CollAFL: Path Sensitive Fuzzing

江嘉诚

摘要

覆盖率指导下的模糊测试是一种广泛使用的、有效的寻找软件漏洞的解决方案。跟踪代码覆盖率并利用它来指导模糊测试，对覆盖指导型模糊测试至关重要。然而，由于插桩的高开销，跟踪完整和准确的路径覆盖率在实践中是不可行的。流行的模糊测试工具 (例如 AFL) 通常使用粗略的覆盖信息，例如存储在紧凑位图中的边命中率，以实现高效的灰盒测试。这种覆盖率的不准确性和不完整性给模糊测试工具带来了严重的限制。首先，它导致了路径碰撞，这使得模糊测试工具无法发现导致新崩溃的潜在路径。更重要的是，它使模糊测试工具无法对模糊策略做出明智的决定。

在本文中，提出了一个覆盖率敏感的模糊处理方案 CollAFL。它通过提供更准确的覆盖率信息来缓解路径碰撞，同时还保持了较低的插桩开销。它还利用覆盖率信息来应用三种新的模糊策略，提高了发现新路径和漏洞的速度。CollAFL 的原型基于流行的模糊测试工具 AFL，并对 24 个流行的应用程序进行了评估。结果显示，路径碰撞很常见，即在一些应用中，高达 75% 的边可能与其他边发生碰撞，而 CollAFL 可以将边的碰撞率降低到接近零。此外，在三种模糊策略的应用下，CollAFL 在代码覆盖率和漏洞发现方面都优于 AFL。平均而言，CollAFL 比 AFL 在 200 小时内覆盖了 20% 的程序路径，发现了 320% 的独特崩溃和 260% 的漏洞。总的来说，CollAFL 发现了 157 个新的安全漏洞以及 95 个新的已分配 CVEs。

关键词：模糊测试；哈希碰撞

1 引言

内存损坏漏洞是许多严重威胁程序的根本原因，包括控制流劫持攻击[1-3]和信息泄露攻击。防御者和攻击者都急切于发现程序中的漏洞。攻击者依靠漏洞来破坏目标程序的执行并进行恶意操作。如果防御者能够提前发现漏洞，他们就可以修补漏洞，以防御潜在的攻击。

覆盖率指导下的模糊测试是最流行的漏洞发现方案之一，在工业界广泛部署。例如，Google 的 OSS-Fuzz 平台[4]采用了几个最先进的覆盖指导型模糊测试工具，包括 libfuzzer[5]、honggfuzz[6]和 AFL[7]，来持续测试开源应用程序。它在 5 个月内通过数千台虚拟机发现了 1000 多个 bug。

首先，跟踪代码覆盖率对于覆盖指导的模糊测试工具来说是至关重要的。准确的路径覆盖信息可以帮助模糊测试工具感知所有独特的路径，并探索这些路径以发现漏洞。然而，由于极高的插桩开销，跟踪路径覆盖率在实践中是不可行的。模糊测试工具必须在性能与覆盖精度之间权衡。libfuzzer 和 honggfuzz 利用 Clang 编译器提供的 SanitizerCoverage 工具，来跟踪块覆盖率。VUzzer[8]使用动态二进制工具 PIN[9]来跟踪块覆盖率。AFL(在 GCC 和 LLVM 模式下) 使用静态工具与紧凑的位图来跟踪边缘覆盖，提供比块覆盖更多的信息。即使是 AFL，也有一个已知的哈希碰撞问题，即两个不同的边可能有相同的哈希，因此在覆盖位图中共享相同的记录。这导致了边覆盖的准确性的损失。实验表明，在某些应用中，高达 75% 的边可能与其他边发生碰撞。

更重要的是，利用覆盖率信息来指导模糊处理，对于覆盖率指导的模糊测试是至关重要的。AFL 利用边缘覆盖率信息来识别种子 (即对覆盖率有良好贡献的测试用例)，并将其加入种子池，等待进一

步的变异和测试。AFLfast[10]进一步利用边覆盖率信息，优先从种子池里选择走过路径不那么频繁的种子进行变异，以提高路径发现的效率。VUzzer 利用块覆盖率信息降低走过错误处理块的测试用例和走过频繁路径的测试用例的优先级。然而，假若代码覆盖率信息不准确，则模糊测试工具无法做出最佳决策。此外，很少有模糊测试工具利用代码覆盖率信息来直接驱动模糊测试工具向非探索的路径发展。

就目前而言，覆盖率不准确的后果被模糊测试工具的巨大成功所掩盖，因此还没有被系统地评估过。在本文中，可以证明它实际上对模糊测试工具的能力有着至关重要的影响。还可以证明，如果能以较低的开销实现准确的边缘覆盖，并部署适当的覆盖率指导的模糊策略，模糊测试工具可以显著提高其探索路径和寻找错误的能力。

2 相关工作

本文研究了覆盖率不准确对覆盖率指导型模糊测试工具的负面影响。特别是，本文证明了 AFL 中的哈希碰撞问题严重限制了其发现路径和漏洞的效率。

本文设计了一种算法来解决 AFL 中的哈希碰撞问题，通过一种低开销的插桩方案 (在大多数情况下比 AFL 更快) 来提高其边覆盖精度。

本文提出了三种新的覆盖率敏感的种子选择策略。通过实验可以证实，基于准确的边覆盖信息对种子进行优先排序，可以显著提高模糊测试工具的性能。

2.1 AFL 的覆盖追踪方法

覆盖率指导的模糊测试工具利用覆盖率信息来驱动模糊测试。如前所述，覆盖率的不准确性模糊了错误的发现。因此，准确的覆盖率对模糊测试工具来说至关重要。

不同的程序路径表现出不同的程序行为，因此可能有不同的漏洞。准确的路径覆盖可以帮助模糊测试工具感知不同的路径。然而，在运行时跟踪所有的路径覆盖率 (尤其是边的顺序) 是不可行的，因为路径的数量非常多，每个路径的存储开销也很高。

对于追踪块覆盖的模糊测试工具，例如 libfuzzer、honggfuzz 和 VUzzer，提高其覆盖精度的解决方案是用边覆盖追踪取代其追踪方案，例如 AFL 使用的方案。然而，AFL 所提供的边覆盖是不完美的。AFL 利用一个位图 (默认大小为 64KB) 来跟踪应用程序的边覆盖。位图的每个字节都代表一个特定边的统计数据 (例如，击中次数)。为每个边计算一个哈希值，并作为位图的键。在这个方案中存在一个哈希碰撞问题，即两条边可能有相同的哈希值。所以模糊测试工具无法区分这些边，导致覆盖率不准确。

更具体地说，AFL 对目标应用程序进行插桩，并为其基本块分配随机键。给定一条边 $A->B$，AFL 计算它的哈希值方式如下所示:

$$cur \oplus (prev \gg 1) \tag{1}$$

pre 和 cur 分别是基本块 A 和基本块 B 的键。由于键的随机分配，两条不同的边可能有相同的哈希值。此外，由于边的数量很多 (例如，可与位图大小 64K 相当)，其碰撞率可能很高。

就目前而言，覆盖率不准确的后果被模糊测试工具的巨大成功所掩盖，因此还没有被系统地评估过。本文实验表明，在现实世界的应用中，由于哈希碰撞问题，高达 75% 的边可能对 AFL 不可见，这

大大限制了 AFL 的能力。

2.2 CollAFL 的覆盖追踪方法

如公式 1所示，AFL 使用一个固定的公式来计算每条边的哈希值，速度快但容易发生碰撞。本文通过对不同的边应用不同的哈希公式来完善它，以消除哈希值碰撞，同时保持哈希值计算的速度和覆盖率跟踪。

一般来说，给定两个具有键 $prev$ 和 cur 的基本块 A 和 B，本文计算边 $A->B$ 的哈希值，如下所示:

$$Fmul(cur, prev) = (cur \gg x) \oplus (prev \gg y) + z \tag{2}$$

其中 $<x, y, z>$ 是待定的参数，对于不同的边可能有所不同。AFL 使用的公式 1是这种算法的一个特定形式，即对于所有的边/块 $<x = 0, y = 1, z = 0>$。$Fmul$ 的计算过程与 AFL 相同，具有相同的开销。

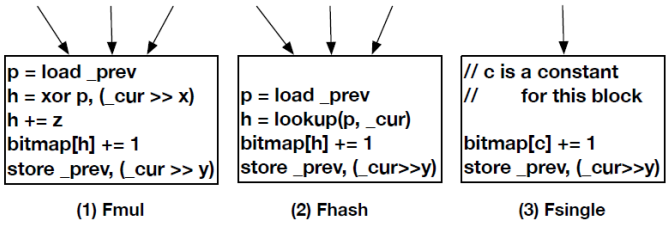

图 1: 新哈希算法

如图 1所示，可以为每个基本块选择一组参数，而不是每个边，来计算边的哈希值。为了简单起见，一个相同的参数 y 将在块之间共享，其值 $(prev >> y)$ 将被缓存在全局变量 $\_prev$ 中。每个块可以有一组不同的参数 $<x, z>$。

因此，给定一个应用程序，可以尝试为每个基本块找到一个参数解决方案，确保通过 $Fmul$ 计算的所有边的哈希值是不同的。本文使用一个贪心算法来逐一搜索每个基本块的参数。一旦找到所有基本块的解决方案，就可以用它们的哈希值来区分任意两条边，从而解决哈希碰撞问题。

2.3 种子选择策略

最近的研究[10-11]表明，种子选择策略对基于覆盖的模糊测试工具至关重要。一个好的种子选择策略可以提高模糊测试工具的路径探索和错误发现速度。

AFL 优先考虑那些体积较小、执行速度较快的种子，因此有可能在一定时间内测试更多的测试用例。Honggfuzz 按顺序选择种子，而 libfuzzer 则优先考虑击中更多新基本块的种子。VUzzer 优先考虑走过较深路径的种子，并降低走过错误处理块和频繁路径的测试用例的优先级，因此难以到达的路径很可能可以被测试到，而无用的错误处理路径将被避开。AFLfast 优先考虑走过频率较低的路径的种子，并且选择被选到较少的种子，因此有可能对偏僻路径进行彻底测试，并且在普遍走过的路径上浪费的能量较少。

种子选择策略也可以加强模糊测试工具在特定方向的能力。例如，QTEP[12]优先考虑覆盖更多由静态分析确定的故障代码的种子，增加测试过程中触发漏洞的概率。SlowFuzz[13]优先考虑使用更多资源 (如 CPU、内存和能源) 的种子，增加触发算法复杂性漏洞的概率。AFLgo[11]优先考虑更接近预定目标位置的种子从而实现高效的定向模糊测试。

3 本文方法

3.1 本文方法概述

对于哈希碰撞问题，首先根据前驱个数将基本块分为多个前驱的基本块集合和单个前驱的基本块集合，然后再对多个前驱的基本块集合计算每一个基本块的参数，由此将其分为可解决的基本块集合和不可解决的基本块集合，最后根据一个基本块对应的类型应用对应的公式来计算哈希值，从而解决哈希碰撞问题。

对于种子选择策略，如果一条路径有许多未探索 (或未触及) 的邻居分支，那么这条路径的变异很可能会探索这些未探索的分支。如果一条路径有许多未被探索 (或未被触及) 的邻居后代，那么来自这条路径的变异就很有可能探索那些未被探索的后代。如果一条路径有许多内存访问操作，就很可能引发潜在的内存损坏漏洞，其变异也是如此。

3.2 单个前驱基本块的哈希算法

如果一个基本块只有一个前驱，如图 1(3) 所示，本文直接对该基本块所对应的边分配一个哈希值，而不是用公式 2来计算，只要这个哈希值不会跟其他边的碰撞。

因此，对于只有一个前驱基本块 A 的基本块 B，不需要寻找一个参数组合 $< x, y, z >$，而只需要为其唯一的入边 $A- > B$ 寻找一个唯一的哈希值。因此，为它引入一个不同的哈希算法，如下所示:

$$Fsingle(cur, prev) : c \tag{3}$$

其中 preh 和 cur 是分配给基本块 A 和 B 的键，参数 c 是一个待确定的唯一常量。

这个哈希值 c 可以被离线解析，然后硬编码到基本块 B 中。因此，CollAFL 比 AFL 计算这种边的哈希值要快得多。正如本文实验所示，在大多数应用中，超过 60% 的基本块只有一个前驱基本块。因此，它可以节省大量的运行时间开销，提高模糊测试工具的吞吐量。

此外，这些哈希值可以在任何时候被解决。因此，为了避免冲突，可以等到所有其他边的哈希值被确定之后再来挑选未使用的哈希值并将其分配给只有一个前驱基本块的块。

3.3 多个前驱基本块的哈希算法

如果一个基本块 B 有多个前驱，即 B 有多个入边，必须动态地计算基本块 B 的哈希值，因为被击中的入边只有在运行时才知道。一般来说，本文将使用上述公式 2来计算哈希值。

有时候并不能保证找到这个公式的解来避免碰撞，即使在删除了只有一个先驱的基本块之后。本文使用一种贪心算法来解决这些块的参数。本文将能解决的基本块表示为可解决基本块，把不能解决的基本块表示为不可解决基本块。

对于一个不可解决基本块 B，本文引入了另一个哈希算法来计算其入边 $A- > B$，如下所示:

$$Fhash(cur, prev) : hash\_table\_lookup(cur, prev) \tag{4}$$

其中 pre 和 cur 是基本块 A 和 B 的键。它离线构建了一个哈希表，为所有以不可解决基本块结束的边提供唯一的哈希值，并与所有其他边的哈希值不同。在运行时，它查找这个预先计算的哈希表，以获得这些边的哈希值，使用它们的开始和结束块作为键。

在运行时，一个哈希表的查找操作比之前的哈希算法 $Fmul$ 和 $Fsingle$ 要慢得多。因此，需要限制不可解决基本块的集合尽量小。根据本文的实验，这个集合通常是空集。

3.4 未触及邻居分支指导策略

在这个策略中，具有更多未触及的邻居分支的种子将被优先模糊测试。本文相信基于这些种子的变异有更高的概率去探索那些未触及的邻居分支。

更具体地说，本文使用未触及的邻居分支的数量作为测试用例 T 的权重，具体如下:

$$Weight\_Br(T) = \sum_{\substack{bb \in Path(T) \\ <bb,bb_i> \in EDGES}} IsUntouched\left(< bb, bb_i >\right) \tag{5}$$

其中函数 $IsUntouched$ 返回 1 当且仅当边 $< bb, bb_i >$ 没有被任何之前的测试用例覆盖, 否则返回 0。

在这个策略中，具有较高权重的种子将被优先模糊测试。值得注意的是，随着测试的进行，之前测试过的测试用例集合会发生变化，所以函数 $IsUntouched$ 的返回值也会改变。因此，一个测试用例的权重是动态的。

3.5 未触及邻居后代指导策略

在这个策略中，具有更多未触及的邻居后代的种子将被优先模糊测试。这些种子的变异有更高的概率去探索那些未触及的邻居后代。

更具体地说，本文使用未触及的邻居后代的数量作为测试用例 T 的权重，具体如下:

$$Weight\_Desc(T) = \sum_{\substack{bb \in Path(T) \\ IsUntouched(<bb,bb_i>)}} NumDesc\left(bb_i\right) \tag{6}$$

其中函数 $IsUntouched$ 返回 1 当且仅当边 $< bb, bb_i >$ 没有被任何之前的测试用例覆盖, 否则返回 0，函数 $NumDesc$ 返回从当前基本块开始的后代路径数量。它的正式定义如下:

$$NumDesc(bb) = \sum_{<bb,bb_i> \in EDGES} NumDesc\left(bb_i\right) \tag{7}$$

这里的权重是不确定的，因为函数 $IsUntouched$ 是动态的。然而，每个基本块的后代子路径的数量是确定的，因此可以使用静态分析来计算这个值，而不需要运行时间的开销。

3.6 内存访问指导策略

在这个策略中，具有更多访问内存操作的种子将被优先模糊测试。

更具体地说，本文使用访问内存操作的数量作为测试用例 T 的权重，具体如下:

$$Weight\_Mem(T) = \sum_{bb \in Path(T)} NumMemInstr\left(bb_i\right) \tag{8}$$

其中函数 $NumMemInstr$ 返回当前基本块访问内存的操作数，这个值可以被静态计算。因此，与前两种策略不同，以这种方式计算的权重是确定的。

4 复现细节

4.1 与已有开源代码对比

本次复现论文并无开源代码，但本次复现借鉴了其他人尝试复现的代码。其他人尝试复现的代码仅包含哈希碰撞算法部分，在本次复现中修正了其代码中对与新建基本块相关的边未分配哈希值的问题以及改进了一部分时间复杂度并独立复现了复现论文的三种种子选择策略。

本次复现的哈希碰撞解决算法如下所示:

Procedure 1 The collision mitigation algorithm.

Input: Original program
Output: Instrumented program
(BBS, SingleBBS, MultiBBS, Preds) = **GetCFG**()
Keys = **AssignUniqRandomKeysToBBs**(*(BBS*)
// Fixate algorithms. Preds and Keys are common arguments.
(Hashes, Params, Solv, Unsolv) = **CalcFmul**(*MultiBBS*)
(HashMap, FreeHashes) = **CalcFhash**(*Hashes, Unsolv*)
// Instrument program with coverage tracking.
InstrumentFmul(Solv, Params)
InstrumentFhash(Unsolv, HashMap)
InstrumentFsingle(SingleBBS, FreeHashes)

Procedure 2 CalcFmul

Input: *MultiBBS, Keys[], Preds[]*
Output: *Hashes, Params, Solv, Unsolv*
for y = 1 to log_2MAPSIZE **do**
  Hashes=∅, Params=∅, Solv=∅, Unsolv=∅
  for BB in MultiBBS **do**
    // search parameters for BB
    for x=1 to log_2MAPSIZE **do**
      for z=1 to log_2MAPSIZE **do**
        tmpHashSet=∅, cur=Keys[BB]
        // hashes for all incoming edges via Equation 2
        for p ∈ Preds[BB] **do**
          edgeHash = (cur ≫ x) ⊕ (Keys[p] ≫ y) + z
          tmpHashSet.add(edgeHash)
        end
        // Found a solution for BB if no collision
        if sizeof(tmpHashSet) == sizeof(Preds[BB]) **and** tmpHashSet ∩ Hashes == ∅ **then**
          Solv.add(BB)
          Params[BB] = <x, y, z>
          Hashes.extend(tmpHashSet)
        end
      end
    end
    Unsolv.add(BB)
  end
  // Found a good solution, if Unsol is small enough.
  if $sizeof(Unsol) < \Delta$ ***or*** $\frac{sizeof(Unsol)}{sizeof(BBSet)} < \delta$ **then**
    break
  end
end
return *(Hashes, Params, Solv, Unsolv)*

Procedure 3 CalcFhash

Input: *Hashes, Unsol, Keys[], Preds[]*
Output: *HashMap, FreeHashes*
HashMap=∅, FreeHashes= BITMAP_HASHES - Hashes
for BB in Unsol **do**
  cur=Keys[BB]
  for p ∈ Preds[BB] **do**
    HashMap(cur, Keys[p]) = FreeHashes.RandomPop()
  end
end
return *(HashMap, FreeHashes)*

4.2 实验环境搭建

实验环境基于 Ubuntu 22.04.1 LTS、Clang 11.0.0、LLVM 11.0.0、AFL 2.57b。

4.2.1 Clang 和 LLVM 的安装

LLVM 项目是模块化、可重用的编译器以及工具链技术的集合。Clang 是 LLVM 项目的一个子项目，基于 LLVM 架构的 C/C++/Objective-C 编译器前端。

从 github 下载预先编译好的二进制文件。

```
wget https://github.com/llvm/llvm-project/releases/download/llvmorg-11.0.0/
   clang+llvm-11.0.0-x86_64-linux-gnu-ubuntu-20.04.tar.xz
```

下载完毕后解压，并将其 bin 目录添加到环境变量中。在终端中输入以下命令即可输出对应版本号，借此判断是否安装成功。

```
clang --version
llvm-config --version
```

4.2.2 AFL 的安装

AFL(American Fuzzy Lop) 是由安全研究员 Michał Zalewski 开发的一款基于覆盖指导 (Coverage-guided) 的模糊测试工具，它通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率，增加发现漏洞的概率。

从 github 下载 AFL 源码。

```
git clone https://github.com/google/AFL
```

在其根目录执行 make 命令编译 AFL 的一系列工具，然后在其 llvm_mode 目录下执行 make 命令以编译基于 LLVM 的插桩工具，最后在其根目录执行 sudo make install 命令全局安装 AFL 的一系列工具。

```
make
cd llvm_mode
make
cd ..
sudo make install
```

4.3 界面分析与使用说明

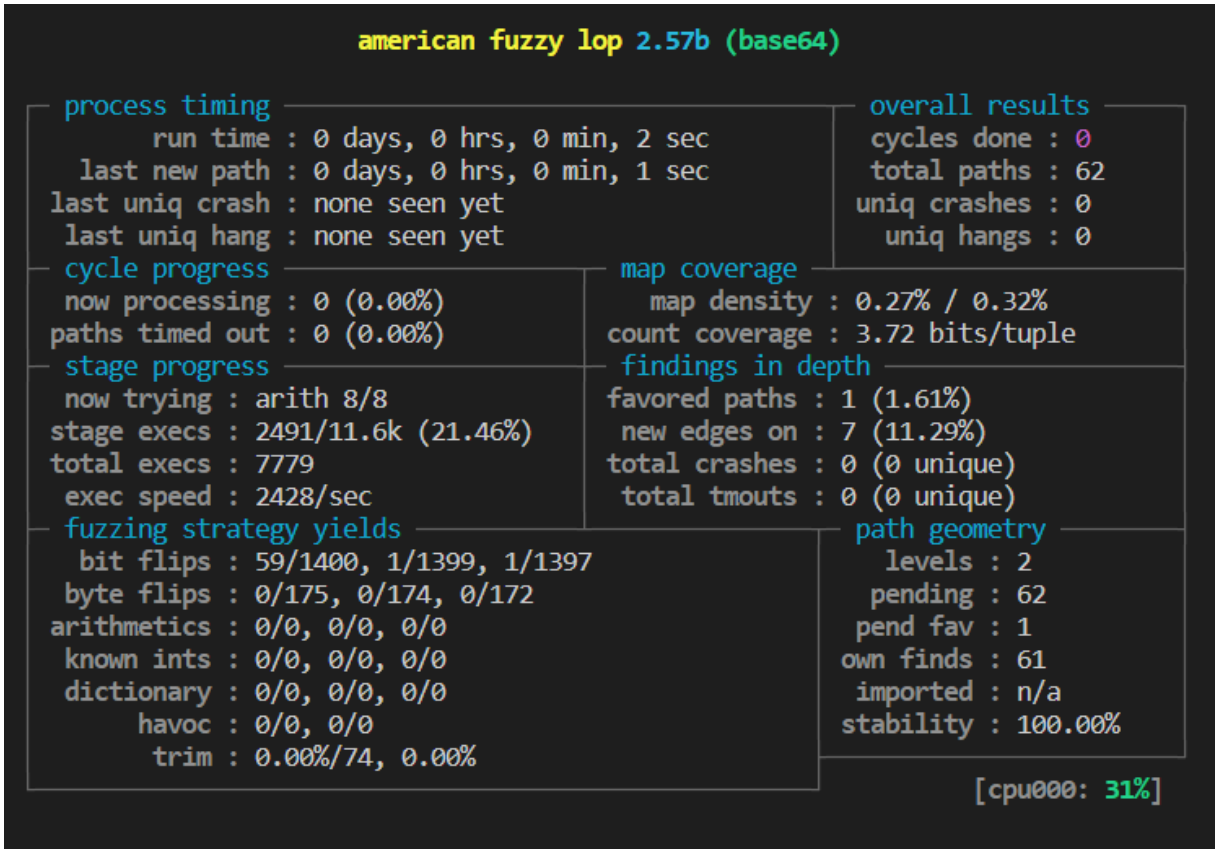

图 2: 界面示意

- Process timing: 运行时长、以及距离最近发现的路径、崩溃和挂起经过了多长时间
  - run time: 运行时长
  - last new path: 自从上次发现新路径过去的时间
  - last uniq crash: 自从上次发现新崩溃过去的时间
  - last uniq hang: 自从上次发现新超时过去的时间
- Overall results: 当前状态的概述
  - cycles done: 完成队列中所有用例一轮模糊测试的次数
  - total paths: 目前已发现的总用例个数
  - uniq crashes: 目前已发现的独特崩溃个数
  - uniq hangs: 目前已发现的独特超时个数
- Cycle progress
  - now processing: 当前正在测试的种子的 ID
  - paths timed out: 当前这一轮模糊测试中被丢弃的用例个数
- Map coverage： 目标二进制文件中的插桩代码所观察到覆盖范围的细节。
  - map density: 当前正在测试用例在位图中字节置位数与总空间大小的比值、全部测试用例位图中字节置位数与总空间大小的比值
  - count coverage: 位图中位置位数与总空间位数的比值，即位覆盖率

- Stage progress：现在正在执行的文件变异策略、执行次数和执行速度。
  - now trying: 当前所处的阶段名称
  - stage execs: 当前阶段的执行速度
  - total execs: 执行目标程序的次数
  - exec speed: 每秒执行目标程序的次数
- Findings in depth：有关找到的执行路径，异常和挂起数量的信息。
  - favored paths: 喜爱的用例个数
  - new edges on: 发现了新边的用例个数
  - total crashes: 所有的崩溃个数
  - total tmouts: 所有的超时个数
- Fuzzing strategy yields：关于变异策略产生的最新行为和结果的详细信息。
  - bit flips: 位翻转新发现的路径和独特崩溃总和
  - byte flips: 字节翻转新发现的路径和独特崩溃总和
  - arithmetics: 算数运算新发现的路径和独特崩溃总和
  - known ints: 有趣值新发现的路径和独特崩溃总和
  - dictionary: 字典新发现的路径和独特崩溃总和
  - havoc: 随便破坏新发现的路径和独特崩溃总和
  - trim: 减小用例大小的百分比、为减小文件大小执行目标程序的次数、虽然不可能删除，但被认为没有影响的字节的比例
- Path geometry：有关找到的执行路径的信息。
  - levels: 路径深度
  - pending: 在队列中等待被模糊测试的用例个数
  - pend fav: 在队列中等待被模糊测试的被喜爱的用例个数
  - own finds: 新发现的路径个数
  - imported: 并行模糊测试时导入的测试用例个数
  - stability: 相同的输入在目标程序中产生不同行为的程度。
- CPU load
  - cpu: CPU 利用率

使用方法：先编译目标程序

```
afl-clang-fast -o afl_test afl_test.c
```

对于直接从 stdin 读取输入的目标程序

```
afl-fuzz -i testcase_dir -o findings_dir /path/to/program […params…]
```

对于从文件读取输入的目标程序，@@ 就是占位符，表示输入替换的位置

```
afl-fuzz -i testcase_dir -o findings_dir /path/to/program @@
```

4.4 创新点

利用插桩来追踪样例走过的基本块。

将三种种子策略融合在一起，集成到复现的 CollAFL 中。

5 实验结果分析

本部分对实验所得结果进行分析，详细对实验内容进行说明，实验结果进行描述并分析。本次实验所测试的程序为 LAVA 测试集[14]中 LAVA-M 下的 base64。通过 LAVA 在 uniq、who、md5sum、base64 四个程序上进行 bug 插入而形成的测试集即为 LAVA-M。LAVA-M 被广泛应用于模糊测试领域的工具效果评估。

在 Ubuntu 终端执行以下命令以开启对 base64 的测试:

```
wget http://panda.moyix.net/~moyix/lava_corpus.tar.xz
apt-get install libacl1-dev
tar -xvf lava_corpus.tar.xz
cd lava_corpus/LAVA-M/base64/coreutils-8.24-lava-safe
sed -i 's/IO_ftrylockfile/IO_EOF_SEEN/' lib/*.c
echo "#define _IO_IN_BACKUP 0x100" >> lib/stdio-impl.h
cd ..
sed -i '1i#include <sys/sysmacros.h>' coreutils-8.24-lava-safe/lib/mountlist.c
sed -i 's/.\/configure/.\/configure FORCE_UNSAFE_CONFIGURE=1/g' ./validate.sh
export CC=afl-clang-fast
export CXX=afl-clang-fast++
./validate.sh
afl-fuzz -m none -t 5000 -i fuzzer_input/ -o outputs coreutils-8.24-lava-safe/
   lava-install/bin/base64 -d @@
```

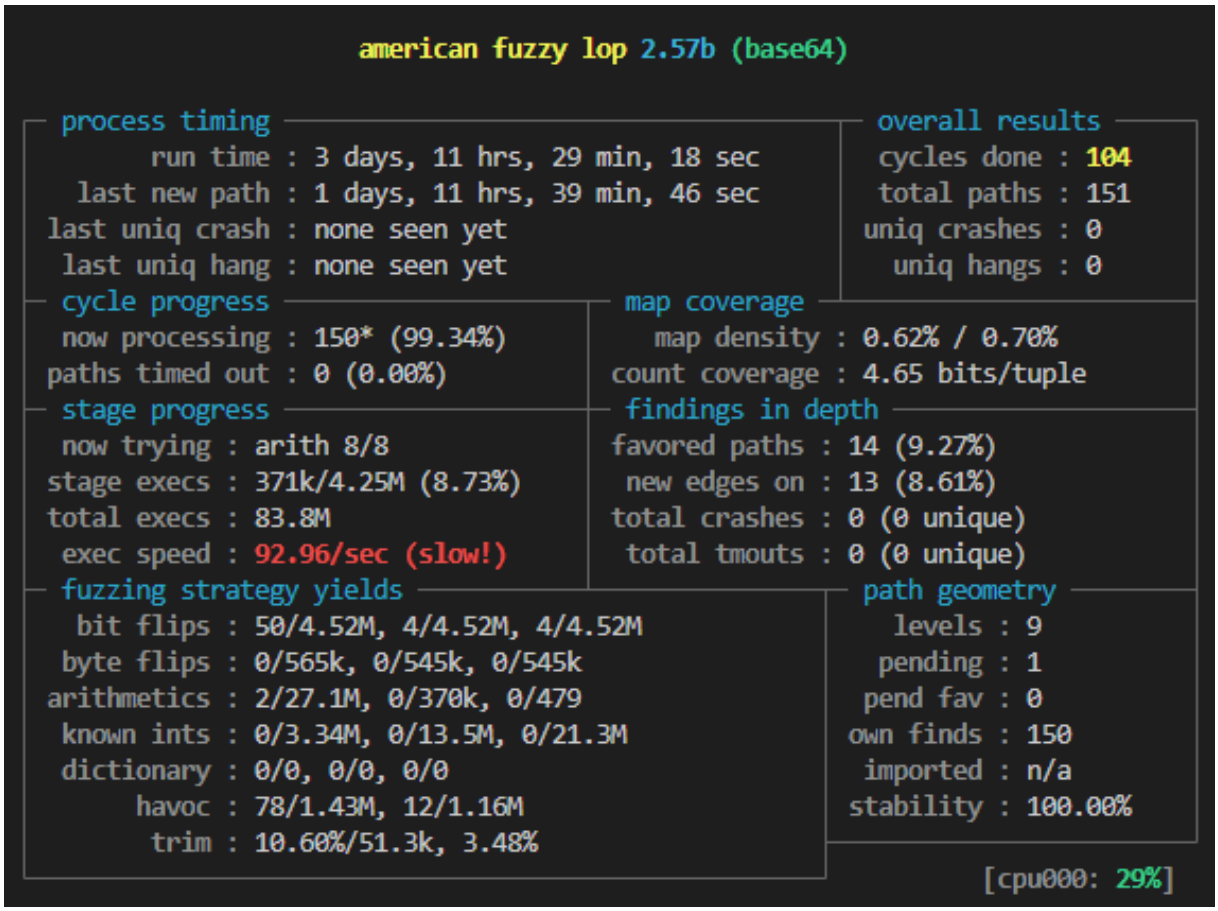

图 3: AFL 实验结果示意

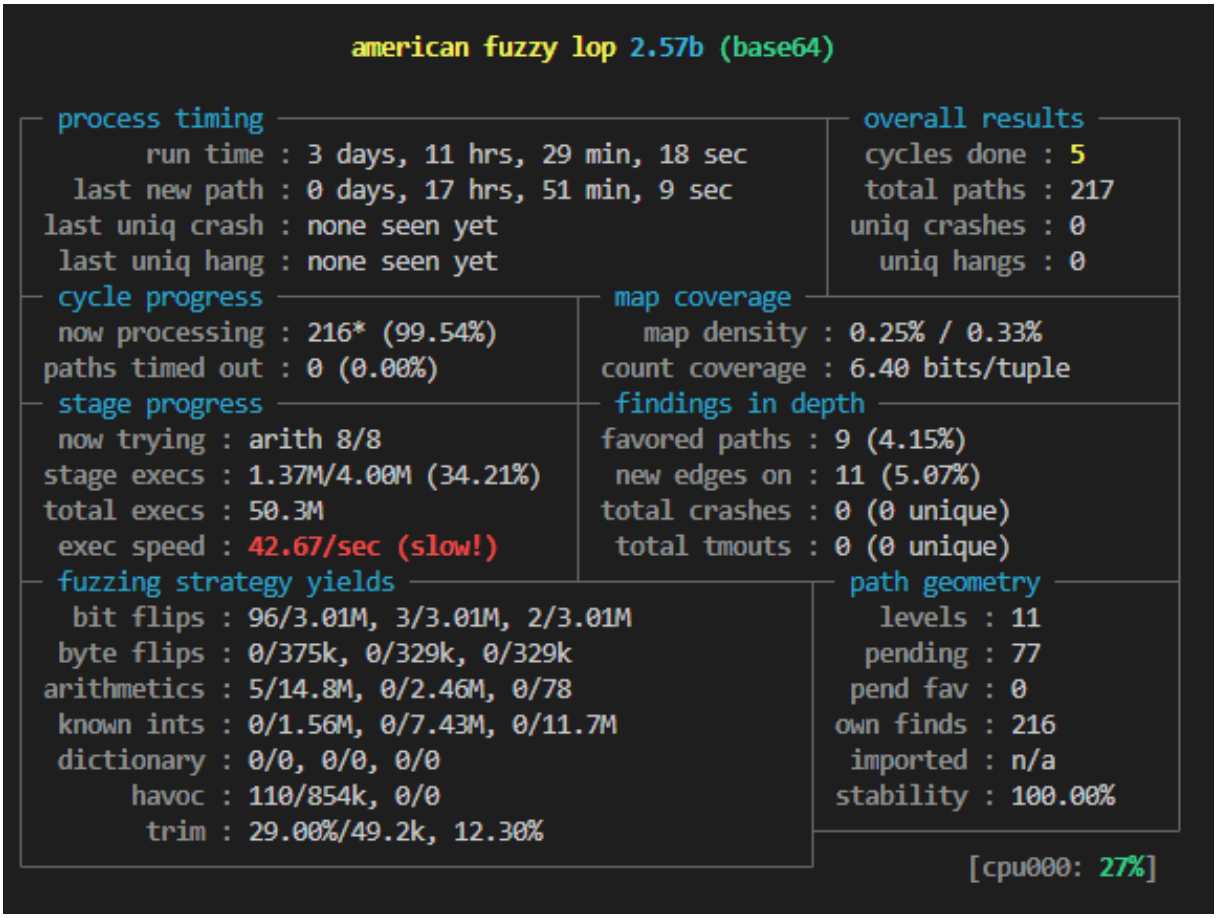

图 4: CollAFL 实验结果示意

图 3是原本的 AFL 对 base64 测试的结果，图 4是基于 AFL 修改的所复现的 CollAFL 对 base64 测试的结果。本文的核心思想在于解决 AFL 的哈希碰撞问题，从图 3的 total paths:151 可得 AFL 找到的路径总数为 151 条，而从图 4的 total paths:217 可得 CollAFL 找到的路径总数为 217 条，这正是由于解决了哈希碰撞问题从而使得找到的路径信息更加准确。

6 总结与展望

本文研究了覆盖率不准确对覆盖率指导型模糊测试工具的负面影响。本文提出了一个覆盖率敏感的模糊测试工具 CollAFL，它解决了最先进的模糊测试工具 AFL 中的哈希碰撞问题，使边缘覆盖信息更加准确，同时仍然保持了低插桩开销。基于准确的覆盖信息，本文提出了三种新的种子选择策略以直接驱动模糊测试工具朝向未探索过的路径。实验表明，在路径发现、崩溃发现和漏洞发现方面，这个解决方案既有效又高效。

本文解决哈希碰撞问题的方法依赖于编译器来获取边的信息，并相应地计算哈希值，以解决边之间的冲突。然而，通过静态分析获得一个准确的控制流图是一个困难的挑战，编译器有可能会遗漏一些边。本文的解决方案只保证解决已知边的冲突。研究如何分析程序以得到一个更准确的控制流图是未来可进一步进行研究的方向。

本文所提出的方法是可以扩展到二进制程序中的，但由于分析二进制程序异常困难，边的信息会变得非常不准确，可以预见程序在运行时哈希碰撞会增加。因此如何将本文所提出的方法扩展到二进制程序是未来可进一步进行研究的方向。

本文所提出的方法使用 AFL 的默认位图来跟踪边覆盖信息，并且没有考虑边的顺序。因此，信

息并没有理想的准确。使用其他解决方案以获得更准确的路径信息，可以预料到模糊测试的速度会减慢，但它也可以给模糊测试工具提供更理想的信息。因此如何减少跟踪路径的运行时开销并获取更准确的信息是未来可进一步进行研究的方向。

参考文献

[1] CHECKOWAY S, DAVI L, DMITRIENKO A, et al. Return-oriented programming without returns[C]//Proceedings of the 17th ACM conference on Computer and communications security. 2010: 559-572.

[2] SHACHAM H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)[C]//Proceedings of the 14th ACM conference on Computer and communications security. 2007: 552-561.

[3] SHACHAM H, PAGE M, PFAFF B, et al. On the effectiveness of address-space randomization[C]//Proceedings of the 11th ACM conference on Computer and communications security. 2004: 298-307.

[4] SEREBRYANY K. {OSS-Fuzz}-Google's continuous fuzzing service for open source software[J]., 2017.

[5] SEREBRYANY K. Continuous fuzzing with libfuzzer and addresssanitizer[C]//2016 IEEE Cybersecurity Development (SecDev). 2016: 157-157.

[6] SWIECKI R. Honggfuzz[J]. Available online a t: http://code. google. com/p/honggfuzz, 2016.

[7] ZALEWSKI M. American fuzzy lop[Z]. 2017.

[8] RAWAT S, JAIN V, KUMAR A, et al. VUzzer: Application-aware Evolutionary Fuzzing.[C]//NDSS: vol. 17. 2017: 1-14.

[9] LUK C K, COHN R, MUTH R, et al. Pin: building customized program analysis tools with dynamic instrumentation[J]. Acm sigplan notices, 2005, 40(6): 190-200.

[10] 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.

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

[12] WANG S, NAM J, TAN L. QTEP: Quality-aware test case prioritization[C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 523-534.

[13] PETSIOS T, ZHAO J, KEROMYTIS A D, et al. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2155-2168.

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