提高 fuzzing 边覆盖率的改进方法

贾春福[1]，严盛博[1]，王志[1]，武辰璐[1]，黎航[2]
（1. 南开大学网络空间安全学院，天津 300350；2. 南开大学人工智能学院，天津 300350）

摘 要：针对 AFL 边覆盖不全、未充分利用边覆盖信息和有效字节信息的问题，提出了改进方法。首先，设计了新的种子选择算法，在一轮循环中可完全覆盖所有已发现的边；其次，按边覆盖热度对路径评分，以此调整种子的测试次数；最后，对有效字节进行更多的变异。基于上述方法实现了新的 fuzzing 工具—efuzz。实验表明，efuzz 的平均边覆盖数比 AFL 和 AFLFast 分别增加了 5%和 9%；在 LAVA-M 测试集中，efuzz 发现的漏洞数超过了 AFL；在常用软件中，efuzz 发现了 3 个新的 CVE 漏洞。所提方法可以有效提高 fuzzing 的边覆盖率、提升漏洞发现能力，具有实用性。

关键词：fuzzing 技术；漏洞；AFL；边覆盖
中图分类号：TP311.1
文献标识码：A
doi: 10.11959/j.issn.1000−436x.2019223

Method to improve edge coverage in fuzzing

JIA Chunfu[1], YAN Shengbo[1], WANG Zhi[1], WU Chenlu[1], LI Hang[2]
1. College of Cyber Science, Nankai University, Tianjin 300350, China
2. College of Artificial Intelligence, Nankai University, Tianjin 300350, China

Abstract: Aiming at the problems of incomplete edge coverage, insufficient uses of edge coverage information and valid bytes information in AFL (American fuzz lop), a novel method was proposed. Firstly, a new seed selection algorithm was introduced, which could completely cover all edges discovered in one cycle. Secondly, the paths were scored according to the frequency of edges, to adjust the number of tests for each seed. Finally, more mutations were crafted on the valid bytes of AFL. Based on the method above, a new fuzzing tool named efuzz was implemented. Experiment results demonstrate that efuzz outperforms AFL and AFLFast in the edge coverage, with the increases of 5% and 9% respectively. In the LAVA-M dataset, efuzz found more vulnerabilities than AFL. Moreever, in real world applications efuzz has found three new security bugs with CVEs assigned. The method can effectively improve the edge coverage and vulnerability detection ability of fuzzer.

Key words: fuzzing, vulnerability, AFL, edge coverage

1 引言

fuzzing[1]是一种有效的漏洞挖掘技术，它通过生成大量的测试用例来对目标程序进行测试，同时监视目标程序的运行过程，发现程序暴露出的缺陷。传统的 fuzzing 工具由于生成测试用例时过于随机，其测试存在盲目性，虽然测试速度很快，但效率很低。针对 fuzzing 存在的问题，研究者提出了很多新的技术和方法，结合覆盖引导、静态分析、动态符号执行、语法表示、动态污点分析和机器学习等多种技术，大大提高了 fuzzing 的效率，并且在现实的软件中发现了大量漏洞[2]。

收稿日期：2019−06−14；**修回日期**：2019−09−13
通信作者：王志，zwang@nankai.edu.cn
基金项目：国家自然科学基金资助项目（No.61972215, No.61702399, No.61972073, No.61872202）；天津市自然科学基金资助项目（No.17JCZDJC30500）；赛尔网络下一代互联网技术创新基金资助项目（No.NGII20180401）
Foundation Items: The National Natural Science Foundation of China (No.61972215, No.61702399, No.61972073, No.61872202), The Natural Science Foundation of Tianjin (No.17JCZDJC30500), CERNET Innovation Project (No.NGII20180401)

覆盖引导的 fuzzing 策略被广泛使用，并且已被证明是非常有效的。它尽可能地测试更多的代码分支，使程序的代码覆盖率实现最大化。目前，代码覆盖率有 2 种基本的测量方式。一种是计算基本块（BBL, basic block）覆盖数，Libfuzzer、honggfuzzy 和 VUzzer[3]都是通过程序插桩来跟踪 BBL 的覆盖信息。另一种是计算边覆盖数，AFL（American fuzzy lop）是第一个将边覆盖引入 fuzzing 的工具，其通过编译时静态插桩，当程序运行时可获取边覆盖信息，提供了比 BBL 覆盖更加精确的信息。

最近几年发表了大量基于 AFL 的研究。一类研究是定向 fuzzing。例如，AFLGo[4]和 Hawkeye[5]通过对程序的静态分析来调整种子排序和种子的测试次数，逐步引导程序到达目标点。另一类研究结合了符号执行技术。Driller[6]使用选择性的混合符号执行技术，当 AFL 被“卡住”时，调用 Angr[7]来生成合法输入，从而有效地分析大规模程序。WildFire[8]先通过单独测试程序中的函数来发现漏洞，再使用 KLEE[9]来校验这些漏洞的可行性。还有一类研究结合了人工智能技术。孙鸿宇等[10]分析了人工智能技术在安全漏洞领域的应用。Godefroid 等[11]基于神经网络的统计机器学习，生成富含格式的文件作为 AFL 的初始种子。Skyfire[12]从样本中学习一种概率上下文敏感语法（PCSG, probabilistic context sensitive grammar）来描述语法特征和语义规则，利用 PCSG 生成具有不同语法结构的种子输入，从而为处理高度结构化输入的程序生成正确、多样和不常见的种子。

然而，AFL 本身也存在一些缺陷，通过对这些缺陷的修复，可以优化上述方法的效果。例如，CollAFL[13]针对 AFL 边覆盖记录存在冲突的问题，通过静态分析生成新的 hash 计算式，把冲突率降低到接近 0，大大增加了覆盖信息的准确率，提升了 AFL 的效果。

此外，本文研究团队也发掘出 AFL 中隐藏的几个缺陷和可以进一步改进的地方，主要有以下几个方面。1) AFL 的种子选择算法存在缺陷。其进行选择时，实际隐含了一个固定顺序，使队列中排在越后面的种子被选中的概率越低。更严重的问题是，执行完一轮循环后，被选中的种子可能并没有覆盖到所有的边。2) AFL 对每个选中的种子执行的变异次数几乎一样，没有考虑到每条边的热度（即这条边被覆盖到的次数）。3) AFL 会记录哪些字节变化时会产生新的状态转移，但是这些记录信息未被有效利用，并且在从进程中无法使用这些记录信息。

针对上述问题，本文着眼于 AFL 算法的缺陷，同时对一些功能进行了改进，具体包括以下几个方面。

1) 提出了完全覆盖种子选择算法。该算法随机打乱边的顺序，以打乱后的顺序进行种子选择，且只选择队列中位于当前位置之后的种子，最后对覆盖情况进行检查并修复覆盖不全的问题。本文所提算法避免了按固定顺序选择优先种子集，同时也避免了对边覆盖不完全的情况。

2) 提出了基于边覆盖热度的能量调度算法。根据路径中边的覆盖热度，对每条路径进行评分，以此为依据动态调整每个种子文件的变异次数。

3) 增加对有效字节的利用。在进行随机性变异时，更多地对有效字节部分进行变异。同时加快了有效字节的产生，并在主进程和从进程之间同步有效字节信息。

基于上述方法，本文在开源的 AFL 工具上，设计和实现了新的 fuzzing 工具—efuzz，并按照 Klees 等[14]提出的测试思想进行实验，实验结果表明 efuzz 的边覆盖和漏洞发现能力都超过了 AFL 和 AFLFast。使用常用软件测试 24 h 后，efuzz 的平均边覆盖数相比 AFL 和 AFLFast 分别提升了 5%和 9%，某些情况下甚至达到 20%以上。使用 LAVA-M[15]测试 7 天后，efuzz 发现的漏洞总数超过了 AFL。在常用软件中，efuzz 发现了 3 个新的 CVE 漏洞，其中一个是 binutils 工具包中的漏洞 CVE-2018-20671，该漏洞从 2001 年起就已存在。

2　相关知识和技术概述

2.1　AFL 概述

AFL 是一个覆盖率引导的灰盒测试工具。它基于遗传算法，采用编译时插桩的方式，通过获取被测试程序运行时的覆盖信息反馈，来判断是否触发了目标程序新的内部状态，从而发现感兴趣的测试用例，以此引导 fuzzing 策略，这大大提高了测试工具的覆盖率。

图 1 展示了 AFL 的简易流程，具体步骤如下。

步骤 1　提供一个初始输入集加入种子队列。

步骤 2　按照种子选择算法从种子队列中选择优先种子集。

步骤 3　按预设概率，顺序从种子队列中选择

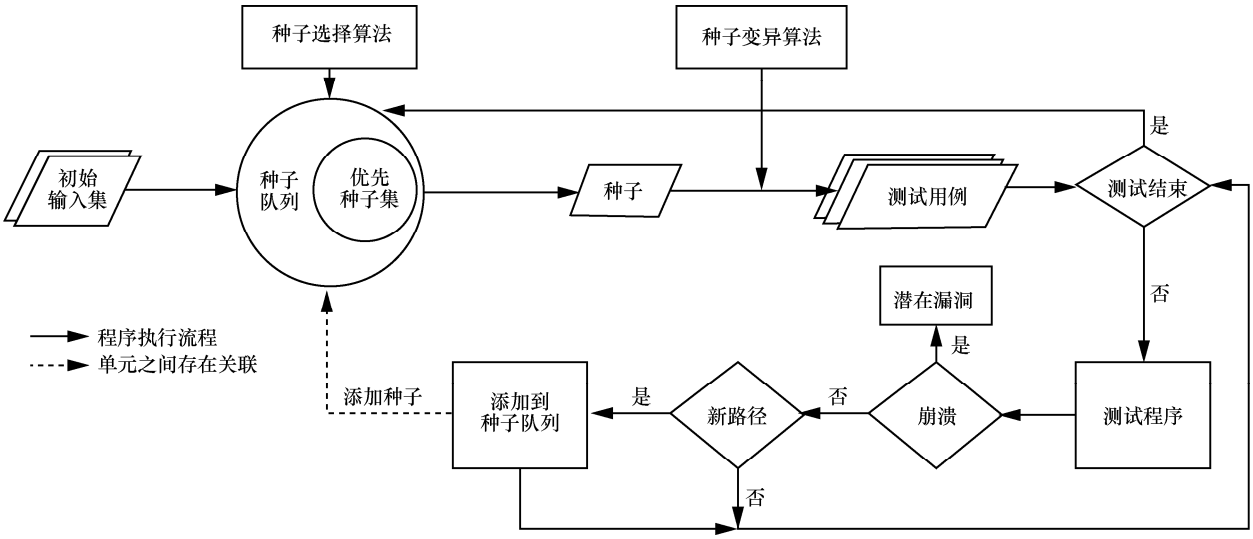

图 1 AFL 简易流程

一个种子文件。

步骤 4 使用多种变异算法对种子文件进行变异，循环生成大量测试用例进行测试。

步骤 5 如果发现程序崩溃，那么这可能是一个潜在的漏洞。

步骤 6 如果发现了新的路径，那么把生成该路径的测试用例加到种子队列。

步骤 7 对该种子生成的所有测试用例，如果测试结束，回到步骤 3。

整个测试是一个无限循环过程，直到人工终止。

2.2 边覆盖记录

AFL 记录的边覆盖信息包括边的 hash 以及这个边被命中的次数。边覆盖信息使用规模为 2^{16} 的数组记录，边 hash 作为数组的索引，数组中的值是边被覆盖的次数。为了加快处理速度，AFL 在每个 BBL 中插入了一个 0~65 535 的随机数，边 hash 是由边的首尾 2 个 BBL 中的随机数通过简单 hash 算法生成的，记录覆盖信息的算法伪代码如下。

```
1) edge_hash = cur_bbl ^ (last_bbl >>1)
2) shared_mem [ edge_ hash ]++
```

很显然步骤 1)中的 hash 算法会存在冲突问题，但这不是本文的研究重点。本文把一个 hash 记录直接对应为一条边覆盖记录。

2.3 种子选择算法

fuzzing 中如何从种子队列中选择种子是一个非常关键的问题。之前的工作已经证明，良好的种子选择策略可以显著提高模糊效率，更快地发现更多的 bug[16]。

为了理解 AFL 的种子选择算法，首先介绍以下 2 个概念。

优先种子。覆盖到某一条边的最好种子称为该边的优先种子。最好种子就是所有能够覆盖到该边的种子中，对其进行一次完整测试所消耗时间最短的种子。

优先种子集。可以覆盖当前已经发现的所有边的优先种子集合，即优先种子集的覆盖面等价于当前所有种子的覆盖面。

AFL 调用种子选择算法从整个种子队列中选出优先种子集，随后按概率优先选择其中的种子来进行测试。该算法本质上是一个贪婪算法，如算法 1 所示。

算法 1 AFL 的种子选择算法

输入 每条边对应的优先种子 T [65 536]、种子队列 Q

输出 选择结果 S

```
S = ∅
C[65536] = {0} //覆盖所有的边记录
for h in 0…65535 do //h: 边对应的 hash
  if T[h] and not C[h] then
    S.add(T[h])
    update_coverage_info(C, S)
return S
```

算法 1 按照边的 hash 值，在 0~65 535 范围内按从小到大的顺序选择该边的优先种子，直到覆盖所有已发现的边。由于被测试程序编译生成后，所有边的 hash 值就已经确定了，因此这实际上是一个

固定的选择顺序。

2.4　能量调度

Böhme 等[17]提出了能量的概念，能量指一个种子被选中后的测试次数。能量调度即按照一定算法对能量进行控制。AFL 给每个种子分配相同的能量，这忽略了对路径稀有度的考量，可能将过多的能量分配给高密度区域[18]。AFLFast 使用马尔可夫模型，给低频路径分配了更多的能量。但是，AFLFast 总是朝着低频路径，可能会错失对一些边的探索，导致代码覆盖率不高。Klees 等[14]也指出了 AFLFast 的这个问题。

2.5　有效字节信息

如果对种子文件中某个字节进行翻转后产生了与原种子对应路径不同的新路径，那么这个字节就是有效字节。下面具体解释相关概念。

确定性阶段和随机性阶段。AFL 分为确定性测试和随机性测试两大类。确定性测试阶段对种子的变异方式是高度确定的，且每个种子只会经历一次确定性测试，而随机性阶段会调用一些随机算法对种子进行变异。

主进程和从进程。AFL 可以有多个进程并行工作，主进程对每个选中的种子先进行确定性测试，再进行随机性测试；从进程只进行随机性测试。同时，主进程和从进程之间定期进行种子同步。

AFL 使用如下方法来记录有效字节信息。当一个种子经过确定性阶段的字节翻转子阶段时，观察种子文件中某个字节的翻转对执行路径是否有影响。如果对这个字节翻转后，产生了与种子对应路径不同的新路径，则使用 eff_map（effector maps）来记录这个字节，在确定性测试的剩余子阶段中只对 eff_map 中记录的字节进行变异。这样一般可以将测试的执行次数减少 10%～40%左右，而不会显著降低覆盖率。在极端情况下，例如块对齐的 tar 文件，执行次数可能减少高达 90%。

3　fuzzing 改进方法

3.1　完全覆盖种子选择算法

AFL 的种子算法本身基于如下完全覆盖性约束：选出的优先种子集必须能完全覆盖所有已经发现的边。通过 2.3 节的介绍以及实验测试，发现算法 1 存在以下缺点。

1) 按照边的 hash 值以 0~65 535 的固定顺序选择，导致种子被选中的概率相差太多，hash 较小的边对应的优先种子总是被先选择。举个极端的例子：按照算法 1，假设 hash 值为 0 的优先种子 $T[0]$被选中，而 $T[0]$对应的种子为 $s0$，如果 $s0$ 生成的路径刚好覆盖了所有已发现的边, 那么每次都只会选择 $s0$ 这一个种子, 其他种子永远无法被选到优先集合。在实验中记录每次选出的优先种子集也发现，每次选择出的优先种子集变化极小。如果采用随机的顺序进行种子选择，可以缓解这个问题。

2) AFL 每测试一个种子后，如果种子队列发生了变化，就会调用一次算法 1，这可能导致执行一轮循环后，有的边没有被覆盖，这种情况违背了完全覆盖性约束。出现这个问题的根本原因在于选择了队列中位于当前位置之前的种子，具体以代码 1 为例进行说明。

代码 1　AFL 种子选择算法缺陷示例代码

```
void main(void) {
    int i, j, a, c, x1, x2, x3, x4;
    input(&i, &j, &a, &c, &x1, &x2, &x3, &x4);
    while(i > 100) {
        if(a > 200) {
            i = x1;   a += 100;
            if (i < 10)   bug();
            continue;
        }
        while(j > x2) {
            j++;
            if( c > 300) { i = x3;   break; }
        }
        i = x4;   break;
    }
    if(i < 100)   do_something();
    return;
}
```

代码 1 编译后对应的控制流程如图 2 所示。

图 2 中每条边上的数字代表该边的编号，以 e_i 表示边 i，只有顺序通过边 e_2、e_6、e_{14} 的路径才会触发 bug。

以 q_j 表示第 j 条路径，假设初始发现的 6 条路径如表 1 所示。

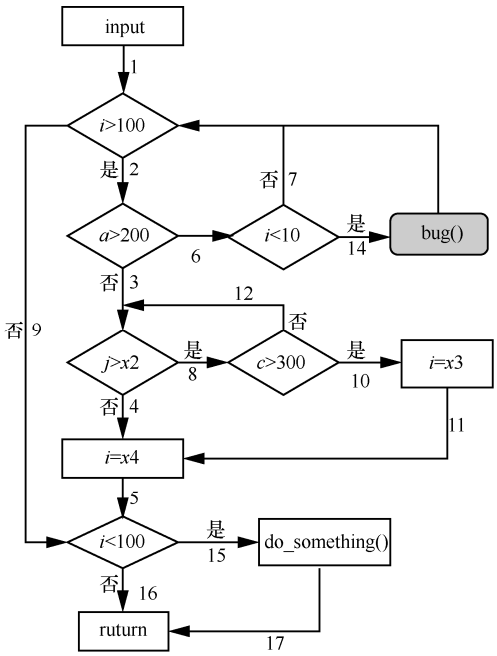

图 2 AFL 种子选择算法缺陷示例

表 1 路径说明

路径	经过的边	说明
q_1	e_1 e_9 e_{15} e_{17}	
q_2	e_1 e_2 $\boldsymbol{e_6}$ e_7 e_2 e_3 e_4 e_5 e_{16}	e_6 的优先种子
q_3	e_1 e_2 e_3 e_4 e_5 e_{15} $\boldsymbol{e_{17}}$	e_{17} 的优先种子
q_4	e_1 e_2 e_6 e_7 $\boldsymbol{e_9}$ e_{16}	q_5 出现前，e_9 的优先种子
q_5	e_1 $\boldsymbol{e_9}$ e_{16}	e_9 的优先种子
q_6	e_1 e_2 e_3 $\boldsymbol{e_8}$ e_{12} e_4 e_5 e_{15} e_{17}	e_8 的优先种子

假设这些边经过 hash 运算后的值，从小到大的顺序为(e_9, e_8, e_{17}, e_6, …)，且测试按照以下步骤进行。

步骤 1 测试 q_1，发现新路径{q_2, q_3, q_4}，这时覆盖的所有边集合为{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_9, e_{15}, e_{16}, e_{17}}。调用算法 1 选择优先种子集，按照边 hash 顺序 (e_9, e_{17}, e_6)，先选择 e_9 对应的优先种子 q_4，再选 e_{17} 对应的优先种子 q_3，这时已经覆盖了所有边，选择结束，优先种子集的结果为{q_3, q_4}。

步骤 2 q_2 不在优先种子集中，被跳过。

步骤 3 测试 q_3，发现新路径{q_5, q_6}，这时覆盖的所有边集合为{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{12}, e_{15}, e_{16}, e_{17}}。再次调用算法 1，按边 hash 顺序(e_9, e_8, e_{17}, e_6)，先选择 e_9 对应的优先种子 q_5，再选择 e_8 对应的优先种子 q_6，由于 e_{17} 已经被 q_6 覆盖，对其跳过处理，最后选 e_6 对应的优先种子 q_2，选择结束，结果为{p_5, q_6, q_2}。

步骤 4 q_4 不在优先种子集中，被跳过。

步骤 5 测试 q_5。

步骤 6 测试 q_6。

整个测试过程中，按照种子队列从前向后的顺序判断是否选择该种子，没有对可到达 e_6 的 q_2 和 q_4 进行测试。虽然 AFL 存在随机性变异，也有可能通过其他种子变异出到达 e_6 的测试用例，但是随机性变异过于盲目，概率过低，导致很难到达 e_{14}，从而无法触发 bug。

上述例子说明了 AFL 在一轮循环中，虽然每次选择的优先种子集能覆盖所有已发现的边，但是真正测试的种子却可能会漏掉某些边。上述情况看似概率很小，但是本文通过实验发现这种情况大量存在。

针对上述 2 个问题，本文提出了完全覆盖种子选择算法，如算法 2 所示。

算法 2 完全覆盖种子选择算法

输入 每条边对应的最好种子 T[65 536]、种子队列 Q、当前位置 queue_cur

输出 选择结果 S

```
q = queue_cur
S_old = S
S = ∅
C[65536] = {0} //覆盖所有边记录
R[65536] = {0, 1, …, 65535}
for s in S_old do
  if s before q then
    S.add(s)
    update_coverage_info(C, S)
disorder(R)
for h in R do
  s = T[h]
  if not C[h] and s don't before q then
    S.add(s)
    update_coverage_info (C, S)
for h in 0…65535 do //h: 边的 hash
  if not C[h] then
s = find_seed(q, h)
S.add(s)
update_coverage_info (C, S)
return S
```

算法 2 共分为 4 个步骤，具体如下。

步骤 1 把本次循环中已经测试过的种子文件直接加入 S 并更新 C。

步骤 2　生成 0~65 535 的乱序排列。

步骤 3　按照步骤 2 生成的序列从 T 中挑选种子加入 S，并更新 C。只有当前位置之后的种子才会被加入 S。

步骤 4　检查每条边是否被覆盖，如果没有覆盖，从当前位置向后找到一个可以覆盖该边的种子文件，添加到 S 并更新 C。

下面使用数学归纳法对上述算法的覆盖完整性进行证明。

1) 测试开始时第一次选择的优先种子集显然可以覆盖到所有已经发现的边。

2) 假设第 n 次选择的优先种子集 Q_n 能够覆盖到所有已经发现的边。当第 n+1 次选择时，当前位置把 Q_n 切分为两部分，一部分是在当前位置之前（不包含当前位置）的集合 Q_1，另一部分是当前位置之后（包含当前位置）的集合 Q_2。如果出现了新的边，这些边对应的种子文件会被增加到队列尾部，这些新种子文件的集合记为 Q_3。根据假设，可推导出 $Q_1 \cup Q_2 \cup Q_3$ 能够完全覆盖当前所有的边。按照算法 2，在步骤 1 中 Q_1 全部被选中，而步骤 4 是检查步骤，当发现某条边 e 尚未被覆盖时，则 $e \notin Q_1$，那么必然有 $e \in Q_2 \cup Q_3$，这说明从当前位置向后查找，一定可以找到一个种子文件对应的路径包含 e。因此，步骤 4 完成后可保证对边的完全覆盖。

上述证明说明算法 2 满足完全覆盖性约束，同时算法 2 采用乱序选择，且从当前位置开始选择新的优先种子，可以有效避免出现算法 1 的 2 个缺点。

3.2　基于边覆盖热度的能量调度算法

王志等[19]分析僵尸网络控制命令时，提出了 BBL 覆盖率的概念，用来描述一个 BBL 被执行路径覆盖的频繁程度。受此思想启发，本文用边的热度来描述一条边被所有路径覆盖的次数，结合 2.4 节的分析，提出了一个新的能量调度算法。

用 $N(e)$表示边 e 的热度，计算式为

$$N(e)=\sum_{q\in Q}((e\in q)?1:0) \tag{1}$$

边的评分与该边的热度成反比，即这条边被覆盖的次数越多，评分越低。每个种子文件对应一条执行路径，一条执行路径由多条边组成。路径评分为该路径中边的评分之和，$p(q)$为路径 q 的评分，定义为

$$p(q)=\sum_{e\in q}\frac{1}{N(e)} \tag{2}$$

将上述评分进行归一化，得到路径 q 的调整系数，定义为

$$\overline{p}(q)=\begin{cases}\overline{p}_{\max}, & p(q)\geqslant p_{\max}\\ \overline{p}_{\text{avg}}+(\overline{p}_{\max}-\overline{p}_{\text{avg}})\dfrac{p_{\max}-p(q)}{p_{\max}-p_{\text{avg}}}, & p(q)\geqslant p_{\text{avg}}\\ \overline{p}_{\min}+(\overline{p}_{\text{avg}}-\overline{p}_{\min})\dfrac{p(q)-p_{\min}}{p_{\text{avg}}-p_{\min}}, & p(q)>p_{\min}\\ \overline{p}_{\min}, & p(q)\leqslant p_{\min}\end{cases} \tag{3}$$

其中，p_{avg} 为所有种子对应路径评分的均值，$p_{\min}$ 为最小值，$p_{\max}$ 为最大值。$\overline{p}_{\text{avg}}$、$\overline{p}_{\min}$、$\overline{p}_{\max}$ 分别为预定义的调整系数。

种子原来的测试次数乘以调整系数 $\overline{p}$，即为种子新的测试次数。该算法的优点如下。

1) 热度越低的边，被给予越高的权重，有可能得到越多的执行机会。

2) fuzzing 往往难以到达很深的代码块，该算法对边评分采用加法，较长路径的评分会有更多的优势，有利于覆盖更深的边。但这也可能导致路径探索被局限在某一些分支上，针对这个问题，在具体实现中可通过对系数的调整来达到一定的平衡。

AFLFast 通过路径覆盖频率来选择种子，而本文方法则是通过边覆盖热度来选择种子，通过实验分析，本文方法在边覆盖上比 AFLFast 更优。

3.3　有效字节信息的进一步利用

2.5 节中提到的有效字节机制非常可靠，并且很容易实现，但是也存在如下问题。

1) 只有确定性阶段使用了有效字节信息，随机性阶段并没有使用，未能充分发挥其效果。

2) 有效字节信息由主进程产生，但主进程速度很慢。在实验中发现，有时主进程用一周时间都无法执行完一次循环，这会导致可利用信息非常少。

本文提出了 3 种进一步利用有效字节信息的方法。

1) 在随机性阶段使用有效字节信息引导变异。在应用随机性算法时，按一定概率只对有效字节进行变异。

2) 在从进程中也使用有效字节信息。保存主进程中记录的信息，供从进程使用。

3) 当存在从进程时，主进程去掉随机性阶段，

只处理确定性阶段，可加快有效字节信息的产生。

针对以上方法，需要在确定性阶段把 eff_map 信息记录到文件，当从进程对这个种子进行测试时，直接加载这个文件。

AFL 中共有 16 种变异方法，如表 2 所示。这些方法中，变异位置都是随机选择的。efuzz 中对此进行了改进，按一定概率只选择 eff_map 中记录的位置进行变异，不再是完全随机选择变异位置。

表 2 随机性阶段的变异方法

序号	变异方法
1	随机选择位进行翻转
2	随机选择 1 B 设置为特殊值，值从预定义的集合随机选取
3	随机选择 2 B 设置为特殊值，值从预定义的集合随机选取
4	随机选择 4 B 设置为特殊值，值从预定义的集合随机选取
5	随机选择 1 B 减去一个较小的随机数
6	随机选择 1 B 增加一个较小的随机数
7	随机选择 2 B 减去一个较小的随机数
8	随机选择 2 B 增加一个较小的随机数
9	随机选择 4 B 减去一个较小的随机数
10	随机选择 4 B 增加一个较小的随机数
11	随机选择 1 B 设置为随机数
12	随机删除一段数据
13	插入常量字节块或克隆文件中一段数据到随机位置
14	覆盖写入常量字节块或文件中原有一段数据到随机位置
15	覆盖写入字典中的数据到随机位置
16	插入字典中的数据到随机位置

随机性阶段会对文件进行多次变异。除了 12、13、16 这 3 种方法可能改变种子文件长度外，其他方法不会改变种子文件的长度。一旦变异过程中文件长度发生变化，eff_map 记录信息将不再准确。因此，efuzz 把随机性阶段又分成了 2 个子阶段：第一个子阶段不采用 12、13、16 这 3 种方法；第二个子阶段采用所有 16 种方法，一旦文件长度发生了变化，就立即恢复到 AFL 原有的方式。

4 系统设计和实现

AFL 的最新开源版本是 2.52b，efuzz 扩展了这个版本，共添加了不到 1 000 行的 C 代码，实现了第 3 节中描述的技术。

在算法 2 中对每个种子进行顺序编号，相比原来的算法，仅增加了种子选取顺序的随机化处理和最后的检查，因此整体增加的 CPU 消耗几乎可以忽略不计。

在能量调度算法中，对路径的评分实际上是很难准确定义的，本文采用了比较保守的策略，设置 3 个调整系统的值分别为 0.2、0.8、4，且只有在发现的总路径数超过 200 时，才启用这一策略。因为该算法中都是浮点计算，为降低计算量，每增加 20 条以上的新路径才重新计算一次评分。其次，该算法中所有种子对应的覆盖信息都要记录下来，每个种子需要占用 8 KB 的内存空间。在本文的实验中，总路径很少有超过 12 000 的，因此，总计算次数不会超过 600，单个进程的内存增长数不会超过 100 MB，对计算机资源的整体消耗非常小。

在确定性阶段测试过的种子，需要把它的有效字节信息记录到一个对应的以.eff 为后缀的文件。eff 文件中每一位表示该种子的 1 B 是否是有效，因此 eff 文件大小只有种子文件的$\frac{1}{8}$，对存储资源的消耗也非常小。

5 实验与分析

Klees 等[14]指出了当前 fuzzing 实验中普遍存在的一些问题，并给出建议的测试方法，希望能建立统一的测试标准。本文中的实验按照他们提出的测试思想，以如下方式进行测试。

1) 共选择了 5 个常用的目标程序，分别是 binutils 程序集中的 objdump、nm、readelf，以及 libtiff 和 tcpdump。

2) 对每个测试程序选择了 5 个有代表性的初始输入，输入分别来源于空种子、随机种子、公开的测试样本、漏洞的 POC（proof of concept），表 3 是实验中具体使用的输入文件。由于有些程序使用空种子和随机种子无法测试，因此本文使用其他种子代替这 2 个种子的实验。

3) 涉及对边覆盖数进行比较的实验，每个实验都测试 30 次，每次测试时间为 25 h。由于实验数据并不是实时更新的，选取第 24 h 这个时间点的数据来进行评估。最终结果取 30 次的平均值。

4) 每次实验使用一个主进程和一个从进程共同测试，更加接近真实使用环境。

5) 以 AFL 和 AFLFast 作为比较对象。AFLFast 和 efuzz 一样，也是针对 AFL 本身算法的改进，且是开源软件，方便进行实验。

6) Stephens 等[6]指出使用唯一状态转换的数量作为边覆盖的度量是合理的，本文实验也使用状态

表 3　　实验使用测试程序和种子文件

测试程序	第一个种子	第二个种子	第三个种子	第四个种子	第五个种子
binutils-objdump	empty	rand100	regdbdump	CVE-2018-1000876	CVE-2018-20673
binutils-nm	empty	rand100	regdbdump	CVE-2018-1000876	CVE-2018-20673
binutils-readelf	pr21135	regdump	CVE-2017-6965	CVE-2017-6966	CVE-2017-6969
tcpdump	heapoverflow-tcp_print	stp-heapoverflow	CVE-2017-11108	bgp_vpn_attrset	EIGRP_subnet_down
libtiff	miniswhite-1c-1b	rgb-3c-8b	palette-1c-1b	CVE-2019-7663	CVE-2019-6128

注：empty: 空种子文件，rand100：100 个随机字节，regdbdump：64 位 Ubuntu 16.04 中的/sbin/regdbdump，CVE-∗：对应 CVE 漏洞的 POC 样本，其他：测试程序自身提供的测试样本。

转换的数量作为边覆盖的度量。

实验共使用了 3 台服务器，都是 64 位 Ubuntu 16.04 LTS 系统，CPU 为 Intel(R) Xeon(R) E5-2620 v4 @ 2.10 GHz，内存 64 GB，硬盘 12 TB。

5.1　边覆盖的完整性测试

为了对 AFL 和 efuzz 的种子选择算法进行测试，在它们的算法中添加了检查代码，每次算法执行完时，都对优先种子集的边覆盖情况进行校验，检查本轮循环中已测试过的优先种子和当前位置之后还未测试的优先种子是否覆盖了所有已经发现的边，如果出现覆盖不全的情况，则进行日志记录，并记下每次选择最多有多少边未被覆盖。

实验用表 3 中各测试程序对应的 5 个文件共同作为该程序的初始化种子，只运行 1 h。由于 efuzz 采用了新的种子选择算法，保证了覆盖的完整性，结果中并未出现覆盖不全的情况。而 AFL 中出现了大量覆盖不全的情况，如表 4 所示。

表 4　　AFL 边覆盖不全的记录

测试程序	覆盖不全总次数	调用算法总次数	覆盖不全的比例	未覆盖的最大边数
binutils-objdump	425	583	72.90%	14
binutils-nm	638	743	85.87%	4
binutils-readelf	1 199	1327	90.35%	7
libtiff	362	458	79.04%	5
tcpdump	1 498	1599	93.68%	10

由表 4 可知，AFL 覆盖不全的比例非常大，虽然每次未覆盖的边数非常少，但依然说明了 AFL 的种子选择算法在一轮循环中并未满足完全覆盖性约束。

5.2　能量调度算法和有效字节信息改进的效果验证

实验中开发了仅启用能量调度算法的程序 efuzz-Enery 和仅启用有效字节信息改进的程序 efuzz-Bytes，用于单独测试这 2 个改进的有效性。每个改进的测试选择了 5 个实验，每个实验进行 30 次，结果如表 5 所示。

表 5　　单独测量的边覆盖数

	tcpdump1	libtiff2	libtiff3	readelf4	readelf5
AFL	11 342.2	3691.4	5154.1	10 222.7	9487.0
AFLFast	10 627.9	3640.9	5279.7	9 753.7	8977.6
Enery	11 756.1	3872	5300.2	10 306.5	9951.7
Bytes	11 601.1	3816.6	5364.1	10 591.8	10068.8
Enery | AFL	3.65%	4.89%	2.83%	0.82%	4.90%
Enery | Fast	10.62%	6.35%	0.39%	5.67%	10.85%
Bytes | AFL	2.28%	3.39%	4.07%	3.61%	6.13%

注：tcpdump1：使用表 3 中的第一个种子文件测试 tcpdump，Fast：AFLFast Enery：efuzz-Enery，Bytes：efuzz-Bytes，Enery | AFL：efuzz-Enery 相对 AFL 的提高。

从比较结果来看，efuzz-Enery 相对 AFLFast 有比较大的提高，同时比 AFL 也有一定提高，证明了本文的能量调度算法比 AFLFast 在边覆盖率上更有优势。

通过比较 efuzz-Bytes 与 AFL 可以看到，efuzz-Bytes 相对 AFL 也有少量的提高，说明有效字节信息的合理利用可以在一定程度上提升 AFL 的效果。

5.3　efuzz 与其他工具的比较

实验对比 efuzz、AFL 和 AFLFast，使用 5 个程序，并把每个程序对应的 5 个种子文件分别作为初始输入，共 25 组实验。进行该测试的资源要求非常高，总开销为 25×30×25×2×3 = 112 500 核·小时 = 4 687.5 核·天。表 6 是 24 h 的平均边覆盖数结果，从整体来看，efuzz 在覆盖数上优于 AFL 和 AFLFast。同时也发现，AFLFast 在平均覆盖数上要略差于 AFL。

表 6　　25 组实验的平均边覆盖数

实验名	AFL	AFLFast	efuzz	比 AFL 提高	比 AFLFast 提高
objdump1	7 288.3	7 223.8	7 852.5	7.74%	8.70%
objdump2	8 314.3	8 098.9	8 569.3	3.07%	5.81%
objdump3	8 371.4	8 403.8	8 623.4	3.01%	2.61%
objdump4	6 969.8	6 701.5	7 458.2	7.01%	11.29%
objdump5	7 599.8	7 031.2	7 829.1	3.02%	11.35%
nm1	5 794.7	5 566.4	6 172.6	6.52%	10.89%
nm2	3 985.2	3 996.5	4 796.1	20.35%	20.01%
nm3	7 306.2	6 450.1	7 618.3	4.27%	18.11%
nm4	6 036.0	5 029.9	6 228.3	3.19%	23.83%
nm5	5 551.9	5 316.3	5 823.7	4.90%	9.55%
readelf1	10 421.2	9 823.0	10 807.8	3.71%	10.03%
readelf2	10 637.9	10 029.5	10 971.2	3.13%	9.39%
readelf3	10 238.0	9 578.3	10 738.4	4.89%	12.11%
readelf4	10 222.7	9 753.7	10 633.0	4.01%	9.02%
readelf5	9 487.0	8 977.6	10 186.9	7.38%	13.47%
libtiff1	5 218.1	5 284.7	5 326.7	2.08%	0.80%
libtiff2	3 691.4	3 640.9	3 875.2	4.98%	6.43%
libtiff3	5 154.1	5 279.7	5 419.0	5.14%	2.64%
libtiff4	5 513.8	5 504.2	5 785.1	4.92%	5.10%
libtiff5	5 299.7	5 376.0	5 489.5	3.58%	2.11%
tcpdump1	11 342.2	10 627.9	11 916.2	5.06%	12.12%
tcpdump2	11 654.2	11 123.8	11 646.5	−0.07%	4.70%
tcpdump3	11 869.0	11 105.9	12 009.9	1.19%	8.14%
tcpdump4	11 844.4	11 183.7	11 894.4	0.42%	6.35%
tcpdump5	11 821.6	11 120.9	11 826.0	0.04%	6.34%
平均	—	—	—	4.54%	9.24%

单独分析每一组实验，efuzz 几乎在所有情况下都优于 AFL 和 AFLFast。在 nm2 中，efuzz 的边覆盖数甚至提高超过 20%，但在 tcpdump 实验中，efuzz 表现一般，仅仅略好于 AFL，tcpdump2 中甚至比 AFL 低 0.07%。这说明本文的改进方法在绝大多数情况下提高了边覆盖率，但是在极少数情况下也引入了一些限制。这也进一步说明了 Klees 等[14]提出的评估方法的合理性，只有通过大量不同条件下的实验，才能比较全面地评估 fuzzing 工具的有效性。

5.4　LAVA-M 测试

对 LAVA-M 中的 4 个程序，使用其提供的默认种子文件进行测试，每个程序各运行 7 天。表 7 中数据显示，AFL 在 base64 和 uniq 中发现的漏洞数超过了 efuzz，但 efuzz 在 who 中发现的漏洞数远超 AFL，且在 4 个程序中发现的漏洞总数也超过了 AFL。

表 7　　LAVA-M 测试发现漏洞数

测试程序	设计 bug 数	AFL 触发数	efuzz 触发数
base64	48	6	4
md5sum	61	0	0
uniq	29	4	2
who	2 517	1	56
总计	2 655	11	62

LAVA-M 在测试程序中插入了许多字符串比较和整数校验代码来阻碍 fuzzing 工具，而 AFL 的变异算法很难产生适当的输入绕过这些代码，导致无法到达新的代码块，因此 AFL 和 efuzz 在 LAVA-M 上的效果都很差。虽然 efuzz 在 who 上的表现超过了 AFL，但发现的漏洞数也仅占总漏洞数的 2%。efuzz 在 base64 和 uniq 上差于 AFL，再次说明了 efuzz 的改进方法在有些情况下反而效果更差，且同样对校验性的代码无法达到很好的绕过效果。

5.5　新的 CVE 漏洞

表 8 是使用 efuzz 发现的 3 个新的 CVE 漏洞，CVE-2018-20671 是 binutils 中的一个整型溢出漏洞，最终会触发堆溢出。CVE-2018-20467 是 ImageMagick 中的一个拒绝服务漏洞，会造成 CPU 和内存资源的耗尽。CVE-2019-6988 是 openjpeg 中的一个拒绝服务漏洞，由于其逻辑问题，会长时间占用大量 CPU 资源。本文研究团队向相关厂商反馈了这些问题，现在这 3 个漏洞都已经得到修复。

表 8　　efuzz 发现的漏洞

CVE 编号	漏洞程序	漏洞说明
CVE-2018-20671	binutils	整型溢出/堆溢出
CVE-2018-20467	ImageMagick	拒绝服务
CVE-2019-6988	openjpeg	拒绝服务

表 8 中 CVE-2018-20671 最早存在于 binutils 2.11 中的 objdump 程序，该版本的发布时间是 2001 年。从那时起，绝大多数的版本都存在这个漏洞。objdump 是一个广泛使用的二进制工具，也是很多相关论文中的测试目标程序，大量的研究者对其进行 fuzzing，都没有发现这个漏洞。本文使用收集整理的种子文件，用一个进程进行 30 次测试，efuzz 平均只需要 519 s 就能发现这个漏洞，而 AFL 和 AFLFast 在相同时间内都无法发现这个漏洞。这进一步证明了 efuzz 的有效性。

6　结束语

目前，fuzzing 是软件漏洞挖掘领域最有效的方式之一，覆盖率的提高一直是研究的重要方向。本文首先基于 fuzzing 中边的完全覆盖性约束，设计了完全覆盖种子选择算法，并基于边覆盖热度提出了新的能量调度算法，最后进一步利用了 AFL 中的有效字节信息。上述改进方法也可推广到其他 fuzzing 工具中。从实验结果来看，本文实现的 efuzz 相对 AFL 和 AFLFast 在边覆盖率和漏洞发现能力上都有提高，并在常用软件中发现了新的漏洞。但 efuzz 依然存在很多改进空间，比如在 tcpdump 上的效果并不明显，在 tcpdump-2、base64、uniq 上的实验结果还要略差于 AFL。同时在不知道被测程序结构的情况下进行 fuzzing 依然存在盲目性，如何结合程序分析来获取更多被测程序信息，以及结合其他方法进一步提高边覆盖率，还需要在接下来的工作中进行更深入的研究。

参考文献：

[1] SUTTON M, GREENE A, AMINI P. Fuzzing: brute force vulnerability discovery[M]. NJ: Pearson Education, 2007.

[2] CHEN C, CUI B, MA J, et al. A systematic review of fuzzing techniques[J]. Computers & Security, 2018, 75(1): 118-137.

[3] RAWAT S, JAIN V, KUMAR A, et al. VUzzer: application-aware evolutionary fuzzing[C]//ISOC Network and Distributed System Security Symposium. ISOC, 2017: 1-14.

[4] BÖHME M, PHAM V T, NGUYEN M D, et al. Directed greybox fuzzing[C]//ACM Conference on Computer and Communications Security. ACM, 2017: 2329-2344

[5] CHEN H, XUE Y, LI Y, et al. Hawkeye: towards a desired directed grey-box fuzzer[C]//ACM Conference on Computer and Communications Security. ACM, 2018: 2095-2108.

[6] STEPHENS N, GROSEN J, SALLS C, et al. Driller: augmenting fuzzing through selective symbolic execution[C]//ISOC Network and Distributed System Security Symposium. ISOC, 2016: 1-16.

[7] SHOSHITAISHVILI Y, WANG R, SALLS C, et al. Sok: state of the art of war: offensive techniques in binary analysis[C]//IEEE Symposium on Security and Privacy. IEEE, 2016: 138-157.

[8] OGNAWALA S, KILGER F, PRETSCHNER A. Compositional fuzzing aided by targeted symbolic execution[J]. arXiv Preprint, arXiv:1903.02981, 2019.

[9] CADAR C, DUNBAR D, ENGLER D R. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs[C]//USENIX Symposium on Operating Systems Design and Implementation. USENIX, 2008: 209-224.

[10] 孙鸿宇,何远,王基策, 等.人工智能技术在安全漏洞领域的应用[J]. 通信学报, 2018, 39(8): 1-17.
SUN H Y, HE Y, WANG J C, et al. Application of artificial intelligence technology in the field of security vulnerability[J]. Journal on Communications, 2018, 39(8):1-17.

[11] GODEFROID P, PELEG H, SINGH R. Learn & fuzz: machine learning for input fuzzing[C]//IEEE/ACM International Conference on Automated Software Engineering. IEEE/ACM, 2017: 50-59.

[12] WANG J, CHEN B, WEI L, et al. Skyfire: Data-driven seed generation for fuzzing[C]//IEEE Symposium on Security and Privacy. IEEE, 2017: 579-594.

[13] GAN S, ZHANG C, QIN X, et al. CollAFL: path sensitive fuzzing[C]//IEEE Symposium on Security and Privacy. IEEE, 2018: 679-696.

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

[15] DOLAN-GAVITT B, HULIN P, KIRDA E, et al. Lava: large-scale automated vulnerability addition[C]//IEEE Symposium on Security and Privacy. IEEE, 2016: 110-121.

[16] LI J, ZHAO B, ZHANG C. Fuzzing: a survey[J]. Cybersecurity, 2018, 1(1):6.

[17] BÖHME M, PHAM V T, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain[C]//ACM Conference on Computer and Communications Security. ACM, 2016: 1032-1043.

[18] WANG M, LIANG J, CHEN Y, et al. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing[C]//International Conference on Software Engineering. 2018: 61-64.

[19] 王志, 蔡亚运, 刘露, 等. 基于覆盖率分析的僵尸网络控制命令发掘方法[J]. 通信学报, 2014, 35(1): 156-166.
WANG Z, CAI Y Y, LIU L, et al. Using coverage analysis to extract Botnet command-and-control protocol [J]. Journal on Communications, 2014, 35(1): 156-166.

[作者简介]

贾春福（1967− ），男，河北文安人，博士，南开大学教授、博士生导师，主要研究方向为计算机网络与信息安全、可信计算、恶意代码分析。

严盛博（1987− ），男，湖北荆州人，南开大学硕士生，主要研究方向为逆向工程与漏洞挖掘。

王志（1981− ），男，山西长治人，博士，南开大学讲师，主要研究方向为计算机病毒的分析与防治。

武辰璐（1997− ），女，河南焦作人，南开大学硕士生，主要研究方向为二进制漏洞挖掘。

黎航（1995− ），男，湖北荆门人，南开大学硕士生，主要研究方向为自然语言处理。