计 算 机 研 究 与 发 展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202220771
60(7): 1501−1513, 2023

HeapAFL: 基于堆操作行为引导的灰盒模糊测试

余媛萍[1,2]　苏璞睿[1,3]

[1](中国科学院软件研究所可信计算与信息保障实验室　北京　100190)

[2](中国科学院大学计算机科学与技术学院　北京　100190)

[3](中国科学院大学网络空间安全学院　北京　100190)

(yuanping2017@iscas.ac.cn)

HeapAFL: Heap-Behavior Guided Greybox Fuzzing

Yu Yuanping[1,2] and Su Purui[1,3]

[1](*Trusted Computing and Information Assurance Laboratory*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)

[2](*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100190)

[3](*School of Cyber Security*, *University of Chinese Academy of Sciences*, *Beijing* 100190)

Abstract　As the software gets more and more complicated, the intricate reference relationship and the interlaced life cycle of numerous data objects are confusing, which makes them prone to program errors and incurs vulnerabilities. Fuzzing is a general vulnerability discovery technique. However, the state-of-the-art fuzzing techniques focus on the full coverage of functionality testing but not the heap-based memory status in the running. It suffers from heap-based memory state information loss to distinguish execution with potential heap-based memory errors and often strays into unrelated paths. In this paper, we propose a heap-behavior diversity-guided fuzzing solution named HeapAFL. It uses static analysis to obtain the control flow and data flow information of heap-behavior to guide fuzzing to generate test cases that trigger more complex heap behaviors. The fuzzing process is guided by basic heap-behavior information so that our method is general and does not require domain knowledge. We test HeapAFL on a dataset of 6 real-world programs and compare it with 6 state-of-the-art fuzzers with a CPU running for 4 032 hours. The results show that HeapAFL is a suitable method for heap-based memory vulnerability discovery, and it performs better than related works. It outperforms AFL, AFLFast, PathAFL, TortoiseFuzz, Angora, and Memlock in vulnerability findings 1.32 times，1.39 times，1.92 times，1.56 times，2.78 times, and 2.08 times, respectively. Moreover, we have found 25 heap-based vulnerabilities, including 19 known (i.e., 1day) and 6 unknown (i.e., 0day) vulnerabilities, and reported them to CVE (common vulnerabilities and exposures). We have 2 CVE numbers assigned, with the others waiting for confirmation.

Key words　fuzzing; heap-behavior; vulnerability discovery; open source software; prioritization

摘　要　随着软件开发环境和业务逻辑的复杂度不断增加，大量的堆内存对象生命周期及其引用关系造成堆内存操作行为错综复杂，极易引发程序错误造成漏洞. 模糊测试作为高效的软件代码错误检测技术，常用于漏洞挖掘. 然而，目前最先进的模糊测试工具专注于代码全覆盖功能测试，忽略了执行时堆内存操作状态信息，从而错过堆内存漏洞发现机会. 针对上述问题，提出了一种基于堆操作行为引导的灰盒模糊

收稿日期: 2022-08-29; **修回日期**: 2023-01-09

基金项目: 国家自然科学基金项目(62232016, 62102406, 61902384); 前沿科技创新专项课题 (2019QY1403)

This work was supported by the National Natural Science Foundation of China (62232016, 62102406, 61902384) and the Frontier Science and Technology Innovation Project (2019QY1403).

通信作者: 苏璞睿(purui@iscas.ac.cn)

测试方法 HeapAFL,在不依赖漏洞先验知识的情况下,其通过静态分析插桩基础堆操作函数及其参数监测执行时控制流和数据流变化,反馈堆操作行为信息,指导模糊测试中种子优先变异阶段,探索多样化堆操作行为从而更高概率触发堆内存错误类漏洞.在 6 个真实应用程序上验证方法效果,并与 6 个最先进的模糊测试工具进行比较,实验中的 CPU 总共测试了 4 032 h.实验结果表明,HeapAFL 在漏洞挖掘效果和崩溃发现效率上优于对比工作.在漏洞挖掘数量上,HeapAFL 相比于基准模糊测试方法 AFL, AFLFast, PathAFL, TortoiseFuzz, Angora, Memlock 分别提升了 1.32 倍, 1.39 倍, 1.92 倍, 1.56 倍, 2.78 倍, 2.08 倍.最终,HeapAFL 在数据集上挖掘到了 25 个堆内存错误类漏洞,其中包括 19 个已知的漏洞 (即 1 day) 和 6 个未知的漏洞 (即 0 day),并报告给 CVE (common vulnerabilities and exposures) 官方漏洞库后已经获得了 2 个 CVE 漏洞编号,其余漏洞正在等待审核.

关键词 模糊测试;堆操作行为;漏洞挖掘;开源软件;优先变异

中图法分类号 TP311

堆内存错误类漏洞作为一种经典的程序漏洞[1],一旦触发会影响程序功能正常执行,甚至被攻击者利用造成系统执行任意代码.其中,C 和 C++开发语言因其执行的高效性,被广泛应用于底层系统、共享库或者性能需求高的软件开发中,但因缺乏边界检查、人工显式内存管理、软件版本的不断功能迭代和开发者能力参差不齐等原因,极少数软件维护者能一直清晰定位到堆内存操作行为能否正常执行,使软件面临大量堆内存错误类漏洞[1-2].因此,开源软件版本发布之前,进行充分的代码安全性测试是发现和预防潜在安全威胁的重要手段,也是目前研究的热点问题.

模糊测试作为一种常用的软件错误检测技术,具有效率高和易部署的优势,被安全厂商广泛使用,例如谷歌开源的 AFL[3], LibFuzzer[4] 项目等,可以基于代码覆盖率导向程序功能覆盖测试,并取得了很好的漏洞测试效果[3].模糊测试是从无限的程序执行状态空间中探索出错状态空间的过程[5].但是现有灰盒模糊测试方法集中在利用代码维度覆盖信息反馈引导.例如, MemFuzz[6] 采用简单的内存读写指令覆盖,如 load, store 指令; TortoiseFuzz[7] 关注重点易错代码区域,如系统调用函数、循环结构等.有一些为细粒度的代码状态探索设计的反馈信息,例如: PathAFL[8] 引入路径 hpath 覆盖率探索新程序执行路径; Angora[9] 引入 n-gram 代码控制流上下文关联信息. Memlock[10] 监测了内存消耗峰值,仅能挖掘潜在的内存消耗类漏洞,如递归漏洞.由此可见,现有信息反馈集中在代码区域执行计数维度的程序状态探索,对于堆内存错误类漏洞感知力有限.

本文提出一种新的堆内存行为信息反馈方法 HeapAFL,在关注代码区域覆盖信息的同时记录执行时堆内存操作状态信息,通过增加模糊测试工具对堆操作行为的控制流和数据流信息敏感性,来更高概率触发堆内存错误类漏洞.为实现 HeapAFL 方法,需要解决 2 个问题:

1)如何快速记录堆操作行为信息

程序执行中堆内存行为量极其大,如何在模糊测试资源、时间有限的环境中快速记录、存储和比较行为信息,从而快速感知和探索新的堆内存状态空间是一个重要的问题.为平衡开销和效率,本文利用静态分析插桩追踪程序运行时的堆操作函数位置及其参数执行信息,设计了一个新的堆操作信息记录结构 *heapmap*,以实际行为参数值作为结构偏移索引来存储信息,以结构索引值内容的累计计数记录当前输入影响的堆内存操作行为.最终, *heapmap* 记录的是一次程序执行中所有堆操作行为信息,包括数量和类别.

2)如何引导堆操作行为空间状态的探索

已有模糊测试通过变异程序输入,探索不同代码覆盖进行代码功能测试,取得了很好的效果.如何利用记录的堆操作行为信息感知程序执行中堆操作敏感的数据流和控制流信息,导向多样化状态空间是一个难题.本文在保持原有代码覆盖率信息的基础上,动态记录一个全局最多样化堆操作行为结构,在堆操作函数执行次数,或者参数值受当前种子输入影响的范围比历史所有种子都大的情况下,更新当前种子为最高变异优先级 favored,获得下轮种子变异及产生后代的优先权,从而探索多样化堆操作空间.

总体而言,本文提出的方法 HeapAFL 具有 2 个特点: 1)引入了一个新颖的堆内存行为信息反馈维度,在关注代码覆盖执行状态的同时感知堆操作行为维度的状态空间信息; 2)设计了多样化堆操作行

为引导策略，用于调整模糊测试中种子变异优先级. 基于这 2 个特点，本文设计和实现了一个堆内存状态探索工具 HeapAFL，通过编写 LLVM 静态代码分析插件来分析源代码信息，识别并插桩堆操作函数及其参数，获取执行时堆操作行为信息，引导模糊测试进行堆内存状态探索并通过详细的实验测试评估了其有效性.

具体而言，本文在 CVE（common vulnerabilities and exposures）官方网站上收集了 6 个最近有新 CVE 号公布的开源应用程序，复现已知漏洞和挖掘新漏洞的效果测试效率. 本文将 HeapAFL 和 6 个最先进的开源模糊测试方法 AFL[3]，AFLFast[11]，PathAFL[8]，Tortoise-Fuzz[7]，Angora[9]，Memlock[10] 进行比较. 本文进行多次实验并统计了 Mann Whitney U-test 值用作统计分析评估. 实验结果表明，HeapAFL 在挖掘堆漏洞效果上的统计显著性优于 6 个基准模糊测试方法. 在崩溃触发和漏洞发现速率上快于相关工作. 实验测试中 CPU 总共运行了 4 032 h，HeapAFL 方法在数据集上挖掘到了 25 个堆内存错误类漏洞，其中包括 6 个未知的堆内存错误类漏洞.

本文做出了 3 方面的贡献：

1）提出了一种新颖的程序状态感知方式，有效挖掘堆内存错误类漏洞. 静态插桩记录轻量级堆操作行为反馈信息，用于种子优选变异策略，导向多样化堆操作状态空间.

2）实现了 HeapAFL 方法的原型系统，并评估了方案的有效性. 其在漏洞数量和挖掘效率上的实验效果优于最先进的模糊测试工具，共发现 25 个堆内存错误漏洞，其中有 6 个未知的堆内存错误漏洞.

3）在线上仓库中开源了工具源代码①和测试数据用于后续工作研究.

1 问题分析

程序的内存空间包括栈内存和堆内存. 栈内存空间在执行期间由编译器自动分配和回收，且已有的栈内存防护机制几乎可以阻断栈内存错误漏洞利用的发生，因此其危害性相对较小. 堆内存生命周期通过程序开发者动态显式管理，即堆内存的分配和释放依靠手工操作，比起栈内存自动回收内存的机制更容易出错. 另外，堆内存操作灵活、管理机制复杂度高，因此堆漏洞的挖掘和防护仍然是目前共有的难题.

本文利用灰盒模糊测试技术挖掘开源软件中的堆内存错误类漏洞. 首先介绍堆操作相关的内存行为和灰盒模糊测试技术流程组件，然后用一个小的漏洞程序阐述已有工作在挖掘堆内存错误类漏洞存在的局限.

1.1 堆内存操作

堆内存错误类漏洞存在历史长达数十年，据 MITRE 公司官方统计，其仍是最危险的软件安全性漏洞之一[1-2]. 漏洞发生在堆内存对象中的内容被某些非预期程序行为意外修改，而后续程序执行中对修改内容的访问行为会造成程序错误.

为方便展示程序执行中的多样堆内存操作行为，本文截取了程序执行中的 1 个时间片 t_1~t_8 中堆对象 obj1，obj2，obj3，obj4 的行为信息，如图 1 所示. 纵轴代表内存中的堆对象，横轴代表执行时刻信息. 在程序运行时，堆操作代码执行产生大量堆对象，每个堆对象涉及大量堆操作行为，包括堆对象自身的使用信息，及与其他堆对象的相互引用信息. 以堆对象 obj1 为例，其内存空间在时刻 t_1 得到分配，在不同时刻被堆对象 obj2 和 obj4 引用，甚至被堆对象 obj3 二级引用，在时刻 t_6 释放回系统内存，从而结束正常堆内存操作行为信息.

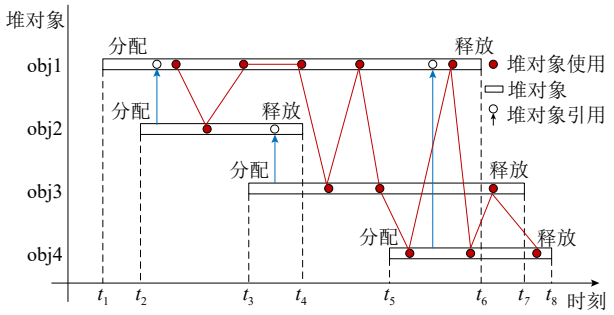

Fig. 1 Heap object operation behavior when program executing

图 1 程序执行时堆对象操作行为

堆内存错误类漏洞与程序执行中堆操作行为关联紧密，表 1 列举了堆操作相关的内存错误类漏洞及其细节信息，包括 CWE（common weakness enumeration）的错误类型描述、漏洞类型、2022 年 CWE 官方统计最危险的 25 种软件错误类型排名和公共漏洞评分系统（common vulnerability scoring system，CVSS）分数值[2]. 堆内存错误类漏洞包括堆溢出（HBO）、空指针解引用（NPD）、多次释放（DF）、释放后重用（UAF）和内存泄露（ML）.

① 仓库的开源网址是：https://github.com/Clingto/HeapAFL.

Table 1 Heap-Based Memory Error Types and Their Details

表 1 堆操作相关的内存错误类型及其细节信息

漏洞类型	错误类型描述	堆操作	CWE 排名[2]	CVSS 分数
堆溢出	越界访问	是	1	64.20
空指针解引用	无效指针解引用	是	11	7.15
多次释放	重复释放	是		
释放后重引用	释放后重引用	是	7	15.50
内存泄露	内存泄露	是	23	3.52

1.2 灰盒模糊测试

灰盒模糊测试是一种常用的漏洞挖掘技术[12], 其使用程序执行中提取的反馈信息,引导模糊测试进程探索更多程序状态从而检测程序错误, 流程如图 2 所示. 其基于程序执行反馈信息, 将保存有意义的种子存进种子队列, 在队列中优选更有价值的种子执行种子变异算法操作, 从而触发新的代码路径或者新的程序状态.

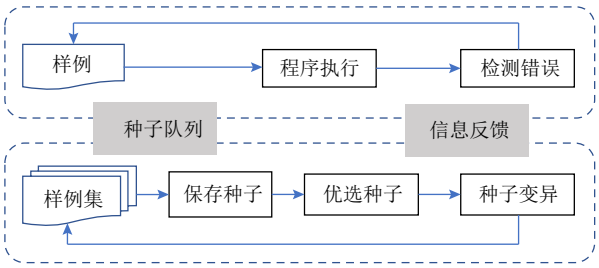

Fig. 2 Workflow of greybox fuzzing

图 2 灰盒模糊测试工作流程

流行的模糊测试工具, 例如 AFL, 采用控制流边覆盖引导程序进行未知代码路径探索, 并取得了显著的漏洞挖掘效果[3]. 但是, 代码覆盖率反馈信息为程序全功能测试设计, 不能区分各个区域代码执行状态的差异性.

除了代码控制流边覆盖率反馈信息, 有一些为细粒度的代码状态探索设计的反馈信息. 例如, Angora 关注代码上下文关联分支覆盖, PathAFL 关注路径覆盖, TortoiseFuzz 关注易错区域程序执行路径覆盖等.Memlock 关注内存消耗敏感的内存操作函数, 用于挖掘潜在内存消耗类漏洞.但是, 集中于代码执行区域探索的信息反馈方式对于程序执行时的堆内存操作状态感知力有限, 从而限制了堆内存错误类漏洞的挖掘能力.

1.3 案例描述

本节用一小段简化后的漏洞代码片段阐述已有灰盒模糊测试工具在挖掘堆内存错误类漏洞上存在的不足. 如图 3 代码所示, 样例中存在一个堆内存错误漏洞, 当非正常访问结构体列表中某个结构成员的数据指针*data 指向数据的时候触发内存错误. 程序通过用户的输入决定 switch 结构里哪个 case 操作将得到执行. 当用户输入数字 1 时, 程序在结构体列表中添加 1 个结构成员, 其包括分配定长的结构体大小的堆内存空间和分配受输入影响的变长堆内存空间 2 次内存分配, 该结构成员后续可以通过输入数字控制执行 case 分支的不同操作, 包括编辑、检查和删除分配到的结构体对象. 在 while 循环不断执行

```
1  struct note{
2      int len;
3      char* data;
4  }Note;
5  int id=0, NUM=100, count=0, select=0;
6  Note* list[NUM], t_note;
7  void add_note() {
8      if(id < NUM) {
9          Note* n=malloc(sizeof(Note)) ;
10         n->len=Get_input();
11         if  (n->len > 0){
12             n->data=malloc(n->len);
13             list [id++]=n;
14         }else
15             printf("Invalid input!");
16     }
17 }
18 void delete_note() {
19     int cur_id=Get_input();
20     if(list[cur_id]) {
21         free(list[ cur_id]->data);
22         free(list[ cur_id]);
23     }
24 }
25 void edit_note()  {
26     int cur_id=Get_input();
27     t_note=list[cur_id];
28     Get_input(*(t_note->data)); //error!
29 }
30 void check_note(){
31     for(int i=0; i<id; i++)
32         if(list[i])
33             count++;
34 }
35 int main(){
36     while(True) {
37         select=Get_input ();
38         switch(select) {
39         case 1: add_note(); break;
40         case 2: delete_note(); break;
41         case 3: edit_note(); break;
42         case 4: check_note(); break;
43         ...
44     }
45 }}
```

Fig. 3 Code lines of case example

图 3 案例代码

过程中，代码分支被覆盖的程度加深，程序状态变多，堆内存行为也变得复杂.

已有模糊测试工作的反馈信息对堆内存行为没有足够的感知力，可能会忽略对漏洞发现有价值的种子及其后代变异机会，减弱堆内存错误漏洞挖掘能力. 以 AFL 为例，种子 1 执行路径代码行为 41—9—12—43—44—42，在多轮种子变异后，AFL 几乎全覆盖程序代码（接近 100% 的代码覆盖率），失去反馈指导力，陷入接近于盲模糊测试阶段；此时，生成另一个与种子 1 代码覆盖率相似且仅有堆操作行为不一样的种子 2，但是由于种子 2 没有引起代码覆盖率变化且执行效率低，因此会被设置最低优先级滞后变异. 据手工分析，种子 2 是模糊测试进程中生成触发漏洞的概念验证（proof of concept, POC）种子的关键先祖测试用例. 其他内存相关的模糊测试工作与上述情况类似，它们集中在代码维度信息覆盖和反馈，或多或少缺失程序执行时的信息反馈. Memlock 虽然关注到了程序执行时内存使用的峰值变化，但仅针对内存资源消耗类漏洞，削弱了多样化堆操作行为探索，例如样例中的种子 2 是产生了新的堆内存操作行为从而接近于 POC，而不是新的内存消耗峰值变化.

2 观察及解决思路

代码覆盖率反馈信息对堆内存操作行为的感知不充分，而使用细粒度易错代码区域敏感的反馈信息也局限于代码执行计数，限制了程序堆内存操作行为的状态空间探索. 本文引入新的堆内存操作行为敏感的反馈信息，引导灰盒模糊测试工具探索开源软件中多样化的堆内存操作状态，从而检测未知错误.

具体而言，模糊测试器通过变异种子收集并反馈程序执行中受种子输入影响的堆操作控制流和数据流信息，观测种子执行的中断信号来判断程序执行异常. 常用的细粒度程序分析方法是根据程序的控制依赖或者数据依赖图，例如 UAFL[13−15] 等利用指针依赖分析或者程序切片指导堆操作行为顺序执行，理论上可以准确预知程序执行状态，但效果受限于程序静态分析技术性能瓶颈. 因此，本文不依赖程序分析而记录粗粒度的堆操作行为反馈信息，该方法在原型系统中被验证是有效的.

程序执行中堆操作行为信息量庞大，如何表示和记录执行时的堆操作行为是一个难题. 为了平衡执行效率和记录开销，本文提出一种轻量级堆操作行为跟踪机制，通过插桩堆操作相关函数（例如 *malloc*( ), *calloc*( ), *realloc*( )等）的操作指令和操作参数，记录执行时堆操作行为信息.

对于追踪到的堆操作行为，快速判断和反馈堆内存操作控制流和数据流变化，可以引导模糊测试工具触发更多堆内存行为. 本文使用程序执行中操作的堆对象数量来表示和记录堆操作函数所在控制流边的执行频度，及其与当前种子输入的关联性. 除了堆操作行为相关的控制流信息，堆操作数据流信息敏感性也是一个重要的堆操作行为维度. 本文通过堆操作函数参数值的变化判断堆操作数据是否受当前输入样本控制，从而判断输入堆操作数据流敏感性.

总体而言，本文将上述解决思想实现为一个整体的模糊测试系统 HeapAFL，通过对系统的评估测试验证方法的有效性. HeapAFL 建立在流行的模糊测试框架 AFL[3] 上，并沿用 AFL 相同的系统框架工作流程.

3 系统概述

本节首先通过系统框架描述本文提出的 HeapAFL 方法，然后通过案例具体描述堆操作行为引导方法的执行过程.

3.1 HeapAFL 方法概述

图 4 是 HeapAFL 的系统框架图，框架图包括静态分析和模糊测试 2 个部分. 静态分析以程序源代码为输入，生成控制流信息和堆操作 2 种程序信息（见 4.1 节）. 静态分析判断程序插桩位置以及插桩指令类型，用于获取程序执行时的反馈信息. 其中控制流信息用于引导代码控制流边覆盖，而堆操作信息用于引导程序堆操作行为多样化探索（见 4.2 节）.

当程序完成插桩操作后，HeapAFL 进入持续的程序模糊测试循环阶段. 对于给定的初始种子集合，HeapAFL 首先从种子队列中优先选择一个种子，利用多种变异策略生成测试集；接着，用插桩后的程序分别执行测试集中每一个测试样例，收集各个测试样例执行时的堆操作行为信息和控制流边覆盖信息，如果执行触发段中断信号（segment fault, SEGV），则将测试样例加入崩溃样本集合，如果测试样例执行产生新代码覆盖则加入种子队列；最后，通过堆内存行为信息和程序执行吞吐量选择有价值的种子优先执行下一次种子变异操作，从而获取产生更多后代种子的机会. HeapAFL 重复上述模糊测试循环过程，直到测试到达预先设定的时间或者手动结束模糊测试操作.

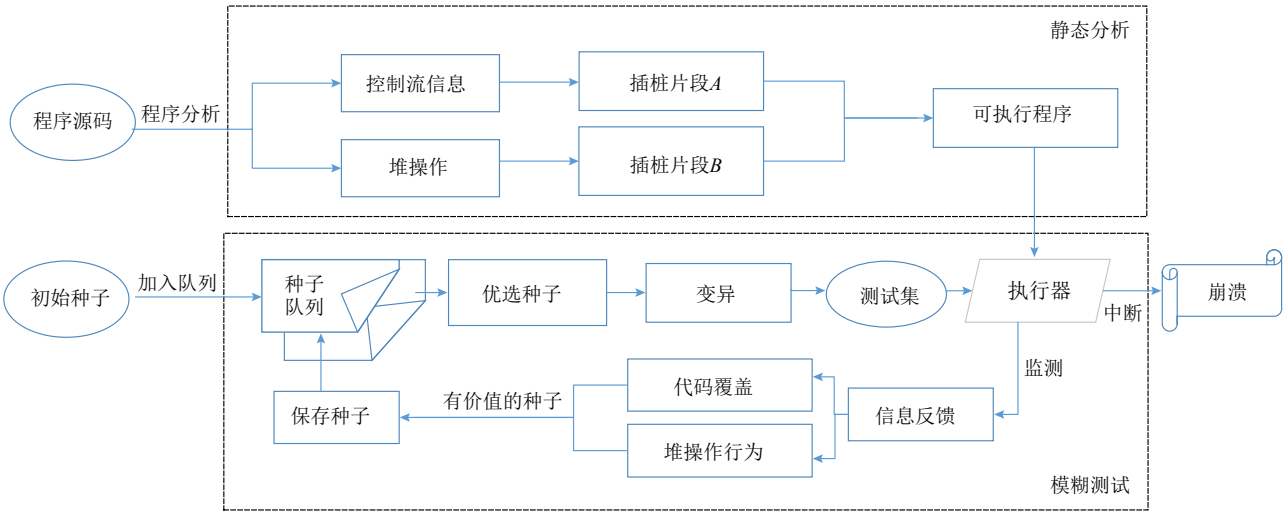

Fig. 4 Framework of HeapAFL

图 4 HeapAFL 的系统框架图

3.2 HeapAFL 方法案例

本节用案例展示 HeapAFL 的执行过程. 图 5 是样例程序的部分控制流转移图，图 5 由基本块和控制流跳转边组成. 模糊测试流程通过变异输入样本 *Input* 中的字节，探索和覆盖受输入影响的控制流和数据流，从而测试程序的不同状态空间.

在静态分析阶段，HeapAFL 给每个基本块头部插桩了一个执行计数器 *bitmap*，记录代码覆盖信息；给每个堆操作指令处插桩了一个指令定位及获取其操作数值的位图 *heapmap*，记录堆操作行为信息. HeapAFL 模糊测试时，通过 *bitmap* 存储的代码覆盖信息判断并保存有新代码覆盖的测试样例，逐步探索不同代码分支，例如 case 1，case 2 等. 基于代码覆盖，case 1 分支上的 2 个受输入控制的堆操作 *malloc*(*const*)和 *malloc*(*size*)得到执行，并产生堆操作行为. 其中，堆操作 *malloc*(*size*)的堆大小参数值受输入 *Input* 控制. *heapmap* 可以记录并感知影响 case 1 分支中堆分配操作的控制流变化，例如堆对象数量变化，引导 case 1 分支执行；*heapmap* 也可以记录堆操作 *malloc*(*size*)中堆大小参数受输入影响的数据流行为

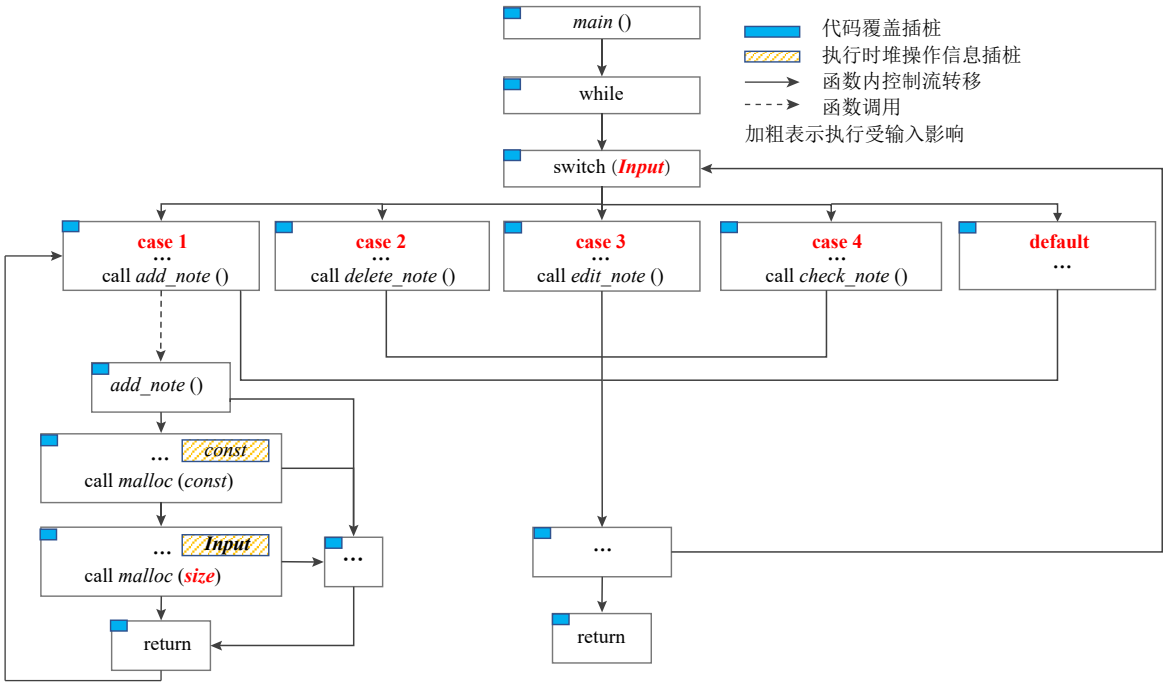

Fig. 5 Partial control flow transition of case example

图 5 样例程序局部控制流转移

变化，例如通过堆大小参数的变化判断当前测试用例输入是否会影响堆操作值，从而引导堆空间大小数值的变化和堆数量的变化，探索多样化堆操作行为.

本文工作是通过增加堆内存对象的分配数量和增加复杂堆操作行为的执行数量，使程序执行时的堆内存行为关联更复杂，从而更高概率出错. 此处样例程序的堆内存错误漏洞发生在 case 3 分支中，结构体指针成员因前面多次操作中的 1 次异常堆内存释放操作的执行，后续访问无效指针内存空间出错.

在原型系统测试中，本文用 AFL 检测出堆内存错误漏洞用了 39.65 min，用 HeapAFL 工具测试仅用了 2.55 min.

4 堆操作行为引导方法

本节介绍堆操作行为引导方法中静态分析插桩和模糊测试循环 2 部分的详细设计及实现.

4.1 静态分析插桩

HeapAFL 静态分析判断待测程序的插桩方式，用于收集导向信息，驱动模糊测试进程. 基于代码控制流信息分析，HeapAFL 插桩收集代码覆盖引导路径探索. 基于堆操作分析，收集执行时堆内存操作行为变化，引导模糊测试在有限时间范围内探索多样化堆内存行为.

为了方便描述方法，本文使用如下术语.

术语 1. 控制流边. HeapAFL 沿用 AFL 基于控制流转移图（control flow graph, CFG）记录控制流边覆盖（branch coverage）信息的方式. 在基本块头部插入以伪随机边数值（即 ID）标识的 8 b 计数器，追踪程序执行时每条控制流边的执行次数，并通过不同计数数量级对计数值设置分类桶，即按照控制流边执行次数 1, 2, 4~7, 8~15, 16~31, …, 128~255 等分为 8 个类别. 最终，1 次程序执行的全部代码控制流边覆盖信息被记录进路径 *bitmap* 中.

术语 2. 路径 *bitmap*. 大小为 2^K 的 8 b 数组计数器，计数值保存在以边 ID 为索引的 *bitmap* 偏移值中（在 AFL 中，K=16）.

术语 3. 堆内存操作. 堆内存分配操作，例如，*malloc*(*size*),*calloc*(*size*),*realloc*(*size*)等，决定可操作的堆内存对象数量. 当堆操作敏感的控制流或者堆操作参数值 *size* 敏感的数据流受程序输入影响时，可以基于影响堆内存操作的输入引导模糊测试探索更多堆内存行为状态.

需要提到的是，本文插桩标准的系统库堆内存操作获取函数操作指令和参数. 部分程序采用自定义堆内存操作函数，但大部分程序仍依赖于系统库与内存交互，因此本文不处理极少数用户自定义但不依赖标准库的内存操作函数.

术语 4. 堆操作行为 *heapmap*. 大小为 2^K 的 8 b 数组计数器，计数值保存在以操作参数 *size* 为索引的 *heapmap* 偏移值中（K=16）. 所有计数器计数累加值增大，或者计数非零项增加，表明此次测试样例中存在影响堆操作的控制流或者数据流信息的字节，可以优先选择变异该种子及关联字节.

信息反馈插桩实现细节如算法 1 所示，本文基于 LLVM[16] 代码分析框架分析应用程序，基于分析结果插桩代码片段收集反馈信息，引导模糊测试进程. 除了 AFL 的控制流边代码覆盖信息，本文还插桩堆内存操作信息. 具体而言，本文沿用 AFL 在每个基本块头部（算法 1 的行⑤）插桩记录边覆盖 *BranchCov*（算法 1 的行②）；除此以外，插桩堆内存操作函数及其操作数值 *opd*（算法 1 的行⑦），以操作数值为索引记录堆操作行为信息存入 *heapmap*（算法 1 的行③）.

算法 1. 反馈信息插桩算法.

① *Initialize*(*BranchCov*, *heapmap*)；
② func *CovFb*(*BranchCov*, *cur_loc*, *prev_loc*)
  /*代码覆盖反馈*/
    BranchCov[*cur_loc*⊕*prev_loc*] ++;
③ func *HeapFb*(*heapmap*, *cur_loc*, *prev_loc*, *opd*)
    /*堆操作行为反馈*/
    heapmap[*opd*]++ ;
④ func *Instrumentation*(*Func*)
⑤ for each *BB* in each *Func* do
    cur_loc = <*CompileTime_Random*>;
      /*控制流边 ID*/
    Insert(*CovFb*(*BranchCov*, *cur_loc*, *prev_loc*))；
⑥ for each *Ins* in *BB* do
⑦   if *IsHeapAllocFunc*(*Ins*(*opd*)) then
        Insert(*HeapFb*(*heapmap*, *opd*))；
      end if
    end for
⑧ *prev_loc* = *cur_loc* >> 1;
⑨ end for

4.2 模糊测试

基于 4.1 节所述的静态插桩阶段获取的反馈信息，模糊测试通过选种、变异等模块引导测试进程逐渐接近目标状态. 模糊测试中每个测试用例有一个选择概率，HeapAFL 基于选择概率判断种子下一轮变异产生后代的概率，概率高的测试用例会被优先

选择变异. 被选中的种子会执行变异操作, 产生一定数量的测试用例, 其中产生新代码覆盖的测试用例会加入种子队列等待被选择.

术语 5. 选择概率 *Prob*. 种子被选择测试的概率, 设置为 favored 的种子一定会被测试到, 否则以大小为 a 的概率被测试执行. 本文设置 $a = 0.01$.

术语 6. 累计堆操作行为. 累计当前测试用例执行的堆操作数量, 用于判断其执行是否影响堆操作控制流信息; 除此以外, 累计堆操作数值 *size* 分类的类别数量, 便于快速比较、判断当前测试用例堆操作参数是否数据流敏感.

术语 7. 优先变异. 如果当前种子的累计堆操作行为, 即堆操作数总量或者堆操作数值类别总量多于历史最高值, 则设置种子的选择概率为 100%.

堆操作行为引导的灰盒模糊测试算法实现细节如算法 2 所示. 本文修改 AFL-2.52b[3] 的核心模糊测试组件, 利用算法 1 所述的插桩获取的反馈信息, 即代码控制流边覆盖 *BranchCov* 和堆操作行为信息 *heapmap*, 升级了 AFL 的种子优选变异策略.

算法 2. 堆操作行为引导的灰盒模糊测试算法.

输入: 插桩后的程序 P, 初始种子集合 T;

输出: 触发堆内存错误类漏洞的测试样例集合 S.

① $S \leftarrow \varnothing$;
② $Queue \leftarrow T$;
③ Repeat
④ for each t in $Queue$ do
⑤ if $Prob_t$ to select t then
⑥ $NumChildren \leftarrow AssignEnergy(t)$;
⑦ end if
⑧ for $i = 0 \rightarrow NumChildren$ do
⑨ $Child_i \leftarrow Mutate(t)$; /*种子变异*/
⑩ $status,time,BranchCov,heapmap \leftarrow Execute(P, child_i)$; /*测试样例执行*/
⑪ if $status$ == CRASHED then
⑫ $S \leftarrow S \cup child_i$; /*崩溃样例保存*/
⑬ else $Is\_Interesting(BranchCov)$ then
⑭ $Queue \leftarrow Queue \cup child_i$; /*种子保存*/
⑮ $HeapBehav \leftarrow Accumulation(heapmap)$; /*累加*/
⑯ if $HeapBehav > global\_HeapBehav$ then
⑰ $HeapBehav = global\_HeapBehav$;
⑱ $UpdateSeedProb(child_i, Time, HeapBehav, Prob_t)$; /*种子队列排序*/
⑲ end if
⑳ end if
㉑ end for
㉒ end for
㉓ until 超时或人工结束 /*fuzzing 循环终止*/

种子变异产生一定数量的后代测试用例, 其中触发程序段中断信号 SEGV 的测试用例保存进崩溃样例集合(算法 2 的行⑫), 产生新代码覆盖的测试样例保存进种子队列集合(算法 2 的行⑭); 接着, 给当前种子计算累计堆操作行为 *HeapBehav*, 根据 *HeapBehav* 值与全局最大累计堆操作行为 *global_HeapBehav* 的比较值, 设置当前种子的选择概率, 即 favored 与否(算法 2 的行⑯~⑲); 最后, 优先选择设置为 favored 的种子进行变异(算法 2 的行⑤⑥⑦). 模糊测试循环会一直执行直到超时或者人工停止测试.

5 实验评估

本文基于真实应用程序组成的测试集进行了详细的实验测试, 评估上述原型系统 HeapAFL. 更详细的细节信息见本文开源仓库. 实验主要回答 3 个问题:

问题 1. HeapAFL 是否可以挖掘到真实应用程序漏洞.

问题 2.优先变异策略是否可以提升堆内存漏洞挖掘效率.

问题 3. HeapAFL 堆内存错误类漏洞挖掘能力是否优于相关模糊测试对比工作.

5.1 实验设置

本文从 4 个角度设置了详细的实验对比方案评估 HeapAFL 效果.

1)相关对比工作. 本文对比 6 个先进的基准模糊测试工具, 包括 AFL, AFLFast, PathAFL, TortoiseFuzz, Angora, Memlock, 表 2 列出了对比工具发表的会议及其年份. 对比工具的选择基于以下考虑: AFL 是一个广泛使用和用于比较的灰盒模糊测试工具; AFLFast

Table 2 Related Work of Fuzzing

表 2 模糊测试相关工作

工作	会议	年份
AFL[3]		2014
AFLFast[11]	CCS	2017
PathAFL[8]	AsiaCCS	2020
TortoiseFuzz[7]	NDSS	2020
Angora[9]	S&P	2018
Memlock[10]	ICSE	2020

是 AFL 的升级版，优化了种子优选变异策略；PathAFL, TortoiseFuzz, Angora 分别基于代码路径、易错代码域和 n-gram 代码控制流边的探索程序状态；Memlock 是基于运行时信息，即内存消耗峰值挖掘资源消耗类漏洞. 需要提到，本文没有选取 MemFuzz 作为对比工具，一方面其没有开放源码，另一方面其关注内存的读写信息，与堆内存操作信息关注点不同. 总结来说，本文选择多种具有代表性、广泛应用于漏洞挖掘实践的先进模糊测试工具作为基准对比工作，用于评估方案效果.

2)测试集合. 本文选取数据集基于下面的因素考虑：应用程序存在已知 1day 堆内存错误类漏洞且可能挖掘到未知 0day 漏洞、流行度高、测试频度高、开源社区活跃度高和功能多样等. 根据已经发现过漏洞的程序再次出现新漏洞的概率更大的判断，本文在 CVE 官方网站上收集了 6 个最近有堆内存错误类 CVE 号公布的开源应用程序，分别包括图片处理工具 jpegoptim、编译器 yasm、物联网固件平台 mjs、知名的开发工具 readelf、视频处理器 mp4box 和文档处理器 mxmldoc.

3)性能指标. 为对比相关基准模糊测试工具，最直观的方式是通过发现的漏洞数量判断工具的漏洞挖掘能力；除此以外，本文设计堆内存行为优先变异策略，因此漏洞挖掘效率评估也是一个重要指标. 本文对比了漏洞数量、程序崩溃 crash[3] 数量和漏洞发现时间. 此外，本文进行 Mann Whitney U-test 统计分析，判断各个工具的执行效果对比的显著性. 本文的模糊测试执行速度通过每秒测试样例的执行数量表示，用于判断测试工具的执行效率.

4)实验配置参数. 由于模糊测试技术依赖随机变异过程，所以测试中可能存在自然的性能抖动. 本文采取 2 个策略，每个程序测试相对较长的时间以达到稳定状态，例如 12 h；每组实验测试 8 次，进行统计分析用于评估工具统计有效性. 实验中 CPU 总共测试 4 032(6×7×8×12)h. 编译测试程序的编译器版本是 clang6.0.0，测试的软件包、程序版本和执行选项如表 3 所示. 实验机器是 64 位 Ubuntu LTS 16.04，CPU 配置为 Intel® Xeon® CPU E5-2630 v3，处理器 2.40 GHz，32 核，内存 32 GB.

5.2 真实程序漏洞挖掘（问题 1）

本文模糊测试沿用 AFL[3] 段中断信号 SEGV 识别程序崩溃样本，并存入崩溃样本集合. 接着，使用 ASAN[17]（AddressSanitizer）工具的程序崩溃栈哈希值筛除集合中重复的程序崩溃样本；然后,手工验证剩余样本崩溃信息，构建其与公布的漏洞 CVE 号或者错误报告的关联性来识别漏洞. 对于没有检索到漏洞信息的崩溃，将其上报给 CVE 官方漏洞库确认后会给漏洞发布一个唯一标识 ID，即 CVE 漏洞编号.

Table 3 Real-world Programs Tested in Experiment

表 3 实验中测试的真实程序

软件包	测试程序	版本	命令
JPEGOPTIM	jpegoptim	d23abf2	-
YASM	yasm	6caf151	-
MJS	mjs	9eae0e6	-f
BINUTILS	readelf	2.28	-w
GPAC	mp4box	4c19ae5	-diso
MXML	mxmldoc	2.12	-

注：“-”后面接执行命令选项，无命令选项代表执行默认命令.

本文实验中，HeapAFL 在测试集中总共挖掘到 30 个漏洞，其中 25 个是堆内存错误类漏洞，5 个是栈溢出漏洞(SO). 表 4 中列举了挖掘到的漏洞的详细信息，包括漏洞 ID、漏洞类型(详见 1.1 节)、漏洞状态、测试程序和程序版本号. 其中，状态标识为 1day 表明前人发现了此漏洞并公开了漏洞信息，标识为 0day 的漏洞表明本文崩溃样例 POC 触发的漏洞是当前版本的未知漏洞，未检索到相关信息. HeapAFL 在当前测试集上共发现 8 个未知漏洞，其中有 6 个堆内存错误类漏洞，在上报给 CVE 官网后，获得了 2 个新的 CVE 编号，即 CVE-2021-33461 和 CVE-2021-33462，其余漏洞在审核当中.

5.3 漏洞挖掘效率（问题 2）

本文基于相同测试集重跑开源工具进行对比实验. 本节基于崩溃数量随时间的增长趋势和堆内存错误类漏洞第 1 次被发现的时间评估 HeapAFL 的漏洞挖掘效率.

图 6 中展示了测试集上 6 个程序在对比工具上的崩溃数量随着时间增长的曲线图. 需要注意的是，AFL 定义了一种崩溃记录方式，该方式是一种常见的模糊测试效率的评估指标，详细信息见 AFL 用户手册文档[3]. HeapAFL, AFLFast, TortoiseFuzz, Memlock 沿用了 AFL 框架流程，Angora 和 PathAFL 的日志文件记录的信息与 AFL 不同，此处不展示. 从图 6 中可以看出，HeapAFL 在 gpac, yasm, binutils, mjs 这 4 个程序上的崩溃数量能快速多于其他工具触发崩溃且持续增长；在 mxml 测试程序的模糊测试过程中前 1 h 能快速触发崩溃，后续速度慢于其他相关工作，但最终能达到与最高数量 124 接近的崩溃数量 104；在

Table 4 Real-world Vulnerabilities Found by HeapAFL

表 4 HeapAFL 挖掘到的真实漏洞

漏洞 ID	程序	版本	类型	状态
Bug_23 869	readelf	2.28	ML	1day
CVE-2019-14 444	readelf	2.28	NPD	1day
CVE-2017-6 966	readelf	2.28	NPD	1day
CVE-2019-20 169	gpac	4c19ae5	UAF	1day
CVE-2019-20 164	gpac	4c19ae5	UAF	1day
CVE-2019-20 168	gpac	4c19ae5	UAF	1day
CVE-2018-11 416	jpegoptim	d23abf2	DF	1day
mjs-issue-78	mjs	9eae0e6	UAF	1day
mjs_crash_1	**mjs**	**9eae0e6**	**NPD**	**0day**
CVE-2021-33 444	mjs	9eae0e6	NPD	1day
CVE-2021-33 439	mjs	9eae0e6	NPD	1day
mjs_crash_2	**mjs**	**9eae0e6**	**NPD**	**0day**
mjs_crash_3	**mjs**	**9eae0e6**	**SO**	**0day**
mjs_crash_4	**mjs**	**9eae0e6**	**NPD**	**0day**
mjs_issue	mjs	9eae0e6	SO	1day
CVE-2021-33 464	yasm	6caf151	HBO	1day
CVE-2021-33 463	yasm	6caf151	NPD	1day
CVE-2021-33 462	**yasm**	**6caf151**	**UAF**	**0day**
CVE-2021-33 461	**yasm**	**6caf151**	**UAF**	**0day**
CVE-2021-33 456	yasm	6caf151	NPD	1day
CVE-2021-33 460	yasm	6caf151	NPD	1day
CVE-2021-33 455	yasm	6caf151	NPD	1day
CVE-2021-33 458	yasm	6caf151	NPD	1day
CVE-2021-33 457	yasm	6caf151	NPD	1day
CVE-2021-33 466	yasm	6caf151	NPD	1day
yasm-crash_1	**yasm**	**6caf151**	**SO**	**0day**
yasm-crash_2	**yasm**	**6caf151**	**NPD**	**0day**
mxml_crash_1	mxml	2.12	SO	1day
CVE-2018-20 593	mxml	6caf151	SO	1day
mxml-issue-235	mxml	6caf151	NPD	1day

注：加粗字体标识测试集中发现的未知漏洞.

jpegoptim 程序上, HeapAFL 最晚触发崩溃, 用时 2 h, 且崩溃数量“1”小于最高值“14”. 据后续手工漏洞成因分析发现, 其他测试工具虽然测试出的崩溃数量多, 但是, 属于触发同一个程序漏洞的多个类似崩溃样例, 且与 HeapAFL 一个崩溃样例触发的漏洞相同. 整体而言, HeapAFL 在发现崩溃效率上有明显的优势.

表 5 中展示了 HeapAFL 和 AFL 发现相同堆内存错误类漏洞花费的时间(此处仅展示 HeapAFL 和 AFL 都挖掘到的漏洞信息). 从表 5 中可以看出, HeapAFL 比 AFL 能更快挖掘漏洞(即 34.93 h<40.06 h). 具体而言, HeapAFL 在挖掘接近 70%(即 11/16)的漏洞上能缩短漏洞发现时间; 在剩余 5 个漏洞的挖掘中, Bug_23869, CVE-2018-11416, mxml-issue-235 仅轻微增加测试时间 6 min; 在 CVE-2021-33462 和 CVE-2021-33455 漏洞上增加的时间稍长, 其大于 1 h. 由于 HeapAFL 仅优化了 AFL 工作流程框架中的种子优选变异策略, 因此从漏洞发现时间可以看出, HeapAFL 能提高堆内存错误类漏洞的挖掘效率.

综上所述, HeapAFL 在崩溃发现速度和堆内存错误类漏洞发现速度上有明显优势, 因此, 优先变异策略可以提升堆内存错误类漏洞挖掘效率.

5.4 漏洞挖掘能力对比(问题 3)

漏洞挖掘数量是判断模糊测试工具漏洞挖掘能力的重要指标, 本节基于各个测试工具挖掘到的漏洞数量的并集、平均值和交集的对比来评估 HeapAFL 的漏洞挖掘能力. 除此以外, 本节计算了多次实验的 Mann Whitney U-test 统计分析数值(即 p-value), 评估实验结果对比的显著性. 最后, 由于 Sanitizer 可以帮助查找和判断堆内存漏洞的发生, 本节进一步探索在模糊测试方法 HeapAFL 中添加知名的 Sanitizer 工具 ASAN[16] 对堆内存漏洞挖掘效果的影响.

各个模糊测试工具的堆内存错误类漏洞数量如表 6 所示, 包括多次实验漏洞数量总和(U)和平均数量(Avg). 从漏洞整体漏洞数量和单次平均漏洞数量来看, HeapAFL 优于所有基准的对比工具. 虽然从漏洞数量平均值上的比较来看, HeapAFL(即 16 个)与第 2 名 AFLFast(即 14.13 个)和第 3 名 AFL(即 13.13 个)差距不算太大, 但从整体实验漏洞并集来看 HeapAFL 明显优于 AFL, AFLFast(即 25/19=1.32 和 25/18=1.39), 且优于接下来的 TortoiseFuzz(即 25/16=1.56)、PathAFL(即 25/13=1.92)、Memlock(即 25/12=2.08)和 Angora(即 25/9=2.78). 除了漏洞数量, 本文计算了 8 次独立重复实验中各次实验结果的 Mann Whitney U-test 统计分析结果, 如表 6 中最后一行 p-value 所示. 其中 p-value 值小于 0.01 表明实验结果显著优于其他工作, 因此 HeapAFL 实验结果显著优于 6 个基准对比工具. 由此可见, HeapAFL 的堆内存错误类漏洞挖掘能力优于其他相关工作.

图 7 是 HeapAFL 与各个对比工具堆内存错误类漏洞比较的韦恩图, 展示了漏洞比较的详细信息, 包括方法对比中相同的漏洞数量和不同的漏洞数量及各个数量占比. 从图 7 中可以看到, HeapAFL 虽然遗

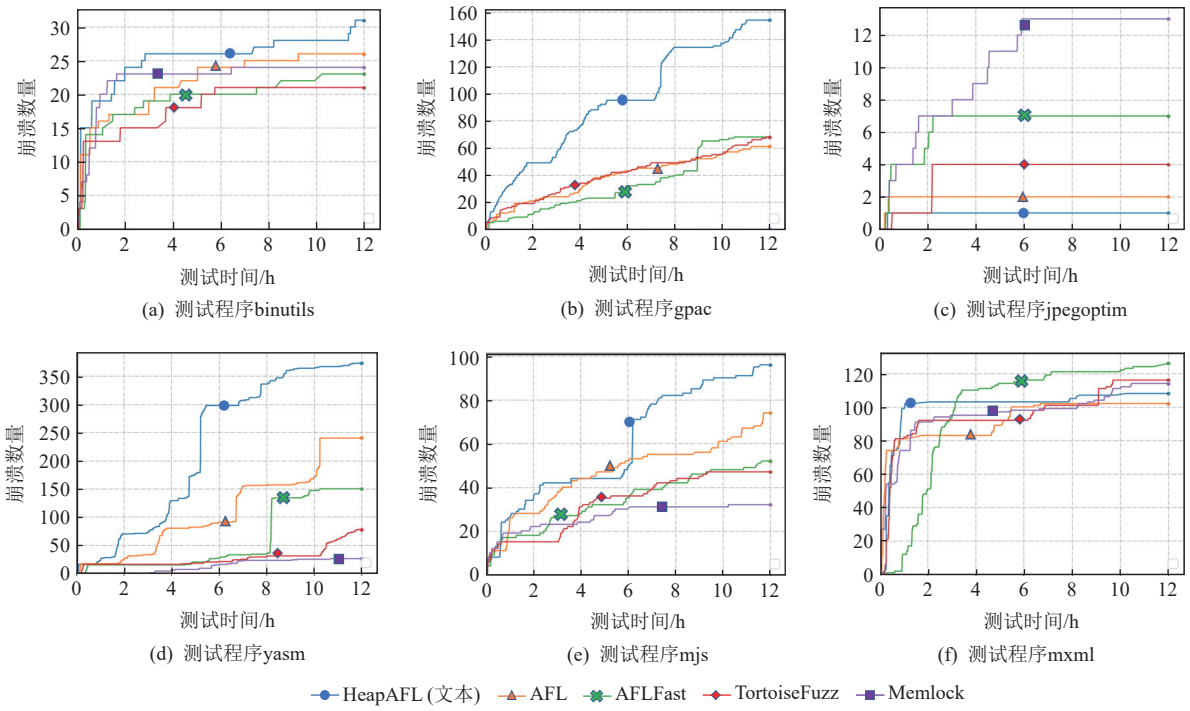

Fig. 6 Curves diagram of crashes number increasing with test time

图 6 崩溃数量随测试时间的增长曲线图

Table 5 Time of Heap-Based Vulnerabilities Found by HeapAFL and AFL

表 5 HeapAFL 和 AFL 挖掘到的堆内存错误漏洞时间 h

漏洞 ID	HeapAFL	AFL	差值
Bug_23869	0.14	0.06	0.09
CVE-2019-14444	0.05	0.06	0.00
CVE-2017-6966	0.14	0.19	−0.05
CVE-2019-20169	0.04	0.11	−0.07
CVE-2019-20164	1.54	9.92	−8.38
CVE-2018-11416	0.35	0.23	0.12
mjs-issue-106	10.43	11.67	−1.24
mjs_crash_1	0.04	0.06	−0.01
CVE-2021-33444	2.89	3.24	−0.36
CVE-2021-33463	0.10	0.16	−0.06
CVE-2021-33462	8.32	6.54	1.78
CVE-2021-33456	0.84	3.37	−2.53
CVE-2021-33455	9.57	3.78	5.79
CVE-2021-33458	0.08	0.26	−0.18
CVE-2021-33457	0.08	0.26	−0.18
mxml-issue-235	0.31	0.16	0.15
合计	34.93	40.06	−5.13

漏了少数漏洞但整体效果优于基准对比工作.

表 7 是 HeapAFL 添加 ASAN 之前和之后堆内存错误类漏洞数量和模糊测试中测试样例的执行速度. 可以看出, HeapAFL 方法添加有代表性的 Sanitizer 工具 ASAN(即 HeapAFL-ASAN)可以挖掘到 21 个漏洞, 略少于原始 HeapAFL 工具挖掘到的 25 个漏洞, 但是多于原生 AFL 工具挖掘到的 19 个漏洞. 进一步研究发现, HeapAFL-ASAN 的测试样例执行速度平均值为每秒 226.72 个, 小于 HeapAFL 的每秒 489.24 个, 也就是添加了 ASAN 的测试工具相对降低了模糊测试速率, 从而在一定程度上减少了样例测试次数. 本文测试出的实验结果与文献 [17] 中提到的添加 ASAN 会降低执行速度的结论相吻合. 另外, 对于 mjs 和 mxml 程序, HeapAFL-ASAN 的执行速度接近于 HeapAFL(即 274.46 对比于 261.50 和 443.40 对比于 475.74), 而其漏洞挖掘实验效果优于 HeapAFL, 可见在执行的测试样例总量接近的时候, HeapAFL-ASAN 漏洞挖掘能力优于 HeapAFL. 因此, 本文建议在时间、计算资源充足的漏洞挖掘场景中, 可以在 HeapAFL 中添加 ASAN 以提高堆内存错误类漏洞挖掘能力.

6 结 论

本文提出了一种基于堆操作行为引导的灰盒模糊测试方法 HeapAFL, 通过堆操作行为的控制流和数据流敏感性, 引导模糊测试工具探索多样化的堆内存操作状态空间, 从而增加堆内存错误类漏洞的

Table 6 Number of Heap-Based Vulnerabilities Found by HeapAFL and Different Fuzzers

表 6 HeapAFL 与各个模糊测试方法发现的堆漏洞数量

测试程序	HeapAFL		AFL		AFLFast		PathAFL		TortoiseFuzz		Angora		Memlock	
	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg	U	Avg
binutils	3	2.13	3	2.13	2	2.13	2	1.50	3	2.13	4	2.00	4	2.38
gpac	3	3.00	2	1.25	3	2.88	2	1.50	2	1.38	5	1.00	0	0.00
jpegoptim	1	1.00	1	1.00	1	1.00	1	0.75	1	1.00	0	0.00	1	0.75
mjs	6	3.13	5	2.75	6	2.50	2	0.88	4	2.50	0	0.00	3	1.88
yasm	11	5.75	6	4.75	5	4.50	4	2.63	4	3.88	0	0.00	3	1.88
mxml	1	1.00	2	1.25	1	1.13	2	1.88	2	0.88	0	0.00	1	0.38
合计	25	16.00	19	13.13	18	14.13	13	9.13	16	11.75	9	3.00	12	7.25
p-value			8.35E−04		8.00E−03		1.57E−04		3.36E−03		1.91E−06		1.52E−06	

注：U 是多次实验漏洞数量总和，Avg 是多次实验漏洞平均值，p-value 是 HeapAFL 与各个基准模糊测试方法多次重复实验的统计分析结果.

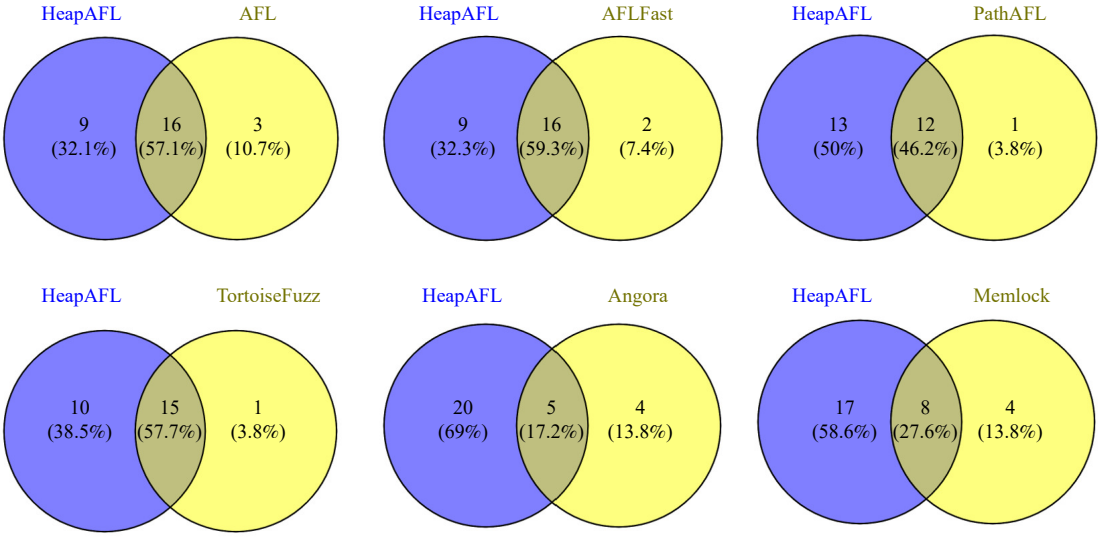

Fig. 7 Comparison of the discovered heap-based vulnerabilities of HeapAFL and each fuzzer

图 7 HeapAFL 与各基准模糊测试工具挖掘堆内存错误类漏洞比较

Table 7 Comparison Experiment of HeapAFL and HeapAFL Combined with ASAN

表 7 HeapAFL 与 HeapAFL 结合 ASAN 的对比实验

测试程序	HeapAFL		HeapAFL-ASAN	
	漏洞数量	每秒测试样例的数量	漏洞数量	每秒测试样例的数量
binutils	3	833.10	2	202.02
gpac	3	1 065.17	2	245.39
jpegoptim	1	0.63	1	0.27
mjs	6	261.50	7	274.46
yasm	11	299.27	6	194.78
mxml	1	475.74	3	443.40
合计	25	489.24	21	226.72

触发概率. 本文在 6 个真实程序组成的测试集上对比了 6 个相关模糊测试工作，实验结果表明，HeapAFL 在崩溃发现效率和漏洞挖掘数量上均优于基准测试工作. 目前 HeapAFL 发现了 6 个堆内存错误类漏洞，并获得了 2 个新的 CVE 编号.

作者贡献声明：余媛萍提出算法思路和实验方案，完成实验并撰写论文；苏璞睿提出指导意见并修改论文.

参 考 文 献

[1] Szekeres L, Payer M, Wei Tao, et al. Sok: Eternal war in memory [C] //Proc of the 34th IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2013: 48−62

[2] Mitre Corporation. 2022 CWE top 25 most dangerous software weaknesses[EB/OL]. 2022[2022-08-26]. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[3] Zalewski M. American fuzzy lop [EB/OL]. 2017[2022-08-26]. https://lcamtuf.coredump.cx/afl/

[4] Serebryany K. Libfuzzer-A library for coverage-guided fuzz testing[EB/OL]. 2015[2022-11-28]. https://github.com/llvm-mirror/llvm/blob/master/docs/LibFuzzer.rst

[5] Zhu Xiaogang, Wen Sheng, Camtepe S, et al. Fuzzing: A survey for roadmap[J]. ACM Computing Surveys, 2022, 54(11): 1−36

[6] Coppik N, Schwahn O, Suri N. MemFuzz: Using memory accesses to guide fuzzing[C] //Proc of the 12th IEEE Conf on Software Testing, Validation and Verification (ICST). Piscataway, NJ: IEEE, 2019: 48−58

[7] Wang Yanhao, Jia Xiangkun, Liu Yuwei, et al. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization[C/OL] //Proc of the 27th Network and Distributed System Security Symp(NDSS 2020). Reston, VA: The Internet Society. 2020 [2022-11-28]. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24422-paper.pdf

[8] Yan Shengbo, Wu Chenlu, Li Hang, et al. PathAFL: Path-coverage assisted fuzzing[C] //Proc of the 15th ACM Asia Conf on Computer and Communications Security. New York: ACM, 2020: 598−609

[9] Chen Peng, Chen Hao. Angora: Efficient fuzzing by principled search[C] //Proc of the 39th IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 711−725

[10] Wen Cheng, Wang Haijun, Li Yuekang, et al. Memlock: Memory usage guided fuzzing[C] //Proc of the 42nd ACM/IEEE Int Conf on Software Engineering. New York: ACM, 2020: 765−777

[11] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain[J]. IEEE Transactions on Software Engineering, 2019, 45(5): 489−506

[12] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32−44

[13] Wang Haijun, Xie Xiaofei, Li Yi, et al. Typestate-guided fuzzer for discovering use-after-free vulnerabilities[C] //Proc of the 42nd IEEE/ACM Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2020: 999−1010

[14] Yang Ke, He Yeping, Ma Hengtai, et al. Guiding directed grey-box fuzzing by target oriented coverage[J]. Journal of Software, 2021, 33(11): 3967−3982 (in Chinese)
(杨克, 贺也平, 马恒太, 等. 有效覆盖引导的定向灰盒模糊测试[J]. 软件学报, 2021, 33(11): 3967−3982)

[15] Nguyen M D, Bardin S, Bonichon R, et al. Binary-level directed fuzzing for use-after-free vulnerabilities[C] //Proc of the 23rd Int Symp on Research in Attacks, Intrusions and Defenses (RAID 2020). Berkeley, CA: USENIX Association, 2020: 47−62

[16] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C] //Proc of the Int Symp on Code Generation and Optimization(CGO 2004). Piscataway, NJ: IEEE, 2004: 75−86

[17] Serebryany K, Bruening D, Potapenko A, et al. AddressSanitizer: A fast address sanity checker[C] //Proc of the USENIX Annual Technical Conf (USENIX ATC 12). Berkeley, CA: USENIX Association, 2012: 309−318

Yu Yuanping, born in 1994. PhD. Her main research interests include system security, vulnerability discovery, and vulnerability analysis.
余媛萍, 1994 年生. 博士. 主要研究方向为系统安全、漏洞挖掘和漏洞分析.

Su Purui, born in 1976. PhD, professor, PhD supervisor. His main research interests include system security, malicious analysis, and vulnerability discovery.
苏璞睿, 1976 年生. 博士, 教授, 博士生导师. 主要研究方向为系统安全、恶意代码分析和漏洞挖掘.