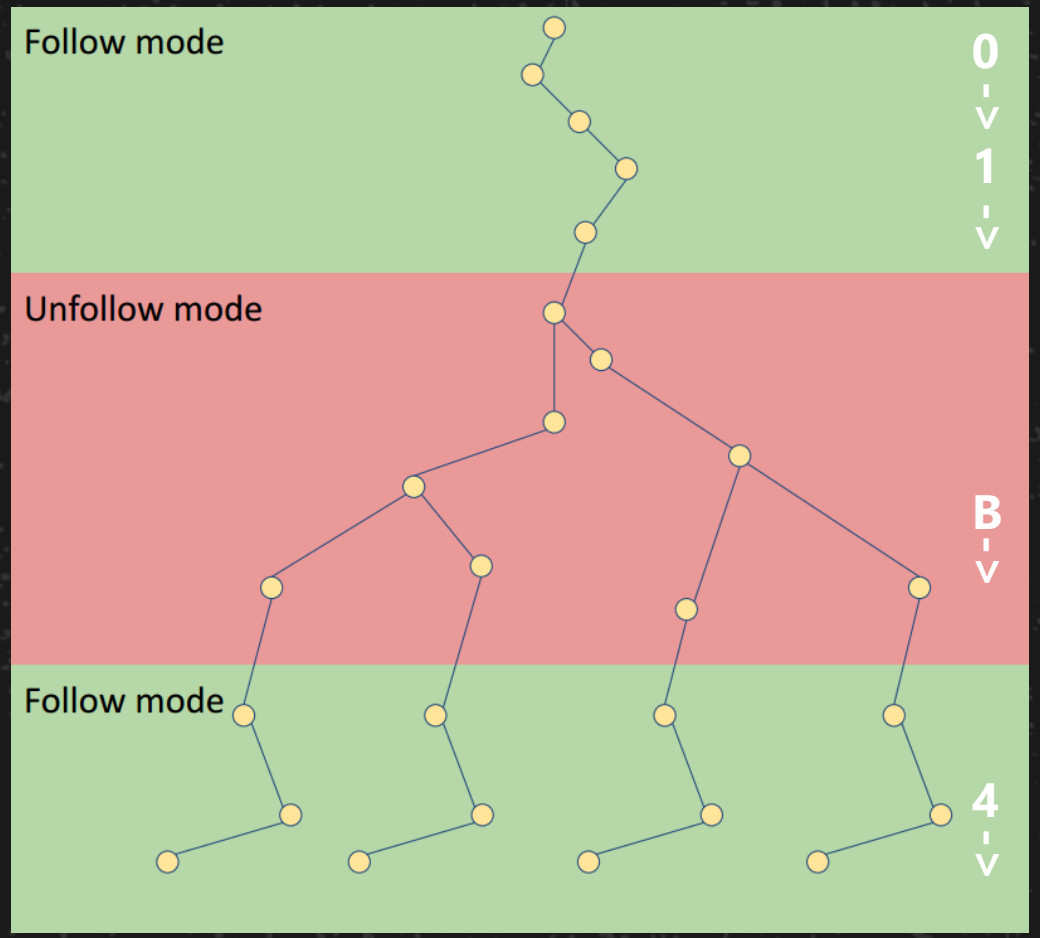
内存布局操控原语分析（动态分析）

- Maze通过动态分析获取程序路径中的内存布局原语属性。基于内存布局原语属性信息，Maze可以实现内存布局的精确操控。



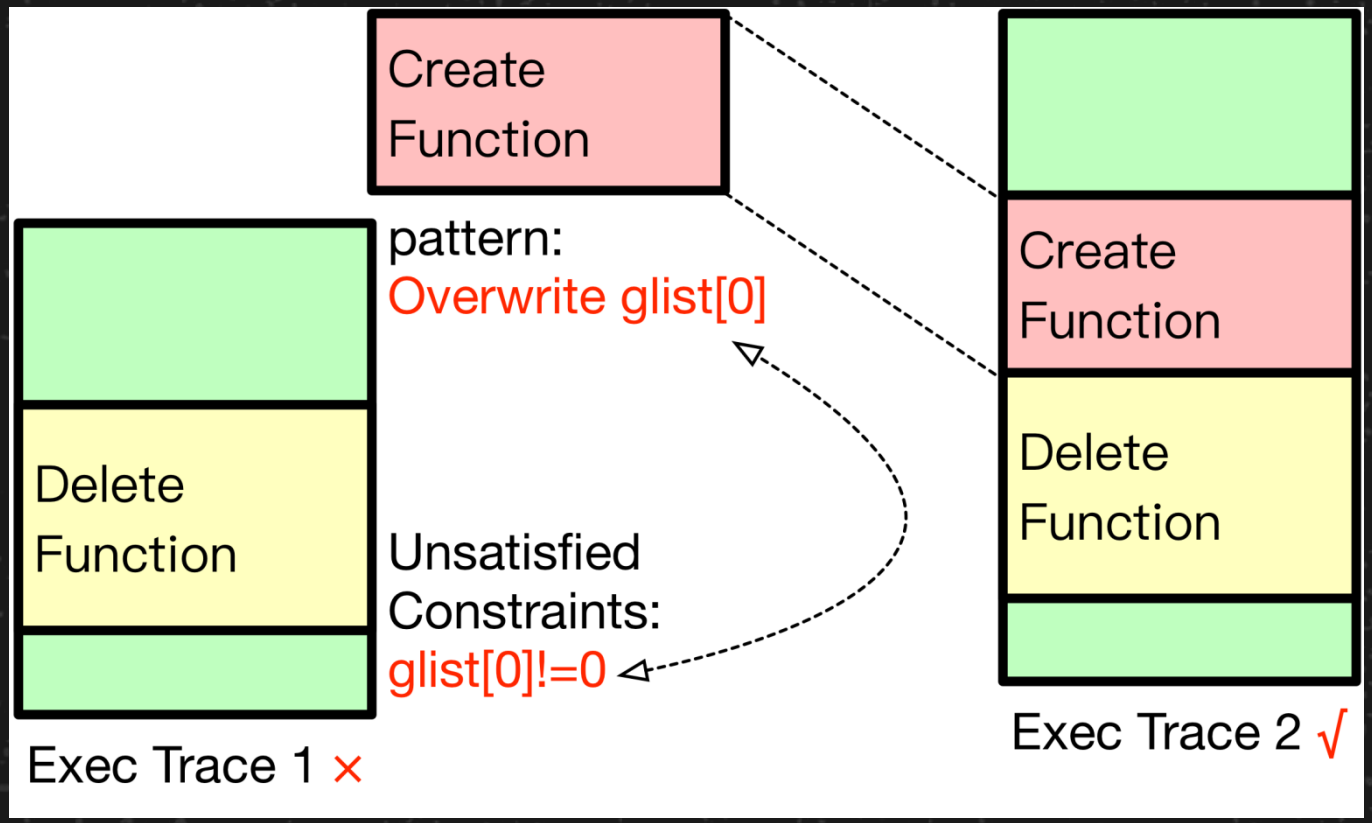
- Maze基于S2E平台分析静态分析中提取的程序路径。
- 对于动态执行失败的路径，Maze分析造成失败的约束条件，并组合路径以满足约束，并重新动态分析。
- Maze通过动态分析提取内存布局操作原语的属性信息。

程序路径的动态分析



- **Maze基于s2e平台，使用符号执行和污点传播动态分析功能单元路径中的内存布局操作原语。**
- **0->1->B->4->L：**
  - 对于concrete的基本块地址，如0、1、4，Maze会进入Follow模式，即按照基本块序列进行剪枝。
  - 对于标记为symbolic的分支和循环，即B、L，则进入Unfollow模式，只要找到一条到达下一个指定基本块的路径即可。

路径约束条件分析



- Maze通过符号执行动态分析功能单元路径，但由于约束不满足而执行失败。
- Maze分析执行失败的原因，并计算出满足约束的数据约束。
- Maze寻找可使数据约束满足的其他功能单元。
- 组合这些功能单元重新符号执行。

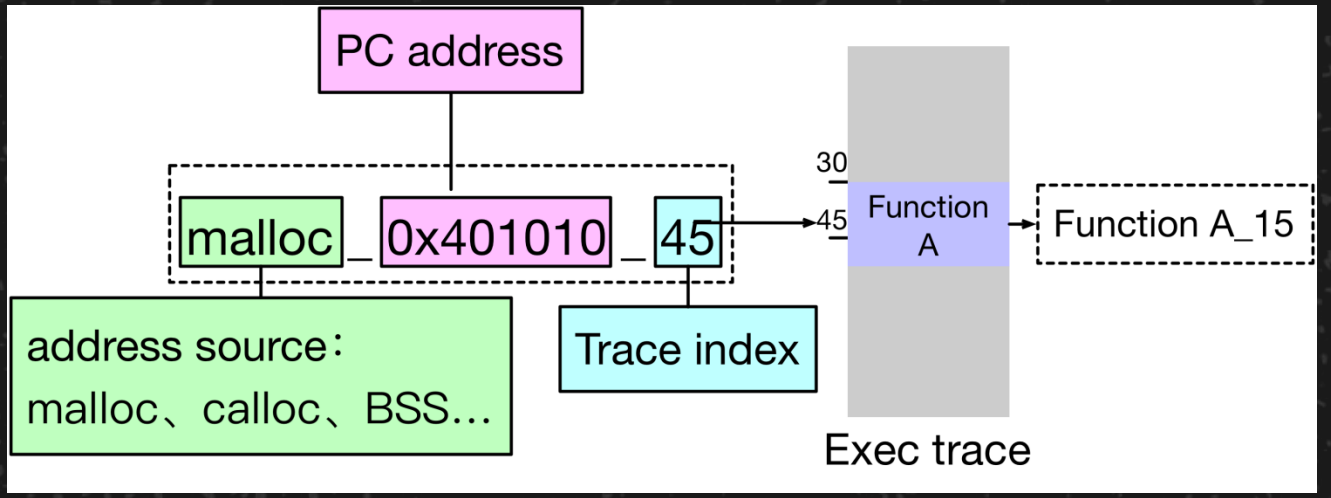
内存布局操作原语分析

内存申请

- Size： 内存申请的长度取值范围
- Addr： 申请到的内存地址污点标签
- Index： 内存申请函数调用在程序执行路径中的位置

内存释放

- Addr： 要释放的内存地址污点标签
- Index： 内存释放函数调用在程序执行路径中的位置

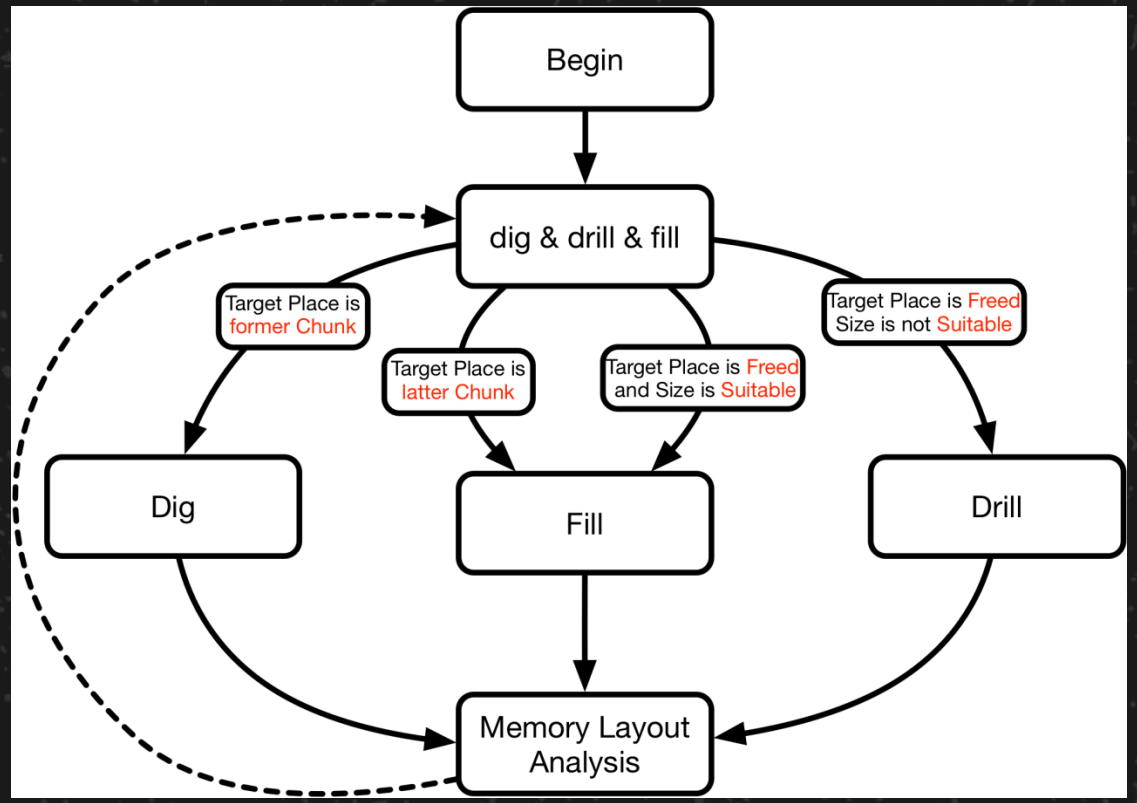


污点标签：

- Maze根据每个内存地址的来源生成污点标签。其中Trace_index为call malloc在当前指令执行序列中的序号。
- 通过Trace_index可以计算得出call malloc所属的功能单元路径及在路径中的偏移。如图中的Function A_15，这个信息是堆块的“类型标识符”

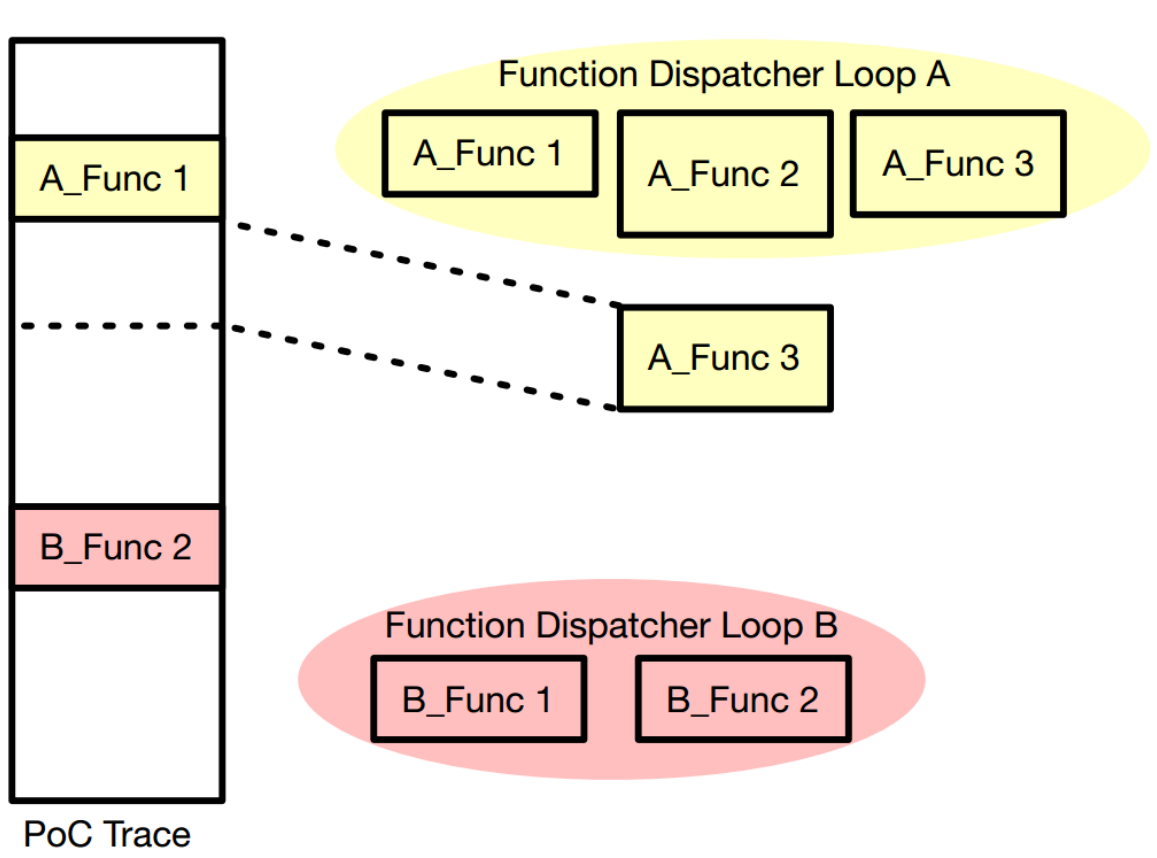
目标内存布局的构造

Maze通过组合程序路径，使用Dig & Fill技术定向构造目标内存布局。



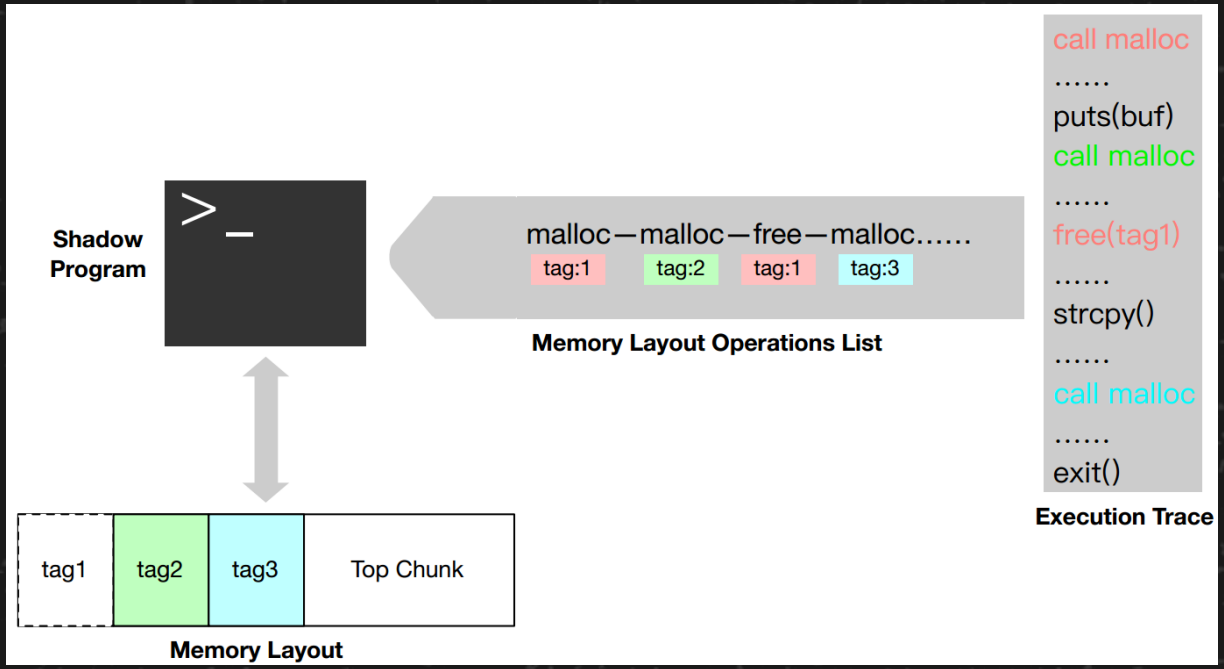
- Maze通过组合程序功能单元路径来操控内存布局。
- Maze通过离线程序模拟的方式获取内存布局状态。
- Maze基于当前内存布局状态，使用Dig & Fill技术定向构造目标内存布局。

程序路径的组合



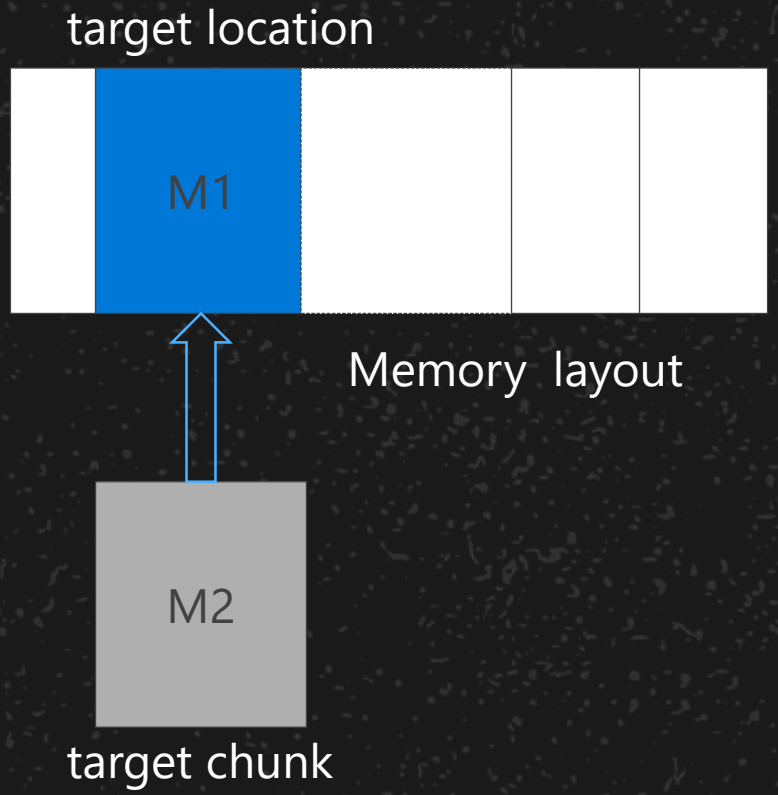
- 通过上述方法已定位功能调度结构及其中的功能单元路径，并通过动态分析获取了路径中的内存布局操作原语。
- 在PoC路径的功能调度结构中，插入含有需要内存布局操作原语的功能单元路径。
- 路径编辑完成后，更新路径中污点标签的Trace index字段。

内存布局状态的获取



- Maze 通过一个影子程序离线模拟编辑后的路径中的内存布局操作，然后获取当前路径对应的内存布局状态。
  - 从路径中获得一个诸如 malloc … free … malloc 的序列。
  - 根据内存地址中的污点标签，确定内存申请和释放对应的关系。
  - 传递给离线模拟程序的内存操作
  - 获取离线程序的内存状态。

目标内存布局转换路径生成



- **Dig & Fill**

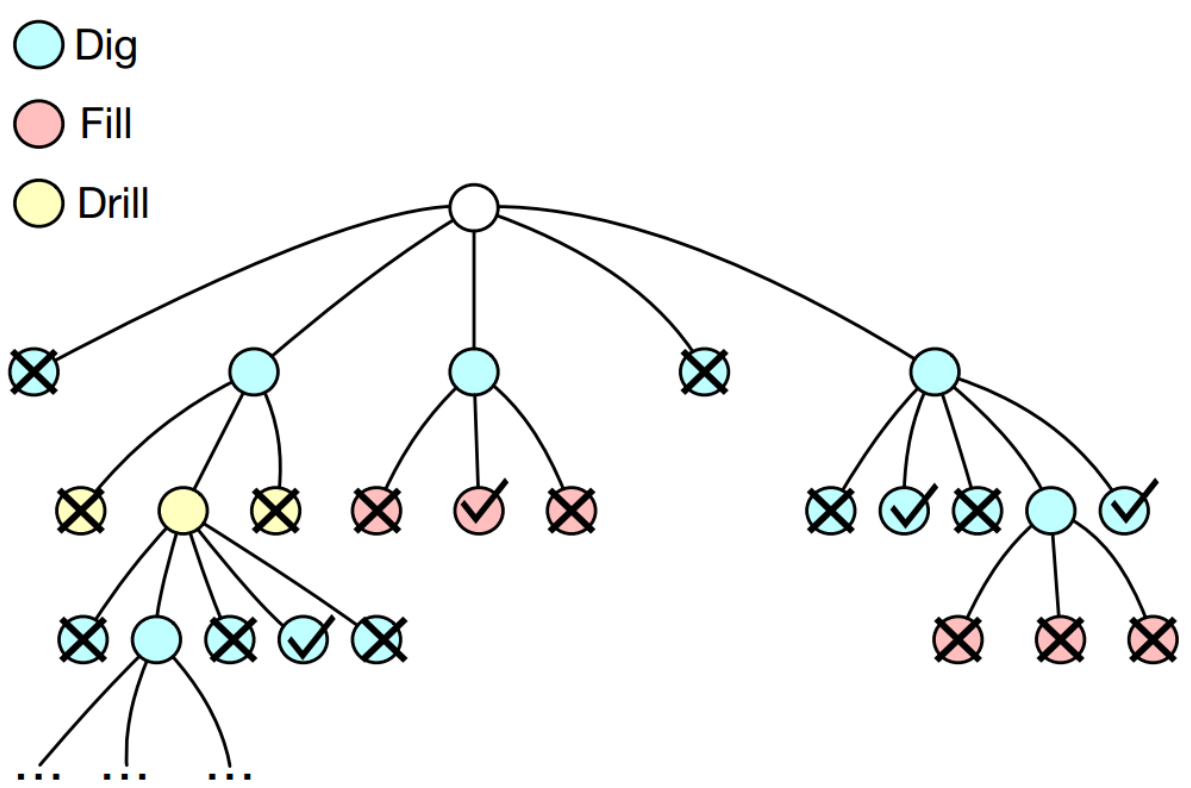
➢ **根据目标堆块和目标内存位置的状态，决定下一步操作。**

➢ **递归进行，直到内存布局逼近目标内存布局。**

➢ **算法：**

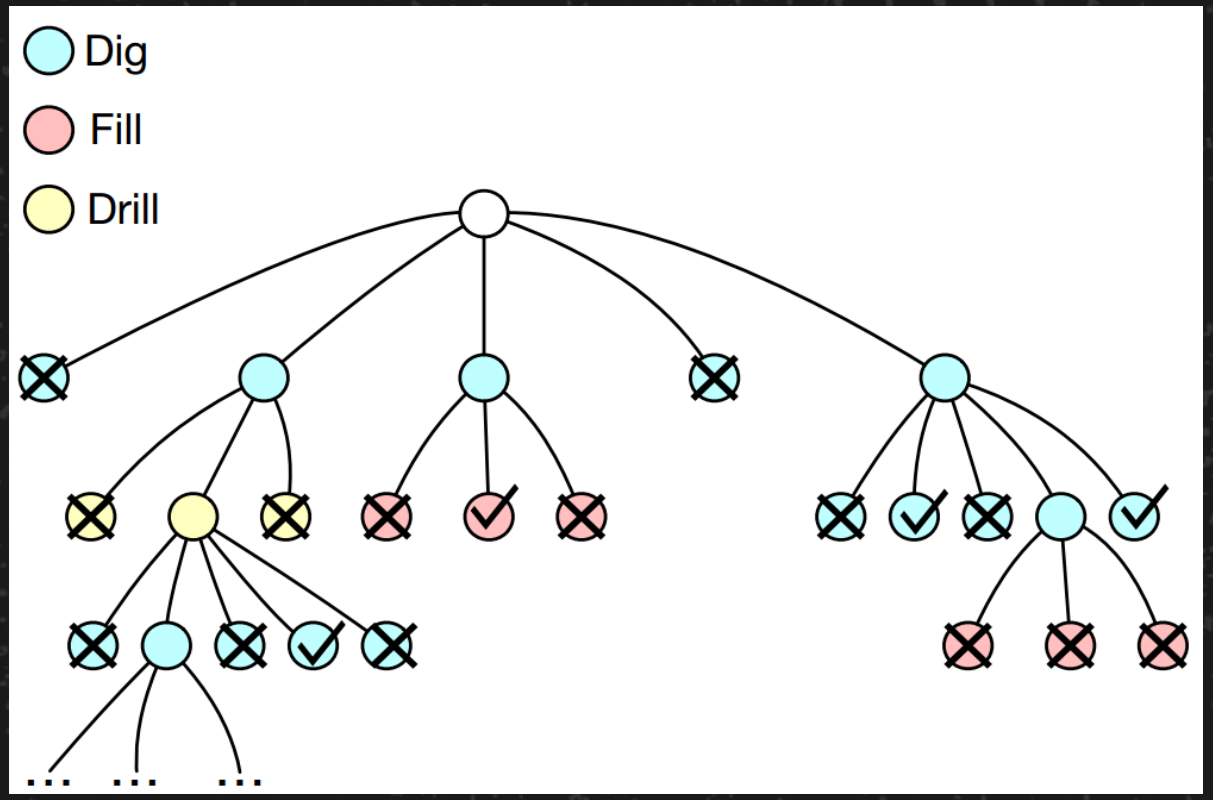
```
if  M1.status == Busy and M1.alloc_index == M2.alloc_index:
    Win()
else if M1.status==Busy and M1.alloc_index < M2.alloc_index:
    Dig()
else if M1.status==Free or M1.alloc_index > M2.alloc_index:
    Fill()
else if M1.status==Free and M1.size is not suitable:
    Drill()
```

内存布局转换路径生成——Dig



- **Dig：增加合适(suitable)的空闲堆块数目**
  - 1 更改释放堆块的长度，增加合适的空闲堆块数目
  - 2 更改内存申请的长度，减少占位的内存申请数目
  - 3 插入释放操作，释放合适的堆块
  - 4 插入内存申请操作，插入释放操作，构造一个合适的空闲堆块

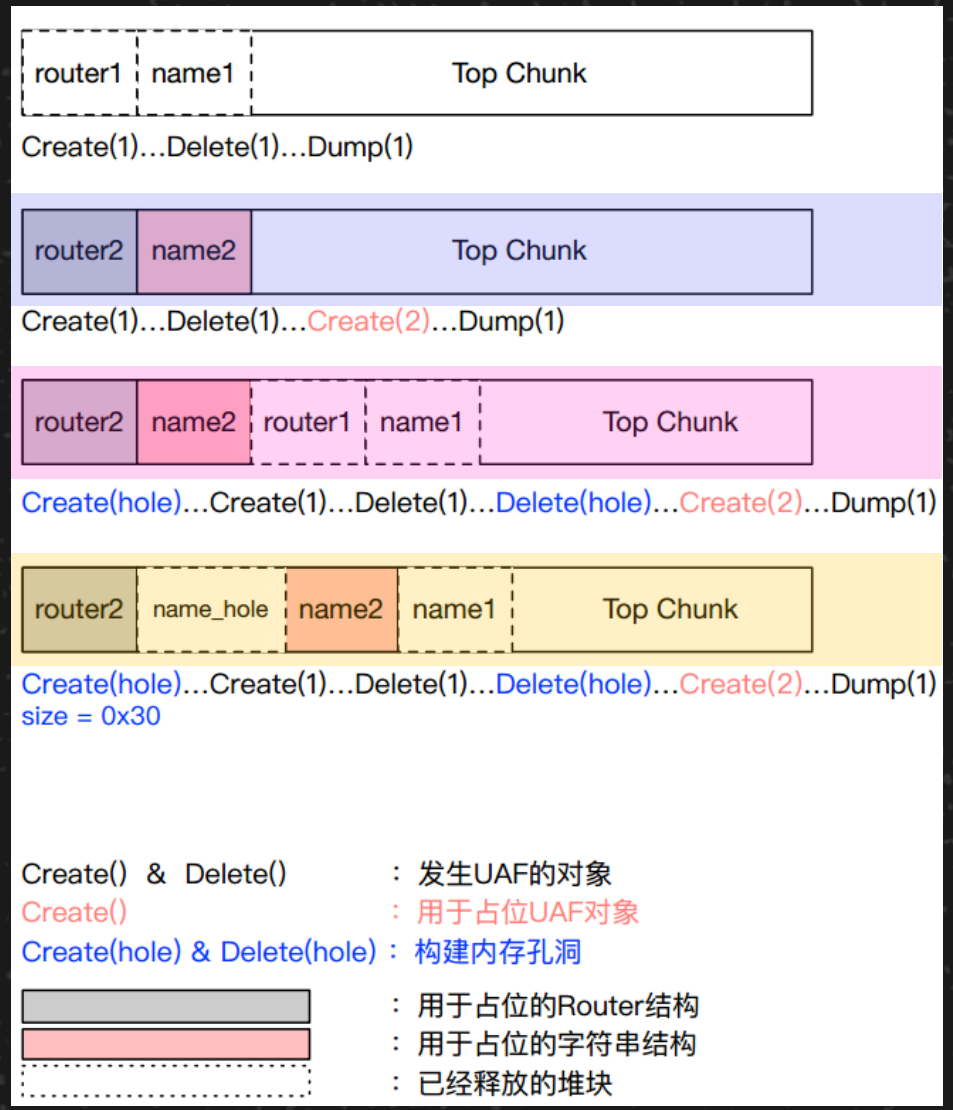
内存布局转换路径生成——Fill，Drill



- **Fill：减少合适的空闲堆块数目**
  - 1 更改释放堆块的长度
  - 2 更改内存申请的长度，减少合适的空闲堆块数目
  - 3 插入申请的操作

- **Drill：使目标堆块变成一个合适的堆块**
  - 1 更改释放堆块的长度
  - 2 构造内存分割
  - 3 构造内存融合

Example：程序路径中内存操作原语“噪音“的处理



- Create功能中含有两个malloc操作。多余的malloc为“噪音”
  - M1：router1，M2：name2
  - M1.status == busy, M1.alloc_index(router2) < name2 → dig(6)
  - M1.status == free → fill(1)
  - M1.status == busy , m1.alloc_index = name2 →win

触发漏洞利用原语

```
00  Router Create(){
01    int size = read_num();
02    ...
03    Router *router = malloc(0x18);
04    router->name = malloc(size);
05    ...
06    router->func_ptr=show;
07    glist[count++]=router;
08    ...
09  }
10
11  void Delete(){
12    ...
13    free(router->name);
14    free(router);
15    ...
16  }
17
18  void Dump(){
19    ...
20    int index = read_num();
21    glist[index]->func_ptr(); //劫持控制流
22    ...
23  }
```

- 通过污点分析获取程序中的敏感操作指令，如存储在堆上的函数指针、内存写指针。
- 构造完成内存布局后，在程序路径中插入含有函数指针或内存写指针解引用操作的功能单元路径。

漏洞利用的生成

约束求解

```
0x555555554cb4
0x555555554cc8
0x555555554d0f
0x555555554d1c
0x555555554d32
0x555555554a20
0x555555554d43
0x555555554d5a
0x5555555549f0
0x555555554d6b
0x555555554bc8
0x555555554c36
0xffffffffffffffff
0x555555554c41
0x555555554c63
0x555555554d7f
0x555555554db3
0xffffffffffffffff
0x555555554dbd
0x555555554dd0
0x5555555551a0
```

```
1 93824992234016 64 121
1 93824992234016 118 16
1 93824992234016 172 104
1 93824992234016 226 24
1 93824992234016 280 240
1 93824992234016 334 16
5 93824992235259 387 5 0 0
3 93824992233936 388 5
5 93824992235259 428 2 0 0
3 93824992233936 429 2
1 93824992234016 470 24
4 93824992234536 478 8 32 1 4000000000000000
4 93824992234536 478 8 33 1 0100000000000000
4 93824992234536 478 8 34 1 0000000000000000
4 93824992234536 478 8 35 1 0000000000000000
4 93824992234536 478 8 36 1 0000000000000000
4 93824992234536 478 8 37 1 0000000000000000
4 93824992234536 478 8 38 1 0000000000000000
4 93824992234536 478 8 39 1 0000000000000000
5 93824992235259 523 6 0 0
3 93824992233936 524 6
1 93824992234016 565 136
5 93824992235112 619 3 0 0
5 93824992235259 662 4 0 0
```

- 求解路径约束
- 求解内存布局原语序列约束
- 求解漏洞利用数据约束
- 必要的求解优化：
  - 内存地址符号化
  - 循环的处理

Demo

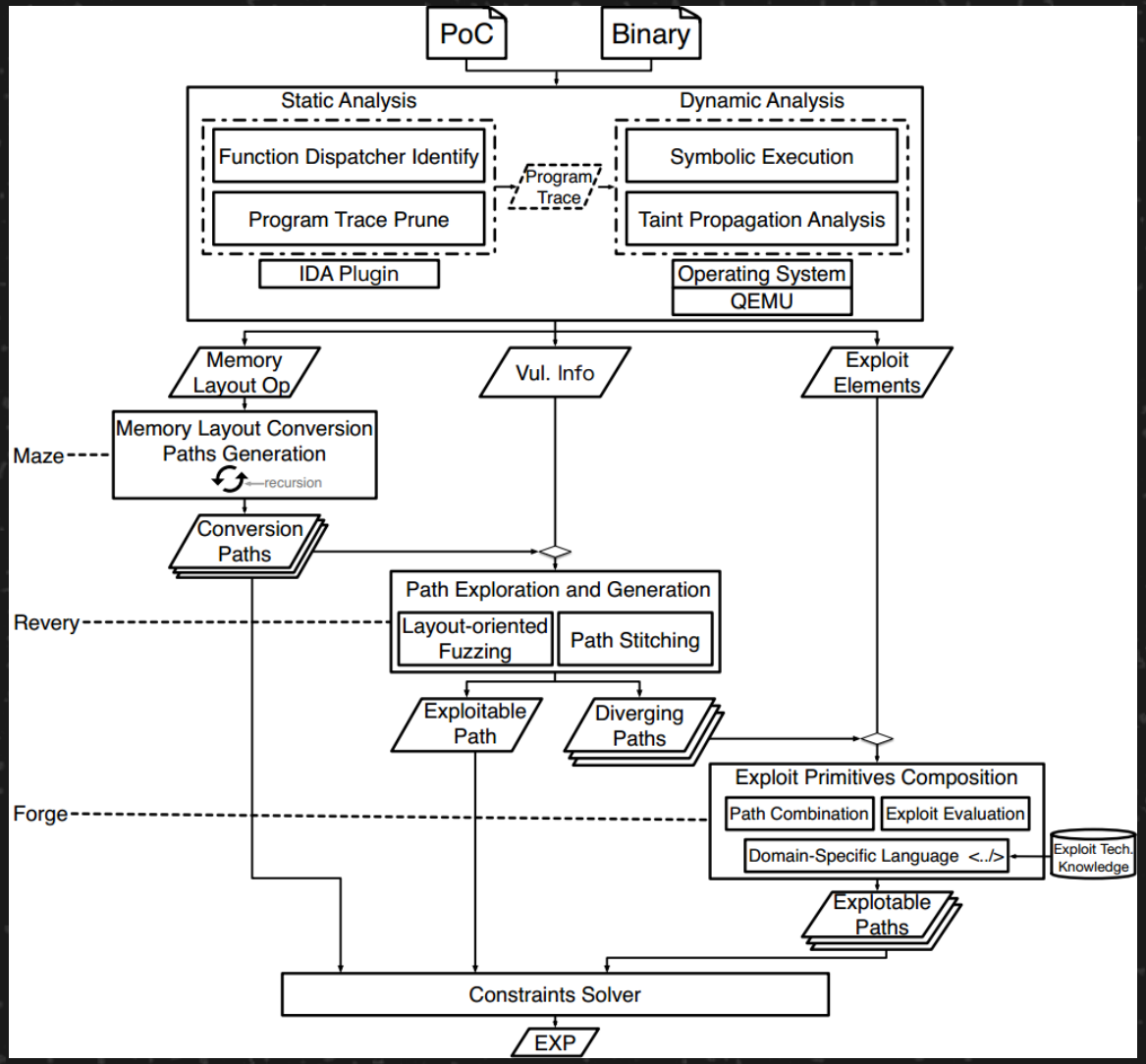
总结

Maze

- Maze通过程序分析技术获取含有内存布局操作原语的程序路径。
- Maze使用Dig & Fill技术，精确操控内存布局，定向构造可利用内存布局。
- Maze可适用于堆溢出、UAF等类型的漏洞，可以把大部分PoC的初始内存布局自动化转换为可利用状态。

下一步工作

下一步工作



- 自动化组合漏洞利用原语，绕过操作系统安全机制。
- 漏洞的自动化类型转换与组合。
- 整合多个关键技术为一个漏洞利用自动化生成系统

谢谢！

Q&A

BLUEHAT
SHANGHAI 2019

Exploring The Maze Of Memory Layouts Towards Exploits

Yan Wang

wangy0129@gmail.com

Twitter @y1su0yanyu

Chao Zhang

chaoz@tsinghua.edu.cn

Twitter @chao_zhang_pku