调研报告：并行与集成模糊测试

作者：李秋豪　时间：2021年7月20号　版本：0.2.0　链接：CN_Parallel_Ensemble_Fuzzing.pdf

摘要

Fuzzing（模糊测试）作为一种测试和漏洞挖掘技术，在各大企业的生产环境中得到了广泛应用。但由于其密集生成随机输入集的特点，通常会给计算机系统带来性能上的压力。同时，不同的 Fuzzing 方案对于不同的软件，或对于同一软件的不同阶段适应性都不同，测试人员难以在运行前决定最优的测试方案。为了解决这两个问题，学术界和工业界出现了一些能够并行不同 Fuzzing 方案进行集成测试的框架。这些框架虽然产生了效果，但各自仍留有一些问题。例如对于单机多核（100核以上）的并行环境，不同 Fuzzing 方案的信息共享，以及测试任务的分隔和分配依然缺乏高效的支持。本文调研分析了并行与集成模糊测试领域的相关论文，总结了该领域主要的挑战和思路，最后提出了一种解决方案。

1 引言

Fuzzing 是一种测试手段，其通过**自动化的构造随机、畸形、非常规的输入**，试图引发程序的异常，从而发现程序中的 Bug。Fuzzing 的普遍算法如 图 1–1 所示，在获得初始语料集（种子）后，Fuzzer 会选择此次测试的语料，并对其进行生成变异等操作。随后以得到的输入运行被测试程序，获得运行的状态信息，比如是否出现内存异常以及覆盖度的变化，以此更新语料集并指导下一次测试。目前，Fuzzing 技术已广泛被测试人员和安全研究人员使用。

ALGORITHM 1: Fuzz Testing

```
Input : ℂ, t_limit
Output : 𝔹 // a finite set of bugs
1 𝔹 ← ∅
2 ℂ ← PREPROCESS(ℂ)
3 while t_elapsed < t_limit ∧ CONTINUE(ℂ) do
4     conf ← SCHEDULE(ℂ, t_elapsed, t_limit)
5     tcs ← INPUTGEN(conf)
      // O_bug is embedded in a fuzzer
6     𝔹′, execinfos ← INPUTEVAL(conf, tcs, O_bug)
7     ℂ ← CONFUPDATE(ℂ, conf, execinfos)
8     𝔹 ← 𝔹 ∪ 𝔹′
9 return 𝔹
```

图 1–1 Fuzzing 工作的普遍算法 [1]

为了提高 Fuzzing 的效率，出现了许多对于 Fuzzing 算法实现上的优化。例如：

1. MOPT [2] 将每一个变异操作看成一个粒子，使用粒子群优化算法动态的去调整各个变异操作所占的比重；
2. REDQUEEN [3] 通过输入和程序的状态关系，以一种“轻量级”的污点分析手段判断输入中哪些字段能够影响程序中的分支判断，然后尝试直接对这些字段进行变异，从而提高变异的效率；
3. AFLGo [4] 在每次测试过程中，会统计当前输入所能到达的基本块和预定义的目标基本块的距离，并以此为反馈指导调度，尝试在最短时间内生成能够到达目标基本块的输入；
4. NAUTILUS [5] 通过使用自己定义的上下文无关语言描述输入结构，并在语法树上对输入进行变异，以此生成具有语法规则的语料集。
5. FIFUZZ [6] 通过使用错误注入的技术，并结合程序上下文对注入进行变异，以此测试程序中的错误处理模块。

以上优化大多局限在单个 Fuzzer 内部。而在现在的生产环境中，多核多机集群已经很常见，于是许多 Fuzzing 工具支持同时运行多个 Fuzzer，以此达到更好的测试效果。例如对于 AFL [7]，其支持并发运行一个主 Fuzzer 和多个从 Fuzzer，这些 Fuzzer 每隔一段时间会扫描其他 Fuzzer 的语料集，并尝试运行这些语料，分析其是否对自己的测试过程有帮助，随后复制到自己的语料集中。Google 的 ClusterFuzz（后端）和 OSS-Fuzz（前端）通过在100,000多个虚拟机 [8] 上对500个开源项目进行测试，产出了超过30,000个 Bug [9]。

本文主要关注并行和集成 Fuzzing，首先我们定义这两个概念。并行 Fuzzing：**同时运行多个 Fuzzer**，这些 Fuzzer 之间通过交互执行信息达到更好的测试效果。集成 Fuzzing：**运行多个不同 Fuzzing 方案的 Fuzzer**，这些 Fuzzer 之间通过交互执行信息达到更好的测试效果。可见，集成 Fuzzing 大多可以在并行的环境下进行。

接下来的文章组织如下：第二节分析近几年关于并行、分布式以及集成 Fuzzing 相关的学术论文；第三节总结出这一领域面临的挑战和通用的思路；第四节描述我们对于这些挑战所提出的解决方案——LilacFuzz 的框架设计；最后一节给出我们对于 LilacFuzz 在实现上的时间计划。

2 相关工作

目前，针对于并行、分布式和集成模糊测试的论文并不多。我们选取了5篇比较典型的方向和论文，按照时间线进行简要讲解。每一个讲解可分为3个部分：现象与问题；设计实现与评估；未来工作与不足。

2.1 提高并行测试的扩展性

现象与问题

Wen xu et al. [32] 通过在单机多核系统上运行 Fuzzer，发现随着并行数量上升，实际用于 Fuzzing 的资源总量和占比都出现了瓶颈，效果甚至不如并行数量更小的情况。作者在1–120核上并发运行 AFL，效果如 图 2–1 所示。

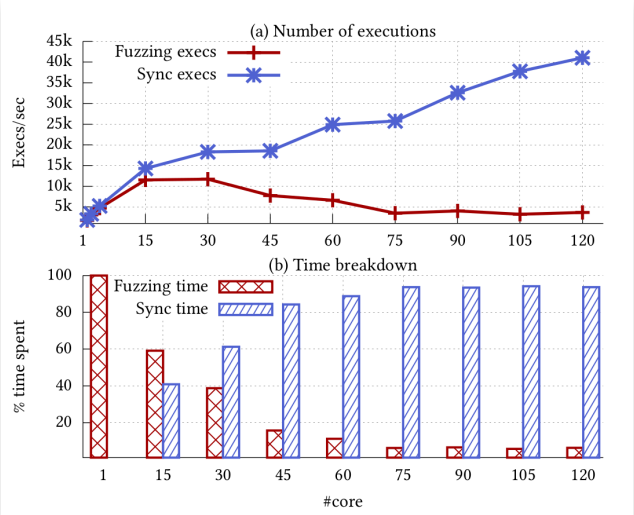

图 2–1 AFL在多核系统上测试libjpeg

随后作者分析了 AFL 的并行工作流程，如 图 2–2 所示。

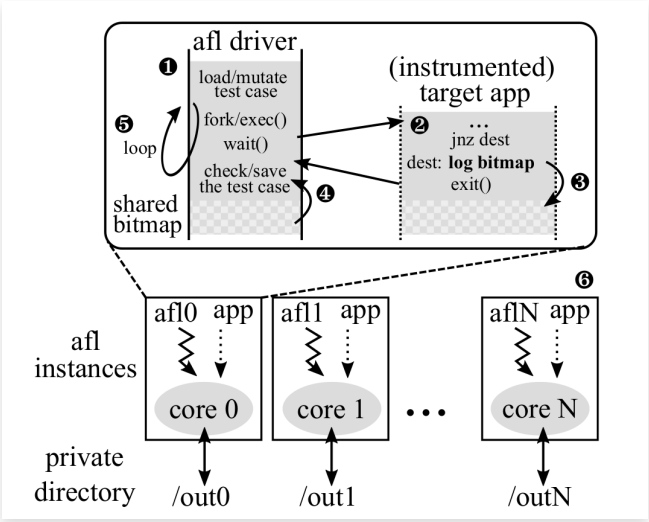

图 2–2 AFL的并行设计

作者认为，AFL 测试过程中需要大量使用 **fork(2)** 系统调用，同时会在**文件系统**上读写创建大量的语料集，而且**同步语料**的时候也需要每个 Fuzzer 重新执行一遍输入，是这三个问题导致了 AFL 在大规模并行的时候产生了性能瓶颈。于是作者分别针对这三点提出了解决方案。

设计实现与评估

（1）对于 fork(2) 所带来的性能瓶颈，主要是由于 fork(2) 本质上是创建了两个进程，在这个过程中会出现大量不利于并行的竞争。例如 POSIX 标准规定 PID 应该是递增的，许多内核数据结构在复制构建的过程也需要串行化。此外内核还会进行许多与 Fuzzing 本身无关的操作，例如对进程进行安全权限检查，将进程放入调度器进行调度等。而作为 Fuzzing 来说，我们真正需要的只是 fork(2) 对进程的镜像功能（Copy-on-Write 等）。所以作者设计并实现了一种新的系统调用 `snapshot(unsigned long cmd, unsigned long callback,struct iovec *shared_addr)`，其作为 Linux 内核模块加载到系统中，工作过程如 图2-3 所示。

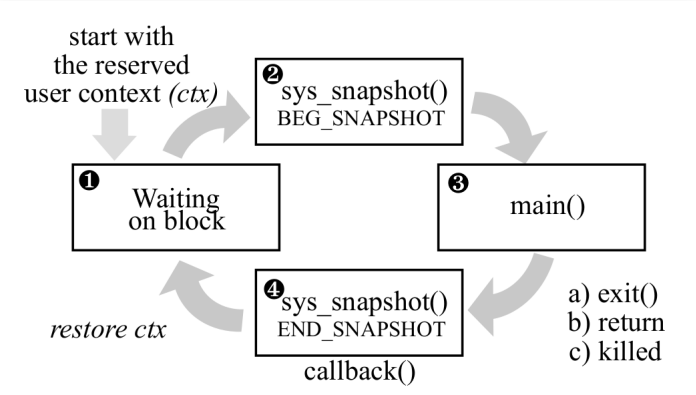

图 2-3 snapshot(2)的工作流程

在该系统调用中，cmd 指定了 snapshot 起始和结束，callback 是当次 fuzzing 结束时回调的函数，shared_addr 描述的是在这次 fuzzing 中不需要镜像保护的内存区域（例如覆盖度反馈位图）。

对于一次 Fuzzing ，首先用户通过 sigsetjmp(3) 保存用户态上下文 ctx，随后调用 cmd 为 BEG_SNAPSHOT 的 snapshot(2)，内核扫描该进程的 VMA，记录这些虚拟内存区域的属性，并设置为只读，以及诸如 brk，文件描述符表等进程相关的内核数据。在测试中，当一个 VMA 被修改时，会触发指令异常，内核随即根据 VMA 的记录进行处理，如果 snapshot(2) 之前是可写的，则会进行 Copy-on-Write 处理。当被测试的程序结束时（例如其主动调用 cmd 为 END_SNAPSHOT 的 snapshot(2)，exit(2)，或被 kill），内核会恢复进程的 VMA 和其他内核属性，并捕获其终止原因，以此为参数调用 callback 函数，在 callback 函数执行后，用户会使用 siglongjmp(3) 恢复用户态上下

文。可见，通过使用该系统调用，我们避免了 fork(2) 中 namespaces，cgroup，scheduler 等模块产生的 Fuzzing 不相关以及对并行不友好的操作。

（2）关于 AFL 对文件系统的压力，作者认为来自于两个方面。一是 AFL 会实时读、写，创建中间语料文件，这些文件大多只有 KB 级别，会对磁盘 IO 和文件系统产生性能上的压力。二是在进行同步的时候，一个 Fuzzer 会遍历其他 Fuzzer 所有的语料集文件夹，以此对文件系统产生同步上的压力。

作者考虑到对于 Fuzzing 过程中产生的中间文件，其并不需要严格的一致性、持久化存储保证，真正需要这些保证的只有最终的语料集和 crash 样本。为了减轻上述问题对于文件系统和磁盘的压力，作者设计了一种“双层”文件系统：对于每个 Fuzzer 的直接语料集，采用 tmpfs [10] 这种内存文件系统，由于 tmpfs 没有持久化文件系统所具有的日志、写回磁盘等操作，使得其能够在高并行处理的情况下保证较好的性能。同时，为了兼顾持久化存储，外部的 ext4 文件系统会定期同步 tmpfs 中的语料集。最后，为了避免同一文件夹下并行操作带来的同步损耗，每一个语料集都有自己单独的分区。

关于 tmpfs 在并发上的优势，可以参考 USENIX ATC 2016 的一篇论文 [11]，如 图 2-4 所示，tmpfs 在所有文件系统中均是性能上限。Jussi Judin 在2018年使用 AFL 对不同文件系统进行了性能测试 [12]，如 图 2-5/6 所示，无论考虑 Fuzzing 性能还是对磁盘的读写损耗，tmpfs 都是最佳方案。

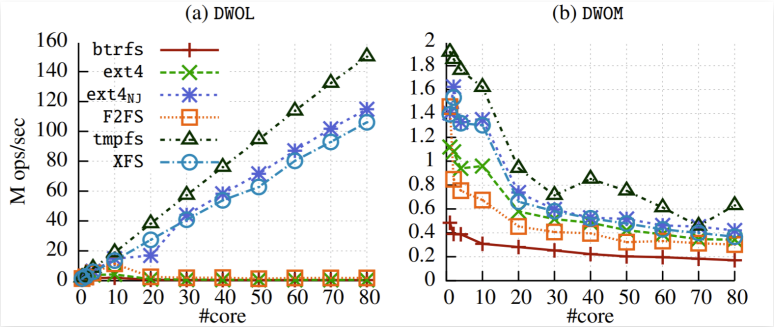

图 2-4 不同文件系统对于私有文件数据块写（DWOL）和共享文件数据块写（DWOM）的性能对比

execs/second	stdin	file
btrfs	20.5 k	14.6 k
ext2	31.6 k	19.2 k
ext3	30.0 k	18.4 k
ext4	28.0 k	18.2 k
JFS	31.7 k	17.1 k
ReiserFS	28.0 k	14.5 k
tmpfs	**34.6 k**	**25.5 k**
XFS	21.1 k	16.8 k

图 2-5 在不同文件系统下，AFL（非并行）2分钟内对于典型被测程序测试次数

writes/month	stdin x4	file x4
btrfs	**9.53 TiB** (78 %)	0.052 TiB (65 %)
ext2	0.072 TiB (**450 %**)	0.120 TiB (**600 %**)
ext3	0.075 TiB (94 %)	0.147 TiB (130 %)
ext4	0.090 TiB (64 %)	0.088 TiB (100 %)
JFS	**44.4 TiB** (110 %)	**339 TiB** (120 %)
ReiserFS	0.069 TiB (82 %)	0.072 TiB (75 %)
tmpfs	-	-
XFS	**786 TiB** (98 %)	0.035 TiB (220 %)

图 2-6 在不同文件系统下，4核并行 AFL 一个月对磁盘的读写大小

（3）对于粗粒度同步（只有语料）所带来的性能压力，主要是来自于其固有的设计。通过简单分析可得，每一个 Fuzzer 的时间复杂度为O(m * n) * ExecTime。其中 m 为 Fuzzer 的数量，n 为每个 Fuzzer 产生的语料数量， ExecTime 为对于每个语料 Fuzzer 的平均执行时间。同时如上文所说，同步中遍历的过程也会对文件系统和磁盘产生压力。

考虑到所有的 Fuzzer 都是同质（Fuzzer 和被测试程序均相同）的，作者设计出一种基于内存的消息队列同步方案，通过同步更细粒度的执行信息——文件名、元数据、覆盖度位图，来避免 Fuzzer 在同步的过程中需要执行其他 Fuzzer 产生的每一个语料。其结果如图 2-5 所示。

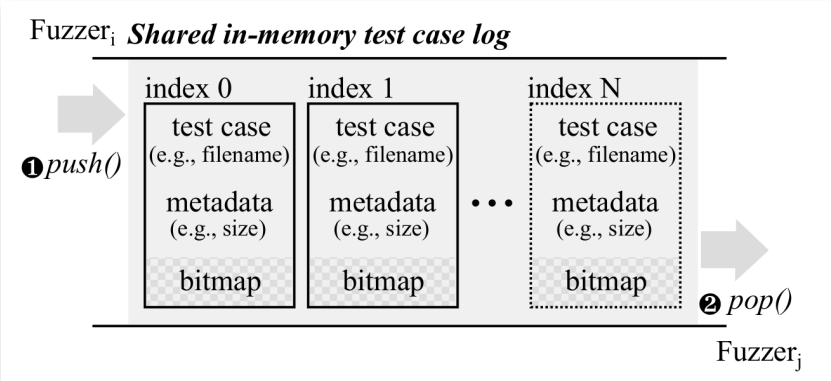

图 2-5 基于细粒度执行信息的同步方案

每个 Fuzzer（master）都会构建自己的环形消息队列，同时维护一个全局的 HEAD 和 TAIL 。当产生新的语料时，master 会向自己的队列上 push 语料信息，而其他Fuzzer在同步的时候会对该队列维护一个局部的 TAIL，从该队列中 pop 出语料信息并判断该语料是否对自己有帮助。只有当所有其他 Fuzzer 都 pop 一个语料后，master 才会更新全局的 TAIL。由于执行信息中包含覆盖度位图，其他 Fuzzer 不用执行语料就能判断该语料是否能给自己带来新的覆盖度，从而将时间复杂度减小到 O(m * n)。

对于评估，作者使用 Google's fuzzer test suite [13] 中的部分程序对采用上述三种优化方案的架构进行测试，效果如 图 2-6 所示。

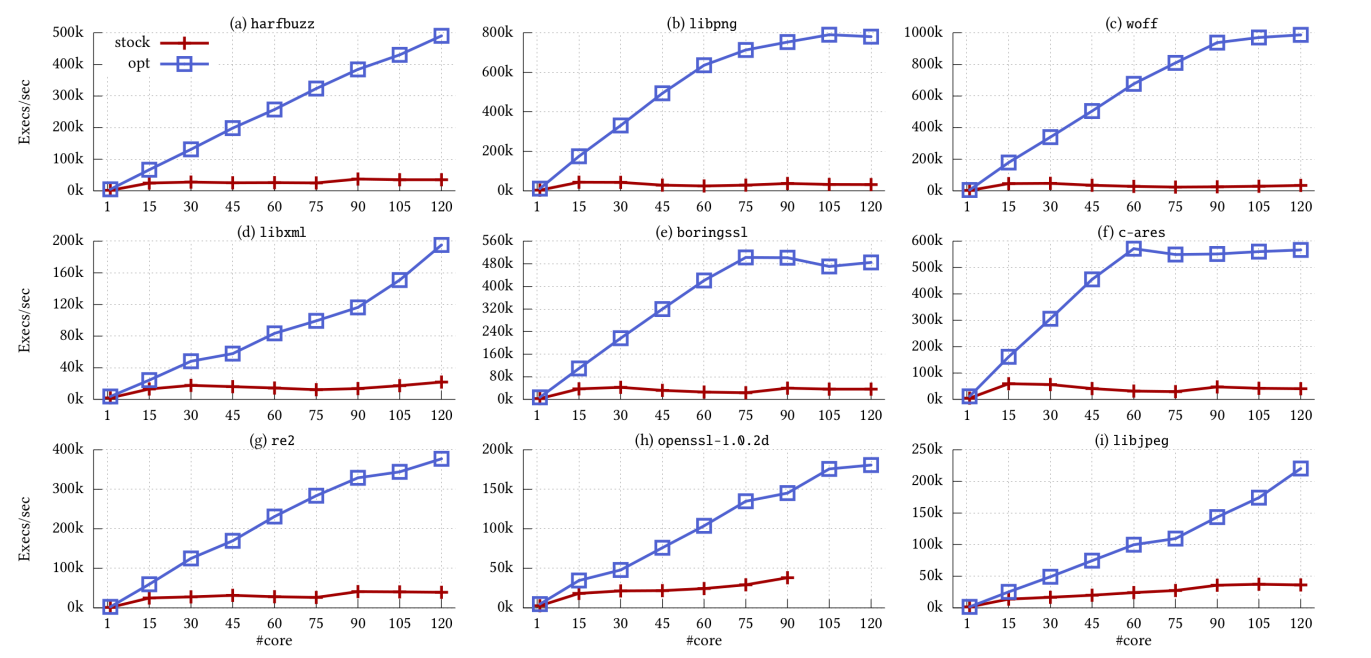

图 2-6 优化方案（opt）和原方案（stock）每秒测试次数对比

可见，对于大多数程序优化后的方案都能随着并行数量呈现线性增长，只有部分程序在60核左右出现了瓶颈，这可能是由于内存申请释放等管理操作带来的性能损耗。此外，为了测试每个优化的必要性，作者还设计了对照组，分别使用未使用 fork 优化，未使用文件系统优化，未使用消息队列优化的方案进行测试，也得到了印证假设的结果。

未来工作与不足

1. 对于未来的工作，作者提出了三点。一是该论文只对 LibFuzzer 和 AFL 这类针对输入型程序的 Fuzzer 进行了优化，对于其他的 Fuzzer，例如针对内核或文件系统的 Fuzzer，并行的性能瓶颈可能会在不同的地方。二是该论文只针对于 GNU/Linux 平台进行了优化。三是现在很多 Fuzzer 是跑在云/虚拟化环境下，客户机与宿主机之间的透明性可能会带来一些性能上的损耗。
2. 我们是否可以将语料去重、分析的任务单独剥离出来，采用独立的线程/核进行处理？这样时间复杂度就会降为 O(n)（参见2.5节）。
3. 本文假设所有的 Fuzzer 的执行信息，如反馈度位图，都是同质（相同格式和意义）的，这导致无法并行多个不同的 Fuzzer。可以采用单独的模块不同的 Fuzzer 语料分析产生同质的执行信息。
4. 由于该文对于 snapshot(2) 的实现采用了 Linux 内核模块，但由于 Linux 内核 API 不稳定，该模块维护起来的难度较高。例如对于 AFL 的后继项目 AFL++ [14]，其已放弃维护对应的模块 [15]。或许我们可以在用户态实现对应的镜像功能 [16]。

2.2 集成不同测试方案

这篇论文虽然发于 USENIX Security 2019，但其预印版在2018年就已发在 arXiv 上，所以放在2.2节。

现象与问题

目前的 Fuzzer 例如 AFL 和 LibFuzzer 均支持并行测试，但其并行的 Fuzzer 都是同样的方案。Yuanliang Chen et al. [33] 认为，通过不同 Fuzzing 方案的种子同步可以达到更好的测试效果。

```
void crash(char* A, char* B){
    if (A == "Magic Str"){              => T1
        if (B == "Magic Num") {
            bug();                      => T4
        }else{
            normal();                   => T3
        }
    }else if (A == "Magic Num"){        => T2
        if (B == "Magic Str"){
            bug();                      => T5
        }else{
            normal();                   => T6
        }
    }
}
```

图 2-7 被测试程序

例如对于 图 2-7 所示程序。假设 Fuzzer1 对于“Magic Str”条件语句有较好的分析能力，能够生成通过 `X == "Magic Str"` 的输入。而 Fuzzer2 对于“Magic Num”条件语句有较好的分析能力，能够生成通过 `Y == "Magic Num"` 的输入。如果我们只运行 Fuzzer1 ，语句覆盖度将是 T1，T3；如果只运行 Fuzzer2，语句覆盖度将是 T2，T6；如果我们同时运行 Fuzzer1，Fuzzer2 但是不同步种子，语句覆盖度将为 T1，T3，T2，T6；如果我们同时运行 Fuzzer1，Fuzzer2 并同步它们产生的种子，那么当 Fuzzer1 通过 T1 语句后，Fuzzer2 将产生通过“Magic Num”语句的输入，到达 T4，同理 Fuzzer1 将到达 T5，最终的覆盖率将为 T1，T2，T3，T4，T5，T6。可见通过同步种子可以达到“1+1 > 2”的测试效果，作者将这种方法称为“集成模糊测试（Ensemble Fuzzing）”。

设计实现与评估

显然，参与集成测试的 Fuzzer 种类越多，我们能够对复杂程序语句的探索能力和测试效果就越好。作者首先定义了 Fuzzer 的多样性：（1）覆盖度反馈的粒度（2）语料生成的策略（3）语料变异和调度的策略。并从三个角度出发，选择了 AFL（作为对比基础），AFLFast [23]（变异与调度策略），FairFuzz [24]（变异与调度策略），libFuzzer [25]（覆盖度反馈粒度）， Radamsa [26]（语料生成），QSYM [27]（语料生成）作为 Fuzzing 组合的 Base Fuzzer。并设计了一种能够全局共享 Base Fuzzer 种子的测试框架 EnFuzz，其整体框架和算法如 图 2-8 和 图 2-9 所示。

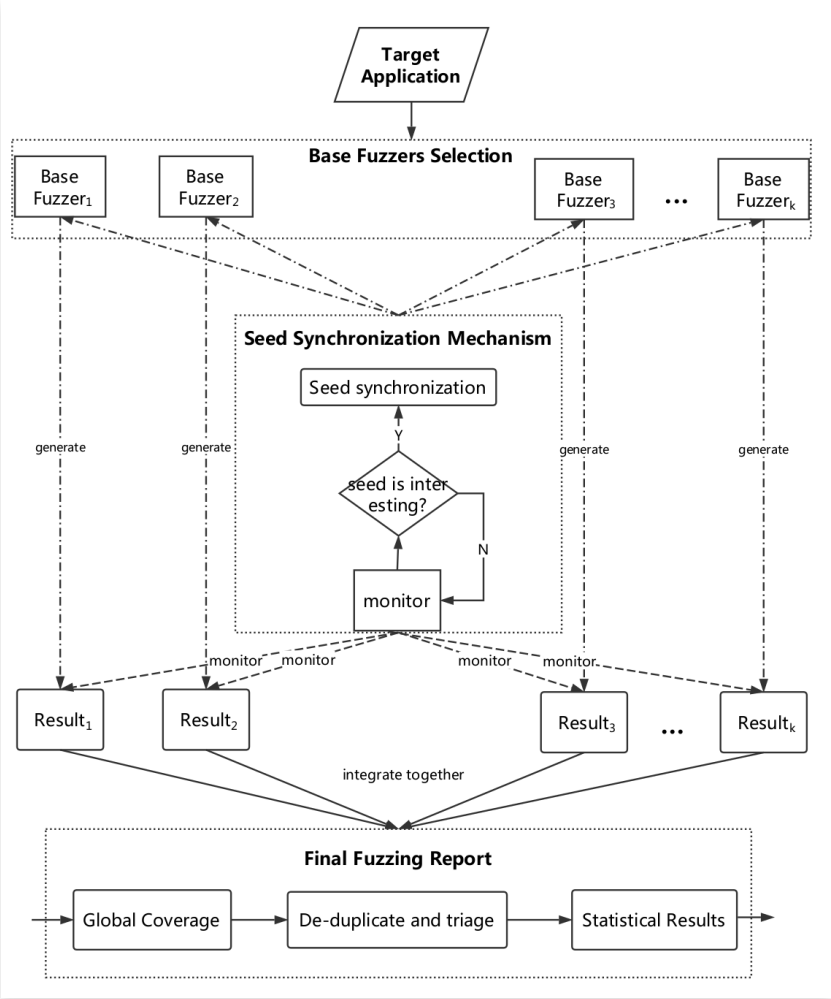

图 2- 8 种子同步整体架构

对于 Base Fuzzer，它们会维护一个一个局部的语料队列，用于保存自己产生的语料。同时，当发现产生新路径的种子后，Base Fuzzer 还会将语料上传到全局种子共享池（基于文件系统）中，使种子能被其他 Base Fuzzer 使用。

```
repeat
    foreach seed s of the GlobalSeedPool do
        // Skip synchronized seeds ;
        if s.isSync() == False then
            foreach base fuzzer f of the BaseFuzzers[] do
                Cover = f.run(s) ;
                // update the global coverage ;
                newCover =
                 (Cover ∪ GlobalCover) − GlobalCover ;
                GlobalCover = Cover ∪ GlobalCover;
                // synchronize the seed s to base fuzzer f ;
                if Cover.causeCrash() and
                 !newCover.isEmpty() then
                    crashes.push(s);
                    f.queue.push(s);
                else if !newCover.isEmpty() then
                    f.queue.push(s);
                else
                    continue;
                end
            end
        else
            continue;
        end
        s.setSync(True);
```

图 2–9 全局种子同步算法

每隔固定时间（作者选择120秒），全局种子共享池的管理器会扫描新的语料，然后依次选择所有的 Base Fuzzer 运行该语料，更新全局覆盖度位图，并判断该次运行是否能够产生新的全局覆盖度。如果产生了新的覆盖度，全局管理器会将该语料推送到 Base Fuzzer 的局部语料队列中，使 Base Fuzzer 能够在未来使用该语料。

作者分别用 Base Fuzzer 和 EnFuzz 对 Google's fuzzer-test-suite 进行4核并行测试，EnFuzz 在覆盖度上超过了所有 Base Fuzzer。

Project	AFL	AFLFast	FairFuzz	LibFuzzer	Radamsa	QSYM	EnFuzz
boringssl	3286	2816	3393	5525	3430	2973	**7136**
c-ares	146	116	146	191	146	132	**253**
guetzli	3248	2550	1818	3844	3342	2981	**4508**
lcms	1682	1393	1491	1121	1416	1552	**2433**
libarchive	12842	10111	12594	22597	12953	11984	**31778**
libssh	110	102	110	362	110	149	**377**
libxml2	14888	13804	14498	28797	17360	13172	**35983**
openssl-1.0.1	3992	3501	3914	2298	3719	3880	**4552**
openssl-1.0.2	4090	3425	3956	2304	3328	3243	**4991**
openssl-1.1.0	4051	3992	4052	2638	3593	4012	**4801**
pcre2	79581	66894	71671	59616	78347	60348	**85386**
proj4	342	302	322	509	341	323	**709**
re2	12093	10863	12085	15682	12182	10492	**17155**
woff2	23	16	20	447	22	24	**1324**
freetype2	19086	18401	20655	25621	18609	17707	**27812**
harfbuzz	12398	11141	14381	16771	11021	12557	**16894**
json	1096	963	721	1081	1206	1184	**1298**
libjpeg	1805	1579	2482	1486	1632	1636	**2638**
libpng	582	568	587	586	547	606	**781**
llvm	8302	8640	9509	10169	8019	7040	**10935**
openthread	268	213	230	1429	266	365	**1506**
sqlite	298	322	294	580	413	300	**636**
vorbis	1484	1548	1593	1039	1381	1496	**1699**
wpantund	4914	5112	5691	4881	4891	4941	**5823**
Total	190607	168372	186213	209574	188274	163097	**271408**
Improvement	–	11% ↓	2% ↓	9% ↑	1% ↓	14%↓	**42%** ↑

图 2–10 Base Fuzzer 和 EnFuzz 对 Google's fuzzer-test-suite 的路径覆盖度对比

另外，作者设计了 EnFuzz-A（AFL，AFLFast，FairFuzz），EnFuzz-Q（AFL，AFLFast，FairFuzz，QSYM），EnFuzz-L（ AFL，AFLFast，FairFuzz，libFuzzer），以及EnFuzz-（关闭种子同步功能）作为对照组进行测试。效果与预期吻合。作者还发现，对于 LAVA-M [28] 这种代码量（2K-4K LOCs）较小的被测试程序，拥有符号执行的 EnFuzz-Q 表现最好，而对于代码量大的程序其表现较差。最后 AFLFast 在并行时性能与原 AFL 的并行模式相比有较大下降。

最后，作者尝试使用 Base Fuzzer 和 EnFuzz 对真实软件进行测试，EnFuzz 的产出再次超过了所有 Base Fuzzer，效果如 图 2-11 所示。

Project	AFL	AFLFast	FairFuzz	LibFuzzer	QSYM	EnFuzz
Bento4_mp4com	5	4	5	5	4	**6**
Bento4_mp4tag	5	4	4	5	4	**7**
bitmap	1	1	1	0	1	**2**
cmft	1	1	0	1	0	**2**
ffjpeg	1	1	1	0	1	**2**
flif	1	1	1	2	1	**3**
imageworsener	**1**	0	0	0	**1**	**1**
libjpeg-05-2018	3	3	3	4	3	**5**
libiec61850	3	2	2	1	2	**4**
libpng-1.6.34	2	1	1	1	2	**3**
libwav_wavgain	3	2	3	0	2	**5**
libwav_wavinfo	2	1	2	4	2	**5**
LuPng	1	1	1	3	1	**4**
pbc	5	5	6	7	6	**9**
pngwriter	1	1	1	1	**2**	**2**
total	35	28	31	34	32	**60**

图 2-11 4核并行测试24小时后各 Fuzzer 发现的 Bug 数量

未来工作与不足

1. 作者认为目前从三个方面定义 Fuzzer 多样性可能不够全面和准确。并提出了一种多样性计算方法如 图 2-12 所示。其中 n 是被测试程序数量，pi 是该 Base Fuzzer 发现的路径数量，pA 是基准 Fuzzer（例如 AFL）发现的路径数量，最后的多样性是对所有程序测试的方差大小。可见，这种方案无法之间测试所有 Fuzzer 集合的多样性大小（参见2.4节），同时路径数量这一单位的粒度也较大，可以尝试使用不同路径的数量。

$$mean = \frac{1}{n}\sum_{i=1}^{n}\frac{p_i - p_{A_i}}{p_{A_i}} \tag{1}$$

$$diversity = \frac{1}{n}\sum_{i=1}^{n}\left(\frac{p_i - p_{A_i}}{p_{A_i}} - mean\right)^2 \tag{2}$$

图 2-12 Base Fuzzer 的多样性计算

2. 作者认为可以根据不同的 Base Fuzzer 的运行状态动态分配不同的计算资源（例如 CPU 和内存），以此达到更快的覆盖度提升。
3. 在作者给出的全局种子同步算法中，只有当种子在 Base Fuzzer 中能对全局覆盖度产生共享时才把该种子推送给 Base Fuzzer。但是对于些覆盖度反馈相同，其他方案不同的 Fuzzer，例如 AFL 和 AFLFast，当 AFLFast 发现一个新的种子并更新全局覆盖度位图后，AFL 再次运行该种子并不能产生新的全局覆盖度，也就不会被推送该种子，但 AFL 是可能需要这个种子的 [17]。
4. 作者在算法和实现中并没有说明对于覆盖度反馈不同的 Fuzzer 如何更新全局覆盖度。
5. 作者只在4核上测试了 EnFuzz 的并行性能。但我们可以看到，从复杂度考虑 EnFuzz 的种子同步方案和 AFL 本来的方案没有区别，每一个 Fuzzer 还是要执行其他 Fuzzer 的所有种子，这在大规模并行测试的时候或许会出现性能瓶颈（参见2.1节）。
6. 作者对于 EnFuzz 同步种子的时间间隔是基于经验静态设计的。或许我们可以参考 AFLSmart [18] 中的“deferred generation”——当 Base Fuzzer 自己能够快速发现新路径时暂缓同步，反之加快同步。
7. 值得注意的是，OSS-Fuzz 的后端 Cluster Fuzzer 称自己使用了 EnFuzz 的同步机制 [19]，但其实用的是原始版本的 AFL++ 和 LibFuzzer 等 Fuzzer 的并行模式。从文档可知 OSS-Fuzz 现在对单个项目的测试环境还是单核 [20]，这在大规模并行时或许会遇到问题（参见2.1节）。
8. 最后，从 图 2-11 中可以看出，AFL 依然能够发现足够多的 Bug，对于公司来说，60和35个 CVE ID 可能并没有本质上的区别……或许现在的研发重点应该放在“安全左移”上，例如使用安全的语言、库以及策略。
9. 闭源。

2.3 分隔测试任务

现象与问题

如引言所说，传统上的 Fuzzing 优化大多集中于单个 Fuzzer 本身的算法和实现上，例如 AFLFast 和 FairFuzz。这些优化在非并行模式下都能取得比原始的 AFL 更好的覆盖度。Jie Liang et al. [34] 通过在4核并行模式下使用 AFL、AFLFast 和 FairFuzz 对12个程序进行测试，发现 AFLFast，FairFuzz 跟 AFL 相比覆盖度降为97%和96%，且产生的Bug数量降为57%和89%（这一点在2.2节 EnFuzz 的评估中也得到了体现）。也就是说，这些针对单个 Fuzzer 的优化方案在并行环境下出现了性能下降。

作者认为，这种性能下降是由于优化算法没有考虑并行的情况，每个 Fuzzer 之间会做很多重复工作，而且这些同步信息可能会影响优化算法本身，加上同步过程本身也需要算力，总体上就导致效率的下

降。于是作者提出两个解决方向，一是采用高效的同步方案，二是对 Fuzzing 总任务进行分隔，将分隔后的任务派给不同的 Fuzzer，每个 Fuzzer 只处理自己的那一部分以减少重复工作。

设计实现与评估

由于对 Fuzzing 任务进行分隔与 Fuzzer 本身的运行逻辑强相关，需要对不同 Fuzzer 进行不同的处理，所以作者只对 AFLFast 和 FairFuzz 进行了改造，并以 FairFuzz 进行了讲解。

首先是执行信息的同步，作者采取了和 EnFuzz 类似的“全局+局部信息存储”设计（毕竟是同一个课题组……），不一样的地方在于 Base Fuzzer 会拉取全局信息更新局部信息，其余的设计不再赘述，框架如 图 2-13 所示。

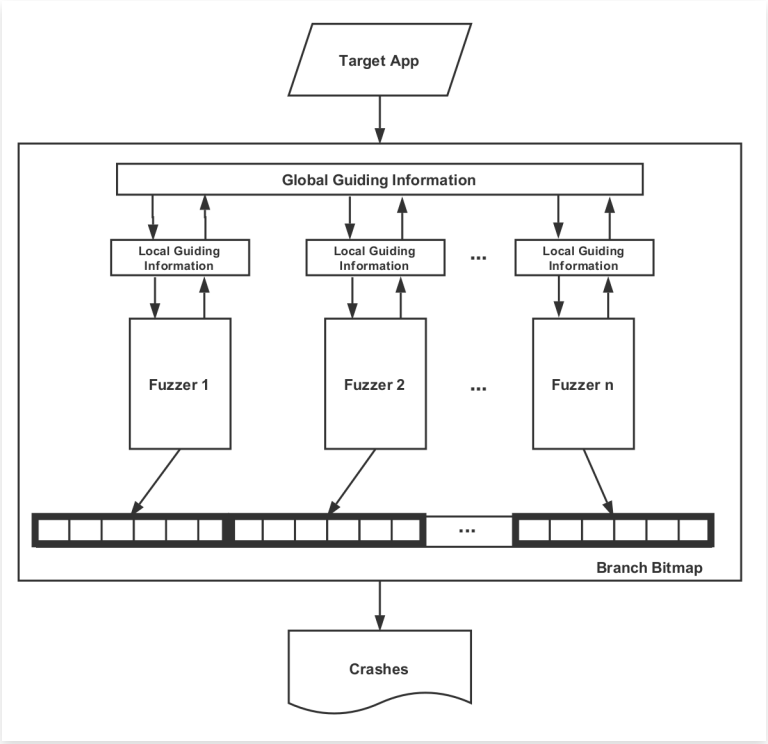

图 2-13 PAFL 框架

对于 FairFuzz 来说，其调度时会选择能够到达被执行次数最少基本块的种子，所以作者对全局覆盖度位图的更新设计为：`B_global_i = max(b_local_i, i)`，即全局位图中保存的是每个基本块被执行的最大数量。这样 Base Fuzzer 通过拉取全局位图就能知道现在有哪些基本块已经被执行多次，从而将资源放在执行次数少以及还没有被执行的基本块上。

其次是对 Fuzzing 总任务的分隔，作者采取的方案是对覆盖度位图进行线性分隔，每个 Fuzzer 在测试语料后，只有当该语料中的目标基本块（被执行次数最少）在自己对应的覆盖度位图间隔中，才会将该

语料保存到局部语料集中参与未来的测试。算法如 图 2–14 所示。

Algorithm 1: Fuzzing Task Division Algorithm

Input : The total number of fuzzer instances: n,
the index of the fuzzer instance: m,
input seed s,
branch hit count map $branch\_hit\_count$.

```
interval = MAP_SIZE/n;
start = (m − 1) ∗ interval;
end = m ∗ interval;
branch_covered = FindCoveredBranch(s);
min_hit_id =
  FindMinHitID(branch_covered, branch_hit_count);
if min_hit_id < start or min_hit_id >= end then
    Try next input seed;
end
```

图 2–14 PAFL任务分隔算法

作者对 AFLFast、FairFuzz 以及使用 PAFL 优化的 AFLFast 和 FairFuzz 对 Google's fuzzer test suite 中的12个程序进行了测试，结果表明使用 PAFL 改进的方案均超过了原方案。

Project	AFLFast	AFLFast with PAFL	FairFuzz	FairFuzz with PAFL
boringssl	3875	3901	3638	4077
c-ares	105	168	105	168
guetzli	2324	2440	1514	4217
lcms	2472	2677	2495	2706
libarchive	9267	10532	8646	10482
libssh	604	667	604	667
libxml2	14204	17216	14351	22803
openssl	4067	4145	4086	4143
pcre2	34725	37273	35124	36253
proj4	324	327	324	327
re2	16498	16520	16452	16670
woff2	175	177	175	175
Total	88640	96043	87514	102688

图 2–15 PAFL（AFLFast和FairFuzz实现）24小时4核并行的分支覆盖度对比

作者也使用 PAFL 强化的 AFLFast 核 FairFuzz 对 Github 上的一些开源项目进行了测试，发现了一些漏洞并获得了25个 CVE 编号。

未来工作与不足

1. 作者认为他们未来的工作将是采用更高效的同步方案，并适配到多机分布式的环境中。
2. 本论文采用分隔覆盖度位图的方法分隔 Fuzzing 任务，但这样的无法适用于不采用覆盖度位图作为反馈的 Fuzzer。使用更原始的种子\语料来分配Fuzzing任务或许是个更普适的方案（参见2.5节）。
3. 本论文的分隔 Fuzzing 任务的方法与 Fuzzer 的设计和实现强相关，无法同时集成运行覆盖度位图粒度和监测方案不同的 Fuzzer。
4. 当 Fuzzer A 认为覆盖度位图中的 b 不是它所属范围时，应该将种子和执行信息直接告知对应 Fuzzer，这样可以将覆盖度降低一个数量级。
5. 这种随机的分隔方法对只针对于覆盖度提升的 Fuzzer 组合比较适合，但对于 Directed Fuzzing [4] 可能会让性能下降，因为多个 Fuzzer 路径的目标是相同的。
6. 在评估中只采用了单机4核的并行方案，没有测试大规模并行性能（参见2.1节）。而且分隔覆盖度位图的方法需要在运行前确定 Fuzzer 的数量，无法进行动态调整。
7. 作者对 Github 项目进行测试时，没有使用原始的 AFLFast 及 FairFuzz 做对照组。
8. 未给出源代码或验证原型。

2.4 选择集成方案

现象与问题

Emre Güler et al. [35] 认为，在进行集成测试的时候，对 Fuzzer 方案组合的选择很大程度上决定了 Fuzzing 的效果——通常来说，Fuzzer 组合的多样性越高，Fuzzer 组合就越能更完全的测试程序。但是，对于 Fuzzer 组合的选取通常是通过测试人员的经验。例如在 EnFuzz 中，仅从三个角度去考虑 Fuzzer 的多样性，这样不仅需要人力去评价，另外未来如果出现新的 Fuzzer 方案后还需要重新设计 Fuzzer 组合。最后，在有限的时间内，Fuzzer 组合的多样性可能无法代表其实际的产出。所以，我们需要设计新的计算 Fuzzer 组合能力的方法，而且要能够自动化的对没有专家知识的 Fuzzer 判定最佳组合。

设计实现与评估

作者首先提出并回答了三个问题。

1. 为什么不直接并行多个覆盖度最高的 Fuzzer（非集成测试）？

这一点实际上在 EnFuzz 中的实验部分就已经证明了。在本文中，作者给出了一个例子，如 图 2–16 所示，Fuzzer A 更能求解左边的4个分支，而 Fuzzer B 更能求解右方的3个分支。如果我们仅运行覆盖度

更高的 Fuzzer A，那么可能无论测试多长时间，右方的3个分支都无法被覆盖到。

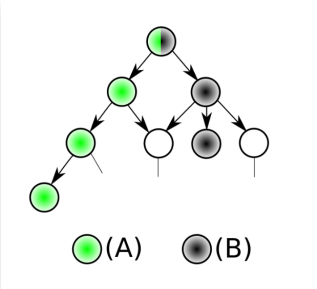

图 2–16 Fuzzer A 与 Fuzzer B 对于路径求解能力的区别

2. 为什么不直接选择多样性最高的组合方案?

这是由于“能覆盖”并不能代表整体的“覆盖效果”。再次以 图 2–16 举例，在一次时间有限的测试会话中（这里假设左方的分支 Fuzzer B 无法求解，反之亦然。即不考虑Fuzzer之间的合作）， Fuzzer A 对于左方的4个分支求解成功的概率都是0.01，那么运行两个 Fuzzer A 能够求解左方分支的概率将为 1 – 0.99 * 0.99 = 0.02，覆盖度期望将为 0.02 * 4 = 0.08；运行一个 Fuzzer A 和一个 Fuzzer B，覆盖度期望将为 4 * 0.01 + 3 * 0.6 = 1.84；而运行两个 Fuzzer B 对于右边的3个分支求解成功的概率为 0.6，那么它的覆盖度期望将为 (1 – 0.4 * 0.4) * 3 = 2.52。可见，即使 Fuzzer A 和 Fuzzer B 的组合是多样性最高的，但是在一个资源有限（包括时间）的测试会话中，其效果并不如两个 Fuzzer B。

3. 如何判定实践中效果最好的组合方案?

结合1和2两个问题，作者认为认为应该通过分析有限时间内，各个 Fuzzer 对于被测程序覆盖度反馈的互补性，并结合 Fuzzer 多样性来计算 Fuzzer 组合的实际能力。并最终提出了一种自动化分析 Fuzzer 组合的框架 Cpuid，其组成如 图 2–17 所示。

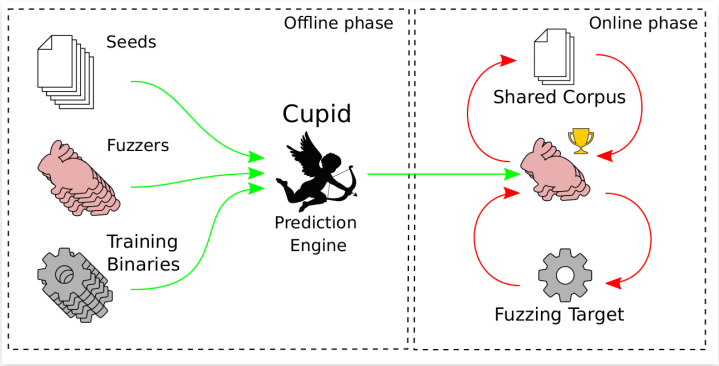

图 2–17 Cpuid的组成

该框架分为两部分，离线部分通过让各个 Fuzzer 用不同的种子测试训练集程序，得到各个 Fuzzer 的覆盖度位图，然后通过这些位图的互补程度确定在 Fuzzer 数量为 n 的时候最佳的 Fuzzer 组合，随后进入在线的Fuzzing。需要注意的是，在训练集程序不变的情况下，每个 Fuzzer 只需要在离线时测试一次训练集程序即可，后续加入新的 Fuzzer 后不需要重新进行测试。

对于互补程度的计算，作者提出了以下两个公式。图 2-18 表示的是对覆盖度位图中每一个单位 b，组合方案 F 能够到达的概率。Π(1-b_f) 中的 b_f 代表 Fuzzer f 在对训练集程序的测试过程中，到达基本块 b 的概率。图 2-19 表示通过求和覆盖度位图中的所有基本块的到达概率。最终，我们就得到了组合方案 F 对于训练集的实际测试能力，也就是 F 对每个分支求解的概率，如 图 2-20 所示。

$$BranchProb_F(b) = 1 - \prod_{f \in F}(1 - b_f)$$

图 2-18 计算组合方案 F 对于覆盖度位图单位 b 的可达概率

$$AverageBranchCov_F = \sum_{b \in B} BranchProb_F(b)$$

图 2-19 计算组合方案 F 对于覆盖度位图的整体可达概率

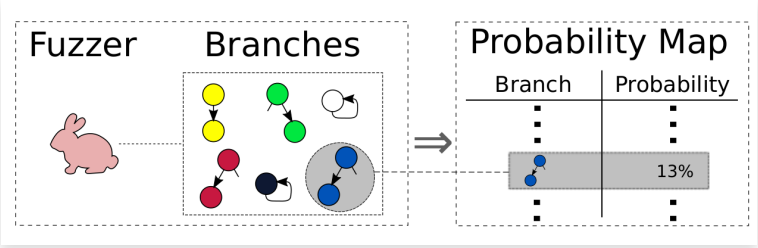

图 2-20 Cupid 计算的最终结果

为了更明显的展示上述公式的意义，作者使用 LibFuzzer 和 Honggfuzz 及其组合进行测试，将覆盖度位图的情况做成热力图，效果如 图 2-22 所示。图中的每个方块代表位图中的一个覆盖度单位，其颜色越深表示被命中次数越多。可见当使用多个相同 Fuzzer 时，位图中可达的单位分布基本不变，但是颜色变深；当使用多个不同 Fuzzer 时，可达分布扩大，且有部分颜色加深（合作的结果）。

图 2-21 LibFuzzer 和 Honggfuzz 对 freetype2 程序测试的覆盖度热力图

考虑到随机性和训练集程序中的噪声干扰，在计算出 Fuzzer 数量为 n 的所有组合方案的分数排行后，作者列出与第一名分数之差不超过5%的方案，然后选取这些方案里面 Fuzzer 组合最多样性的那一个作为最终方案。另外，为了得到在生产环境中效果更好的方案，使用 Fuzzer 对训练集程序进行测试的时候需要限制测试时间。

作者选用的测试时间为30分钟，训练集程序为 reetype2，re2，boringssl，llvm-libcxxabi，libjpeg-turbo，pcre2，wpantund，lcms，vorbis，harfbuzz，Fuzzer 集为 AFL，QSYM，AFLFast，RADAMS，FairFuzz，LAF-Intel，LibFuzzer，Honggfuzz，每个程序测试30次。因为要使用覆盖度反馈粒度和意义不同的Fuzzer，作者将所有 Fuzzer 最终的覆盖度反馈用AFL的方式（64KB边覆盖度位图）进行统计。

为了和 EnFuzz 对比评估，在训练所有 Fuzzer 后，作者设 Fuzzer 数量为4（n=4），计算出所有组合的得分，结果如 图 2-22 所示。

Position	Combination	Complementarity
1.	FairFuzz,libFuzzer,libFuzzer,libFuzzer	99691.98
2.	AFL,libFuzzer,libFuzzer,libFuzzer	99404.65
	...	
26.	FairFuzz, FairFuzz, libFuzzer, radamsa	96236.07
27.	**FairFuzz, libFuzzer, QSYM, AFL**	96176.07
28.	FairFuzz, libFuzzer, AFL, AFL	96064.53
	...	
124.	AFLFast, QSYM, QSYM, QSYM	72796.42
125.	QSYM, QSYM, QSYM, radamsa	71395.88
126.	QSYM, QSYM, QSYM, QSYM	66091.62

图 2-22 n = 4 时 Cupid 得到的组合排行

综合考虑到 Fuzzer 组合的多样性，作者采用27号方案（FairFuzz，LibFuzzer，QSYM，AFL）和 EnFuzz 对 Google's fuzzer test suite 中的部分程序进行测试，结果如 图 2-23 所示。要注意的是，为了避免被测程序的代码量不同带来的统计误差（大程序覆盖度变动对结果影响大），作者使用了各程序覆盖度的几何平均数进行评估。

Binary	Fuzzer Combination EnFuzz	Fuzzer Combination Cupid	Cupid Coverage Speed-up 90% Coverage	95% Coverage	97% Coverage	99% Coverage	p-value
c-ares	**58**	**58**	0.00%	**+10.00%**	**+10.00%**	**+10.00%**	-
guetzli	**2617**	2603	-6.28%	-11.73%	-2.58%	-3.62%	0.26
json	707	**711**	**+135.91%**	**+59.89%**	**+45.03%**	**+1827.58%**	**0.01**
libarchive	3161	**3577**	**+36.54%**	**+10.50%**	**+3.06%**	**+2.39%**	**< 0.01**
libpng	668	**697**	**+64.57%**	**+543.47%**	**+1135.38%**	**+54.97%**	**< 0.01**
libssh	809	**811**	**+51.18%**	**+16.96%**	-20.29%	-44.23%	0.48
libxml2	2014	**2123**	**+160.85%**	**+268.99%**	**+169.16%**	**+74.59%**	**< 0.01**
openssl-1.0.2d	**786**	784	-5.26%	**+6.38%**	-3.39%	**+8.22%**	0.08
openssl-1.1.0c	777	**779**	**+29.17%**	**+29.17%**	**+29.17%**	**+84.92%**	0.07
openthread	**864**	863	**+69.23%**	**+27.68%**	-33.14%	-14.83%	0.48
proj4	2715	**2819**	-2.80%	**+57.57%**	**+45.21%**	**+28.68%**	0.11
sqlite	**913**	**913**	**+489.64%**	**+489.64%**	**+489.64%**	**+489.64%**	0.08
woff2	1058	**1102**	**+75.32%**	**+432.28%**	**+351.31%**	**+48.58%**	**< 0.01**
Total	17147	**17835**					
Improvement (sum)	-	**+4.01%**					
Improvement (geomean)	-	**+2.31%**	**+59%**	**+90%**	**+74%**	**+64%**	

图 2-23 Cupid 与 EnFuzz 测试10小时后的分支覆盖度（p-value 为曼-惠特尼U检验 [29]）

可见，Cupid 选择的方案在整体效果上和EnFuzz相比略有提升，但在达到90%，95%，97%，99%的速度上有较大提升。说明 Cupid 方案选择出了更优的方案。另外，为了测试 图 2-19 和 图 2-20 算法在真实环境中计算的耗时，作者将 n 设为 10000，在5分钟内就得到了结果。

最后，作者还在 n=2 的情况下用 Cupid 测试出21个方案排名，然后对程序进行测试得到实际的方案排名，并计算这两个向量的皮尔森系数 [30]，得到了正相关的结果。

未来工作与不足

1. Cupid 是在 Fuzzing 前静态决定组合方案，运行时动态决定可能也是一个选择。
2. 本文没有对大规模集成 Fuzzing 的性能进行评估。
3. 在对训练集程序进行测试的时候时间只有30分钟，这对于一些需要长时间求解的 Fuzzer 不友好。
4. 本文对于覆盖度位图中的所有单位一直同仁（值域为 [0, 1]），我们或许可以对于目标单位加大权值（Directed Fuzzing? [4]）。
5. 我曾觉得可以针对于特定程序，例如图像处理程序，使用特定的训练集进行评估，这样得出的结果对某类的程序效果可能会更好。但是作者在结尾表示，对于特定的训练集，由于时间限制等因素，训练集中会有一部分代码我们无法覆盖，这样训练出来的效果可能不能反映程序的特点。所以作者还是决定使用多样化的训练集程序对Fuzzer进行评估。

2.5 独立调度和分析模块

现象与问题

Xu Zhou et al. [36] 认为，目前支持并行的 Fuzzer 大体具有以下缺点：

1. 任务重复：由于 Fuzzer 的同质性，许多 Fuzzer 在运行过程中处理的语料和进行的变异都是相同的，这会导致计算资源的浪费。PAFL 通过分隔覆盖度反馈位图来解决这一问题，让不同的 Fuzzer 负责不同的基本块，但这依然需要每一个 Fuzzer 运行所有新产生的语料。
2. 同步消耗：在同步的过程中，由于 Fuzzer 之间需要传递诸如语料名称，语料大小、覆盖度等执行信息，会对 Fuzzing 本身的进度产生影响（参见2.1节）。
3. 分布式可扩展性：传统 Fuzzer 的并行模式并不支持多机并行，这会限制对计算资源的利用。
4. 负载均衡：传统 Fuzzer 的并行模式并没有对单个 Fuzzer 以及全局的负载进行监测和调控，而是依靠经验进行参数调整。

作者提出了一种基于中心调度的并行 Fuzzing 架构（非集成）——UniFuzz，较好的解决了上述问题。UniFuzz 和其他 Fuzzer 的特性对比如 图 2-24 所示。

Program	Shared data					Synchronization		Inter-machine	Task conflicts
	Seeds	Crashes	Hangs	Fuzzer_stats	Coverage	sync method	sync time		
AFL-P [34]	✓					share memory	update after fuzz_one()	×	✓
Roving [29]	✓	✓	✓	✓		master node	periodical update	✓	✓
disfuzz [30]	✓	✓	✓	✓		master node	periodical update	✓	✓
PAFL [32]	✓				✓	share memory	periodical update	×	×
Li et.al. [35]					✓	master node	periodical update	✓	×
Clusterfuzz [36]	✓	✓	✓	✓		cloud resource	–	✓	×
EnFuzz [33]	✓	✓			✓	share memory	periodical update	×	×
P-fuzz [31]	✓				✓	database	periodical update	✓	×
UniFuzz	✓	✓	✓	✓	✓	database	live update	✓	×

图 2-24 支持并行的 Fuzzer 对比

设计实现与评估

UniFuzz 分为四个模块，其架构如 图 2-25 所示。其中数据库负责保存语料和执行信息；Fuzzing 节点（多个 AFL 组成，无局部语料队列和调度功能）负责对语料进行测试，并将语料和对应的执行信息上传到中心数据库中；评测节点根据语料的哈希值和覆盖度，对 Fuzzing 节点上传语料进行去重；中心调度节点负责维护全局的语料优先队列，并向 Fuzzing 节点和评估节点委派测试和去重任务。这四个模块之间依靠 RPC 交互信息，所以可以在多机上分布式执行。

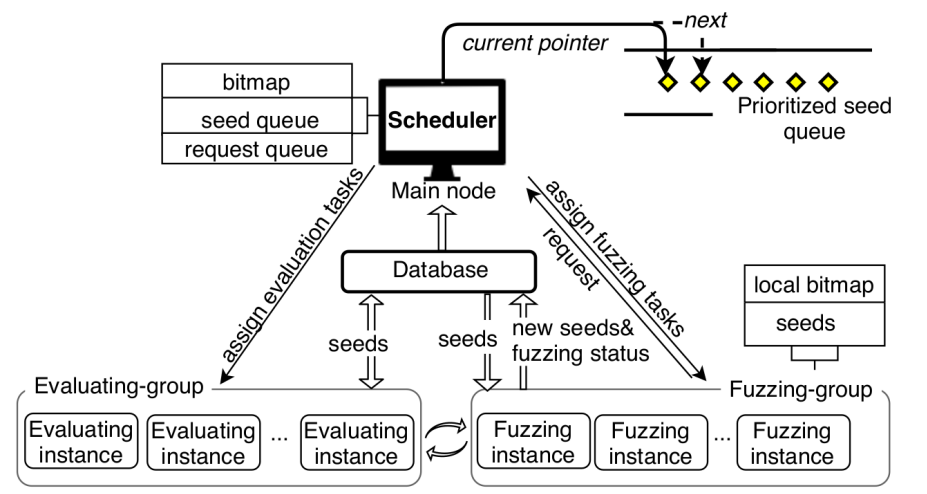

图 2–25 UniFuzz 整体架构

为了保证 Fuzzing 节点和评测节点的负载均衡，Fuzzing 任务的委派被设计为“申请–响应”模式，即 Fuzzing 节点主动向中心调度器申请任务，中心调度器根据优先级队列的情况，向 Fuzzing节 点委派语料在数据库中的索引和能量（对该语料测试的计算资源大小）。同时，评估节点的数量是动态变化的——中心调度器根据当前需要评测的种子数量决定，当去重任务大的时候，中心调度器会将一部分 Fuzzing 节点转换为评测节点。

中心调度器节点的算法如图 2–26 所示，值得注意的是，调度器向 Fuzzing 节点和评测节点委派语料时，只需要告知其在数据库中的索引（哈希值），不需要传输整个语料。另外，调度其在委派语料时对 Fuzzer 是没有区分的（不考虑 Fuzzing 节点的局部状态）。其他部分不再赘述。

Algorithm 2 The working procedure of the scheduler

```
Request_Queue ← ∅
Seed_Queue ← ∅
while True do
    if receive new_seeds from fuzzing instances then
        Evaluating_Nodes ← allocate_node(new_seeds)
        evaluated_seeds ← evaluate(Evaluating_Nodes, new_seeds)

        Database ← upload(evaluated_seeds)
    end if
    for seed in evaluated_seeds do
        fuzzing_status ← get_from_database(seed)
        Seed_Queue ← update_queue(seed, fuzzing_status)
    end for
    if receive request from fuzzing instance then
        Request_Queue.push_back(request)
        seed_id ← Seed_Queue.pop_front()
        energy ← perform_score(seed_id)
        fuzzer ← get_next_request_fuzzer(Request_Queue)
        dispatch_task(seed_id, energy, fuzzer)
    end if
end while
```

图 2–26 中心调度器算法

对于评测节点数量的变化频率，作者设置每1000次 Fuzzing 节点上传种子更新一次，算法如 图 2-27 所示。其主要是根据需要去重的种子，之前去重的速度以及去重比例来决定，最终将这些新的评测节点的标志位置为“evaluation”。

Algorithm 3 Dynamic evaluating instance expanding

```
updates ← 0
while True do
    if receive new_seeds from fuzzing instances then
        updates ← updates + 1
    end if
    if updates ⩾ 1000 then
        eval_nodes ← new_seeds_to_evaluate/evaluate_speed
        unique_rate ← total_evaluated_seeds/unique_seeds_num
        if eval_nodes > unique_rate/2 then
            eval_nodes ← unique_rate/2
        end if
        if eval_nodes ⩽ 2 then
            eval_nodes ← 2
        end if
        updates ← 0
    end if
    for each node in eval_nodes do
        node.flag ← "evaluating"
    end for
end while
```

图 2-27 评测节点动态扩展

最后，作者为了减少 Fuzzing 节点和数据库的语料传输，设置了一种缓存策略：当 Fuzzing 向数据库上传语料和执行信息时，将语料保存到本地的缓存队列中，以后 Fuzzing 节点收到中心调度器委派的测试任务时，先判断该语料是否在本地的缓存队列中，只有不命中时才会从数据库中读取语料，否则直接从缓存中获得语料。

在对比评估 UniFuzz 性能时，作者遇到了两个问题。一是原始版本的 PAFL 和 EnFuzz 在作者的测试环境下（64-bit CentOS Linux release 7.7.1908）无法成功运行，于是作者自己按照其设计实现了对比版本。二是 PAFL 和 EnFuzz 不支持多机模式，而单机上进行测试会很快达到硬件核心数的上限（Intel Xeon E5-2620 v4），于是作者提出了“计算资源”的概念——将 CPU 核数乘以测试时间作为计算资源，例如单核测试128小时和128核测试1小时的计算资源等同。

图 2-28 表示了 UniFuzz，AFL 非并行模式，PAFL 以及 EnFuzz 对12个程序进行测试的结果。可以看出 UniFuzz 在大多数情况下都能在路径覆盖度上取得优势。

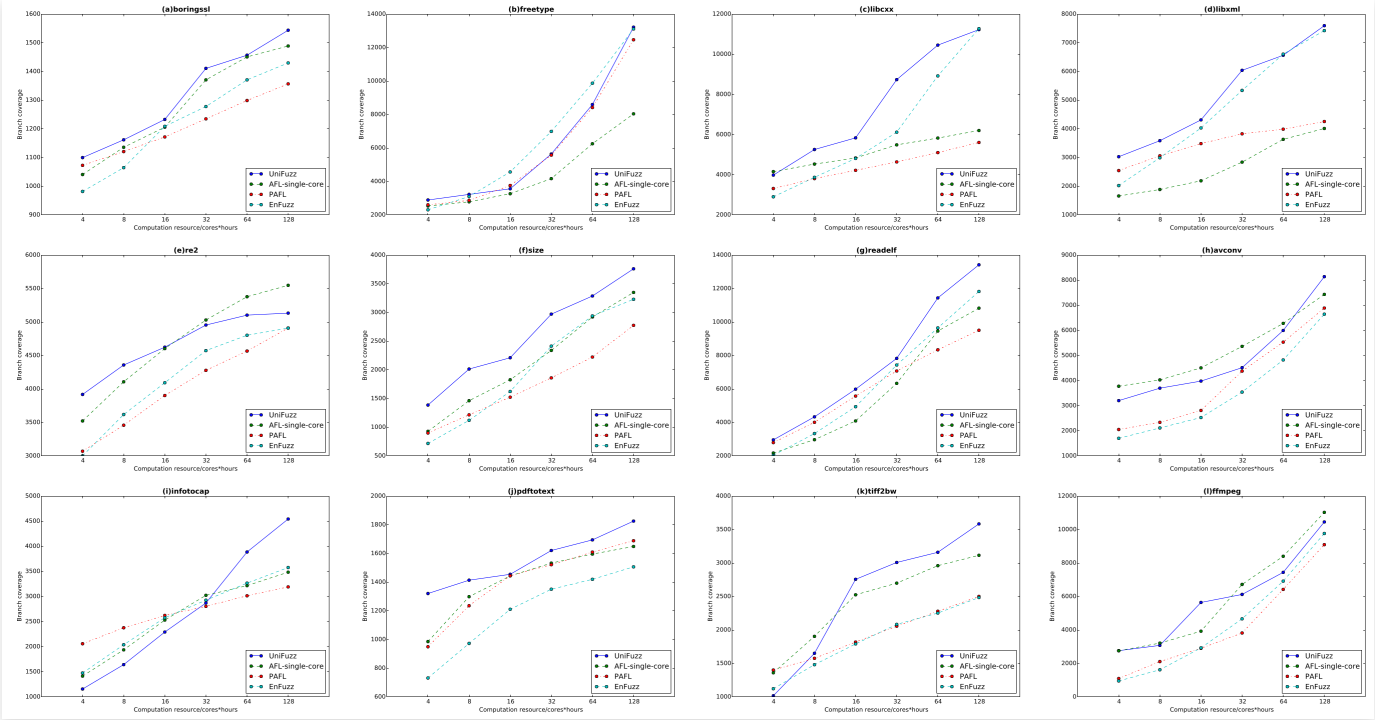

图 2-28 同等计算资源下 UniFuzz 和其他 Fuzzer 的路径覆盖度对比

作者还将 UniFuzz 和非并行模式的 AFL 在不同计算资源下的路径覆盖度进行对比，得到了几乎线性的结果，如 图 2-29 所示。作者认为这是因为 UniFuzz 在并行模式下会给语料赋予更多的能量，例如对于种子 s1 和 s2，它们在运行过程中都能发现新的路径 p3，对于非并行的AFL来说，当运行 s1发现 p3 后，会给 s1 的能量翻倍，但是 s2 发现 p3 后并不会，所以总的能量是 e1 * 2 + e2。但是如果并行执行 s1 和 s2，UniFuzz会 给每个种子的能量都翻倍，所以总能量变为 e1 * 2 + e2 * 2。后续作者的实验也证明了这一观点。

Program	4 units	8 units	16 units	32 units	64 units	128 units
boringssl	1,100(1.06x)	1,162(1.02x)	1,233(1.02x)	1,411(1.03x)	1,457(1.00x)	1,544(1.04x)
freetype	2,895(1.14x)	3,230(1.16x)	3,566(1.09x)	5,644(1.35x)	8,601(1.37x)	13,221(1.64x)
libcxx	3,985(0.96x)	5,260(1.16x)	5,840(1.21x)	8,740(1.59x)	10,458(1.79x)	11,230(1.81x)
libxml	3,031(1.82x)	3,589(1.90x)	4,314(1.97x)	6,042(2.13x)	6,567(1.81x)	7,599(1.89x)
re2	3,919(1.11x)	4,359(1.06x)	4,625(1.00x)	4,956(0.99x)	5,104(0.95x)	5,134(0.93x)
size	1,387(1.49x)	2,015(1.38x)	2,212(1.21x)	2,972(1.27x)	3,288(1.13x)	3,762(1.12x)
readelf	2,962(1.36x)	4,338(1.46x)	5,990(1.46x)	7,831(1.24x)	11,443(1.21x)	13,423(1.24x)
avconv	3,205(0.85x)	3,701(0.92x)	3,983(0.88x)	4,518(0.84x)	5,999(0.96x)	8,143(1.10x)
infotocap	1,158(0.82x)	1,644(0.85x)	2,293(0.91x)	2,871(0.95x)	3,887(1.21x)	4,544(1.30x)
pdftotext	1,321(1.34x)	1,414(1.09x)	1,455(1.01x)	1,621(1.06x)	1,695(1.06x)	1,826(1.11x)
tiff2bw	1,018(0.75x)	1,652(0.87x)	2,757(1.09x)	3,009(1.11x)	3,161(1.07x)	3,585(1.15x)
ffmpeg	2,766(1.00x)	3,082(0.96x)	5,647(1.44x)	6,126(0.91x)	7,445(0.89x)	10,457(0.95x)
Average increase	1.09x	1.13x	1.19x	1.19x	1.20x	1.28x

图 2-29 UniFuzz 与非并行模式的 AFL 覆盖度随计算资源变化的对比

未来工作与不足

1. 在 FSE 2020上，Böhme 等人 [21] 提出了一个经验定律——假设所有 Fuzzer 完全相同，在不考虑同步损耗的情况下，要线性更快的找到同样的 Bug，需要线性更多的 Fuzzing 机器，而要线性找到更多的 Bug，需要指数级更多的 Fuzzing 机器。作者认为他的实验结果反驳了这一定律，因为如图 2-29 所示，作者的覆盖度增长是线性的。但这里有个问题，图 2-29 是跟非并行模式的 AFL 进行对比，而AFL随着资源的线性增长覆盖度并不是线性增长的，所以 UniFuzz 的覆盖度也并不是随着计算资源（机器）的线性增长而线性增长。例如计算资源从64个单位增长到128个单位后，对于boringssl的覆盖度从1457仅增长到1544，远小于一倍。 *(或许需要读一下Böhme的论文再评价😃)*
2. 在 图 2-26 的中心调度器算法中，调度器并没有分析 Fuzzing 节点的局部属性，例如覆盖度位图。但不同 Fuzzing 节点对于同一个种子的需要程度是不一样的，例如一个种子对于上传它的 Fuzzing 节点来说不会提升局部的覆盖度，但是对于另一个 Fuzzing 节点来说却可能产生新的路径。（种子有优先级，但是 Fuzzer 没有对应的优先级）
3. 在 图 2-27 的评测去重算法中，我们只对新的语料进行去重，如果能够定期对全部语料进行去重或许能带来更好的去重效果（性能会有损失）。
4. 在 图 2-27 中，没有对最大的评测节点数量进行限制，可以设置为系统的硬件级线程的数量 - x，其中 x 为预定义常数，避免系统处于满载。
5. 作者提出的“计算资源”概念对于 Fuzzing 这种“选择和探索”的问题可能不太合适，因为一些 Fuzzer 可能需要大量时间去求解更深的路径。而且在生产环境中时间是有限的，例如我们可能无法容忍一个 Fuzzer 跑128小时，但是能够容忍32个 Fuzzer 跑4小时。
6. 在 图 2-28 中，很多 Fuzzer 的覆盖度增长并未趋于稳态，甚至增长速率还有上升的趋势，可能多测试一段时间会比较好。
7. 唯一的全局调度器分隔了Fuzzer的任务，但也带来了局限性，例如对于AFLFast这种有自己的调度算法的Fuzzer，全局调度器会使其失去特点。
8. 该系统没有支持集成测试。另外，集成测试的覆盖度随着计算资源的增长会是线性的吗?

3 挑战与思路

随着 Fuzzing 方案以及被测试软件逐渐变多，为了达到更好的测试效果，能够利用更多计算资源的并行和集成模糊测试将成为主流。但是，由于框架设计和实现上的限制，传统 Fuzzer 的并行模式很难在计算资源线性增长的情况下达到线性或超线性的测试效果。从上面5篇论文可以看出，在有限的硬件和时间资源限制下，我们可以以三个方向出发对并行和集成模糊测试框架进行设计与优化：信息同步，任务分配，组合方案与资源决策。

3.1 Fuzzer之间的信息同步

通过在 Fuzzer 之间共享执行语料的信息，使得 Fuzzer 可以通过合作求解一些单个 Fuzzer 还没有或者很难求解出来的路径。这里主要关注三个问题：

1. 对于同质的信息（例如使用相同粒度和意义的覆盖度反馈），我们可以直接同步该数据。但对于异质的信息，我们可能需要将最原始的信息——语料重新在一个统一的 Fuzzer 执行，从而得到相同格式和意义的信息，然后才能进行同步。
2. 同步操作的复杂度。最原始的同步要求每一个 Fuzzer 同步其他 Fuzzer 的所有信息，这会使得同步操作消耗时间占比随着 Fuzzer 数量增加以及 Fuzzing 的进展逐步增高，除了将同步功能从 Fuzzer 中分离（参见2.5节），我们或许还可以部分同步信息，通过分析语料和执行信息只同步特定信息给特定的 Fuzzer。
3. 可扩展性。同步操作如果过于依赖需要串行化的服务，例如文件系统（参见2.1节），会在大规模并行的时候产生性能瓶颈，所以需要尽量减少对底层系统的依赖，例如在用户态实现自己的同步协议，通过共享内存和数据库进行同步等。

3.2 Fuzzing任务的分隔与委派

在 Fuzzing 的过程中，许多 Fuzzer 会对相同路径进行测试，或者产生相同的语料，又或者对相同的路径约束进行求解。如果我们能将 Fuzzing 任务较好的分隔为不同任务，并委派给最合适的 Fuzzer 去解决，将极大的提高 Fuzzing 的效率。这里主要有两个思路：

1. 使不同 Fuzzer 专注于预定义的不同领域。例如在 PAFL 的设计中，通过对覆盖度位图进行分隔，并要求 Fuzzer 抛弃分析结果不处于其所属位图的语料，从而使得不同的 Fuzzer 可以对程序中不同的位置进行求解，避免了重复工作。此外，或许我们还可以定义其他不相交的目标，比如不同 Fuzzer 对程序中不同的危险函数进行导向性测试 [4]。
2. 给不同 Fuzzer 动态的委派不同的任务。例如在 UniFuzz 的设计中，通过动态的对所有 Fuzzing 节点上传的节点进行去重并重新委派，避免了不同 Fuzzer执行覆盖度相似的的语料，从而减少了重复工作。CollabFuzz [22] 通过数据流分析等手段，从全局的角度将 Fuzzing 节点的种子进行优先级排序，并委派 Fuzzer 处理优先级高的种子。

3.3 Fuzzer的组合及其资源决策

不同的 Fuzzer 对不同的程序，或者同一程序测试的不同阶段的效果会所有区别，如果我们能选择合适的 Fuzzer 组合方案，就能对程序产生更好的测试效果（参见2.2 节，2.4 节）。这个方向的思路有二：

1. 通过分析 Fuzzer 本身的特性来决定Fuzzing组合方案。例如在 EnFuzz 中是通过专家知识选择多样性更高的组合，在 Cupid 中是通过测试训练集程序，由各个 Fuzzer 的覆盖度反馈互补程度决定最佳组合方案。这种思路的缺点是取决于专家知识或训练集程序的质量。
2. 通过 Fuzzing 过程中各个 Fuzzer 的表现动态分配不同的计算资源。例如我们可以参考 MOpt，将不同的 Fuzzer 看成不同的粒子，目标为更高的覆盖度提升速度，使用粒子群优化等算法动态的分配不同 Fuzzer 的数量和计算资源，以此达到更好的计算效果。该思路的缺点是需要运行所有的 Fuzzer，并且运行时的资源调度算法会带来一定的性能损耗。

4 LilacFuzz: Scalable Framework for Ensemble Fuzzing

4.1 整体设计

从第3节“挑战与思路”的总结可知，我们可以从信息同步，任务分隔与分配，组合方案与资源决策三个方向出发进行设计。经过分析，我们认为任务分隔与分配和 Base Fuzzer 的实现强相关，不太适用于大规模集成测试框架。所以我们决定从信息同步和 Fuzzer 的动态资源决策两方面出发进行改进，并设计出了一种模糊测试框架 LilacFuzz。

LilacFuzz 是一种运行在单机多核环境下，支持并行集成模糊测试的可扩展框架。对于集成模糊测试，其使用基于覆盖度反馈的双层调度设计，开发者只需要对 Base Fuzzer 进行少许改动就可以适配到框架中。对于可扩展性，由于采用了启发式的信息同步和资源决策，该框架能够（*who knows?*）在100核以上的并行环境下取得较好的性能提升。最后，该框架不需要修改内核服务，可维护性强。

和上文中的模糊测试框架相比，LilacFuzz 创新体现在：

1. **双层调度与分析框架**。传统的并行与集成模糊测试框架要么仅支持同质的 Fuzzer [32] [36]，要么对于集成测试的同步粒度较大 [33] [35]。LilacFuzz为了解决这两个问题，设计了一种双层结构的框架。局部模块负责调度和分析同质的 Base Fuzzer，即一个 Fuzzer 组，对组内的语料进行去重和执行信息同步（例如覆盖度位图，以避免重复测试）。而且同步由 Base Fuzzer 主动发起，从而避免了单个 Fuzzer 负载过高。全局模块通过分析不同底层模块当前的覆盖度情况，中心调度 Fuzzer 组之间的信息同步，并根据 Fuzzer 组当前的路径探索情况，动态的给不同 Fuzzer 组分配不同的计算资源。
2. **启发式信息同步**。以往的测试框架大多是将新语料同步到所有的 Fuzzer [33] [32]，或者分配给空闲的 Fuzzer [36]，其缺点在于同步操作过多，无法将语料同步给最需要的 Fuzzer。LilacFuzz 从 Fuzzer 组的覆盖度差异性以及最大化全局覆盖度两个方向出发，动态决定哪些组互相同步，以此减

轻同步压力并提高同步的效率。同时，LilacFuzz 还根据不同 Fuzzer 组对新路径探索的情况，实现了延迟同步，进一步减少了同步操作。

3. **动态资源决策**。上文中的测试框架大多没有对不同 Base Fuzzer 的计算资源进行动态决策。由于不同 Fuzzer 对于不同软件，或同一软件的不同测试阶段的适应性不同，我们可以动态的去调整不同 Fuzzer 组的计算资源。在 LilacFuzz 中，全局模块负责根据覆盖度决策 Fuzzer 组可获得的资源，例如该 Fuzzer 组中 Fuzzer 的数量，能够使用的内存等。

4.2 全局模块

LilacFuzz 的全局模块设计如 图 4-1 所示，包含全局调度器和 Base Fuzzer 组成的多个局部模块。其中全局调度器不直接与 Base Fuzzer 交互，而是与每一个 Fuzzer 组的局部模块通信。通信内容主要有三类：

1. 局部模块定期上传给全局调度器该组当前的覆盖度，该覆盖度经过局部模块标准化，所以各个局部模块的覆盖度位图是同质的；
2. 中心调度模块在分析各个 Fuzzer 组的当前覆盖度后，会选择部分 Fuzzer 组，发送语料同步信号，通知其两两进行语料同步；
3. 中心调度器会根据各个 Fuzzer 组的覆盖度变化趋势，给每一个 Fuzzer 组分配不同的计算资源。下面我们分别就语料同步和资源决策进行解释，局部模块覆盖度的标准化参见4.3节。

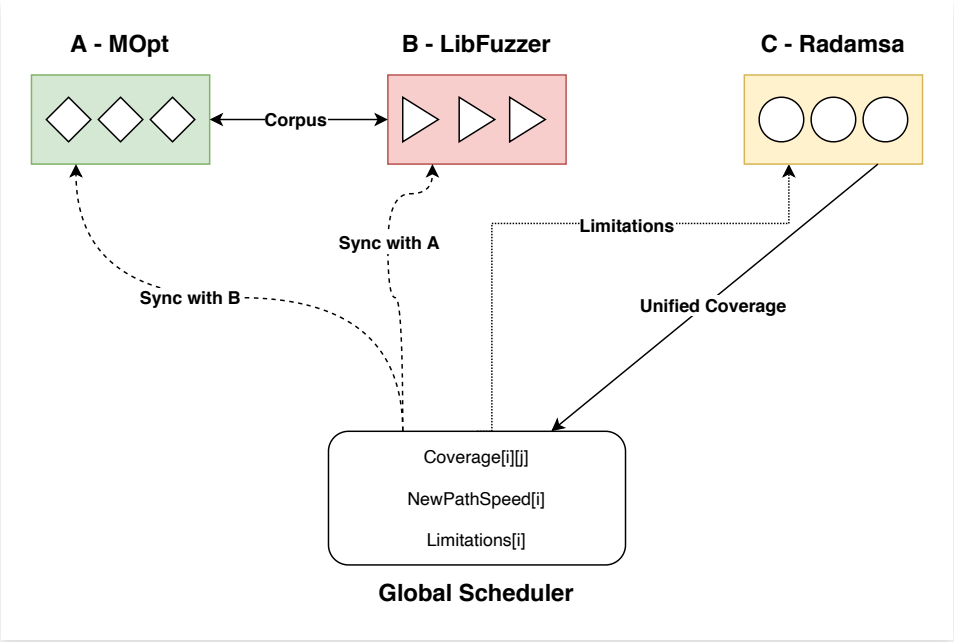

图 4-1 LilacFuzz 全局模块设计

全局语料同步

对于语料同步的 Fuzzer 组别选择，全局调度器主要从两个方面考虑。首先是哪些 Fuzzer 组在交换后语料后能达到目前全局最大覆盖度，二是哪些 Fuzzer 组的覆盖度存在较大的差异性，图 4–2 和 图 4–3 分别表示了这两种情况，其中的颜色深度代表位图单位被命中的次数。虽然 图 4–2 的两个语料交换后并不能达到 图 4–3 的覆盖度，但是这两个 Fuzzer 组的语料互补性非常好，这代表了这两个 Fuzzer 组的路径探寻差异性较大，互相交换可能会产生较好的效果。另外，考虑到位图大小有限，根据差异性进行同步能更好的解决测试中存在碰撞和溢出的问题。

为了减轻全局同步的性能压力，从差异性和覆盖度出发对两两 Fuzzer 组合进行排名后，我们可以只选择部分组合进行同步，然后随机选择少量组合进行同步，以避免局部最优。

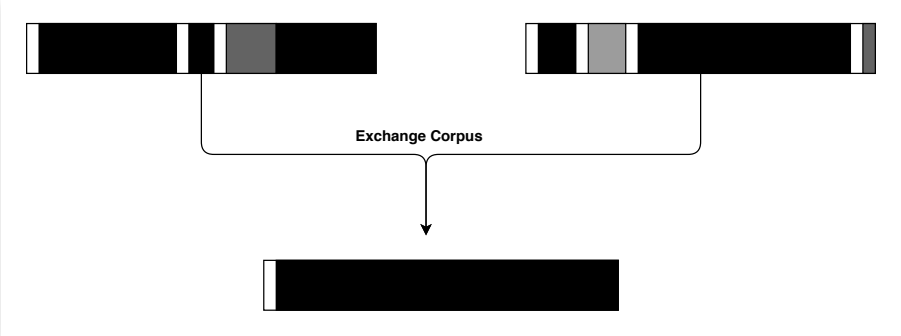

图 4–2 根据全局覆盖度同步

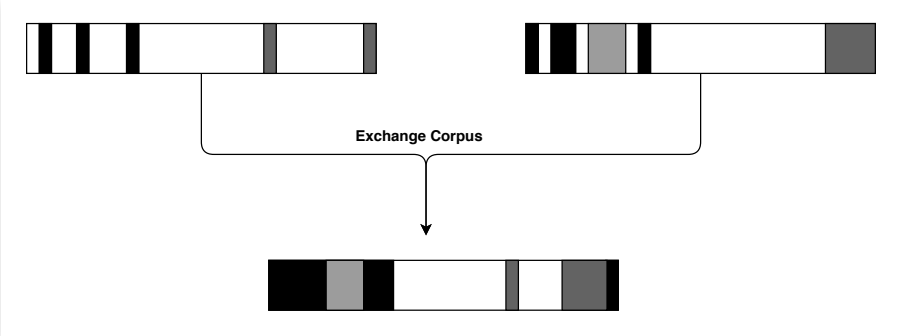

图 4–3 根据差异性同步

另外，为了继续减轻同步压力，LilacFuzz 参考了 AFLSmart [18] 中的“deferred parse”，也实现了类似的延迟同步策略，算法如 图 4–4 所示。其中，ProbExchage 为 Fuzzer 组合 a,b 经过全局调度器分析后同步的概率，rank 为 a, b 的排名，t 为 Fuzzer 距离上次发现新路径的时间，ε 是一个预定义的常数。可见，排名越靠前和目前难以发现新路径的组合被同步的概率更高，同时我们也能避免对不依靠同步就能快速发现新路径的 Fuzzer 组产生影响。

$$ProbExchange(a,\ b)\ = min\left(\frac{1}{rank} \times \frac{max(t_a,\ t_b)}{\epsilon},\ 1\right)$$

图 4–4 延迟同步算法

最后，不同 Fuzzer 组之间的语料交换并不需要 Base Fuzzer 直接参与，而是先由局部模块的调度器互相同步，然后再通过局部模块内部的方案进行同步（见4.3节）。

全局资源决策

对于 Fuzzer 组的资源决策，全局调度器参考 MOpt 采用粒子群优化算法 [31]。我们将每一个 Fuzzer 组看成单个粒子，每个粒子的搜索空间 y（例如 Fuzzer 的数量，每一个硬件级线程对应一个 Fuzzer）为[1, 硬件级线程数 – x]，局部最优解 local_y 为该粒子过去单位时间内覆盖度增长最多时的 y 值，全局最优解 global_y 为过去单位时间内全局框架覆盖度增长最多时该粒子的 y 值。要注意的是，和传统的粒子群优化算法不同，每一个粒子的搜索空间是不同的，即每一个粒子都有自己当前的全局最优解，所以最后全局调度器决策 Fuzzer 组资源的时需要将其向 硬件级线程数 – x 进行标准化。

4.3 局部模块

图 4–5 是 LilacFuzz 的局部模块设计，其中局部调度器的任务有：接受 Fuzzer 的同步请求与数据并在组内广播；分析 Fuzzer 组中的执行信息，对语料进行去重，并保存到局部数据库中；根据全局调度器的命令对 Fuzzer 进行启停控制，或和其他的局部调度器进行语料同步。

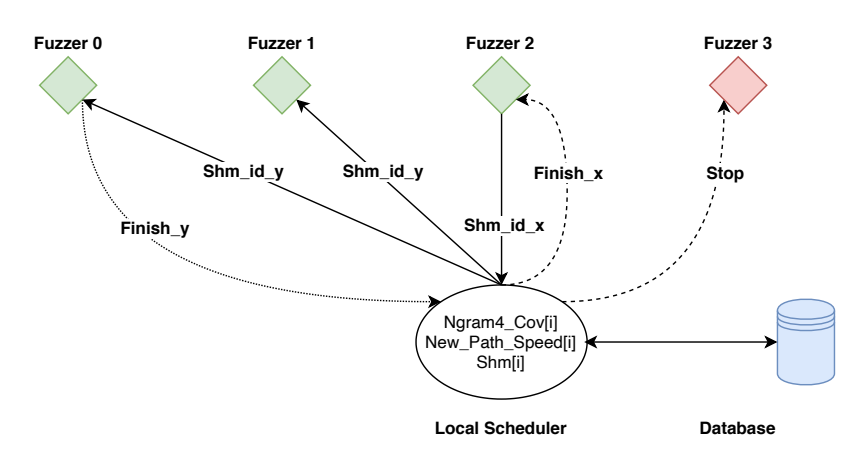

图 4–5 LilacFuzz 局部模块设计

局部语料同步与分析

由于在局部模块中，每一个 Fuzzer 都是相同的，所以可以通过细粒度的同步，例如语料的大小、执行时间、覆盖度位图，以避免重复工作。为了加快同步，语料数据使用共享内存实现，其数据结构是一种隐式链表，如 图 4-6 所示。

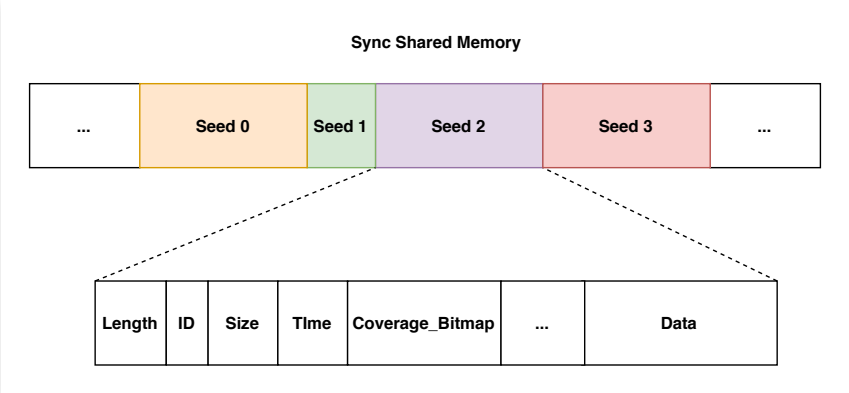

图 4-6 共享内存数据结构

为了减少 Fuzzer 的处理逻辑并降低同步损耗，当 Fuzzer 隐式链表的大小达到预定义的阈值时（*阈值可否由局部调度器动态决定?* ），会主动向局部调度器发送共享内存的 Shm_id，局部调度器语料数据进行复制，并在完成时通知 Fuzzer。随后局部调度器对这些语料进行重新测试和去重，更新全局覆盖度图并传送给全局调度器，将去重后的语料和执行信息保存在局部数据库中，最后使用共享内存的方法广播通知其他 Fuzzer 对去重的语料进行同步。

为了使全局调度器能够处理不同 Fuzzer 组的覆盖度，所有的局部调度器必须使用相同的覆盖度反馈机制（例如使用 Ngram4 [14]） 。另外，在去重的过程中，局部调度器还可以对种子进行分析和排序，有选择的进行组内广播和存储。

这种分层结构不仅解决了不同反馈/监测方案之间的同步问题，也减轻了同步的压力。由于全局语料同步不会对全部 Fuzzer 组的语料进行交换，而且还有延迟同步等优化，所以同步的压力主要来自于高频率的局部语料同步。假设整个架构有 m 个 Fuzzer，每个 Fuzzer 单位时间内产生的语料数量为 n，那么传统测试架构在单位时间内同步的复杂度就是 O(n * m^2)。在 LilacFuzz 中，由于我们将仅在同质的 Fuzzer 之间共享语料，其全局同步复杂度将变为 O(k * n * (m / k)^2) 即 O(n / k * m^2)，其中 k 为 Fuzzer 组的数量。虽然复杂度增长的趋势没有变化，但是我们可以将同步操作数量缩小 k 倍。另外在 Fuzzer 组中的同步也不需要 Fuzzer 重复执行语料，缩短了同步操作的时间。

产生优秀的语料后，Fuzzer也会将语料保存在文件系统中。参考 Wen Xu, et al. 的工作，我们使用内存文件系统作为缓冲以减小持久型文件系统和磁盘IO的压力，其结构如 图 4–7 所示，具体设计和实现请参考2.1节。

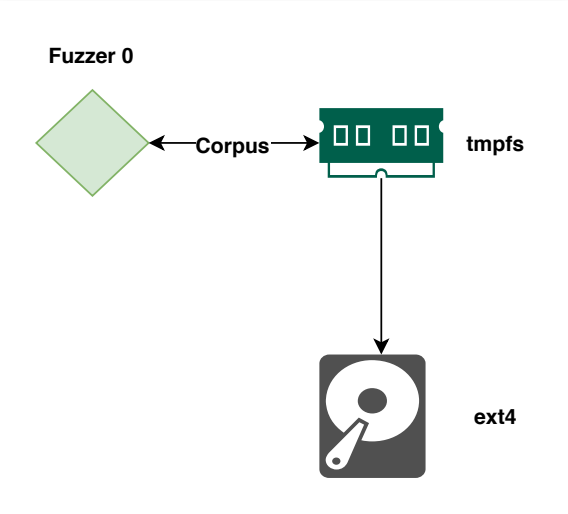

图 4–7 双层文件系统

响应全局调度器

如4.2节所说，全局调度器会不定期通知局部模块之间进行语料共享，通知方式是告知局部模块另一个局部模块的数据库句柄。每一个局部调度器都使用同样的数据库和表结构对语料和执行信息进行存储，并对其他局部调度器维护一个当前已同步的索引，以便下次同步时使用。在同步过程中，局部调度器会像局部语料同步一样，进行语料去重、通知全局调度器新的覆盖、语料排序，部分存储和组内广播等操作。要注意的是，由于局部调度器是相同的，在去重的时候我们不用重新测试语料，而是根据执行信息中的覆盖度直接去重。

当全局调度器向局部模块发送资源限制后，局部调度器会分析组内 Fuzzer 发现新路径的速度和覆盖度，对其进行排序，最后挂起部分 Fuzzer。

5 计划安排

待公开

参考

[1] TSE 2019, The Art, Science, and Engineering of Fuzzing: A Survey. Valentin J.M. Manes, HyungSeok Han, KAIST.

[2] USENIX Security 2019, MOpt: Optimized Mutation Scheduling for Fuzzers. Chenyang Lyu, Shouling Ji, Zhejiang University.

[3] NDSS 2019, REDQUEEN: Fuzzing with Input-to-State Correspondence. Cornelius Aschermann, Sergej Schumilo, Ruhr-Universitat Bochum.

[4] CCS 2017, Directed Greybox Fuzzing. Marcel Böhme, Van-Thuan Pham, National University of Singapore.

[5] NDSS 2019, NAUTILUS: Fishing for Deep Bugs with Grammars. Cornelius Aschermann, Tommaso Frassetto, Ruhr-Universität Bochum.

[6] USENIX Security 2020, Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. Zu-Ming Jiang and Jia-Ju Bai, Tsinghua University.

[7] Github, AFL. Michal Zalewski.

[8] Github, ClusterFuzz. Google LLC.

[9] Github, OSS-Fuzz Trophies. Google LLC.

[10] kernel.org/doc/Documentation, Linux Documentation file systems tmpfs.

[11] USENIX ATC 2016, Understanding Manycore Scalability of File Systems. Changwoo Min, Sanidhya Kashyap, Georgia Institute of Technology.

[12] 2018, afl-fuzz on different file systems. Jussi Judin.

[13] Github, Google's fuzzer test suite.

[14] USENIX WOOT 2020, AFL++ : Combining Incremental Steps of Fuzzing Research. Andrea Fioraldi, Sapienza University of Rome; Dominik Maier, TU Berlin; Heiko Eißfeldt; Marc Heuse, The Hacker's Choice.

[15] Github, AFL++ Snapshot LKM. Andrea Fioraldi, Marc Heuse, Joey Jiao.

[16] Fuzzing Like A Caveman 4: Snapshot/Code Coverage Fuzzer!. h0mbre.

[17] Github, EnFuzz Github issue 2. Paweł Płatek.

[18] TSE 2019, Smart Greybox Fuzzing. Van-Thuan Pham, Marcel Bohme, National University of Singapore.

[19] Github, ClusterFuzz issue 2182. taotao gu.

[20] Github, OSS-Fuzz what-are-the-specs-on-your-machines

[21] FSE 2020, Fuzzing: On the Exponential Cost of Vulnerability Discovery. Marcel Böhme, Brandon Falk, Monash University.

[22] EuroSec 2021, CollabFuzz: A Framework for Collaborative Fuzzing. Sebastian Österlund, Elia Geretto, Vrije Universiteit Amsterdam.

[23] CCS 2016, Coverage-based Greybox Fuzzing as Markov Chain. Marcel Böhme Van-Thuan Pham, National University of Singapore.
[24] ASE 2018, FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. Caroline Lemieux, Koushik Sen, University of California, Berkeley.
[25] LibFuzzer. LLVM.
[26] Gitlab, Radamsa.
[27] USENIX Security 2018, Qsym : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. Insu Yun, Sangho Lee, Georgia Institute of Technology.
[28] LAVA-M. Google LLC.
[29] Wikipedia, Pearson correlation coefficient.
[30] Wikipedia, Mann—Whitney U test.
[31] R. Eberhart and J. Kennedy, “A new optimizer using particle swarm theory,” in MHS, 1995.
[32] CCS 2017, Designing New Operating Primitives to Improve Fuzzing Performance. Wen Xu, Sanidhya Kashyap, Georgia Institute of Technology.
[33] USENIX Security 2019, EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. Yuanliang Chen, Yu Jiang, Tsinghua University.
[34] ESEC/FSE 2018, PAFL: Extend FuzzingOptimizations of Single Mode to Industrial Parallel Mode. Jie Liang, Yu Jiang, Tsinghua University.
[35] ACSAC 2020, Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. Emre Güler, Philipp Görz, Ruhr-Universität Bochum.
[36] arXiv 2020, UniFuzz: Optimizing Distributed Fuzzing via Dynamic Centralized Task Scheduling. Xu Zhou, Pengfei Wang, National University of Defense Technology.

ChangeLog

2021年7月2号 0.1.0 -> 2021年7月20号 0.2.0

1. Fix: 第三节“相关工作”改为按方向分类；
2. Add: 第四节“LilacFuzz”初步设计思路；
3. Fix: 部分错别字和缩进。