T-Fuzz: 基于程序变换的模糊测试

作者：Hui Peng， Yan Shoshitaishvili， Mathias Payer
单位：Purdue University， Arizona State University
来源：2018 IEEE Symposium on Security and Privacy (SP)

摘要

当前 fuzz 随机生成测试用例，不能通过条件检查（sanity check），覆盖受限，不能发现深层路径中的 bug。现有方法通过启发式规则或者复杂输入变异技术（符号执行或污点分析）通过这些检查。

本文通过轻量级动态跟踪技术识别 fuzzer 不能通过的输入检查，从二进制程序中移除这些检查，在变换后的程序上继续 fuzz，探测深层次路径。

程序变换 fuzz 面临两个挑战：1.溢出检查导致过度近似和误报。2.真实的 bug，转换程序上的 crash 输入可能不能触发潜在 bug。作为后端处理步骤，T-Fuzz 使用符号执行过滤误报并重现程序中潜在真实 bug。

通过程序变换和变异输入，T-Fuzz 比当前技术覆盖更多代码，发现更多真实 bug。使用 CGC 测试集，LAVA-M 数据集和 4 个真是程序进行评估。CGC 数据集：发现 166 个二进制的 bug，driller 发现 121 个，AFL105 个。另外，在之前 fuzz 过的程序中发现 3 个新 bug。

1.Introduction

Fuzz 是一种有效的自动化漏洞挖掘技术。当前策略分为两种：基于生成的（PROTOS，SPIKE，PEACH）和基于变异的（AFL，honggfuzz，zzuf）。基于生成依赖于输入格式，局限性大，当前研究集中于基于变异的。当前 fuzz 面临的问题是不能通过复杂合理性检查，覆盖受限，不能发现深层次 bug。AGL 采用覆盖导向的启发式方法辅助变异，取得了巨大成功。另有采用符号执行或污点分析来帮助通过合理性检查。但是，仍存在局限，在 CGC 测试集中 AFL 和 driller 仅能发现不到一半的程序 bug。

当前研究都是关注输入的变异和评估，其实也可以对程序进行变化。本文提出一种新 fuzz 技术，通过移除程序中输入检查，在变换后的程序中进行 fuzz，发现 bug。移除检查会导致误报，本文使用处理后符号执行分析来删除误报。基于程序变换 fuzz 技术，我们实现了 T-Fuzz 原型系统，它采用现有的覆盖导向的 fuzzer，当 fuzzer 不能发现新代码路径，采用轻量级动态跟踪技术发现所有输入不能通过的输入检查。移除这些检查，继续 fuzz。

相比于符号分析，T-Fuzz 有两个优势：1 更好的扩展性，轻量级适用于更大规模程序。2.通过“hard”检查。为了评估和现有方法相比的有效性，在 CGC、LAVA-M 数据集和四个真实程序依赖库上进行测试（pngfix/libpng, tiffinfo/libtiff, magick/ImageMagick, paftohtml/libpoppler）。在 CGC 数据集中，T-fuzz 发现 296 个程序中的 166 个存在 bug，比 driller 多 45 个比 afl 多 61 个。在 LAVA-M 数据集上的测试显示 T-Fuzz 比 steelix 和 vuzzer 可以通过“hard”输入检查，如校验和。实验证明了以少量漏报为代价过滤掉误报。

本文贡献：

1.采用程序变换的方法比重量级程序分析技术高效。

2.提供变异输入和程序的技术：自动检测程序中的合理性检查；移除检查变换程序；重现程序中 bug 通过顾虑误报。

3.在 CGC，LAVA-M 和四个实际程序中进行评估。

4.发现 3 个新的 bug，ImageMagick 两个，libpoppler1 个。

2.Motivation

在如图的实例中，31 行存在栈溢出漏洞，若要到达 31 行必须通过是三个检查：C1 魔数检查、C2Keys 内容检查，C3 CRC 校验。AFL 很难通过 C1，具有符号执行的 driller 可以通过 C1、C2，但很难通过 C3。

观察可得：

1.条件检查分为两种：非关键检查 NCC，如魔数检查；关键检查 CC，如 TCP 包解析长度检查。

2.NCC 可以直接被移除而不产生误报，移除这些检查可以将存在漏洞的代码暴露出来。

3.上例中检查都不是为了防止漏洞，所以移除之后所发现的漏洞在原程序中可以复现，需要使用可以通过检查的内容填充 header、keys 和 CRC 字段。

4.移除关键检查可能会产生误报，因此需要对产生的 bug 进行过滤。

3.T-Fuzz Intuition

T-Fuzz 包含三个模块：

Fuzzer：使用覆盖导向的 fuzzer，如 AFL 或 honggfuzz。依赖 fuzzer 跟踪所有输入的路径和实时状态信息，判断是否阻塞。Fuzzer 输出所有 crash 输入，再进行进一步分析。

程序变换：当 fuzzer 阻塞后，T-Fuzzer 调用程序变换模块对程序进行变换，跟踪 fuzzer 生成的输入检测 NCC，然后移除程序副本的 NCC。

Crash 分析：使用符号执行技术过滤误报。

T-Fuzz 工作流程，首先循环检测目标程序中的 NCC，使用一个队列保存需要被测试的程序（包括原程序及其所有变换）。T-fuzz 从队列中选择一个程序并加载 fuzzer 进程进行 fuzz，直到不能发现新路径。使用阻塞的输入来检测 NCC，通过移除不同的 NCC 生成多个转换后的程序。转换后的程序被添加到队列中。所有的 crash 将通过 crash 分析器进行过滤误报。

4.T-Fuzz 设计

T-Fuzz 使用现有版本的 AFL 作为 fuzzer。

检测 NCC：是 T-Fuzz 的关键，使用上近似：所有 fuzzer 不能通过的检查都认为是 NCC，因此存在一定误报。使用轻量级、不精确的动态跟踪方法检测它们。

程序变换：使用二进制重写来取反 NCC。

过滤误报重现 bug：上述生成的 crash 中存在误报，采用符号执行过滤误报，不能通过“hard”检查，还需要一定的手工分析。

A.检测 NCC

合理性检查在 CFG 中表现为一个节点两条分支，如果一条分支没有被任何输入执行，则认为是 NCC。

在 CFG 中，CN 定义为执行节点的集合，CE 定义为执行边的集合，边界边定义为满足以下条件的边 e：

1. e 不在 CE 中；
2. e 的源节点在 CN 中。

B.对 NCC 进行修剪

1.修剪不在期望部分的执行的 NCC。

2.剪枝错误检查，导致程序直接结束。

C.程序变换

考虑使用不同方法溢出 NCC，比如动态二进制插桩，静态二进制重写，简单的翻转条件跳转指令。前

两种都过重，不适用于大规模程序。T-Fuzz 采用第三种方法进行程序变换。

D.过滤误报重现 bug

移除 NCC 可能在转换程序中导致新 bug，所以需要对 crash 进行验证。T-Fuzz 结合了 driller 的预约束跟踪技术和 shellswap 的路径融合技术，通过跟踪 crash 输入在转换程序上的执行来收集路径约束。如路径约束满足，则说明 crash 可以在原程序中重现。

Crash 分析器跟踪转换程序，收集两类约束：在转换程序中的约束 CT，在原程序中的约束 CO。开始前，将输入改变为预先约束并添加到 CT 中？在跟踪过程中，如果基本块包含翻转的条件跳转，翻转路径约束添加到 CO 中，否则添加原约束到 CO 中。当到达 crash 触发指令时，将 crash 触发条件约束添加到 CO 中。如果 CO 有解，则说明可以生成输入执行相同路径来触发原程序中的 crash，否则认为是误报。

Crash 分析器可能会导致漏报，在下节进行详细讨论。

E.运行例子

为了说明误报过滤、重现 bug 的过程，我们提供了一些具体例子。

5.实现和评估

基于 python 实现了原型系统 T-Fuzz，采用了开源的 AFL，程序变换使用 angr tracer 和 radare2，crash 分析器使用 angr 实现。

为了评估 T-Fuzz 发现 bug 的有效性，我们在三个数据集中进行了测试：CGC、LAVA-M 和四个真实程序，并且和当前 fuzz 工具进行了比较。

A.DARPA CGC 数据集

包含了 248 个题目 296 个二进制程序，包含各类漏洞，提供了漏洞实情和 POV。使用三个不同 fuzz 工具进行比较：启发式 fuzz AFL、符号执行 fuzz driller 和本文 T-Fuzz。使用相同的种子运行 24 小时。

AFL：每个二进制占用一个 CPU，fuzz 前使用 angr 创建字典帮助识别魔数字节。

Driller：每个二进制程序分配一个 fuzz CPU 和一个符号执行的 CPU。

T-Fuzz：使用和 AFL 一样的 CPU 核数，多个转换程序按照队列先进先出进行处理。

a)与 AFL、driller 比较。

发现 296 个程序中 166 个漏洞，比 afl 和 driller 优势：其中 45 个包含复杂检查，约束求解很难生成输入。局限性：1.如果 NCC 翻转导致真实的 bug，转换程序将终止而不能执行漏洞代码。2.如果 bug 隐藏很深，则会导致转换爆炸，太多转换程序需要被执行。

b)与其他工具比较。Steeix 对 8 个 cgc 赛题进行了测试，仅发现了一个 bug。在人工提供种子的前提下 10 分钟发现 bug，T-Fuzz 在不需要提供种子的情况下 15min 发现 bug。

Vuzzer 对 63 个 CGC 赛题进行了 6 小时的测试，发现了 29 个 bug，T-Fuzz 发现 47 个。

B.LAVA-M 数据集

LAVA 数据集包括了一组自动插入漏洞的程序。LAVA-M 是它的一个子集，包括四个工具。比较对象：覆盖导向的 fuzz 工具 FUZZER、基于符号分析工具 SES、Vuzzer 和 Steelix。

T-Fuzz 在校验和程序中表现突出。

C.真实程序

在真实程序中评估了 T-Fuzz，并和 AFL 进行比较。

T-Fuzz 发现了很多 crash，并发现了三个新的 bug。

D.减少误报

E.实例研究

6.相关工作

大量研究来听声 fuzz 效率，如 Rebert 和 Woo 提出种子选择和 fuzz 调度发现更多 bug，但是不能通过复杂检查。

A.基于反馈的方法

AFL、libFuzzer、AFL-lafintal、Steelix，运行后反馈，不能控制检查，很难通过严格检查。

B.基于符号和具体执行的方法

KLEE、Veritesting、SAGE、DART、SYMFUZZ、CUTE、SmartFuzz 和 Driller。上述工具中，driller 和 T-Fuzz 最相近，使用符号执行帮助 fuzzer 生成可以通过检查的测试用例，但是不适用于大规模、严格检查的程序。

C.基于污点分析的方法

动态污点分析被用来辅助 fuzz，Tainscope 关注安全敏感部分的变异。Vuzzer 使用数据流和控制流分析，Dowser 和 BORG 使用符号分析来识别依赖关系帮助生成测试用例。但是，这类技术都过重。

D.基于学习的方法

从大量输入中进行学习，例如 Skyfire 和 Learn&Fuzz 从样本中学习不同值的可能分布，GLADE 基于综合语法学习来生成输入。基于学习的方法在生成格式较好（如 XML）的输入表现很有效，但是很难学习无结构的数据，例如校验和，并且要求大量的学习样本。

E.程序转换

现有程序转换克服检查都需要很多人工辅助。Flayer 依赖用户对检查提供地址，TaintScope 依赖有能够通过检查的输入。MutaGen 依赖程序代码可用性和标识。另外，都需要动态插桩修改目标程序，降低了执行效率。

7.结论

基于变异的 fuzz 受限于生成新的输入，很难通过严格的检查。本文提出了程序转换 fuzz，扩展了变异 fuzz，变异输入的同时也改变程序，实现了 T-Fuzz 原型系统。在 CGC、LAVA-M 和四个真实程序中做了评估，证明了 T-Fuzz 的有效性。

开源代码：https://github.com/HexHive/T-Fuzz。