软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(6):1978−1995 [doi: 10.13328/j.cnki.jos.006564]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

TaintPoint: 使用活跃轨迹高效挖掘污点风格漏洞*

方浩然, 郭 帆, 李航宇

(江西师范大学 计算机信息工程学院, 江西 南昌 330022)

通信作者: 郭帆, E-mail: fguo@jxnu.edu.cn

摘 要: 覆盖反馈的灰盒Fuzzing已经成为漏洞挖掘最有效的方式之一. 广泛使用的边覆盖是一种控制流信息, 然而在面向污点风格(taint-style)的漏洞挖掘时, 这种反馈信息过于粗糙. 大量污点无关的种子被加入队列, 污点相关的种子数量又过早收敛, 导致 Fuzzing 失去进化方向, 无法高效测试 Source 和 Sink 之间的信息流. 首先, 详细分析了现有反馈机制在检测污点风格漏洞时不够高效的原因; 其次, 提出了专门用于污点风格漏洞挖掘的模糊器 TaintPoint. TaintPoint 在控制流轨迹的基础上加入了活跃污点这一数据流信息, 形成活跃轨迹(live trace)作为覆盖反馈, 并围绕活跃轨迹分别在插桩、种子过滤、选择和变异阶段改进现有方法. 在 UAFBench 上的实验结果表明: TaintPoint 检测污点风格漏洞的效率、产出和速度优于业界领先的通用模糊器 AFL++及定向模糊器 AFLGO; 此外,在两个开源项目上发现了 4 个漏洞并被确认.

关键词: 静态分析; 模糊测试; 覆盖反馈; 信息流安全; 污点分析

中图法分类号: TP311

中文引用格式: 方浩然, 郭帆, 李航宇. TaintPoint: 使用活跃轨迹高效挖掘污点风格漏洞. 软件学报, 2022, 33(6): 1978−1995. http://www.jos.org.cn/1000-9825/6564.htm

英文引用格式: Fang HR, Guo F, Li HY. TaintPoint: Fuzzing Taint Flow Efficiently with Live Trace. Ruan Jian Xue Bao/Journal of Software, 2022, 33(6): 1978−1995 (in Chinese). http://www.jos.org.cn/1000-9825/6564.htm

TaintPoint: Fuzzing Taint Flow Efficiently with Live Trace

FANG Hao-Ran, GUO Fan, LI Hang-Yu

(School of Computer and and Information Engineering, Jiangxi Normal University, Nanchang 330022, China)

Abstract: Coverage-guided fuzzing has become one of the most effective ways of vulnerability detection. The widely used edge coverage is a kind of control flow information. However this feedback information is too coarse when detecting taint-style vulnerabilities. A large number of taint-independent seeds are added to the queue, and the number of taint-related seeds converges prematurely, which leads to the loss of evolutionary direction of fuzzing and unable to efficiently test the information flow between source and sink. Firstly, the reasons why the existing feedback mechanism is not efficient enough in detecting taint style vulnerabilities are analyzed. Secondly, TaintPoint, a fuzzer dedicated to taint style vulnerability detection, is proposed. TaintPoint adds live taint as data flow information on the basis of control flow traces to form the live trace as coverage feedback, and the live trace is used to improve the existing method in the instrumentation, seed filtering, selection, and mutation stages respectively. Experimental results on UAFBench show that the efficiency, output, and speed of TaintPoint in detecting taint-style vulnerabilities surpass the industry-leading general-purpose fuzzer AFL++ and directed fuzzer AFLGO. In addition, four vulnerabilities arefound and confirmed on two open source projects.

Key words: static analysis; fuzzing; coverage feedback; information flow security; taint analysis

随着现象级工具 AFL[1]的出现, 覆盖反馈的灰盒模糊测试(CGF)逐渐成为漏洞挖掘最有效的方式之一, 并被广泛集成进软件研发周期, 用以保障程序的质量与安全. CGF 通过种子变异随机生成大量测试用例, 观察

* 基金项目: 国家自然科学基金(61562040); 江西省教育厅科技项目(GJJ200313)
本文由“系统软件安全”专题特约编辑杨珉教授、张超副教授、宋富副教授、张源副教授推荐.
收稿时间: 2021-09-04; 修改时间: 2021-10-15; 采用时间: 2022-01-10; jos 在线出版时间: 2022-01-28

程序执行是否崩溃(crash), 从而发现缺陷和安全漏洞. 对于非崩溃漏洞, 如数组越界、内存未初始化、数据竞争等, Sanitizers[2−4]系列动态分析工具(如 MSan/TSan/ASan)通过编译期插桩, 能够在运行时让这些更隐蔽的漏洞暴露出来, 因此常常被 Fuzzing 工具集成以提高其发现漏洞的能力.

DFSan[2]是 Sanitizers 系列中的动态污点分析工具(DTA). 污点分析[5]是一种经典的程序安全分析方法, 能够发现如 SQL 注入、隐私泄露等非 Crash 漏洞, 我们称其为污点风格的漏洞(taint-style vulnerability). 如果不可信的外部输入(source 引入的污点数据)未经验证(validate)或消毒(sanitize), 通过数据/控制依赖传播, 流入敏感操作(sink), 说明存在污点传播路径, 从而产生潜在的信息流安全问题. 造成心脏滴血漏洞[6]的缓冲区溢出即可被视为污点风格漏洞. 如同其他程序分析技术, 静态和动态污点分析存在其固有局限性. 静态分析通常是可靠的(sound), 但是由于路径不敏感性(如经典的不动点迭代式数据流分析)、缺少运行时信息等因素产生误报; 动态分析依赖于测试集的完备性, 只能分析执行的程序路径, 因而产生漏报.

Fuzzing 通过变异或语法模板大量生成测试用例, 可以用于验证静态分析结果, 消除误报并降低漏报. 然而我们发现: 如果将现有 Fuzzer 如 AFL 直接用于 Source 和 Sink 之间的污点传播路径搜索, 则其有效性和准确性都有待提高. 现有 CGF 广泛采用边覆盖反馈机制, 多数漏洞被视为存在于某个分支下的特殊代码, 当特定的输入探索到此分支时, 漏洞就会被触发. 然而, 对于污点风格漏洞, 是否触发与到达漏洞点的路径强相关, 我们称之为路径敏感. 一对 Source 和 Sink 之间通常存在多条路径, 可能只有其中少数几条能够传播污点到 Sink, 并且由于污点清洁函数 Sanitizer 的存在, 使得这种精确的路径搜索十分困难. 边覆盖作为信息反馈, 在面对此类漏洞时过于粗糙: 大量与污点传播无关的种子被加入队列, 对这些种子做 Fuzzing 是低效的; 当高频的边覆盖完成后, 后续测试用例在 Source 和 Sink 之间产生的新路径无法被识别, 这使得队列中与污点传播相关的种子数量又过早收敛, 不能过滤出真正执行污点传播的好种子, 从而导致 Fuzzing 失去进化方向, 无法系统而高效地搜索污点传播路径. 定向Fuzzer通过优化种子选择和能量调度策略, 使Fuzzing朝着特定的目标进行, 但很多实现基于 AFLGo[7], 从而继承 AFL 的边覆盖反馈机制, 仍然存在上述问题. 正如 Neutaint[8]中所说: “不论任何形式的 DTA, 找到一条新的污点传播路径都是一个困难的问题”.

本文提出一种高效的 Fuzzing 方法挖掘污点风格漏洞, 并实现原型系统 TaintPoint. 该方法的关键是去除覆盖反馈信息中的噪音, 过滤出污点传播问题上真正的好种子. 边覆盖产生的轨迹是一种无差别的控制流信息, TaintPoint 在控制流的基础上加入活跃污点这种问题相关的数据流信息, 只记录存在活跃污点信息的有效执行轨迹, 形成活跃轨迹(live trace)作为反馈信号, 并且围绕活跃轨迹在插桩、种子过滤和选择、变异阶段分别改进现有 Fuzzing 方法.

首先, TaintPoint 通过静态分析识别出 Source-Sink 相关的潜在危险区域. 基于系统依赖图 SDG[9], 我们实现了一个静态Chopper组件, 识别出与Source-Sink依赖相关的基本块. 与标准后向切片(backward slicing)不同的是, Chopper 在 SDG 上进行两次可达性分析. 根据指定的 Source 和 Sink 点, 分别进行正向和反向切片, 取其交集基本块并做危险标记, 表明从 Source 引入的污点数据将沿着这些潜在的危险基本块到达 Sink. 在插桩阶段, 我们将这些危险基本块之间形成的轨迹(trace)在单独的共享内存中进行标记. 分离式插桩区分出了两类轨迹, 同时在危险基本块之间形成逻辑边, 从而减弱无关部分对污点传播路径搜索的影响.

Source-Sink 之间存在的消毒函数(sanitizer)会截断污点传播, 受影响的变量不再带有污点属性. 若在某个程序分支点, 所有处于后续危险基本块中的活跃变量都是未被污染的, 则说明在此程序点以后不会存在污点传播路径到达 Sink. TaintPoint 进行活跃变量分析, 在危险标记的每个分支基本块出口处插桩, 以获取运行时的活跃污点信息. 若某个分支处的有效活跃变量都未被污染, 则不记录该分支点及以后的轨迹, 只记录有活跃污点传播的活跃轨迹(live trace).

在种子过滤阶段, 如果测试用例在危险区域产生了新的覆盖, 那么在加入队列的同时, 给予该种子特殊标记以提升其优先级. 部分种子因为中途离开危险区域或被消毒而中断了 Live Trace 的记录, 若存在新的种子重新回到危险路径上或者绕过了消毒函数, 则产生的活跃轨迹是前者的超集, 该种子对于污点传播问题来说是一个更好的种子. 在种子选择阶段, TaintPoint 提出了新的进化方向, 优先选择活跃轨迹更长的种子来变

异. Fuzzing 将沿活跃轨迹增长的方向进化, 不断扩大污点传播范围, 直到发现到达 Sink 点的测试用例触发漏洞为止.

在种子变异阶段, TaintPoint 的活跃轨迹反馈机制能够灵活地适配各种变异算法.目前, 我们适配了 RedQueen[10]的染色方法, 专注于变异活跃轨迹中的分支, 避免变异后续不存在污点传播的分支, 从而能够高效地发现更多污点传播路径.

我们将 TaintPoint 与业界领先的 ALF++[11]及定向模糊器 AFLGo 在 UAFBench[12]中 13 个真实 CVE 漏洞上进行了实验对比, TaintPoint 在效率、产出和性能上都优于 AFL++和 AFLGO; 另外, 在两个开源项目中发现了 4 个漏洞并被 Google 和 Red Hat 确认.

本文的主要贡献如下:

- 详细分析了现有 Fuzzer 广泛使用的边覆盖反馈机制在面向污点风格漏洞检测时不够高效的原因;
- 面向污点风格漏洞检测, 首次提出使用活跃轨迹作为覆盖反馈. 围绕 Live Trace, 在插桩、种子过滤、选择、变异这 4 个阶段分别提出新的 Fuzzing 策略;
- 提出了新的插桩策略: 基于 SDG 系统依赖图, 实现了一个基本块级别的静态 Chopper 组件, 该组件标记 Source 和 Sink 之间可能存在污点传播的危险基本块, 并对这些基本块单独插桩. 分离式的插桩在危险基本块之间形成逻辑边, 从而消除污点无关边的影响;
- 提出了新的种子过滤、选择和变异策略: 产生新的或更长活跃轨迹的测试用例将被加入种子队列; 优先选择未被 Fuzz 过的活跃轨迹增长的种子, 使得 Fuzzing 朝着活跃轨迹增长的方向进化; 仅变异活跃轨迹中对应的分支, 以提高变异有效性;
- 实现了原型系统 TaintPoint, 在 UAFBench 上的实验结果表明: TaintPoint 在面向污点风格的漏洞检测时优于 AFL++和 AFLGo.

1 研究基础

1.1 污点分析

污点分析[13−24]是经典的信息流分析方法, 被广泛应用于漏洞挖掘(如 XSS、SQL 注入漏洞)、软件工程(指导 Fuzzer 变异)、隐私保护等领域. 它定义了一个三元组⟨Source,Sink,Sanitizer⟩: Source 通常为不可信的外部输入(如用户输入、网络文件), 作为源头引入污点数据; Sink 通常为一组敏感函数(如 syscall、memcpy)或关键寄存器/变量(如 PC、函数指针); Sanitizer 为一组消毒函数(如 encrypt、assert), 能够消除相关变量的污点属性, 表示不再带有潜在的危害. 污点属性随着数据依赖和控制依赖关系在程序中传播. 当未经检查的污点数据到达 Sink 时, 表明存在污点传播路径, 具有潜在的信息流安全问题. 静态分析工具, 如 SVF[15]、TAJ[16]通常对程序进行数据依赖分析, 污点属性沿着依赖关系进行传播. 动态污点分析如 Libdft[17]、BitBlaze[18]等事先定义污点传播策略, 执行时通过影子内存传播污点标签, 进行字节级的污点跟踪. Iodine[25]通过静态分析消除不必要的污点传播插桩, 减少了 DTA 运行时开销. Neutaint、GREYONE[26]采用轻量级的污点推断, 根据 Sink 处变量值的变化来建立 Source-Sink 之间的污点传播模型, 能够发现控制依赖造成的隐式污点传播, 进而指导 Fuzzing 的种子变异. 污点分析不仅在学术界被广泛地加以研究, 在工业界的实践也倍受认可. 很多商业分析器如 Coverity、HP Fortify、CodeSonar 等均将污点分析作为其分析引擎的关键方法.

Sanitizers 是 LLVM 编译器框架中的一系列动态分析工具, 包括 ASan/UBSan/DFSan 等, 能够发现变量未初始化、读写越界、整数溢出、隐私泄露等非崩溃漏洞. Sanitizers 与编译器紧密结合且使用方便, 因此通常被集成进 Fuzzing 工具以提高其漏洞挖掘能力, 如 Google 将 LibFuzzer 及 Sanitizers 全套工具纳入 Android 系统的安全测试中[27,28]. DFSan 作为其中的动态污点分析工具, 其具体实现是一个 LLVM Pass 及运行时库. 在编译时, 利用 Pass 框架进行 LLVM IR 中间代码级的插桩, 为每个变量分配影子内存以存储污点标签, 并指定每条指令的污点传播语义; 在运行时, 通过影子内存传播污点标签, 进行字节级的污点跟踪. 以二元操作符 BinaryOperator 为例, "*result*=*a*+*b*"对应的污点传播动作为"*Shadow*(*result*)=*Combine*(*Shadow*(*a*),*Shadow*(*b*))", 相

应操作数的污点标签进行合并计算. 只要有一个操作数是污染的, 那么结果操作数将被污染. DFSan 在影子内存中传播标签的同时建立一棵污点传播树, 通过标签值来追溯传播链, 查找污点源头.

1.2 DTA的固有问题

尽管针对 DTA 存在很多改进方法, 然而它们都没有正面回答以下问题: 如何尽可能多地发现新路径以充分测试程序中的信息流? 实际上, 这是动态分析的固有问题, 即 DTA 的分析结果依赖于测试集的完备性.静态分析通常是可靠的(sound), 但是由于分析算法在权衡精度和速度后所带来的各种不敏感性(如 flow/context/path-insensitive)、缺乏运行时信息等因素容易产生误报. 动态分析可以发现程序的错误, 却无法证明程序的正确性. 对于污点分析来说, 如果尽可能多地发现和执行潜在的污点传播路径, 则越有可能减少动态分析漏报, 消除静态分析的误报. 如: Neutaint 通过随机字节翻转得到 2 000 个测试用例并收集 Sink 处变量值的变化, 试图发现更多的污点传播路径作为神经网络的训练集.

那么, 如何才能自动化地构造测试用例以充分测试潜在的污点传播路径呢?

1.3 Fuzzing

Fuzz Testing 的本质仍然是动态测试, 其原理为自动化构造大量测试用例, 通过观察程序的执行是否崩溃或违反相关安全准则来挖掘漏洞. Fuzzing 在测试用例生成方法上可分为两类.

1) 基于模板, 通过程序输入的模板自动化生成得到. 模板的优点是测试用例的合法性高, 代码覆盖率高, 但模板的构建较为困难;
2) 基于变异, 由种子测试用例大量随机变异而来. 变异的优点是较为简单, 但是覆盖率低, 容易产生无效的输入而被程序拒绝, 无法深入探索程序.

从程序分析的角度看, Fuzzing 可以分成 3 类: (1) 黑盒, 无程序分析, 如 Peach[29]; (2) 白盒, 使用符号执行引擎 Klee[30]、Angr[31]等重量级的程序分析技术, 如 Driller[32]; (3) 灰盒, 轻量级的程序分析, 基于覆盖反馈的遗传算法. 随着 AFL 的出现, 基于覆盖反馈的灰盒变异 Fuzzer 逐渐成为主流, 学术和工业界围绕 AFL 在 Fuzzing 的各个阶段都做了大量改进. 如在插桩阶段, LAF-Intel[33]将分支语句拆分成逐个字节的比较,降低突破分支的难度. 在覆盖反馈阶段, CollAFL[34]消除边覆盖的 Hash 冲突, 提高覆盖精度. UnTracer[35]和 CSI-Fuzz[36]使用中断机制, 只跟踪触发新基本块/边的种子执行 Trace, 避免无效的比较, 极大地提高了 Fuzzing 速度. 在种子变异阶段, RedQueen 的染色算法使用随机变异来增加输入熵, 以此定位 cmp 等分支指令的参数对应输入字段中的位置, 有效突破了 CheckSum 和 MagicBytes. GREYONE 指出了传统 DTA 的缺点,并提出了使用轻量级的 Fuzzing 驱动的污点推断, 确定了变异字节与分支的对应依赖关系, 提高了变异的有效性. Angora[37]为分支增加上下文信息, 并且使用污点跟踪和梯度下降算法, 专注于突破未覆盖的分支. AFLFast[38]将种子的选择看作马尔可夫链, 优先选择低频路径的种子.

我们想知道的是, 自动化生成测试用例的 Fuzzing 可以解决上述 DTA 的固有问题吗?

1.4 覆盖反馈

覆盖反馈机制是灰盒 Fuzzing 中的关键部分, 给予 Fuzzer 进化的方向. 如果当前测试用例产生了新覆盖, 那么将被视为一个好的种子(认为它的变异结果将更有可能产生新覆盖), 加入种子队列, 以供后续变异. 绝大多数 Fuzzer 使用边覆盖, 认为漏洞代码存在于某个特殊的分支中, 是一次不正确的状态转移. 当特定的输入执行到这个分支时, 漏洞就会被触发. 所以通用 Fuzzer 的目标是尽可能地提高边覆盖率, 让更多的边得到测试. 下面我们以 AFL 为例介绍边覆盖的细节.

在测试之前, AFL 对程序进行基本块级别的插桩, 以便在运行时记录执行轨迹. 它使用随机数作为基本块的 ID, 将上个基本块 ID 右移 1 位并与当前基本块 ID 进行异或运算, 其结果作为边的 ID. AFL 使用名为 trace_bits 的 64 KB 哈希表(共享内存)存储每次执行的 Trace, 边的 ID 作为表索引, 每个字节中存储当前边的命中次数. AFL 对边命中进行[1,2,3,4–7,8–15,16–31,32–127,128+]分组, 以确保用单个 bit 表示该组计数. 每个 bit 位表示一种命中次数区间, 如[0*x*01]表示 1 次命中, [0*x*08]表示 4–7 次命中, 共 8 种; 使用名为 virgin_bits 的 64 KB

哈希表存储 Fuzzing 过程中累计的边覆盖命中计数, 每个字节初始值置为 0*x*FF, 表示该条边的 8 种命中次数区间都没有在历史执行中出现过. 当一次执行结束后, 对 trace_bits 和 virgin_bits 按位与运算(&), 若结果非 0, 则表示产生了新的边覆盖命中计数, 对应的测试用例将被加入种子队列, 以供后续变异. 同时, 对 trace_bits 和 virgin_bits 进行异或计算并写入 virgin_bits, 使得当前边的命中计数在 virgin_bits 中对应的 bit 被置为 0, 从而更新本次执行后各条边的历史命中计数.

与通用的 Fuzzer 希望尽可能地扩大覆盖率不同, 定向 Fuzzer 的目的是尽可能多地测试目标点, 应用场景如补丁测试、漏洞复现等. AFLGo 和 Hawkeye[39]等基于目标位置的距离来优先选择种子及能量分配, 使得更接近目标位置的种子获得更多变异机会.

我们想探究的是, 边覆盖这种普适的控制流信息反馈机制能够适应路径敏感的 DTA 吗?

2 核心方法

2.1 研究动机

首先, 我们用一个简单的例子说明研究动机, 阐述在挖掘污点风格漏洞时, 现有覆盖反馈方法存在的问题. 接着, 详细说明本文方法的核心概念——活跃轨迹. 围绕活跃轨迹, 分别在插桩、覆盖反馈、种子过滤与选择、变异这 4 个阶段改进现有 Fuzzing 方法. 最后, 回归例子来说明原型系统 TaintPoint 如何使用活跃轨迹高效地搜索污点传播路径.

如图 1 程序所示, Source 和 Sink 之间存在 4 个选择结构, 由分支变量 *cond*0/1/2/3 的符号分别决定具体执行的分支. 污点属性由 *Source*(·)引入, 沿着 *S*1→*S*2→*S*3 传播. 同时, 4 个分支变量的符号分别决定了 *S*1、*S*2 和 *S*3 的污点属性. 当分支变量中的任一变量符号为正值时, 都将截断 Source-Sink 之间的污点传播, 使得到达 Sink 点的 *S*3 未被污染, 信息流安全问题不会被检测到. 如当 *cond*1>0 时, *S*1 被赋值为常量 1, *S*1 的污点属性被消除.只有 4 个分支变量同时为负时, Source-Sink 之间存在的污点传播路径才会被发现.

```
01.  int cond0 = getInt();
02.  int cond1 = getInt();
03.  int cond2 = getInt();
04.  int cond3 = getInt();
05.  S1 = Source();       // S1 is tainted by source
06.
07.  if(cond0 > 0)
08.     t++;
09.  else
10.     t--;
11.
12.  if(cond1 > 0)
13.     S1 = 1;           // sanitize
14.  else
15.     S1++;
16.
17.  S2 = S1;            // propagate s1->s2
18.
19.  if(cond2 > 0)
20.     S2 = 2;           // sanitize
21.  else
22.     S2++;
23.
24.  S3 = S2;            // propagate s2->s3
25.
26.  if(cond3 > 0)
27.     S3 = 3;           // sanitize
28.  else
29.     S3++;
30.
31.  Sink(S3);           // is S3 tainted?
```

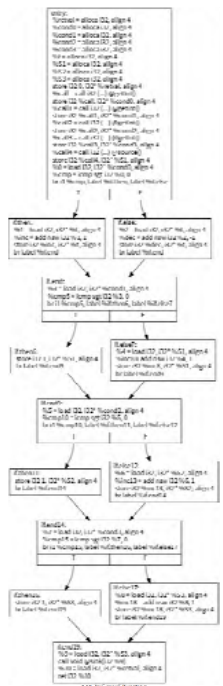

图 1 例子程序及其 LLVM IR 控制流图

假定初始种子输入(*cond*0,*cond*1,*cond*2,*cond*3)的符号都为正(+ + + +), 以 AFL 中的确定性变异(如 bitflip

1/1)每次翻转 1 个 bit 为例. 由于输入变量的最高位符号位被翻转(0→1), 容易得到如表 1 所示的 *cond* 分支变量的符号组合.

表 1 分支变量变异后的符号值组合

序号	*COND* 变量 0/1/2/3 符号
0	(+ + + +)
1	(− + + +)
2	(+ − + +)
3	(+ + − +)
4	(+ + + −)
…	…

当AFL对种子变异后首次得到如表1所示的符号组合序列时, 对应测试用例因为探索到了新的边(每个负号代表另一个分支)而被加入种子队列. 在 4 号种子被加入队列后, Source-Sink 之间所有的边都被覆盖. 此时, 队列中污点传播相关的种子数量收敛. 当一个更好的测试用例到来时, 如(+ − − +)将污染变量传到了 *S*3, 比 1、2、3、4 号种子都要传得更远, 此时只需对其变异 *cond*3 的符号位, 即可发现到达 Sink 点的污染流, 却因为没有触发新的边覆盖而被丢弃. 由于以后 Fuzzer 失去反馈, 对于污染到达 Sink 的符号组合(− − − −)的探索只能靠运气.

程序中的变量 *cond*0 及所控制的两个分支与 Source-Sink 污点传播无关. 1 号测试用例因触发了 *cond*0≤0 分支而被加入队列. 然而, 真正影响污点传播的 *cond*1/2/3 符号与初始种子相同(+ + +). 显然, 在污点传播问题上, 1 号种子的加入以及对其进行 Fuzz 是低效的. 3/4 号种子的 *cond*1 分支变量符号都为+, *S*1 的污点属性被清洁, 并不是污点传播问题上的好种子, 但却因为在后续没有实际污染传播的区域产生了新的边覆盖而被加入. 这种加入导致了 Source-Sink 之间的边覆盖达到 100%, 使得污点传播相关区域的种子数量收敛, Fuzzing 失去了进化方向. 当整个程序规模扩大时, Source-Sink 之间的分支增多, 路径数量随着分支的增长而呈指数倍增长. 假定有 n 个连续分支, 极端情况下, 2 条不同的路径即可覆盖全部边, 而剩下对(2^n-2)条路径的搜索都是盲目的. 无关部分如变量 t 所关联的 1 号种子的情况及 Source-Sink 之外区域产生的新覆盖也会显著增多, 对潜在污染流的探索更加困难. 值得注意的是, 上述问题同样存在于定向 Fuzzing 中. 在这个简单的例子中, 所有的测试用例都会经过 Source 而到达 Sink, 基于距离的定向机制并不能解决无关变量 t 及边覆盖的粗糙性带来的问题.

现有 Fuzzer 无法系统搜索 Source-Sink 之间污点传播路径的主要原因包括: (1) 边覆盖反馈机制选出来的很多种子与污点传播无关, 对这些种子的 Fuzzing 十分低效; (2) Source-Sink 之间的高频边覆盖完成后, 种子队列收敛, Fuzzing 失去进化的方向, 无法找出传播污点的好种子.

本文提出了面向污点风格漏洞的专用 Fuzzing 方法, 并实现了原型系统 TaintPoint. 其核心思想是: 在边覆盖这一普适的控制流信息上加入动态数据流信息作为反馈, 只记录存在活跃污点信息的轨迹, 我们称之为活跃轨迹, 用来最小化与污染传播无关的边覆盖对种子过滤造成的影响; 同时, 最大程度地区分出污染传播相关的路径, 充分测试 Source-Sink 之间的信息流. 围绕活跃轨迹, 我们分别从覆盖反馈插桩、种子过滤与选择、种子变异等阶段改进现有的 Fuzzing 方法.

2.2 分离式插桩

与表 1 中 1 号种子被加入的原因类似, 队列中存在大量与污点传播无关的种子. 这些种子是队列中的“噪音”, 使得 Fuzzing 无法专注于搜索真正存在污点传播的路径. 在 Fuzzing 前的插桩阶段, 我们提出使用 Chopping 引导插桩, 将真正传播污点的轨迹与无关轨迹分离.

Program Chopping[40]的概念源自于 Slicing[41]. 程序切片是一种经典的程序分析技术, 其结果是程序的一个子集, 即那些只影响切片准则处变量值的程序语句集合. 标准的后向切片(backward slicing)能够回答哪些语句影响了切片准则的问题, 却不能回答其中某个语句是如何影响切片准则的. 在污点传播问题中, 后向切片能够计算出影响 Sink 的语句, 然而却无法知道 Source 是如何影响 Sink 的. 也就是说, 对 Sink 后向切片得到的语句中仍然有大量与 Source 无关的部分, 这些语句并不会传播污点. Chopping 在对 Sink 点后向依赖分析

的基础上, 再对 Source 点进行正向依赖分析, 取得二者的交集. 在污点传播问题中, Chopping 的结果表明由 Source 引入的污点属性如何沿着这些语句间的依赖关系流动到 Sink 点.

如图 2 所示, Source-Sink 依赖相关的红色危险基本块 BB1–BB4 之间形成逻辑边, 其轨迹的命中次数将在 trace_bits 中对应的 index 处累计. 同时, 我们在 trace_bits 后分配了同等长度的共享内存 live_trace, 并在 live_trace 中相应的偏移量处(index′)进行 0*x*FF 标记, 以表示这是一条危险逻辑边. 无论程序执行经过 BB2 或 BB3 基本块到达 BB4, 都不会影响到真正传播污点的 1–4 之间逻辑边的轨迹记录.

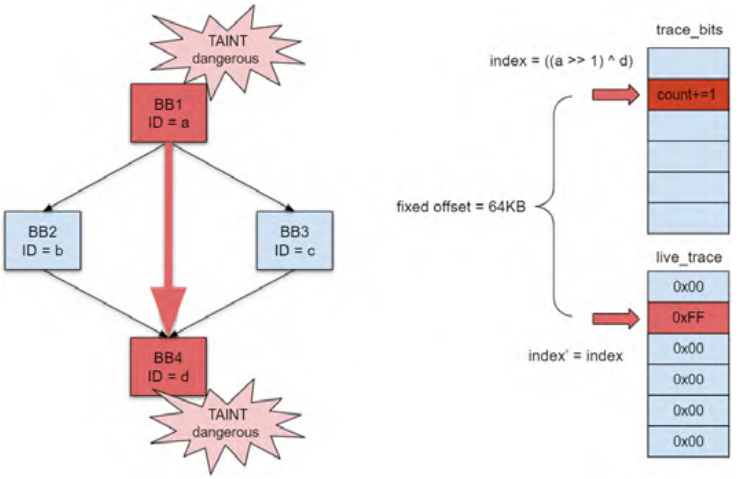

图 2 分离式插桩形成逻辑边及相应标记

我们基于系统依赖图 SDG 实现了一个静态 Chopper 组件. 与上述标准的指令级 Chopping 不同, 该组件只进行 IR 基本块级的标记, 并不删除无关语句. 在 SDG 上进行图可达性分析时, Chopper 标记对应语句所属的基本块. 首先对 Sink 点进行后向依赖分析, 标记出 Sink 函数参数的依赖节点所在的基本块. 再对 Source 进行正向依赖分析, 计算出所有依赖于 Source 的节点所在的基本块. 同时判断这些基本块是否被后向标记过: 若是, 则标记成为危险基本块. 在 Fuzzing 的插桩阶段, 我们将这些基本块之间逻辑边所形成的轨迹在 trace_bits 相对应的 live_trace 中进行标记, 从而将污点传播相关边与无关边的 Trace 分离, 并且在非危险区域上取得的新覆盖将不会影响到污点传播轨迹的记录.

2.3 活跃轨迹

虽然 Chopping 引导的插桩能够有效分离出真正进行污点传播的边, 但是通用的边覆盖反馈机制仍然会使得 Source-Sink 之间危险区域的污点种子数量过早收敛, 从而无法挑选出更好的测试用例作为种子加入队列. 如表 1 程序中的 2–4 号种子, 都因为发现了新的边而被加入种子队列. 此时, 危险区域的边覆盖率达到 100%. 当后续更好的测试用例到来时, 由于无法产生新的边覆盖从而被丢弃.

我们发现 3 号、4 号种子中的分支变量 *cond*1 符号都为正, *S*1 被赋予常量, 污点传播在此处被截断, 后续路径不再有活跃污点变量. 因为 *cond*2 和 *cond*2 变量为负而产生新覆盖的边中, 并没有实际的污点传播. 基于这个观察, 我们利用活跃变量信息, 以活跃轨迹作为反馈机制, 能够真正识别执行污点传播的测试用例, 防止队列中的污点相关的种子过早收敛.

活跃轨迹的本质与分离式插桩的想法类似, 即: 如果只记录存在污点传播的 Trace, 那么没有污点传播的边将不会影响到后续 Fuzzing. 分离式插桩通过保守的静态分析(chopping)将不可能存在污点传播的基本块轨迹与潜在的危险轨迹分开, 而活跃轨迹对污点信息的检查是动态的、实时的, 进一步精化了危险轨迹. 我们应用经典的活跃变量分析(liveness analysis), 在危险基本块中插入当前程序点活跃变量(LiveOut)的污点属性检查, 如图 3 所示, 计算基本块出口处的活跃变量集合, 并插入动态检查这些变量的污点属性的计算过程. 在运

行时, 如果出口处存在活跃污点信息, 表示污点数据成功传播过了此基本块, 后续仍然有潜在的污点传播. 此时, 正常记录与前一危险基本块所形成的逻辑边. 如果不存在任何活跃污点信息, 说明此程序分支点后不会有污点传播, 当前边及后续轨迹都不会被记录, 所形成的活跃轨迹是原有完整轨迹的一个前缀. 图3中的测试用例由于进入污点清洁分支 $D1$, 在到达 E 出口处已经不存在活跃污点变量, 此时中断 $D1 \rightarrow E$ 及以后的 Trace 记录. 如果后续存在新的 Source 而引入污点时, 轨迹又将按此方法被重新记录. 我们进一步优化了经典不动点迭代式的活跃变量分析结果, 只检查那些后续在危险区域被使用的活跃变量的污点信息(称其为有效活跃变量), 减少了运行时开销. 如果在某个程序点中活跃变量在后续被使用时并不处于危险区域, 则即使被污染, 也不会传播到 Sink.

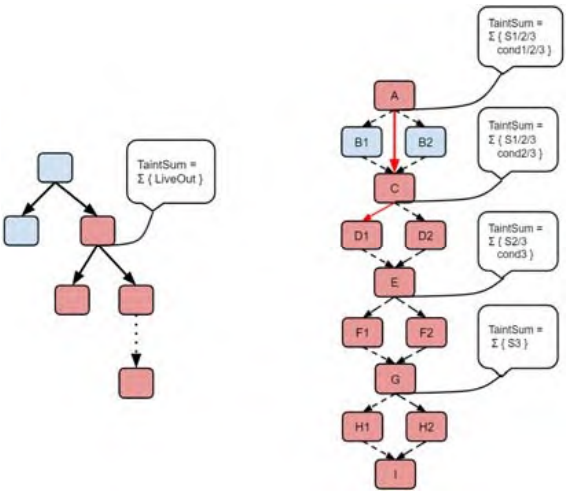

图3 活跃污点分析插桩及活跃轨迹

基本块级的静态 Chopping 结果是保守的. 执行过程中的活跃污点信息检查, 作为一种动态验证手段, 进一步提高了污点传播的分析精度. 活跃轨迹使得真正传播污点的危险基本块之间形成的路径被反馈给 Fuzzer, 消除了 Source-Sink 危险区域中没有污点传播的边覆盖对种子过滤的影响.

在原型系统 TaintPoint 中, 对于危险区域的每个分支基本块, 插入对当前分支点的有效污点变量标签的求和计算过程, 得到 TaintSum, 并将 TaintSum 转成 Bool 值写入特定全局变量 isLive, 表示在当前程序点是否仍然存在活跃的污点变量. 危险区域的每个分支基本块在运行时更新 isLive, 同时在每个基本块中, 根据 isLive 的值来决定是否记录危险基本块之间的逻辑轨迹.

2.4 种子过滤与选择

种子过滤策略是灰盒 Fuzzing 中的关键, 它直接决定了进化的方向. AFL 通过比较 trace_bits 和 virgin_bits 来判断此次执行是否产生了新覆盖(save_if_interesting), 从而决定是否要将测试用例视为种子加入队列, 以供后续变异.然而, 这种普适的覆盖反馈和种子过滤策略无法考虑不同边的特殊性. 在分离式插桩中, 我们将真正执行污点传播的活跃轨迹从整个轨迹 trace_bits 中分离, 其目的之一就是希望 Fuzzing 能够找到在危险区域产生新覆盖的种子, 新覆盖污点传播逻辑边的种子将比无关边新覆盖更有变异价值. 在 AFL 种子过滤策略的基础上, 我们对在 live trace 中产生新覆盖的种子进行特殊标记, 以提高其在队列中的优先级.

部分测试用例因为离开危险区域或遇到消毒验证函数而中断活跃轨迹的记录, 此时, 若有新的种子重新回到危险区域或者绕过了消毒函数, 将会扩大污点传播范围, 从而产生更长的活跃轨迹或是原来活跃轨迹的超集. 在种子选择阶段, 我们提出了一种新策略: 优先选择活跃轨迹更长的种子去变异. Fuzzing 将沿着活跃轨迹增长的方向进化, 不断探索更多更长的污点传播路径, 直到有更多的测试用例到达 Sink, 以发现污点传播相关的安全问题.

如图 4 所示, 3 个测试用例都因为进入消毒基本块($D1$,$F1$,$H1$)而失去所有活跃污点变量, 后续的 Trace 将不被记录. 第 3 个测试用例将污点属性传播得更远, 其活跃轨迹最长, 最为接近 Sink 点. 在种子选择阶段, 若此时队列中存在尚未被 Fuzz 的、携带更长活跃轨迹的种子, 则直接优先变异这些种子. 若队列中所有携带 Live Trace 的种子都被 Fuzz 过, 则采用 AFL 自带的策略. 该策略在两个维度为 Fuzzing 提供进化方向: (1) 优先选择在危险区域产生新覆盖的种子, 减少队列中非危险区域种子的低效 Fuzzing, 在横向上, 使得 Fuzzing 的方向朝着可能传播污点的危险区域进行; (2) 优先选择具有更长活跃轨迹的种子, 在纵向上使得 Fuzzing 朝着 Sink 进行, 直到有更多的测试用例到达 Sink, 发现污点传播漏洞.

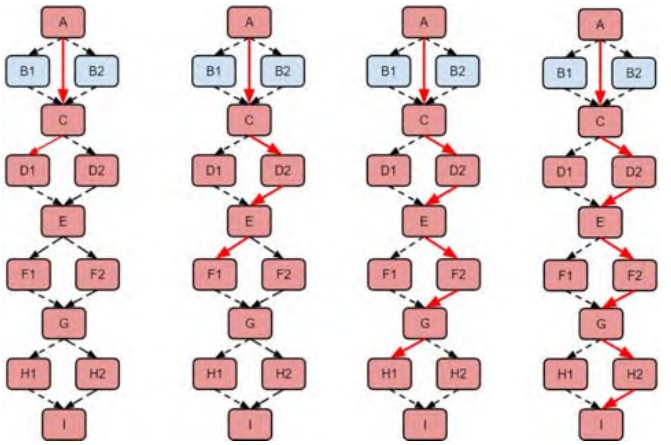

图 4 Source→Sink 活跃轨迹逐步增长

2.5 种子变异

随着 Fuzzing 研究的深入, 很多种子变异策略被提出. 一种主流策略是分析输入字段与分支变量之间的关系, 如 Angora、Vuzzer[42]中的 DTA、GREYONE 中 Fuzzing 驱动的污点推断等. 分支变量的值直接决定了不同分支的执行, 如果能够得到输入字节偏移量与这些分支变量之间的依赖关系, 就能有针对性地变异对应字节, 从而更有效地突破分支, 提高覆盖率. 活跃轨迹机制很容易与这些变异策略相结合, 提高变异的有效性. 如图 5 所示: 当输入变量 $cond1>0$ 时, 如表 1 中的 3 号和 4 号种子进入 $D1$ 基本块, 污点属性被清洁, 活跃轨迹在此处中断记录. 此时, 只有变异分支变量 $cond1$ 在输入中对应的字段, 才能使得 $cond1\leqslant 0$, 从而让污点继续传播下去, 只对变量 $cond2$ 或 $cond3$ 变异是无意义的.

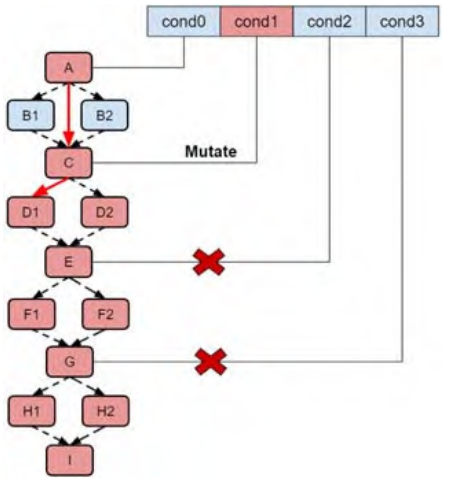

图 5 变异活跃轨迹中分支变量对应的输入

在原型系统 TaintPoint 的实现中, 我们将活跃轨迹机制与 RedQueen 的染色变异策略相结合. RedQueen 使用随机变异增加输入的熵, 通过染色算法定位 cmp 等分支指令变量在输入字节中的位置, 从而有针对性地变异这些字节, 有效突破分支. 我们在 cmp 等分支指令处的 Hook 函数中, 额外传入该指令所属基本块的 TaintSum (活跃污点标签之和). 程序执行时, 若 TaintSum 为 0, 则在 Hook 中使这些分支失效, 不进行后续染色. 如表 1 中 3 号和 4 号种子执行时, 会跳过针对 *cond*2>0, *cond*3>0 比较指令的染色, 只专注于活跃轨迹中存在的 *cond*0/1 分支变量.

2.6 回归例子

我们回到图 1 程序, 说明如何使用活跃轨迹系统化搜索污点传播路径. 如表 2 所示, 0 号为初始种子, 变异得到 1 号测试用例, 因为没有产生新的 Live Trace 而被丢弃. 2 号因为产生新的活跃边命中而被加入种子队列, 3 号和 4 号丢弃, 5 号产生了新的活跃边命中被加入种子队列, 并且活跃轨迹是 2 号的超集, 在下一轮变异中优先被选中, 专注于变异变量 *cond*3 对应的偏移字节, 从而能够较快地突破该分支, 得到 6 号测试用例. 最终, 污点变量 *S*3 到达 Sink, 触发污点传播漏洞.

表 2 基于活跃轨迹的污点路径搜索策略

序号	*cond* 变量(0 1 2 3)符号	TaintPoint 动作
0	(+ + + +)	初始种子, 加入队列
1	(– + + +)	丢弃
2	(+ – + +)	Live Trace 新命中, 加入队列
3	(+ + – +)	丢弃
4	(+ + + –)	丢弃
5	(+ – – +)	Live Trace 新命中且增长, 加入队列
6	(+ – – –)	污点到达 sink, 发现漏洞

3 原型实现

3.1 整体架构

TaintPoint 的架构如图 6 所示.

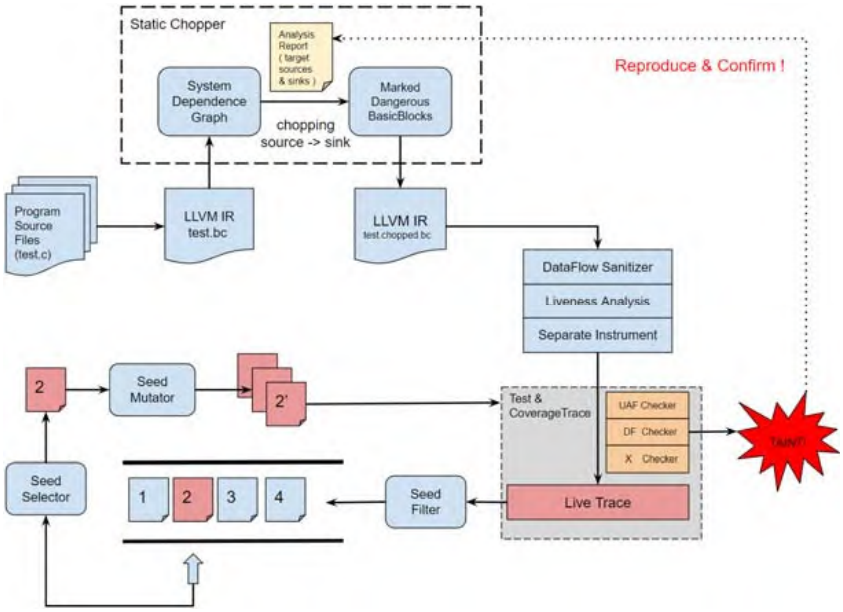

图 6 TaintPoint 整体架构图

首先, TaintPoint 中的 Chopper 组件对被测程序的中间代码进行静态分析, 根据目标 Source-Sink (如等待验

证的静态分析报告)进行危险区域标记; 接着, 依次经过 DFSan 插桩、活跃变量分析及分离式插桩, 并完成编译得到可执行文件. 在 Fuzzing 过程中, TaintPoint 的活跃轨迹反馈及相关优化策略将指导 Fuzzing 的种子过滤、选择、变异等过程. 此外, TaintPoint 的污点风格漏洞挖掘能力是可以扩展的: 各类 Checker 能够根据不同的污点风格漏洞的⟨Source-Sink-Sanitize⟩以及传播语义进行相应的建模. 目前, 我们为其构建了 DoubleFree 和 UseAfterFree Checker, 以挖掘这两种内存破坏漏洞.

3.2 动态污点分析

DFSan 是 LLVM 编译器架构 Sanitizers 系列中的动态污点分析工具. TaintPoint 集成了 DFSan 的 10.0.1 版本, 并进行了适当修改, 提升了其精确性和易用性. DFSan 在进行 BinaryOperator 二元运算符的污点传播时, 进行污点表标签合并(combination). 只要两个操作数中的任意一个污染, 则结果就是污染的. 然而对于 XOR、SUB 指令, 如 *a* XOR *a* 的结果为定值, 并不会根据 *a* 的值而变化. 所以无论 *a* 是否是污染的, 其结果都不会污染. 我们修复了这些不精确的污点传播语义.

DFSan 没有提供便于使用的 Sanitizer 污点清洁机制来去除污点. 例如 encrypt (&password)加密函数, password 被加密后, 即使泄露到外部也不会造成严重的安全问题, 所以其污点属性应该被移除. 在 DFSan 中, 若自定义实现此类函数需要给每个清洁函数标记 Custom, 并自定义 wrapper 函数完成清洁. 我们扩展了 DFSan 内置的标签机制, 增加了 Sanitizer1/Sanitizer2/Sanitizer3 标签, 分别清洁前 3 个参数的污点属性, 避免了大量手工定义 Wrapper.

DFSan 需要手工在程序中使用 API 如 dfsan_set_label 引入污点源、dfsan_has_label 检查标签值等, 难以应用于大规模程序分析. 我们基于 Custom Wrapper 机制, 在 Runtime Library 中自定义添加 Source 和 Sink 函数的污点行为, 如在 Source 的 Wrapper 中为相关变量影子内存设置污点标签, 在 Sink 的 Wrapper 中进行污点标签检查. 这样, 无需手动修改被测程序, 即可形成污点风格漏洞判定的全自动化.

3.3 活跃轨迹

TaintPoint 基于 AFL++ 3.0 实现. AFL++是 AFL 的社区版本, 集成了其他优秀 Fuzzer 的众多特性, 在 Google Fuzzbench 中的表现十分突出[43]. 为了区分特定的污点风格漏洞和其他 Crash 漏洞, 我们增加了一个特别的 exit 信号. 在 Fuzzing 过程中, 每次程序执行结束后, TaintPoint 根据此信号判定是否有污点传播漏洞的发生. 基于 SDG 实现的 Source-Sink 之间的两次可达性分析及标记是 LLVM IR 基本块级别的. 我们在 LLVM 插桩模式中实现了 Chopping 引导的分离式插桩, 给污点传播相关的危险基本块标记一个特定的“dangerous”标签, 在覆盖反馈插桩中, 通过 IR 中基本块的标签来判定是否处于危险区域.

我们通过 LLVM Pass 实现了活跃变量分析, 对于 CallInst 函数调用指令, 保守地认为其会使用所有全局变量, 同时对分析结果进一步优化, 只计入后续在危险基本块上使用的活跃变量. 因为其他活跃变量即使被污染, 如果不处于危险区域, 也不会传播至 Sink. 我们对危险基本块出口处有效活跃变量的污点标签进行累计, 得到活跃污点的标签值的算术和(TaintSum). 若 TaintSum 为 0, 表示后续不存在任何活跃污点信息. 在覆盖插桩中, 根据 TaintSum 的实时值, 决定对此条边命中计数是否加 1. 类似 AFL 的 trace_bits 哈希表, 危险区域的活跃轨迹被额外标记在 64 KB 的 live_trace 哈希表中, 0*x*FF 表示属于此条逻辑边危险区域, 同时根据压缩后的 live_trace_mini 进行污点传播路径长度比较.

3.4 Checker扩展

TaintPoint 具有扩展性的根源在于其核心的污点分析方法能够对不同的漏洞进行建模. 不同的 Checker 实现即为 Source-Sink 污点传播模型适配不同种类的漏洞. 污点分析能够检测的主要漏洞种类如下所示.

- 双重释放 DF (CWE-415);
- 释放后使用 UAF (CWE-416);
- SQL 注入(CWE-98);
- 跨站脚本 XSS (CWE-79);

- 跨站请求伪造 CSRF (CWE-352);
- 栈/堆缓冲区溢出(CWE-121/122);
- OS 命令注入(CWE-78);
- 敏感数据未加密(CWE-311);
- …

值得注意的是, TaintPoint 并不能全自动化识别 Source-Sink, 其输入是静态分析报告, 即等待验证或复现的 Source-Sink 对, 以函数、变量名及行号为输入, 且需要为此类漏洞在 TaintPoint 的运行时库中提前构建相关函数的截断器(interceptor), 从而在 Fuzzing 执行时自动化地在关键函数处设置或检查相关的污点属性. 例如: TaintPoint 复现缓冲区溢出漏洞时, 需要提前构建缓冲区溢出 Checker 为 memcpy/strcpy 或我们插桩的特定指针解引用的 use 函数构建截断器作为 Sink 点; 同时, 为潜在的输入函数如 scanf、fread 编写截断器, 作为 Source 设置污点源.

如何全自动化识别各类漏洞模型中的 Source-Sink, 是一个十分精巧且困难的问题, 也是污点分析固有的问题. 如: 对于著名的 Heartbleed 漏洞, 在被披露前长达 2 年的时间内, 都没有被各种类型的程序分析器发现. 漏洞被披露以后, Synopsys 公司首先提供了解决方案, 在其静态分析器 Coverity 的污点分析引擎中加入 ntohs/ntohl 交换字节序的函数作为污点源. 因为被交换字节序的数据通常来自网络, 即可以被视为缓冲区溢出中的污染源(source). 这种“人工在漏洞发生后为分析器加入 Source”的行为, 也说明了自动化确定污点分析模型中 Source-Sink 的困难. 其更倾向于更高层次的语义分析, 并不是本文研究的重点.

4 实验评估

4.1 测试基准

现有 Fuzzer 中流行的边覆盖是一种通用的控制流信息, 目的是尽可能地扩大程序覆盖率, 从而有更大的可能触发各种漏洞. 然而数据流信息是问题相关的, 对于不同种类的漏洞, 需要建立各自的 Source-Sink 污点传播模型. 例如: 针对隐私泄露问题, Source 为用户输入的关键数据, Sink 为敏感操作, 如通过网络发送; 针对内存破坏漏洞, 如 UseAfterFree, Source 为 free 调用点, 而 Sink 为相关指针变量的解引用点. 在 C/C++语言上, 缺乏专用的污点分析 Benchmark 去评估各类安全分析工具的有效性. Juliet Test Suite C/C++ 1.3v[44]是美国标准技术研究所 NIST 发布的测试套件, 它包括了 CWE 分类中的 64 099 个漏洞, 且将 Source-Sink 显示地标记在代码中, 最为接近我们的需要. 然而 Juliet 测试集主要用于评估静态分析工具, 其漏洞相关的数据流和控制流障碍都是自动化组合生成, 对于 Fuzzer 来说过于简单. LAVA-M[45]测试集只包括缓冲区溢出, 且造成漏洞触发的条件都是 MagicBytes, 并不适用于 Fuzzer 上污点风格漏洞挖掘的评估. 最终, 我们选择了 UAFuzz 中推出的 UAFBench, 其包含 14 个真实的 UAF/DF 漏洞, 专门用于评估 Fuzzer 对于内存破坏漏洞 UseAfterFree 及 DoubleFree 的发现能力. 每个漏洞包括 CVE 号、程序源代码、PoC、动态分析结果(valgrind trace)以及相关 Fuzzing 脚本.

4.2 实验设置

我们为 DF、UAF 这两类漏洞分别制定相应的检测规则, 并在 TaintPoint 中实现相应的 Checker. 对于 Double Free, 我们根据静态分析报告中两次 free 的调用点(source-sink)进行 Chopping, 通过 Hook free 调用点, 为该指针变量指向的地址空间所对应的影子内存设置污点标签, 并跟踪其在程序中的动态传播. 在每次 free 之前, 检查其是否存在污点标记: 若存在, 说明此次 free 的地址区域在之前已被释放, 则检测到 Double Free 漏洞, 最后通过 exit 特殊信号反馈给 Fuzzer. 对于 UAF, 在静态分析结果中的使用处, 我们对其插桩一个简单的 use 函数, 并进行 Hook, 以运行时检查对应地址空间的污染情况.

TaintPoint 能够将分析报告中的 Source-Sink 对作为目标, 来识别危险区域并进行复现验证. 为此, 我们选择了流行的 SVF 静态分析框架进行安全分析. 我们对 SVF 自带的 DoubleFree Checker 进行了一定的改进并构

建 UseAfterFree Checker, 分析结果见表 3. 注意: 分析器的选取以及分析报告结果并不会影响 TaintPoint 的漏洞挖掘过程, 主要区别在于潜在目标 Source-Sink 的位置不同.

表 3 UAFBench 漏洞类型及静态分析结果

Bug-ID	Program	Crash	Bug-type	Results (true positive)
CVE-2019-6455	rec2csv	NO	DF	18 (1)
Giflib-bug-74	gifsponge	NO	DF	9 (1)
CVE-2018-11416	jpegoptim	NO	DF	3 (1)
gifsicle-issue-122	gifsicle	NO	DF	0 (0)
CVE-2019-20633	patch	NO	DF	9 (1)
CVE-2018-20623	readelf	NO	UAF	10 (0)
yasm-issue-91	yasm	NO	UAF	0 (0)
CVE-2016-4487	cxxfilt	YES	UAF	3 (1)
mjs-issue-78	mjs	NO	UAF	6 (0)
mjs-issue-73	mjs	NO	UAF	4 (0)
CVE-2018-11496	lrzip	NO	UAF	1 (0)
CVE-2018-10685	lrzip	NO	UAF	1 (0)
CVE-2016-3189	bzip2	YES	UAF	5 (1)

SVF 采用稀疏数据流分析(sparse value-flow analysis)算法, 其路径不敏感性导致静态分析结果中存在较多误报和漏报. 实验将报告中含有真实漏洞结果(表 3 中 True Positive=1)的被测程序直接送入 Chopper, 根据报告结果中的 Source-Sink 漏洞点进行危险区域识别; 对于静态分析结果不包含真实漏洞的被测程序, 我们在结果之上加入 Benchmark 中已有的 Valgrind 动态分析报告, 希望通过 Fuzzing 的方式验证这些报告的真实性, 评估不同模糊器复现漏洞的各项表现.

TaintPoint 围绕活跃轨迹的一系列策略能够高效搜索指定 Source-Sink 之间的污点传播路径, 验证静态分析结果, 且尽可能地消除静态分析误报, 降低动态分析漏报. 我们将 TaintPoint 与业界领先的通用模糊器 AFL++及定向模糊器 AFLGO 进行对比, 以评估围绕活跃轨迹的各种优化策略在挖掘污点风格漏洞的有效性. AFL++是 AFL 的后续社区版本, 集成了众多 Fuzzer 特性, 在 Google FuzzBench 中表现极佳. AFLGo 是近年来最流行的开源定向 Fuzzer, 基于距离的能量调度策略能够制导 Fuzzer 朝着目标点进行. 由于 UAFBench 上推出的 UAFuzz 是针对二进制程序的模糊器, 所以我们没有将其纳入实验对比. UAFBench 中的大多数漏洞都不会造成程序 Crash, 因而无法被检测到. 为了捕获这两种内存破坏漏洞, 在实验中, 我们为 AFL++分别结合 ASan 及 DFSan. 由于 AFLGo 版本较旧, 结合 DFSan 形成全自动化挖掘十分困难, 我们只为其结合 ASan.

AFLGo 依赖版本较旧, 测试系统环境为 Ubuntu 16.04 LTS, Clang/LLVM 4.0; AFL++和 TaintPoint 的测试系统环境为 Ubuntu 20.04 LTS, Clang/LLVM 10.0.1, CPU 为 AMD Ryzen 4800H@2.9 GHz, 8C16T, 物理内存为 16 GB.

4.3 研究问题及实验结果

为了评估 TaintPoint 挖掘污点风格漏洞的有效性, 我们提出以下 3 个研究问题.

- RQ1. 效率: TaintPoint 能够更快复现 Benchmark 中的漏洞吗?
- RQ2. 产出: TaintPoint 能够产生更多的 Crash 吗?
- RQ3. 性能: TaintPoint 的性能表现/执行速度如何?

对于 UAFBench 中的每个漏洞, 设置 Fuzzer 相关运行数据(Crash 个数、速度统计)的基本记录时间为 600 s, 总计运行 10 轮, 实验数据取平均值.

- RQ1. TaintPoint 能够更快复现 Benchmark 中的漏洞吗?

复现 Benchmark 中漏洞的时间开销是最重要的问题, 能够更快地触发漏洞, 证明了 TaintPoint 围绕 Live Trace 的一系列优化策略的有效性. 与 AFLGo 和 AFL++采用的边覆盖这种普适的控制流信息不同, TaintPoint 利用污点风格漏洞相关的静态 Chopping 结果和动态活跃污点等数据流信息指导 Fuzzing 的各个阶段.

实验结果见表 4. 对于漏洞 gifsicle-issue-122 和 CVE-2018-11496, TaintPoint 触发漏洞的用时长于结合了 ASAN 的 AFL++. 在其他 11 个漏洞测试中, TaintPoint 均比结合了 ASAN 的 AFLGO 和 AFL++更快地复现了

漏洞, 平均用时为 AFLGO 的 41.3%, 为 AFL++的 64.2%. AFLGo 基于较旧版本的 AFL, 即使其拥有基于距离的定向机制, 但仍然用时较长, 在所有测试中均用时最长. 为 AFL++结合 DFSan 消除了 Sanitizer 差异所带来的影响, 能够相对公平地评估 TaintPoint 中活跃轨迹优化机制的有效性. DFSan 由于其本身的开销低于 ASan, 使得 AFL++结合 DFSan 的漏洞复现时间普遍少于结合 ASan. 然而除去 gifsicle-issue-122、CVE-2016-4487、CVE-2018-11496, 在其他 10 个漏洞测试中, TaintPoint 的表现都优于 AFL++&DFSan 组, 平均用时为其 67.1%.

表 4　在 UAFBench 上的漏洞触发时间对比

首次触发时间(s)	AFLGO & ASAN	AFL++ & ASAN	AFL++ & DFSAN	TaintPoint
CVE-2019-6455	572	102	89	65
giflib-bug-74	33	13	14	6
CVE-2018-11416	27	11	8	8
gifsicle-issue-122	2 289	439	402	561
CVE-2019-20633	3 751	1 951	1 751	1 325
CVE-2018-20623	121	94	88	79
yasm-issue-91	289	212	193	167
CVE-2016-4487	523	345	237	291
mjs-issue-78	662	496	415	312
mjs-issue-73	4 143	2 210	2 006	1 943
CVE-2018-11496	3	2	2	4
CVE-2018-10685	55	29	26	17
CVE-2016-3189	41	33	29	19

AW1: TaintPoint 的活跃轨迹反馈机制及相关优化策略有效提升了污点风格漏洞挖掘的效率, 在漏洞复现效率上优于结合 DFSan 和 ASAN 的 AFLGO 及 AFL++.

- RQ2. TaintPoint 能够产生更多的 crash 吗?

Fuzzing 的产出也是我们重点关注的指标, 越多的 Unique Crash 表明 Fuzzer 的路径搜索能力越强. 对于污点风格的漏洞来说, Crash 数量越多, 表明有更多的携带污点的测试用例到达 Sink 点. 实验结果见表 5.

表 5　在 UAFBench 上的 Crash 数量对比

Crash 数量(个)	AFLGO & ASAN	AFL++ & ASAN	AFL++ & DFSAN	TaintPoint
CVE-2019-6455	9	24	41	53
giflib-bug-74	2	2	3	3
CVE-2018-11416	4	10	13	12
gifsicle-issue-122	1	1	1	1
CVE-2019-20633	2	1	3	4
CVE-2018-20623	5	12	10	10
yasm-issue-91	2	3	3	1
CVE-2016-4487	1	4	5	3
mjs-issue-78	4	5	5	11
mjs-issue-73	1	1	1	1
CVE-2018-11496	7	6	29	41
CVE-2018-10685	4	5	12	19
CVE-2016-3189	2	3	5	8

我们手工检查并重放了这些 Crash, 以确认真实性. 除去 CVE-2016-4487、yasm-issue-91、CVE-2018-11416, TaintPoint 在其他 10 个漏洞测试程序上产生的 Crash 数量均多于 AFLGO 和 AFL++. 活跃轨迹反馈机制所带来的系列优化方法能够使得 TaintPoint 高效搜索危险区域, 区分更多不同的污点传播路径, 从而有更多携带污点信息的测试用例到达 Sink 点触发 Crash. 同时, 我们将这些 Crash 结果与 SVF 静态分析报告进行对比, 确认了报告中的虚警(false positive). 更多的 Crash 也更能证明 TaintPoint 在潜在 Source-Sink 危险区域的搜索能力, 从而有更高的可能性来消除误报.

AW2: 相比 AFLGo 和 AFL++, TaintPoint 能够产生更多的 Crash, 有着更强的漏洞挖掘能力.

- RQ3. TaintPoint 的性能表现/执行速度如何?

执行速度是 Fuzzing 的重要指标, 直接影响漏洞挖掘的最终结果. Fuzzing 方法的本质就在于持续大量地生成测试用例来触发漏洞, 所以 Fuzzing 速度十分重要. 实验结果见表 6.

实验结果表明: TaintPoint 的执行速度远远超过结合 ASAN 的 AFLGo 和 AFL++, 约为 AFLGo & ASAN 的

320%、AFL++ & ASAN 的 266%. 主要原因在于: TaintPoint 的 DTA 引擎 DFSan 只在定义的 Sink 点作污点信息检查, 而 ASAN 插桩需要在每个潜在的问题点检查相关的影子内存, 增加了每次执行的开销, 在一定程度上影响了 Fuzzer 的执行速度. 由于活跃轨迹机制在危险区域的分支基本块处插桩检查活跃污点, TaintPoint 的执行速度约为 AFL++ & DFSan 的 87%. 结合 RQ1/RQ2 中的实验结果表明: TaintPoint 在速度不占优势的情况下, 仍然有着更强的污点风格漏洞挖掘能力.

表 6 执行速度对比

速度(千次/10min)	AFLGO & ASAN	AFL++ & ASAN	AFL++ & DFSAN	TaintPoint
CVE-2019-6455	419	481	1 790	1 550
giflib-bug-74	620	850	3 910	3 680
CVE-2018-11416	559	633	1 330	1 170
gifsicle-issue-122	53	76	99	82
CVE-2019-20633	611	804	1 730	1 610
CVE-2018-20623	422	402	1 810	1 490
yasm-issue-91	255	289	270	263
CVE-2016-4487	1 092	1 394	3 310	3 035
mjs-issue-78	483	630	1 920	1 690
mjs-issue-73	519	697	1 950	1 810
CVE-2018-11496	495	612	2 160	1 980
CVE-2018-10685	480	570	2 080	1 750
CVE-2016-3189	1 201	1 190	3 010	2 840

AW3: TaintPoint 的速度远超结合了 ASan 的 AFLGo 和 AFL++, 其污点分析引擎 DFSan 开销低于广泛使用的 ASan; 活跃轨迹机制本身具有一定的开销, 但其带来的挖掘收益是相对值得的. 较高的执行速度为 Fuzzing 的漏洞挖掘提供了重要支持.

4.4 实验总结

我们以 UAF/DF 类型漏洞为例, 评估 TaintPoint 中的活跃轨迹反馈机制在面向污点风格漏洞的挖掘能力. 实验结果表明: 在控制流的基础上结合数据流信息的覆盖反馈及其相关优化策略, 能够有效指导 Fuzzing 高效挖掘污点传播漏洞. 此外, 我们使用 TaintPoint, 在开源项目 OpenJPEG 和 PDFium 上发现了 4 个漏洞并得到了 Google 和 RedHat 的确认. TaintPoint 是可扩展的, 对于其他种类的污点风格漏洞, 如 SQL 注入、隐私泄露等, 我们可以在上层建立更多更完善的污点传播模型, 构建不同种类的 Checker 来发现不同的漏洞, 这是未来的工作之一. 此外, TaintPoint 目前的局限性在于: (1) Chopping 阶段的精准依赖分析开销较大, 且缺乏精准的过程间静态活跃变量分析; (2) 实时污染状态检查所带来的运行时开销过大; (3) 内置的 DTA 无法跟踪控制依赖造成的隐式流. 我们期待进一步研究这些关键问题并继续改进 TaintPoint.

5 相关工作

5.1 Taint-based Fuzzing

污点分析在 Fuzzing 研究中有着重要的作用, 很多工作根据污点分析结果确定程序的输入字段与分支变量之间的依赖关系, 提高变异的有效性. Angora 使用 DFSan 作字节级动态污点跟踪, GREYONE 采用轻量级 Fuzzing 驱动的污点推断(FIT), 取代传统 DTA 以减少开销, 并且能够观察到由于控制依赖关系而产生的隐式污点传播. RedQueen 使用随机变异的策略增加输入熵, 基于染色算法定位输入偏移量. 这些 Fuzzer 广泛采用边覆盖, 用数据流/污点传播信息辅助 Fuzzing 的变异阶段, 其目标仍然是扩大整个程序的覆盖率. 所不同的是, 我们发现在检测污点风格漏洞时, 流行的边覆盖反馈机制存在本文讨论的问题, 提出在普适的控制流信息的基础上加入污点数据流信息作为反馈, 目标是更加高效地搜索 Source-Sink 之间的污点传播路径. TaintPoint 也能作用于通用 Fuzzing 的种子变异阶段, 得到更精准的污点分析结果来指导变异, 例如为 Neutaint 的神经网络提供质量更高的训练集.

5.2 Directed Greybox Fuzzing

相比通用 Fuzzing 的目标为扩大整个程序的覆盖率, 定向 Fuzzer, 如 ALFGo、HawkEye 能够指导 Fuzzer

更多地生成靠近目标点的测试用例, 其应用场景为补丁测试、静态分析报告验证和漏洞复现等. AFLGo 利用静态分析计算出每个基本块与目标位置的距离, 在程序运行时, 根据距离的不同为种子分配相应的能量, 从而使之能够朝着目标点进行 Fuzzing. 然而, 如何指导 Fuzzer 朝着目标点测试并不是本文研究的目标, 边覆盖反馈的粗糙性在这些 Fuzzer 中同样存在. TaintPoint 的目标是更加精确、高效地搜索 Source-Sink 之间的污点传播路径从而发现漏洞, 其集成了数据流/污点信息的活跃轨迹反馈策略同样可以作用于定向 Fuzzer 如 AFLGo. 将 TaintPoint 与基于距离的定向 Fuzzer 结合是未来工作之一, 我们相信可以取得更好的效果.

5.3 DataFlow Testing

TaintPoint 本质上是基于 Fuzzing 的数据流测试. 数据流测试重点关注的是数据在程序中的流动, 分析并测试程序中变量的定义(def)到使用(use)之间的合理性. 如 Survey[46]中所说, 数据流测试填补了全路径测试和分支/语句测试之间的空白, 旨在精确地定位数据流异常, TaintPoint 的目标即如何高效搜索 Source-Sink 之间潜在的污点传播路径. 根据不同的测试用例生成方法, 数据流测试可分为基于随机变异、符号执行[47]、模型检测[48]等. 符号执行和模型检测的缺点在于可伸缩性差, 存在状态和路径爆炸等问题, 而随机测试中的灰盒 Fuzzing 通过覆盖反馈的方式极大地提高了测试效率. 据我们所知, TaintPoint 是第一个基于 AFL 的、以数据流测试(污点风格漏洞)为目标的灰盒 Fuzzer.

6 总 结

本文详细分析了主流灰盒 Fuzzing 广泛采用的边覆盖(控制流反馈)在检测污点风格漏洞时存在的问题, 并提出在边覆盖反馈的基础上结合数据流信息, 形成的活跃轨迹(live trace)作为覆盖反馈信号. 同时, 围绕活跃轨迹分别在 Fuzzing 的插桩、种子过滤、选择、变异这 4 个阶段改进现有策略, 优化 Source-Sink 之间的信息流搜索. 我们实现了原型系统 TaintPoint, 并在 UAFBench 上评估了 TaintPoint 在漏洞挖掘效率、产出和性能这 3 个方面的表现. 实验结果表明: 结合了污点数据流的活跃轨迹作为反馈信息, 能够有效指导 Fuzzing 周期的各个阶段, 高效挖掘污点风格漏洞.

References:

[1] American fuzzy lop. http://lcamtuf.coredump.cx/afl

[2] Data flow sanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html

[3] Address sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html

[4] Memory sanitizer. https://clang.llvm.org/docs/MemorySanitizer.html

[5] Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: Proc. of the 31st IEEE Symp. on Security and Privacy (S&P 2010). Berkeley: IEEE, 2010.

[6] HeartBleed vulnerability. https://heartbleed.com/

[7] Directed greybox fuzzing. In: Proc. of the ACM SIGSAC Conf. on Computer & Communications Security. ACM, 2017. 2329–2344.

[8] She D, Chen Y, Shah A, Ray B, Jana S. Neutaint: Efficient dynamic taint analysis with neural networks. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2020.

[9] Chalupa M. DG: A program analysis library. Software Impacts, 2020, 6: 100038.

[10] Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T. REDQUEEN: Fuzzing with input-to-state correspondence. In: Proc. of the Network and Distributed Systems Security Symp. (NDSS). 2019.

[11] Fioraldi A, Maier D, Eifeldt H, *et al*. AFL++: Combining incremental steps of fuzzing research. In: Proc. of the 14th USENIX Workshop on Offensive Technologies (WOOT 2020). 2020.

[12] Nguyen MD, Bardin S, Bonichon R, *et al*. Binary-level directed fuzzing for use-after-free vulnerabilities. arXiv: 2002.10751. 2020.

[13] Wang L, Li F, Li L, Feng XB. Principle and practice of taint analysis. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 860−882 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[14] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[15] Sui Y, Xue J. SVF: Interprocedural static value-flow analysis in LLVM. In: Proc. of the Int'l Conf. on Compiler Construction (CC). 2016.

[16] TAJ: Effective taint analysis of Web applications. ACM SIGPLAN Notices, 2009, 44(6): 87−97.

[17] Kemerlis VP, Portokalidis G, Jee K, Keromytis AD. Libdft: Practical dynamic data flow tracking for commodity systems. In: Proc. of the ACM SIGPLAN/SIGOPS Int'l Conf. on Virtual Execution Environments (VEE). 2012.

[18] Song D, Brumley D, Yin H, *et al*. BitBlaze: A new approach to computer security via binary analysis. In: Proc. of the Int'l Conf. on Information Systems Security. Berlin, Heidelberg: Springer, 2008. 1−25.

[19] Kang M, Mccamant S, Poosankam P. DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the Network and Distributed Systems Security Symp. (NDSS). 2011.

[20] Chua Z, Wang Y, Blu T, Saxena P, Liang Z, Su P. One engine to serve'em all: Inferring taint rules without architectural semantics. In: Proc. of the Network and Distributed System Security Symp. (NDSS). 2019.

[21] Zhu H, Dillig T, Dillig I. Automated inference of library specification for source-sink property verification. In: Proc. of the Asian Symp. on Programming Languages and Systems (APLAS). 2013.

[22] Arzt S, Bodden E. StubDroid: Automatic inference of precise dataflow summaries for the android framework. In: Proc. of the 38th Int'l Conf. on Software Engineering (ICSE). 2016.

[23] Arzt S, Rasthofer S, Fritz C, Bodden E. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proc. of the in Int'l Conf. on Programming Language Design and Implementation (PLDI). 2014.

[24] Zhu H, Dillig T, Dillig I. Automated inference of library specification for source-sink property verification. In: Proc. of the Asian Symp. on Programming Languages and Systems (APLAS). 2013.

[25] Banerjee S, Devecsery D, Chen PM, *et al*. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In: Proc. of the 2019 IEEE Symp. on Security and Privacy (SP). IEEE, 2019.

[26] Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z. GREYONE: Data flow sensitive fuzzing. In: Proc. of the 29th USENIX Security Symp. 2020.

[27] https://source.android.com/devices/tech/debug/sanitizers

[28] https://source.android.com/devices/tech/debug/libfuzzer

[29] Peach fuzzing framework. https://github.com/MozillaSecurity/peach

[30] Cadar C, Dun Ba RD, Engler DR. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. of the Usenix Conf. on Operating Systems Design & Implementation. USENIX Association, 2009.

[31] Wang F, Shoshitaishvili Y. Angr—The next generation of binary analysis. In: Proc. of the IEEE. IEEE, 2017. 8–9.

[32] Stephens N, Grosen J, Salls C, *et al*. Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. of the Network and Distributed System Security Symp. 2016.

[33] https://lafintel.wordpress.com/

[34] Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z. CollAFL: Path sensitive fuzzing. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2018.

[35] Nagy S, Hicks M. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2018.

[36] Zhu X, Feng X, Meng X, Wen S, Camtepe S, Xiang Y, Ren K. CSI-fuzz: Full-speed edge tracing using coverage sensitive instrumentation. IEEE Trans. on Dependable and Secure Computing, 2020.

[37] Chen P, Chen H. Angora: Efficient fuzzing by principled search. In: Proc. of the IEEE Symp. on Security and Privacy (S&P). 2018.

[38] Bhme M, Pham VT, Roychoudhury A. Coverage-based greybox fuzzing as Markov chain. In: Proc. of the 2016 ACM SIGSAC Conf. ACM, 2016.

[39] Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y. Hawkeye: Towards a desired directed grey-box fuzzer. In: Proc. of the ACM SIGSAC Conf. on Computer and Communications Security (CCS). 2018.

[40] Jackson D, Rollins EJ. Chopping: A generalization of slicing. 1994. https://www.researchgate.net/publication/2500242_Chopping_A_Generalization_of_Slicing

[41] Weiser M. Program slicing. IEEE Trans. on Software Engineering, 1984.

[42] Rawat S, Jain V, Kumar A, Cojocar L, Guiffrida C, Bos H. VUzzer: Application-aware evolutionary fuzzing. In: Proc. of the Network and Distributed Systems Security Symp. (NDSS). 2017.

[43] Metzman J, Szekeres L, Simon L, *et al*. FuzzBench: An open fuzzer benchmarking platform and service. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. 2021. 1393–1403.

[44] https://samate.nist.gov/SRD/testsuite.php

[45] Dolan-Gavitt B, Hulin P, Kirda E, *et al*. LAVA: Large-scale automated vulnerability addition. In: Proc. of the Security & Privacy. IEEE, 2016.

[46] Su T, Wu K, Miao WK, *et al*. A survey on data-flow testing. ACM Computing Surveys, 2017, 50(1): 1–35.

[47] Su T, Fu Z, Pu G, *et al*. Combining symbolic execution and model checking for data flow testing. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Software Engineering, Vol.1. IEEE, 2015. 654–665.

[48] Hong HS, Cha SD, Lee I, *et al*. Data flow testing as model checking. In: Proc. of the 25th Int'l Conf. on Software Engineering. IEEE, 2003. 232–242.

附中文参考文献:

[13] 王蕾, 李丰, 李炼, 冯晓兵. 污点分析技术的原理和实践应用. 软件学报, 2017, 28(4): 860−882. http://www.jos.org.cn/1000-9825/5190.htm [doi: 10.13328/j.cnki.jos.005190]

[14] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80−109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

方浩然(1996－), 男, 硕士生, 主要研究领域为编译器, 程序分析.

李航宇(1995－), 男, 硕士生, CCF 学生会员, 主要研究领域为模糊测试, 静态分析, 机器学习.

郭帆(1977－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为网络安全, 程序分析, 漏洞挖掘.