arXiv:2301.05060v1 [cs.CR] 12 Jan 2023

评估覆盖引导模糊测试工具的分叉感知能力

Marcello Maugeri[1] [a], Cristian Daniele[2] [b], Giampaolo Bella[1] [c], 以及 Erik Poll[2] [d]
[1]数学与计算机科学系，卡塔尼亚大学，意大利卡塔尼亚
[2]数字安全系，拉德堡德大学，荷兰奈梅亨
marcello.maugeri@phd.unict.it, cristian.daniele@ru.nl, giampaolo.bella@unict.it, erikpoll@cs.ru.nl

关键词: 模糊测试、分叉、安全测试、软件安全

摘要: 模糊测试是一种用于发现安全漏洞的有效技术。它通过向被测软件提供畸形输入，并等待出现异常的系统行为（通常是系统崩溃）来发挥作用。多年来，人们开发了各种不同的方法，其中最流行的是基于覆盖率的方法。该方法依靠对系统进行插桩，生成尽可能覆盖更多代码的输入。这种方法之所以成功，也得益于其易用性：模糊测试技术探索无需人工交互（或只需少量人工交互）的方法。尽管付出了诸多努力，设计出完全自动化的模糊测试器仍然是一项艰巨的任务。目标系统可能非常复杂；它们可能集成密码学原语、计算并验证校验和，还可能通过 fork（创建子进程）来增强系统安全性、提升性能，或同时管理多个不同的连接。本文提出了分叉感知属性，用以描述模糊测试器管理使用 fork（创建子进程）的系统的能力。我们利用这一属性评估了 14 种最广泛使用的覆盖率引导模糊测试器，并指出当前的模糊测试器在面对使用 fork（创建子进程）的系统时效果不佳。

1 引言

近年来，人们开发了大量模糊测试器来应对复杂的软件；如今，网络系统通常会使用 fork（创建子进程）来同时处理不同的连接。这带来了两方面需求：1）设计精准且针对性强的模糊测试器；2）根据模糊测试器应对此类先进系统的能力对其进行评估。

然而，正如[Hazimeh 等人 2020], 指出的那样，由于这些模糊测试器彼此差异很大，因此很难对它们进行全面基准测试。Metzman 等人通过构建 FuzzBench [Metzman 等人 2021], 应对了这一问题。FuzzBench 是一个用于评估无状态模糊测试器的开源服务。随后，Natella 和 Pham 提出了 ProFuzzBench [Natella 和 Pham 2021], ，它与 FuzzBench 类似，提供了一个用于评估有状态模糊测试器的服务。

尽管 FuzzBench 包含一些真实世界程序的样本，而 ProFuzzBench 包含不同的网络系统（即经常使用 fork（创建子进程）来处理多个连接[Tanenbaum, 2009]的系统），但它们并未评估模糊测试器应对使用 fork （创建子进程）的程序的能力。

尽管 fork（创建子进程）是创建新进程的唯一方式 [Tanenbaum, 2009], 实验结果表明，当前的模糊测试器无法处理派生进程。

现有方法仅依赖修改代码来移除 fork（创建子进程）。遗憾的是，这种方法违背了在模糊测试过程中减少人工工作、提高自动化程度的初衷。 [Boehme 等人， 2021]。

在这项工作中，我们探究并归类了当前模糊测试器在处理会创建子进程的程序时所表现出的局限性。

总而言之，本文：
1. 提出一种新颖的属性，用以刻画模糊测试器妥善处理 fork（创建子进程）的能力；2. 基于这一属性评估 14 种覆盖率引导模糊测试器；3. 提出改进当前最先进水平的可行方案及未来研究方向。

本文的组织结构如下。第2节介绍相关背景，第3节阐述

[a] https://orcid.org/0000-0002-6585-5494
[b] https://orcid.org/0000-0001-7435-4176
[c] https://orcid.org/0000-0002-7615-8643
[d] https://orcid.org/0000-0003-4635-187X

我们在知识方面的贡献，第4节展示了试图应对分叉问题的现有方法，最后，第5节讨论研究结果并提出可能的未来方向。

2 背景

2.1 模糊测试

模糊测试是由 Miller 等人开创的一种自动化测试技术。 [Miller 等人， 1990] 于 1990年 将其用于测试 UNIX 实用工具。如图 1 所示，覆盖率引导的模糊测试至少由种子选择、输入生成和系统执行组成。

1) 种子选择。用户必须提供一些能够代表系统常见输入的输入消息（种子）。

2) 输入生成。每个模糊测试器的核心，都是生成略有错误的输入消息，并将其发送给被测软件。模糊测试器的效率取决于生成的输入能否使系统崩溃。根据生成消息所采用的方法，模糊测试器可分为:

- 简单型：生成随机字符串（正如第一个模糊测试器所做的那样[Miller 等人， 1995] ）；
- 盲目变异：对用户提供的种子消息进行盲目变异；
- 基于语法：利用系统语法来构造输入消息；
- 智能变异（通常称为进化式）：需要一组输入样本，并利用反馈机制构造针对系统的消息。代码覆盖率反馈就是一种反馈机制，详见第2.2节。

3）系统执行。模糊测试器的每次执行都涉及三个组件:

- 错误检测器：用于报告发现的漏洞。大多数错误检测器只报告崩溃；然而，对于许多系统而言，偏离协议正常流程的异常行为也可能构成重大的安全问题；
- 挂起检测器：用于检测程序执行是否挂起；
- 代码覆盖率检测器：正如第2.2节进一步解释的那样，代码覆盖率是模糊测试器用来提升输入消息质量的一种反馈。

2.2 覆盖率引导模糊测试

智能变异模糊测试器利用反馈机制来引导消息的生成。存在多种不同类型的反馈机制[Shahid等人， 2011], 而且人们常用不同术语来表达相同的概念。为避免造成更多混淆，本文使用“代码覆盖率”一词来表示特定消息所触达的代码行。

代码覆盖率模糊测试器需要使用专用编译器（例如AFL编译器）重新编译代码，以插桩代码并获取运行时信息。

AFL [Zalewski, 2017], 例如，会对代码进行插桩，填充一个位图，用于表示输入所覆盖的代码行。

随后，它会利用此位图，为能够探索此前未见代码行的消息赋予更高分数。

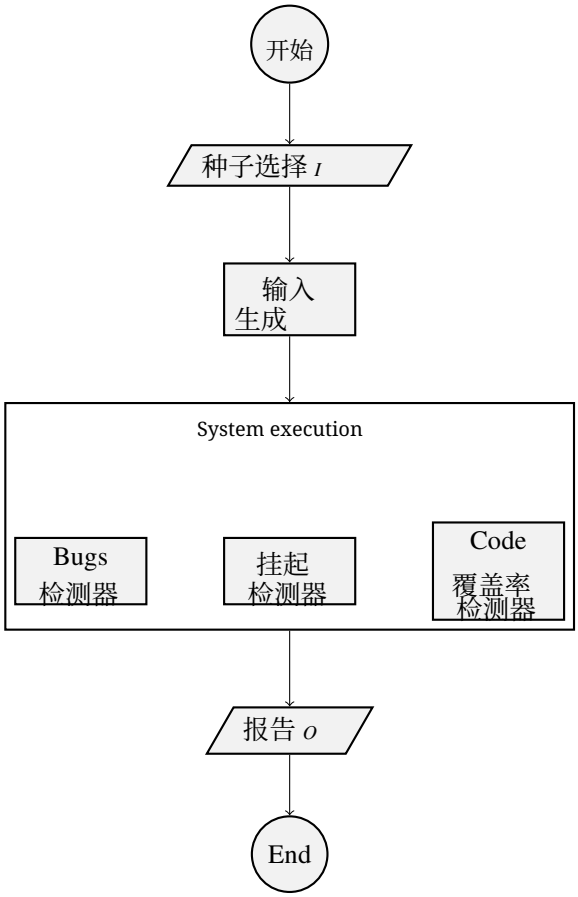


图1：覆盖率引导模糊测试过程

2.3 进程间通信

操作系统提供系统调用来执行各种任务（例如读写文件、访问硬件服务、创建和执行新进程）。在 UNIX 系统中，新进程通过 fork 系统调用创建 [Tanenbaum, 2009]。简而言之，第一个进程称为父进程，它会生成一个克隆，称为子进程，该克隆是父进程的精确副本。fork 之后，文件描述符和寄存器都会被复制，因此一个进程中的更改不会影响另一个进程。此外，父进程和子进程将沿着不同的执行路径运行。

3 我们的贡献

本文旨在了解当前最先进的覆盖引导型模糊测试器如何处理包含分叉的被测软件。

如何比较和对照各种工具并不显而易见。我们提出了一项新颖的属性——fork 感知能力；模糊测试器只有满足这一属性，才能有效且高效地处理分叉。正如下文将要阐述的，fork 感知能力体现在三个方面，它们代表了处理子进程的能力。

此外，我们还在来自两个基准测试框架、使用最广泛的模糊测试器上评估了这项新颖属性，共评估了 14 种工具，其中 11 种来自 FuzzBench，3 种来自 ProFuzzBench。

3.1 fork 感知能力

概括而言，fork 感知能力要求每个模糊测试器都应将子进程视同父进程。在系统执行期间，系统监视器应检测漏洞或挂起，而不论它们发生在何处；覆盖率也应在子进程中进行测量。这通过定义 1 进行了形式化。

定义 1. 如果覆盖率引导的模糊测试器能够以相同的方式检测漏洞和挂起，并测量子进程与父进程分支中的覆盖率，则称其具备 fork 感知能力。

该定义包含的三个方面称为：

[C.1] 子进程漏洞检测：即使异常发生在子进程中，也会报告；**[C.2]** 子进程挂起检测：即使无限挂起发生在子进程中，也会报告；**[C.3]** 子进程代码覆盖率：也会测量子进程的代码覆盖率。

3.2 示例挑战

我们编写了三个简单的 C 程序，作为模糊测试器的测试对象，用于测试这些模糊测试器是否满足上述各项要求。

a) 漏洞检测挑战：

```
1 if(fork()==0){ //子进程 2
raise(SIGSEGV); //模拟崩溃 3 }else {//父进
程 4 wait(NULL); //等待子进程 5 //终止 6 }
```

这段代码向子进程发送 SIGSEGV 信号，以模拟一个错误。该信号用于报告段错误，即内存访问违规，这在用低级语言编写的程序中很常见。模糊测试器还必须在父进程终止后检测到这个错误。

b) 挂起检测挑战：

```
1 if(fork()==0){ //子进程 2 while(1){ ;}
// 模拟 3 // 阻塞代码 4 }
```

这段代码模拟了子进程中的无限循环。模糊测试器必须报告循环结束后仍在运行的进程，并在模糊测试活动结束时终止子进程，避免遗留进程继续执行。

c) 代码覆盖率挑战：

```
1 pid_t pid =fork(); 2 if(pid==0){ //子进程
3 if(data %2 == 0){do_something(); } 4
else { do_something(); } 5 if(data %3 == 0)
{do_something(); } 6 else { do_
something(); } 7 if(data %5 == 0){do_
something(); } 8 else { do_something(); }
9 if(data %7 == 0){do_something(); } 10
else { do_something(); } 11 } 12 else { //父
进程 13 wait(NULL); //等待子进程 14 //终止
15 }
```

这段代码模拟了一个具有多个分支的子进程。模糊测试器必须覆盖并考虑每个子进程的所有分支。

我们让 14 个模糊测试器应对这些挑战，并将结果整理在表 1 中。我们注意到，没有任何一个模糊测试器成功通过全部三项挑战。

模糊测试器	基于	监控技术	Bugs 检测 (C1)	挂起检测 (C2)	Code 覆盖率 (C3)
AFL [Zalewski, 2017]	-	POSIX 信号	×	×	✓
AFL++ [Fioraldi 等人, 2020]	AFL	POSIX 信号	×	×	✓
AFLFast [Bohme 等人, 2017]	AFL	POSIX 信号	×	×	✓
AFLSmart [Pham 等人, 2021]	AFL	POSIX 信号	×	×	✓
Eclipser [Choi 等人, 2019]	AFL	POSIX 信号	×	×	✓
FairFuzz [Lemieux 和 Sen, 2018]	AFL	POSIX 信号	×	×	✓
lafintel [Besler 和 Frederic, 2016]	AFL	POSIX 信号	×	×	✓
AFLnwe[1]	AFL	POSIX 信号	×	×	✓
AFLNet [Pham 等人, 2020]	AFL	POSIX 信号	×	×	✓
MOpt-AFL [Lyu 等人, 2019]	AFL	POSIX 信号	×	×	✓
StateAFL [Natella, 2022]	- AFL - AFLNet	POSIX 信号	×	×	✓
LibFuzzer[2]	-	- 未定义行为检测器 - 地址消毒器 - 内存消毒器	✓	×	✓
Entropic [Bohme 等人, 2020]	LibFuzzer	- 未定义行为检测器 - 地址消毒器 - 内存消毒器	✓	×	✓
Honggfuzz[3]	-	ptrace（Linux）	✓	×	✓

表 1：覆盖率引导模糊测试器评估

3.3 测试平台

尽管这一性质适用于所有覆盖率引导模糊测试器，我们还是决定只分析 FuzzBench [Metzman 等人、2021]和 ProFuzzBench [Natella 和 Pham 2021] 中包含的覆盖率引导模糊测试器。所有模糊测试器均在 Ubuntu 20.04 服务器上运行，我们的所有源代码也都可在网上免费获取[4]，因此我们的实验完全可复现。

3.4 模糊测试器评估

我们使用选定的所有模糊测试器对三个示例挑战进行测试。表 1 总结了我们的发现。

所有基于 AFL 的模糊测试器都使用 POSIX 信号和位图来报告漏洞并跟踪代码覆盖率。

如表 1 所示，虽然位图能够跟踪子进程的代码覆盖率，但由于 AFL 只捕获主进程发出的信号，因此子进程中触发的漏洞不会被检测到，正如文档[5]中所指出的。唯一能够检测子进程中漏洞的模糊测试器是 LibFuzzer[6], Entropic [Bohme 等人提出的，以及 Honggfuzz[7], ，下文将对此进行更详细的讨论：

- LibFuzzer [8] 和 Entropic [Bohme 等人 2020]采用一组消毒器[9] 来报告漏洞。这些机制使模糊测试器能够发现挑战1中的漏洞，并测量挑战3中的不同代码路径，从而满足上文所述的挑战 C.1 和 C.3。遗憾的是，挑战 C.2 未能满足，因为模糊测试器无法检测子进程中的挂起。

- Honggfuzz 支持多种软件/硬件反馈机制，以及用于监控目标的低级接口。在 Linux 机器上运行时，Honggfuzz 使用 ptrace 系统调用来管理进程。这一机制使模糊测试器能够捕获范围广泛的信号。如表 1 所示，使用 ptrace（以及 SanitizerCoverage）使模糊测试器能够检测漏洞，并将子进程中的覆盖率也纳入考量。遗憾的是，这两种机制都无法检测子进程中的挂起。

总而言之，尽管所有选定的模糊测试器都能检测代码覆盖率（C3），但没有一个能检测挂起（C2），而且只有少数能检测子进程中的漏洞（C1）。评估结果表明：

[4]https://github.com/marcellomaugeri/forks-break-afl [5] https://github.com/google/AFL/blob/master/README.md [6] https://llvm.org/docs/LibFuzzer.html

[7]https://honggfuzz.dev/ [8] https://llvm.org/docs/LibFuzzer.html[9]AddressSanitizer、UndefinedBehaviorSanitizer 和 MemorySanitizer

- 循环检测挑战最为困难，因为模糊测试器不会等待所有子进程结束，而只等待主进程结束；
- 代码覆盖率挑战最容易应对，因为无论涉及哪个进程，都可以通过插桩测量执行过程中的代码覆盖率；
- 漏洞检测挑战取决于用于观察漏洞的技术，以及是否使用消毒器。

我们将这一总体结果解读为对未来研究与开发的明确呼吁。

4 现有解决方案

如今，对使用 fork（创建子进程）的程序进行模糊测试，唯一的解决方案是手动修改代码，或破坏系统的多进程特性（例如使用 defork[10] 等工具），以消除 fork（创建子进程）。

遗憾的是，正如 AFLNet 文档 [11], 所指出的，修改代码以移除所有 fork（创建子进程）是一项困难且容易出错的任务；而破坏系统的多进程特性往往会导致奇怪的系统行为。因此，唯一的解决方案仍然是修改模糊测试器。

5 结论

本文从三个不同方面分析了覆盖率引导模糊测试器的 fork 感知能力。对 14种知名模糊测试器的分析表明，尽管处理多进程程序的重要性显而易见，但大多数模糊测试器都忽视了这一问题。14 个模糊测试器中有 11 个无法检测子进程中的漏洞。这些结果背后的直觉与这些模糊测试器检测漏洞的方式有关。所有 AFL衍生模糊测试器都使用信号（SIGSEGV、SIGABRT 等）检测漏洞，而这种机制会漏掉子进程中的漏洞。我们注意到，处理 fork（创建子进程）并非唯一的问题，其他问题可能与进程间通信调度有关。例如，进程间通信可能会影响模糊测试过程的成功与否，因为有些漏洞只有在特定的进程调度下，且只有在访问某个特定内存单元后才会触发。我们认为，本文是朝着设计能够感知软件可能具有多进程特性的模糊测试器迈出的第一步

实现这一目标的第一步，或许是在早期阶段实现一个循环检测器，例如，利用动态库跟踪所有派生进程的进程标识符。总而言之，这项工作不仅首次提供了评估模糊测试器 fork 感知能力的具体方法，还首次揭示了一类迄今被忽视的问题，并指出了值得关注的未来研究方向。

[10]https://github.com/zardus/preeny/blob/master/src/defork.c [11] https://github.com/aflnet/aflnet

参考文献

Besler 和 Frederic（2016）。通过编译器转换绕过模糊测试障碍。https://lafintel.wordpress.com。Boehme、M.、Cadar、C. 和 Roychoudhury、A.（2021）。模糊测试：挑战与思考。《IEEE软件》，38(3)：79–86。Bohme、M.、Manes、V. J. 和 Cha、S. K.（2020）。提升模糊测试器效率：信息论视角。载于第 28 届 ACM 联合会议论文集：欧洲软件工程会议与软件工程基础研讨会，第 678–689 页。Bohme、M.、Pham、V.-T. 和 Roychoudhury、A.（2017）。作为马尔可夫链的基于覆盖率的灰盒模糊测试。《IEEE软件工程汇刊》，45(5)：489–506。Choi、J.、Jang、J.、Han、C. 和 Cha、S. K.（2019）。二进制代码上的灰盒具体符号测试。载于 2019 年 IEEE/ACM 第 41 届国际软件工程大会（ICSE），第 736–747 页。IEEE。Fioraldi、A.、Maier、D.、Eißfeldt、H. 和 Heuse、M.（2020）。Afl++：融合模糊测试研究的增量步骤。载于第 14 届 USENIX攻防技术研讨会（WOOT 20）。Hazimeh、A.、Herrera、A. 和 Payer、M.（2020）。Magma：真实基准模糊测试基准。《ACM计算系统测量与分析》论文集，4(3)：1–29。Lemieux、C. 和 Sen、K.（2018）。Fairfuzz：一种有针对性的变异策略，可提高灰盒模糊测试覆盖率。载于第 33 届 ACM/IEEE 国际自动化软件工程大会论文集，第 475–485 页。Lyu、C.、Ji、S.、Zhang、C.、Li、Y.、Lee、W.-H.、Song、Y. 和 Beyah、R.（2019）。{MOPT}：面向模糊测试器的优化变异调度。载于第 28 届 USENIX安全研讨会（USENIX Security 19），第 1949–1966 页。

Metzman、J.、Szekeres、L.、Maurice Romain Si-mon、L.、Trevelin Sprabery、R. 和 Arya、A.（2021）。FuzzBench：开放式模糊测试器基准测试平台与服务。载于第 29 届 ACM 联合会议论文集：欧洲软件工程会议与软件工程基础研讨会，ESEC/FSE 2021，第 1393–1403 页，纽约，美国纽约州。计算机协会。Miller、B. P.、Fredriksen、L. 和 So、B.（1990）。Unix 工具可靠性的实证研究。《ACM通信》，33(12)：32–44。Miller、B. P.、Koski、D.、Lee、C. P.、Maganty、V.、Murthy、R.、Natarajan、A. 和 Steidl、J.（1995）。重新审视 Fuzz：重新考察 Unix 工具与服务的可靠性。技术报告，威斯康星大学麦迪逊分校计算机科学系。Natella、R.（2022）。Stateafl：面向有状态网络服务器的灰盒模糊测试。《实证软件工程》，27(7)：191。Natella、R. 和 Pham、V. T.（2021）。Profuzzbench：有状态协议模糊测试基准。载于 ISSTA 2021——第 30 届 ACM SIGSOFT国际软件测试与分析研讨会论文集，第 662–665 页。计算机协会公司。Pham、V.-T.、Bohme、M. 和 Roychoudhury、A.（2020）。Aflnet：面向网络协议的灰盒模糊测试器。载于 2020 年 IEEE 第 13 届国际软件测试、验证与确认会议（ICST），第 460–465 页。IEEE。Pham、V.-T.、B¨ohme、M.、Santosa、A. E.、C˘aciulescu、A. R. 和 Roychoudhury、A.（2021）。智能灰盒模糊测试。《IEEE软件工程汇刊》，47(9)：1980–1997。Shahid、M.、Ibrahim、S. 和 Mahrin、M. N.（2011）。软件测试中测试覆盖率的研究。高级信息学院（AIS），马来西亚工艺大学，国际校区，Jalan Semarak，吉隆坡，马来西亚。Tanenbaum、A.（2009）。现代操作系统。培生教育公司。Zalewski、M.（2017）。American fuzzy lop。https://lcamtuf.coredump.cx/afl/。

这张图“orcid.png”提供“PNG”格式，来源：

http://arxiv.org/ps/2301.05060v1

