计算机科学
COMPUTER SCIENCE

**结合动态分析的内存安全漏洞模糊测试方法**

殷家乐, 陈哲

引用本文

殷家乐, 陈哲. 结合动态分析的内存安全漏洞模糊测试方法[J]. 计算机科学, 2025, 52(11): 382-389.

YIN Jiale, CHEN Zhe. Dynamic Analysis Based Fuzz Testing for Memory Safety Vulnerabilities[J]. Computer Science, 2025, 52(11): 382-389.

---

**相似文章推荐（请使用火狐或 IE 浏览器查看文章）**

**Similar articles recommended (Please use Firefox or IE to view the article)**

基于神经元覆盖指标的测试用例生成优化研究

Research on Optimization of Test Case Generation Based on Neuron Coverage Index

计算机科学, 2025, 52(11): 339-348. https://doi.org/10.11896/jsjkx.240900006

嵌入式软件模糊测试研究综述

Survey on Fuzzing of Embedded Software

计算机科学, 2025, 52(7): 13-25. https://doi.org/10.11896/jsjkx.240800068

CNFED:一种基于条件数的浮点表达式误差检测工具

CNFED:An Error Detection Tool for Floating-point Expressions Based on Condition Number

计算机科学, 2025, 52(6A): 240800070-8. https://doi.org/10.11896/jsjkx.240800070

基于配置引导的实时Linux内核靶向模糊测试

Configuration-guided Directed Kernel Fuzzing for Real-time Linux

计算机科学, 2025, 52(6A): 240400161-8. https://doi.org/10.11896/jsjkx.240400161

基于SDR句嵌入的挖矿恶意软件早期检测方法

Cryptomining Malware Early Detection Method Based on SDR

计算机科学, 2024, 51(12): 303-309. https://doi.org/10.11896/jsjkx.231200041

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.241000003

# 结合动态分析的内存安全漏洞模糊测试方法

殷家乐[1]　陈　哲[1,2]
1 南京航空航天大学计算机科学与技术学院　南京 211106
2 软件新技术与产业化协同创新中心　南京 211106
(yinjiale1215@163.com)

**摘　要**　C语言编写的系统往往包含潜在的内存漏洞。模糊测试集成动态分析工具可以挖掘内存漏洞,但会引入巨大的性能开销。同时,当前流行的模糊测试方法更多地专注于提高整体代码的覆盖率,而高效触发已覆盖代码的内存漏洞也是一项重要能力。为此,改进了动态分析工具Movec并与AFL结合,创新工作主要为使用指针元数据引导模糊测试高效挖掘内存漏洞。核心步骤包括使用源码级别的哈希表和二级树管理指针元数据,以减少模糊测试结合动态分析的缺页中断;然后在汇编级别上取消对动态分析代码的覆盖率插桩,以减少冗余插桩对覆盖率计算的影响;随后增加最小指针边界距离,内存分配峰值指标引导模糊测试高效挖掘缓冲区溢出、内存分配失败漏洞;最后优化种子队列的筛选逻辑,精简队列规模并提高内存漏洞相关种子的优先级。对CVE程序进行实验,Movec结合AFL的执行吞吐量为原生的54%,而Asan,Msan只有10%和4%。对比先进模糊测试器,可以在更短的时间内挖掘内存漏洞,平均减少48.4%的时间消耗。

**关键词**:内存安全;动态分析;模糊测试;源代码插桩
**中图分类号**　TP311

# Dynamic Analysis Based Fuzz Testing for Memory Safety Vulnerabilities

YIN Jiale[1] and CHEN Zhe[1,2]
1 College of Computer Science and Technology,Nanjing University of Aeronautics and Astronautics,Nanjing 211106,China
2 Collaborative Innovation Center of Novel Software Technology and Industrialization,Nanjing 211106,China

**Abstract**　Systems written in C often contain potential memory vulnerabilities. Fuzz testing integrated with dynamic analysis tools can uncover memory vulnerabilities but introduce significant performance overhead. Meanwhile,current popular fuzz testing methods focus more on improving overall code coverage,while efficiently triggering memory vulnerabilities in already covered code is also an important capability. To this end,the dynamic analysis tool Movec is improved and combined with AFL,with the innovative work primarily using pointer metadata to guide fuzz testing for efficient memory vulnerability detection. The core steps include using a source-level hash table and secondary tree to manage pointer metadata to reduce page faults caused by combining fuzz testing with dynamic analysis. Then it removes coverage instrumentation of dynamic analysis code at the assembly level to reduce the impact of redundant instrumentation on coverage calculation. Subsequently it adds minimum pointer boundary distance and memory allocation peak indicators to guide fuzz testing in efficiently detecting buffer overflows and memory allocation failure vulnerabilities,and finally optimizes the seed queue filtering logic to streamline queue size and prioritize seeds related to memory vulnerabilities. Experiments on CVE programs show that the execution throughput of Movec combined with AFL is 54% of native,while Asan and Msan are only 10% and 4%,respectively. Compared with advanced fuzzers,it can uncover memory vulnerabilities in a shorter time,reducing the time consumption by an average of 48.4%.

**Keywords**　Memory safety,Dynamic analysis,Fuzzing,Source-level instrumentation

## 1　引言

C语言具有对底层内存控制的能力,但开发人员容易忽略对内存访问合法性的检测,导致在运行过程中引发内存安全性错误。在CVE(Common Vulnerabilities and Exposures)报告中,常见的内存安全性错误包括缓冲区溢出、指针释放后使用、重复释放指针以及内存泄漏等。

为了保证软件的可靠性,现今有越来越多的技术开始关

到稿日期:2024-10-08　返修日期:2024-12-13
基金项目:国家自然科学基金(62172217);国家自然科学基金委员会-中国民航局民航联合研究基金(U1533130);CCF-华为胡杨林基金形式化专项资助
This work was supported by the National Natural Science Foundation of China(62172217),Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China(U1533130) and CCF-Huawei Populus Euphratica Fund Formalization Project.
通信作者:陈哲(zhechen@nuaa.edu.cn)

注检测软件系统的内存安全性，其中动态分析技术[1-4]是指在目标程序运行过程中，对程序中的特定信息进行提取并分析。在内存安全场景下，这些特定信息可以是指针的安全上下界、内存分配大小等，以此检测程序运行时出现的内存漏洞。模糊测试是一种软件测试技术，它对程序输入进行比特级别的变异，将改造后的程序输入作为参数提供给目标程序多次运行，在程序运行过程中监测程序的状态，最终输出可以使得程序崩溃的测试用例。研究人员通过分析测试用例和程序崩溃时的上下文状态来定位程序漏洞。

结合动态分析的模糊测试是当前一种主流方法[5-6]，动态分析技术可以将不导致程序崩溃的内存安全性错误报告为程序异常，辅助模糊测试工具生成相应的测试用例，但是目前模糊测试支持集成的主流动态分析工具会引入巨大的性能开销。同时，基于覆盖率的模糊测试工作结合动态分析技术尽可能地生成拥有更高程序覆盖率的测试用例，挖掘深层代码分支下的程序漏洞。如果包含内存漏洞的代码已经被覆盖到，那么，如何高效触发其内存漏洞也是一项重要能力[7]。鉴于动态分析拥有分析程序运行时行为和状态的能力，可以将其收集到的内存安全相关信息作为模糊测试运行过程中的额外反馈信息，以生成更具针对性的测试用例，提高对内存漏洞的检测效率。因此本文将动态分析工具 Movec 和模糊测试工具 AFL 结合，目标是显著降低模糊测试结合动态分析后引入的性能开销，并使用指针元数据信息引导模糊测试高效挖掘 C 程序的内存漏洞。本文的主要贡献如下：

1）提出了一种结合动态分析的内存安全漏洞模糊测试方法，通过动态分析维护程序运行时的指针值和指针合法边界的距离、已分配内存值两种元数据信息，并以它们作为额外反馈信息指导模糊测试收集、精简种子队列的过程，提高模糊测试对内存漏洞的挖掘效率。

2）发现目前流行的内存安全检测工具 AddressSantizer[8]（Asan）等结合模糊测试会产生大量的缺页中断，从而严重影响模糊测试的运行性能。本文实现的智能状态指针算法[9-11]使用哈希表、二级树维护指针的元数据，有效缓解了性能损耗问题。

3）发现基于覆盖率的模糊测试工具会对动态分析插桩代码进行覆盖率插桩，从而严重影响模糊测试过程中对目标程序新代码路径的探索效率。本文通过在覆盖率插桩阶段定向识别并跳过动态分析的插桩代码，显著降低了对目标程序覆盖率信息计算的影响。

4）基于以上方法实现了源代码级别插桩的动态分析工具 Movec，改进了 AFL，并将两者结合实现针对 C 程序内存漏洞的模糊测试框架。使用 CVE 报告作为实验集，对比 AFL 开启 Asan 选项，本文实现的模糊测试框架减少了平均 92% 的缺页中断次数，并且对比先进模糊器，在更短的时间内揭露了 CVE 报告中的内存漏洞。

## 2 背景和相关工作

### 2.1 动态分析工具 Movec

Movec 是一个基于智能状态指针方法的源代码级动态分析工具，对于程序中的每一个指针变量，Movec 都维护着一个与之对应的指针元数据（Pointer Metadata，pmd）来存储指针的状态信息，包括当前指针的合法上下界、指针指向的内存对象状态、被引用的计数。内存对象的状态即编程语言角度下的内存状态，包括无效类型、堆栈类型、全局类型等。内存对象的引用计数是指程序运行过程中所有指向此内存对象的指针数量，它们共享着状态节点。当最后一个指向此内存对象的指针解除引用后，引用计数会归零，即没有指针指向此内存对象，程序也没有入口对此内存对象执行内存释放，因此如果此时内存对象的状态仍然是合法的，说明程序出现了内存泄漏问题。如果程序访问的指针值在此指针 pmd 的合法上下界之外，会报出空间错误；如果在对指针解引用时发现内存对象的状态是非法的，会报出时间错误。智能状态指针的一系列元数据可以作为衡量程序是否倾向于发生内存错误的指标，来引导模糊测试收集和变异测试用例的过程。

### 2.2 基于覆盖率的模糊测试

AFL（American Fuzzy Lop）是覆盖率引导模糊测试的代表工具。它记录测试用例的代码覆盖率，收集并变异发现新程序执行路径的测试用例并多次运行，以此不断提高对目标程序的覆盖率，增加发现漏洞的概率，同时输出导致程序异常的测试用例。其大致流程如图 1 所示。

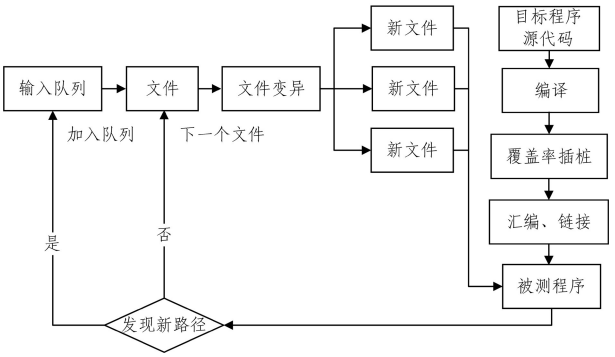


图 1 AFL 流程图

Fig. 1 Flowchart of AFL

目前，在 AFL 基本思想上进行拓展的模糊测试工作逐年增加。AFLFast[12]将模糊测试过程建模为一个马尔可夫链，将测试用例变异导致的程序执行路径转移概率视作马尔可夫链上的状态转移概率，以此优化 AFL 的能量分配策略，使得模糊测试更倾向于探索低访问频率的代码区域。FairFuzz[13]识别覆盖过稀有分支的测试用例，提高其变异优先级，并进行变异测试，从而保留测试用例中能够覆盖稀有分支的关键字节。PATA[14]利用路径感知的污点分析[15-16]技术确定测试用例的关键字节，对关键字节进行变异以覆盖程序的更多路径。

基于覆盖率的模糊测试结合动态分析，可以提高对整体程序的覆盖率，但是当内存漏洞所在代码被覆盖到时，如何高效生成可以触发此处内存漏洞的测试用例也是一项重要能力。以 AFL 为例，当测试用例覆盖到新的程序路径后，即命中程序的分支跳转处、分支跳转目的地、函数入口 3 种基本块会被收集为有价值种子并变异为新的测试用例。重复上述操作，以发现更多的路径。其中的一处细节是，测试用例被收集为有价值种子的条件，不仅是命中先前没有被命中过的新路

径。AFL 在运行过程中记录上述基本块被命中的次数，并且对命中次数预设了对应的数量级，只要当前测试用例产生了比先前所有测试用例更高的数量级时，就会被收集。表 1 列出了具体的映射规则。例如基本块先前被命中 3 次，对应的数量级为 4，当前测试用例使得基本块被命中 4 次，对应的数量级为 8；则产生了更高的命中数量级，则当前测试用例会被收集为有价值种子。如果基本块先前被命中 4 次，对应的数量级为 8；当前测试用例使得基本块被命中 7 次，对应的数量级为 8，没有产生更高的命中数量级，则当前测试用例不会被收集。

表 1 命中次数与对应数量级

Table 1 Number of hits and corresponding magnitudes

| 命中次数 | 数量级 |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 4 |
| 4，…，7 | 8 |
| 8，…，15 | 16 |
| 16，…，31 | 32 |
| 32，…，127 | 64 |
| 128，…，+∞ | 128 |

AFL 只根据覆盖率信息盲目探索新的程序代码执行路径[17]，并不能高效地生成可以触发内存漏洞的程序输入。图 2 中示例程序遍历字符串 str，计算其中字符与字符 A 的 ASCII 值的差，并且存放在数组 arr 中。如果不对字符串长度做检查，字符串长度大于数组长度时，会引起缓冲区溢出错误。

```
void func(char * str,int * arr,int i){
while(str[i]!='\0'){
arr[i++]=str[i++]-65;}}
```

图 2 存在缓冲区溢出漏洞的程序

Fig. 2 Program with buffer overflow vulnerability

假设数组 arr 的长度为 50，模糊测试提供的初始输入为“ABC”，那么 while 分支的基本块会被命中 4 次，while 分支出口的基本块被命中 1 次。AFL 也因为两个基本块先前没有被命中过，会将此输入加入价值种子队列。字符串 ABC 输入被变异为长度为 10 的字符“ABCCE….”，运行时会命中 while 分支 11 次。根据 AFL 的命中次数分类逻辑，命中 11 次对应的数量级 16 大于命中 4 次对应的数量级 8，因此输入会被加入价值种子队列。后续输入继续被变异为拥有 11，12，13，14 个字符的字符串。由于存在命中次数分类，因此 11 到 14 个字符的字符串输入都不会被加入价值种子队列，因为在 AFL 看来，命中 12 到 15 次与命中 11 次都属于同样的命中数量级 16，即 11 到 14 个字符的字符串输入将被丢弃。因此后续 AFL 将不会高效地产生或者不产生更长的字符串输入，也就不会产生大于等于 50 的 $i$ 变量引发缓冲区溢出错误，从而影响内存漏洞的揭露效率。

### 2.3 AFL 和 Asan

AFL 支持集成 Asan 等内存安全检测工具，Asan 将内存漏洞报告为程序崩溃，使得 AFL 输出揭露内存漏洞的测试用例。

这种结合会产生两种代价：一是会降低模糊测试整体的运行性能[18]；二是 Asan 会产生冗余插桩，影响覆盖率信息的计算。

降低模糊测试运行性能的主要原因是内存管理耗时的增加。AFL 使用 fork server 机制，可以避免不必要的 execve 函数调用，提高目标程序的装载速度。同时，写时拷贝可以进一步优化 fork+execve 产生的不必要的页面拷贝。而各类 Sanitizer 开辟了大体积的影子内存空间用来装载内存元数据信息，并且与程序内存空间保持一定的偏移量。Asan 每次执行时，额外使用了 20TB 虚拟内存，Msan 为 72TB。较大的堆内存会导致页表项稀疏，从而增加页表数量和内存管理开销。首先，fork server 在复制出目标程序进程时，由于写时拷贝的存在，父子进程共享相同的物理页面。目标程序进程在运行中修改内存对象时，可能会发生缺页中断，并且对应影子内存中此内存对象的元数据会被修改，可能再次发生缺页中断。此外，目标程序初始化时，Asan 会对影子区域的全局变量、堆区域等进行中毒操作，代表目标程序对应的内存区域还没有被分配，这样的操作也会产生大量的缺页中断。

对于 Asan 产生的冗余插桩影响覆盖率计算的问题，是因为 Asan 在 LLVM-IR 级别对原程序中访问内存的操作进行插桩，以实现对内存漏洞的检测，而这些插桩代码会被编译为汇编代码。AFL 会在汇编级别对原程序插桩覆盖率标记块的相关代码。因此，AFL 会将 Asan 的插桩代码当作原程序的一部分，对其中的函数入口、分支跳转处、分支跳转目的地等进行插桩，得到冗余的覆盖率信息，从而对 AFL 的覆盖率结果产生影响。当前 AFL 的解决办法仅仅是在开启 Asan 选项后，将整体的覆盖率插桩概率调整为原先的 1/3，以缓解 Asan 冗余插桩带来的问题，但精确度并不高，很有可能忽视原程序的代码路径而采纳 Asan 的代码路径。afl-2.52b 源代码如图 3 所示。

```
if(getenv("AFL_USE_ASAN")){
sanitizer=1;
inst_ratio /=3;}
```

图 3 AFL 启用 Asan 选项

Fig. 3 AFL enables Asan option

## 3 Movec 结合 AFL

### 3.1 指针元数据引导模糊测试

为了解决覆盖率引导的模糊测试在生成触发内存漏洞的输入时效率低下的问题，本文提出使用指针元数据作为额外反馈信息来引导模糊测试。在图 2 中的示例程序，11 到 14 个字符的字符串从引发缓冲区溢出错误的角度看，应当被加入价值种子队列，以便后续生成更长的字符串，从而揭露内存漏洞。因此本文提出将指针值与指针合法上下界的距离作为额外反馈信息。此外，还将程序运行过程中已分配的最大内存值作为额外反馈信息。这些额外反馈信息用于引导模糊测试评估是否将当前测试用例作为有价值种子收集。具体的评判标准如下。

1)指针边界距离。当测试用例使得指针值越靠近合法指针上下界时，测试用例被收集。

$$\exists\, p_i \in P, |p_i - bounds_i| < pointer_min_dist(p_i) \quad (1)$$

其中，$P$ 代表所有指针操作的指针集合，指针操作包含指针解引用、数组下标访问、结构体指针变量的成员访问等。每一处指针操作的最小指针边界距离存储于 pointer_min_dist，初始值为无穷大。每次操作指针时，计算出较小的指针距离存储于 pointer_min_dist。需要注意的是，当指针距离小于无穷大初始值时，需要更新 pointer_min_dist，但是不将测试用例收集，否则测试用例队列中会保留大量指针距离小于无穷大的测试用例。另外，更新 pointer_min_dist 的最小阈值也是值得讨论的一点，测试用例经过模糊测试中的 bitflip，arithmetic 等确定性变异策略后，有较大概率导致指针每次靠近指针合法边界的步长为指针指向对象的一个或几个单位，如图 2 所示，假设字符串 str 每次被变异后只自增一个字符的长度，arr[i++]会随之以 sizeof(int)的步长靠近 arr 的合法上界，测试用例也会被收集。测试用例队列变得庞大，导致变异出足够长度，并使得数组越界的字符串的效率随之下降。因此，式(1)更新为：

$$\exists\, p_i \in P, |p_i - bounds_i| < pointer_min_dist(p_i) - n \times sizeof(*p_i) \quad (2)$$

其中，$sizeof$ 计算指针指向对象的内存大小；$*p$ 代表解引用指针；$n$ 是可变参数，即每当指针靠近合法边界的步长大于 $n$ 个指针指向对象内存大小时，才更新 pointer_min_dist。通过对包含缓冲区溢出错误的 CVE 报告进行实验，本文将 $n$ 设置为 2 时平均效果最好。

2)已分配内存值。当测试用例使得已分配的内存值越来越大时，测试用例被收集。

$$\begin{aligned} &\sum_{i=1}^{n} alloc_i - \sum_{i=1}^{m} dealloc_i > max_cur_alloc \\ &\exists\, alloc_i \in \{malloc, calloc, realloc\} \\ &\exists\, dealloc_i \in \{free, realloc\} \end{aligned} \quad (3)$$

其中，$max_cur_alloc$ 存储着当前程序运行过程中的内存分配峰值；$alloc$ 代表 C 语言中内存分配函数的调用；$dealloc$ 代表内存释放函数的调用；$n$ 和 $m$ 代表目前为止，$alloc$ 和 $dealloc$ 的调用总数。当某一个 $alloc$ 函数被调用，则计算是否产生了比先前都更大的内存分配值。

(1)指针边界距离的实现

AFL 在运行目标程序前初始化一个共享内存数组，即最小指针距离数组 min_dist，用于存储每一处指针操作的最小指针距离。通过 Movec 源代码级插桩，在目标程序的 main 函数入口获取此共享内存数组和 AFL 建立进程间通信。由于 Movec 对每一个指针都有 pmd 记录指针元数据，可以在操作指针时很方便地获取指针的合法上下界。Movec 通过在指针解引用、访问数组下标、访问结构体指针变量的成员等操作指针的场景进行插桩，生成的 check 函数既可以检测指针操作处是否有内存漏洞，也可以进行本文提出的指针距离计算。本文使用指针操作所在行号和列号的位运算作为 key 值，将行号作为 key 值的低 32 位，列号作为 key 值的高 32 位，再通过哈希函数_PRFhashtbl_hash 将 key 值转换为 min_dist 的下标。当前指针值与边界值的距离比 min_dist 存储的旧值更小时，会更新其为较小值。每次程序运行到此处都会进行同样的操作，直到程序运行结束，得到一个包含所有最小指针距离的 min_dist 数组，传递给 AFL。代码如图 4 所示。

```
void* PRFcheck(PRFpmd* pmd,PTR* ptr,size_t key){
/*计算当前指针值与合法指针上界的距离*/
ptrdiff_t dist=pmd->bound-ptr;
int idx=PRFhashtbl_hash(key);
/*更新此处指针操作对应的最小指针距离*/
min_dist[idx]=min(dist,min_dist[idx]-2*sizeof(*ptr));
return ptr;}
```

图 4 check 函数定义

Fig. 4 Definition of check function

AFL 在程序运行结束之后开始处理覆盖率信息和 Movec 提供的 min_dist 数组。覆盖率信息仍然是第一优先级，因为只有探索到新的程序路径才有机会发现内存漏洞。因此，当本次程序执行后发现某些分支首次被命中或者被命中次数产生了更高的数量级，则一定会将本次测试用例收集为价值种子；若没有，则再来观察本次程序执行是否有若干个指针更加靠近其合法边界，如果有则将本次测试用例收集；若两者都没有发生，则丢弃本次测试用例。代码如图 5 所示。

```
int save_if_interesting(FILE* case){
/*检测当前测试用例是否发现了新执行路径*/
hnb=has_new_bits(case);
/*测试用例是否发现了更小的指针距离*/
hnm=has_new_min(case);
if(!hnb && !hnm) return 0;
/*将测试用例加入待变异种子队列*/
add_to_queue(case);}
```

图 5 收集测试用例的函数定义

Fig. 5 Definition of the function to collect test cases

如此一来，在覆盖率引导的基础上，指针最小距离可以作为额外反馈信息，以增强模糊测试对缓冲区溢出错误的探索能力。

(2)已分配内存值的实现

已分配的内存值在 AFL 端的实现方式与指针最小距离类似，声明一个共享内存变量 max_alloc 用于 Movec 和 AFL 进行信息反馈。Movec 对内存分配相关库函数，如 malloc，realloc，alloc，free，都进行了包装，可以获取到分配和释放的内存值。包装函数定义如图 6 所示。

```
void *PRFmalloc(size_t size){
void *ret=malloc(size);
cur_alloc+=size;
/*更新运行时内存分配峰值*/
*max_alloc=max(*max_alloc,cur_alloc);
return ret;}
void PRFfree(_PRFpmd *pmd,void *ptr){
free(ptr);
cur_alloc=cur_alloc-(pmd->bound-pmd->base);}
```

图 6 malloc 和 free 的包装函数定义

Fig. 6 Definition of the wrapper functions for malloc and free

PRFmalloc 这类包装函数可以获取对应 malloc 函数的参数，即分配内存量，以此更新 cur_alloc。库函数 free 的参数中不存在即将释放的内存值，但包装函数 PRFfree 提供了即将被释放指针的 pmd，通过计算其上下界距离，可以得出即将释放的内存对象占用了多少内存值，以此更新 cur_alloc。

### 3.2 消除 Movec 冗余插桩

与 Asan 情况相同，Movec 的插桩代码也会被 AFL 进行覆盖率插桩。本文从 AFL 的汇编级别插桩来消除冗余插桩。首先可以确定 afl-as 会对目标程序的 3 种代码块进行覆盖率插桩：

1）函数入口；

2）分支跳转的目的基本块；

3）非无条件分支跳转处。

而 Movec 导致冗余插桩代码的原因主要有以下 3 点：

1）智能状态指针函数；

2）普通包装函数；

3）内联包装库函数。

智能状态指针函数用于在程序运行时创建、修改、销毁指针的 pmd，跟踪对应的指针并进行内存安全检测。普通包装函数是指对源代码的函数进行包装，提取指针类型的函数返回值、函数参数，将其 pmd 传递给原函数，再执行原函数并返回原函数的返回值。智能状态指针函数和普通包装函数会产生与原程序无关的分支，并且其函数入口也会增加一个覆盖点的插桩。示例汇编程序如图 7 所示。规避上述两种函数导致的冗余插桩的方法主要是识别并跳过，当 AFL 识别到当前分支跳转和分支跳转目的基本块或函数入口属于上述两种 Movec 函数时，则取消覆盖率插桩。Movec 的函数名拥有特定前缀，汇编代码中的 C 函数开头和结尾分别有. LFB〈num〉和. LFE〈num〉标识，本文将从 Movec 函数名到. LFE〈num〉之间的代码识别为 Movec 函数体，以此取消对 Movec 冗余代码的覆盖率插桩。

```
__RV_check:
.LFB221        ;AFL 在函数入口插桩
jnb.L3853      ;AFL 在分支跳转处插桩
.L3853:        ;AFL 在分支跳转目的基本块插桩
.LFE221
```

图 7 包装函数的汇编代码

Fig. 7 Assembly code for wrapper function

内联库包装函数是由于 Movec 无法获取库函数的源代码进行插桩，因此为库函数预设库包装函数。库包装函数是 inline 函数，原程序调用库包装函数时，会将其内联嵌入原程序，导致原程序函数中包含了库包装函数的代码，这些代码中的分支跳转同样会影响 AFL 的覆盖率插桩。示例汇编代码如图 8 所示。

```
.LVL3508:          ;内联的库包装函数代码
.loc 1 2501 7 is_stmt 1 view.LVU13808
cmpq          9640(%rbx),%rsi
```

图 8 内联库包装函数的汇编代码

Fig. 8 Assembly code for inline library wrapper function

为了规避此类冗余插桩，本文的解决方法是通过识别此类代码，进而取消覆盖率插桩。在汇编代码中，内联代码块会使用. loc 信息来表示文件来源。如第 2 行的. loc 1 2501，其中 1 代表此代码块的源文件 ID。而此 ID 和对应的文件名会在汇编代码头部进行罗列，通过预存储此映射信息，在后续 AFL 插桩中识别当前. loc 代码块是否属于 Movec 的库包装函数文件，如果是则取消覆盖率插桩，直到下一个. L〈num〉的出现。通过消除 Movec 的冗余插桩，最终产生的覆盖率插桩相比原生 AFL 插桩只多出 1 倍。

### 3.3 避免缺页中断带来的性能损耗

开启 AFL 的 Sanitizer 选项后，可以使用 Asan 等动态分析工具辅助模糊测试检测内存漏洞，但这会引入大量额外的缺页中断，导致模糊测试性能降低，原因是 Asan 的影子内存和 red zone 设计。Movec 使用基于智能状态指针的方法对目标程序进行内存安全检查，同样使用 disjoint-metadata 方法避免改变目标程序原有的内存布局。但 Movec 在实现方面没有使用影子内存这种会创建大体积虚拟内存的方案，而是在源代码级别使用两种数据结构来存储指针元数据，即哈希表和二级树，并默认使用二级树。由于指针变量的地址值是唯一的，Movec 使用指针的地址值作为 key。

在哈希表模式下，Movec 使用链地址法处理冲突，并且当表元素数量达到桶数的 1/3 以上时，将桶数扩容为二倍，哈希表初始容量为 1 024 * 8 字节。

$$index = hash(addr, capacity) \tag{3}$$

在二级树模式下，Movec 将 key 从低位的 26 到 48 位作为一级索引值，从低位的 4 到 25 位作为二级索引值。当一级索引值对应的二级表缺失时，才会给二级表分配内存。一级表和二级表的初始容量分别为 $2^{23}$ 字节和 $2^{22}$ 字节。

$$primaryIndex = (addr \gg 25)0x7FFFFF \tag{4}$$

$$secondaryIndex = (address \gg 3)0x3FFFFF \tag{5}$$

相比于 Asan 的做法，Movec 在程序初始化期间没有对较大堆区域进行访问操作，而是在程序运行到指针变量创建时申请堆内存。同时，Movec 没有因创建较大堆内存而导致页表项稀疏的情况，访问指针元数据时，大大降低了缺页中断次数的发生。本文使用原生 AFL、AFL＋Asan，以及 AFL＋Movec 对 4 个 CVE 报告中的程序进行模糊测试，并且采集在执行目标程序 1 000 次后发生的缺页中断次数，如表 2 所列。对比可知，AFL 在开启 Asan 选项后，最大产生相比原生 AFL 29 倍的缺页中断次数，而 AFL 结合 Movec 则是 2 倍。

表 2 缺页中断次数对比

Table 2 Comparison of page fault counts

| CVE | AFL | AFL＋Asan | AFL＋Movec |
|---|---|---|---|
| 2019-14275 | 94 390 | 1 755 861 | 236 725 |
| 2022-30045 | 59 856 | 1 487 568 | 89 030 |
| 2021-45385 | 47 407 | 1 405 590 | 69 057 |
| 2021-44956 | 72 881 | 1 526 870 | 165 926 |

### 3.4 AFL 种子队列精简逻辑优化

AFL 在生成新的测试用例时，在原有的逻辑中首先会判断其是否发现了新的程序执行路径，如果没有则会丢弃。通过本文的优化后，无论测试用例是否发现了新的程序执行路

径，都会将更容易引发内存漏洞的测试用例加入种子队列，但这样会导致种子队列的规模变得庞大，降低模糊测试的运行性能。因此需要对种子队列进行精简。AFL 通过 update_bitmap_score 和 cull_queue 两个函数进行种子队列的精简与优化。基本思想是，使用种子的运行时间和文件大小的乘积作为种子的评价指标，在某一条程序执行路径上使用评价最高的种子，此种子会替换掉其覆盖到的所有程序路径上的被淘汰种子，以此形成一个精简后的种子队列。在此基础上，本文在判断是否将新生成的测试用例加入种子队列时，增加一层预筛选逻辑。

1）AFL 发现具有新路径的测试用例后，一定会将其加入价值种子队列。

2）AFL 发现测试用例不具有新路径，但更倾向于引发内存漏洞时，会替换具有相同路径但不容易引发内存漏洞的种子。

此外在精简种子队列过程中，由于 AFL 未考虑种子是否更容易引发内存漏洞，因此本文将指针值与指针边界的距离、内存分配峰值作为额外因素加入种子的评价指标计算过程。通过此种子队列精简策略，价值种子队列中可以始终保留具有新路径且更容易引发内存漏洞的种子，有利于后续变异出同种性质的种子，并且也降低了种子队列的规模，避免对模糊测试的性能损耗。种子队列更新如图 9 所示。

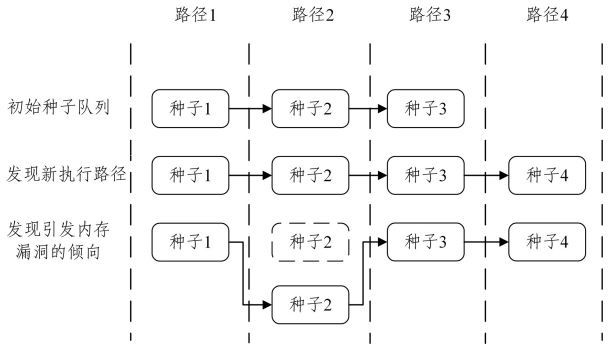


图 9　种子队列更新图

Fig. 9　Seed queue update diagram

## 4　实验与结论

本文参考了 Evaluating Fuzz Testing[19] 关于模糊测试实验的意见，从 4 个方面准备本次实验。

1）基准动态分析工具与模糊测试器

选取 Asan，SoCets，Valgrind 动态分析工具为基准线，验证 Movec 检测内存漏洞的有效性。选取支持与主流模糊测试器集成的动态分析工具 Asan 和 Msan 为基准线，验证 Movec 与模糊测试器结合使用时可以显著缓解缺页中断和冗余插桩导致的性能开销。选取 AFL，AFL＋＋[20]，AFLFast，FairFuzz 先进的模糊测试器为基准，验证指针元数据引导方法可以高效挖掘 C 程序内存漏洞。

2）基准测试集

本文方法专注于检测 C 程序内存漏洞，因此选取 22 个包含内存漏洞的 C 程序 CVE 作为实验集，真实的漏洞报告更具有说服力。

3）评价指标

本文将改进后的 Movec 与模糊测试结合进行针对 C 程序内存漏洞的挖掘，首先使用检测内存漏洞的有效性作为评价指标进行实验。Movec 使用源码级插桩实现的二级树维护指针元数据，并通过前缀识别的方式解决冗余插桩的问题，从而显著缓解缺页中断和冗余插桩导致的性能开销问题，因此使用模糊测试过程每秒执行目标程序的次数为评价指标进行实验。Movec 实现了最小指针边界距离、内存分配峰值的指针元数据引导方法，增强模糊测试针对 C 程序内存漏洞的挖掘效率，因此使用模糊测试框架发现 CVE 报告中的内存漏洞耗时作为评价指标进行实验。

4）实验时间与环境配置

使用 5 * 24 h 的方式运行每个 CVE 报告的模糊测试。因为模糊测试存在随机突变过程，模糊测试的结果会产生抖动的情况，所以使用一个较长的时间作为实验时间，并且运行多次来降低结果的随机性。本实验平台为 64 位 Ubuntu22.04 系统，处理器为 AMD RYZEN 4000，内存为 16 GB，编译器为 gcc-12.3.0。

### 4.1　内存漏洞检测有效性实验

动态分析工具识别不会导致程序崩溃的内存漏洞，并上报给模糊测试器，因此内存漏洞检测能力十分重要。本文在 22 个 CVE 报告上进行实验，比较 Valgrind（memcheck），Asan，SoCets 这 3 个工具和 Movec 针对多种内存漏洞类型的检测能力，实验结果如表 3 所列，Y 代表成功检测，N 代表无法检测。

表 3　有效性对比

Table 3　Validity comparison

| 漏洞类型 | Asan | SoCets | Valgrind（memcheck） | Movec |
|---|---|---|---|---|
| 缓冲区溢出 | Y | Y | Y | Y |
| 内存分配失败 | Y | N | N | Y |
| 子对象越界 | N | N | N | Y |
| 指针释放后使用 | Y | Y | Y | Y |
| 指针双重释放 | Y | Y | Y | Y |
| 非法指针解引用 | N | Y | Y | Y |
| 空指针解引用 | Y | Y | Y | Y |
| 内存泄漏 | Y | N | Y | Y |

由实验结果可知，Movec 可以检测出表 3 中所有类型的内存漏洞。其中子对象越界代表指向结构体成员的指针被解引用到结构体其他成员的内存空间，由于 Movec 能维护每一个指针的合法上下界，因此可以有效检测此类错误，而 Asan 和 Valgrind 只检测指针是否指向整个结构体可访问内存空间内，无法识别结构体成员之间的访问越界。

### 4.2　目标程序执行吞吐量实验

为了衡量模糊测试运行性能，本文使用原生 AFL、AFL＋Movec、AFL 集成 Asan，Msan 两种主流动态分析工具进行对比，对比指标为 24 小时运行平均目标程序每秒的执行次数。实验结果如图 10 所示。对于选取的 5 个 CVE 报告程序，平均运行速度由快到慢依次为 Native，Movec，Asan，Msan。结合 Movec 后为原生的 54%，结合 Asan 后为 10%，结合 Msan 后为 4%。

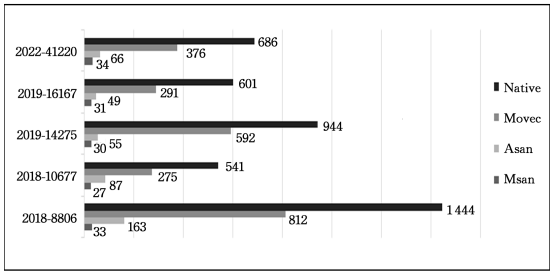


图 10 平均执行速度(运行次数/秒)对比

Fig. 10 Comparison of average execution speed(execs/sec)

### 4.3 内存漏洞挖掘性能实验

为了验证所提模糊测试框架挖掘 C 程序内存漏洞的效率，此实验将其与 AFL，FairFuzz，AFLFas，AFL++[20] 4 个模糊测试器开启 Sanitizer 进行对比，对比指标为揭露 CVE 报告内存漏洞的耗时。实验结果如表 4 所列，其中 NaN 代表模糊测试长时间未揭露特定漏洞。

由实验结果可知，本文提出的模糊测试框架可以比其余 4 种模糊测试器更快地挖掘大部分 C 程序内存漏洞。在本文模糊测试框架领先的 9 个 CVE 报告中，对于每一个报告，选取除了 AFL 的最慢对标工具计算耗时减少比例，再减去最高值和最低值并计算平均值，可知平均减少了 48.4%的挖掘耗时。其中表现稍差的 2021-44956 和 2022-41220 报告，分析原因为除了发现新路径的测试用例，因为指针元数据指标而被收集的测试用例也加入了种子队列，增加了模糊测试对种子队列筛选、能量分配、文件变异的耗时；而 2023-1010 耗时过多、2021-36531 没有发现内存漏洞，是因为上述指针元数据测试用例影响了代码覆盖率。所以，探索新的代码分支要保证为第一优先级。

表 4 揭露内存漏洞时间对比

Table 4 Comparison of memory vulnerability exposure times

| CVE 测试集 | 类型 | Movec+AFL | AFL++ | AFLFast | FairFuzz | AFL |
|---|---|---|---|---|---|---|
| 2018-10677 | 缓冲区溢出 | **0.99 h** | 2.89 h | 3.01 h | 3.22 h | 3.43 h |
| 2018-11576 | 缓冲区溢出 | **0.017 h** | NaN | NaN | NaN | 0.021 h |
| 2018-14052 | 缓冲区溢出 | **0.0006 h** | 3.21 h | 3.04 h | 3.33 h | 5.63 h |
| 2023-1010 | 内存分配失败 | **0.00001 h** | NaN | NaN | NaN | NaN |
| 2019-7581 | 内存分配失败 | **0.001 h** | 0.072 h | 0.12 h | 0.11 h | 0.038 h |
| 2019-16167 | 内存分配失败 | **13.40 h** | 16.12 h | 16.82 h | 23.11 h | NaN |
| 2019-14275 | 缓冲区溢出 | **20.19 h** | NaN | NaN | 21.22 h | NaN |
| 2019-1010302 | 缓冲区溢出 | **16.22 h** | 18.75 h | NaN | NaN | NaN |
| 2021-3404 | 缓冲区溢出 | **19.55 h** | NaN | NaN | NaN | NaN |
| 2021-44956 | 缓冲区溢出 | 0.005 h | **0.002 h** | 0.003 h | 0.003 h | 0.003 h |
| 2022-41220 | 缓冲区溢出 | 0.003 h | **0.002 h** | 0.002 h | 0.015 h | 0.002 h |
| 2021-36531 | 缓冲区溢出 | NaN | **11.20 h** | NaN | NaN | NaN |
| 2023-1010 | 缓冲区溢出 | 5.68 h | 0.021 h | 0.018 h | 0.005 h | **0.002 h** |

**结束语** 针对 C 语言程序的模糊测试，提出了结合动态分析的技术和算法，采取的技术包括源代码级插桩、指针元数据。使用 CVE 报告作为测试集进行有效性和性能实验，结果表明，本文提出的动态分析与模糊测试结合的方法能够有效挖掘 C 程序的内存漏洞，并且相比现有技术具有更好的性能。

未来的工作将专注于进一步提高模糊测试的性能，降低动态分析技术带来的额外性能损耗，同时降低内存漏洞倾向因素对新执行路径探索的影响，保证模糊测试对程序代码中的新分支探索能力。

## 参 考 文 献

[1] CHEN Z, TAO C Q, ZHANG Z Y, et al. Beyond spatial and temporal memory safety[C]// Proceedings of the 40th International Conference on Software Engineering. 2018: 189-190.

[2] NETHERCOTE N, SEWARD J. Valgrind: Aprogram supervision framework[J]. Electronic notes in Theoretical Computer Science, 2003, 89(2): 44-66.

[3] NAGARAKATTE S, ZHAO J Z, MARTIN M M K, et al. SoftBound: Highly compatible and complete spatial memory safety for C[C]// Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2009: 245-258.

[4] XU W, DUVARNEY D C, SEKAR R. An efficient and backwards-compatible transformation to ensure memory safety of C programs[C]// Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Enginee-ring. 2004: 117-126.

[5] CUI Z Q, ZHANG J M, ZHENG L W, et al. A Survey of Research on Coverage-Guided Greybox Fuzzing[J]. Chinese Journal of Computers, 2024, 47(7): 1665-1696.

[6] ZHU X G, WEN S, CAMTEPE S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys, 2022, 54(11s): 1-36.

[7] WEN C, WANG H J, LI Y K, et al. Memlock: Memory usage guided fuzzing[C]// Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 765-777.

[8] SEREBRYANY K, BRUENING D, POTAPENKO A, et al. AddressSanitizer: A fast address sanity checker[C]// 2012 USENIX Annual Technical Conference (USENIX ATC 12). 2012: 309-318.

[9] CHEN Z, WANG C, YAN J Q, et al. Runtime detection of memory errors with smart status[C]// Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2021: 296-308.

[10] CHEN Z, YAN J Q, KAN S L, et al. Detecting memory errors at runtime with source-level instrumentation[C]// Proceedings of the 28th ACM SIGSOFT International Symposium on Software

Testing and Analysis. 2019:341-351.

[11] CHEN Z,YAN R,MA Y Z,et al. A smart status based monitoring algorithm for the dynamic analysis of memory safety[J]. ACM Transactions on Software Engineering and Methodology, 2024,33(4):1-47.

[12] BÖHME M,PHAM V T,ROYCHOUDHURY A. Coverage-based greybox fuzzing as markov chain[C]// Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016:1032-1043.

[13] LEMIEUX C,SEN K. Fairfuzz:A targeted mutation strategy for increasing greybox fuzz testing coverage[C]// Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. 2018:475-485.

[14] LIANG J,WANG M Z,ZHOU C J,et al. Pata:Fuzzing with path aware taint analysis[C]// 2022 IEEE Symposium on Security and Privacy(SP). IEEE,2022:1-17.

[15] CHEN P,CHEN H. Angora:Efficient fuzzing by principled search[C]// 2018 IEEE Symposium on Security and Privacy (SP). IEEE,2018:711-725.

[16] GAN S T,ZHANG C,CHEN P,et al. GREYONE:Data flow sensitive fuzzing[C]// 29th USENIX Security Symposium(USENIX Security 20). 2020:2577-2594.

[17] LIANG G C,LIAO L J,XU X,et al. Effective fuzzing based on dynamic taint analysis[C]// 2013 Ninth International Conference on Computational Intelligence and Security. IEEE,2013:615-619.

[18] BA J S,DUCK G J,ROYCHOUDHURY A. Efficient greybox fuzzing to detect memory errors[C]// Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. 2022:1-12.

[19] KLEES G,RUEF A,COOPER B,et al. Evaluating fuzz testing [C]// Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018:2123-2138.

[20] FIORALDI A,MAIER D,EIßFELDT H,et al. AFL++:Combining incremental steps of fuzzing research[C]// 14th USENIX Workshop on Offensive Technologies(WOOT 20). 2020.

**YIN Jiale**, born in 1999, postgraduate. His main research interest is verification of software.

**CHEN Zhe**, born in 1981, professor, is a member of CCF (No. 22234M). His main research interests include verification of software, software engineering and network security.

(责任编辑:喻藜)