第六章 变异与模糊测试（3学时）

1. 变异测试
2. 模糊测试

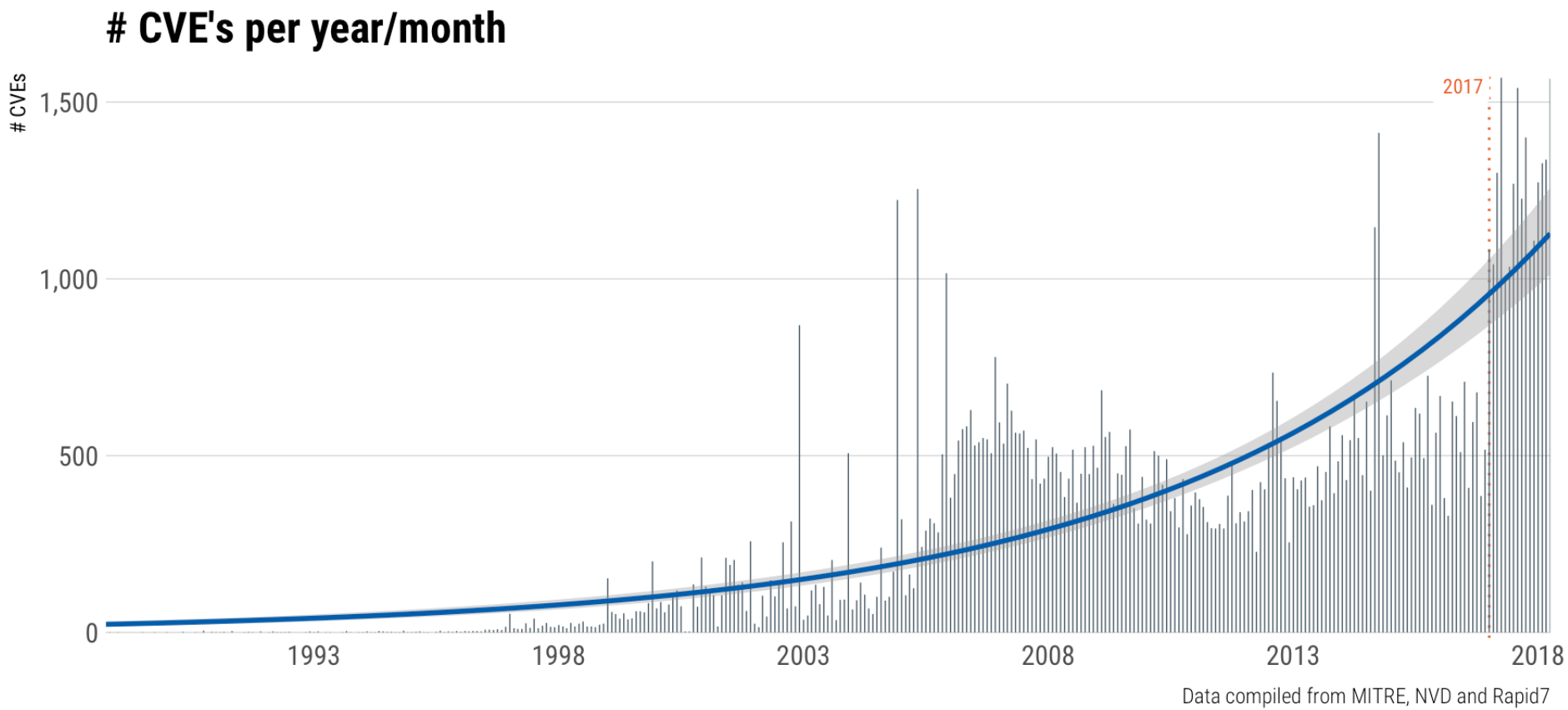
1.背景

- 模糊测试 （fuzzing）
  - 一种软件测试技术
  - 将自动或半自动生成的随机数据输入程序，监视程序异常
  - 发现*程序安全问题*

1.背景

- 程序安全问题:
  - 系统在设计、实现或管理过程中存在的缺陷与不足
  - 导致系统中存在可能被攻击者利用的弱点
  - 程序安全问题具有广泛性
    - 现代软件规模增长、结构日趋复杂
    - 漏洞的数目和多样性日渐增加

1.背景



图片来源：https://www.rapid7.com/blog/post/2018/04/30/cve-100k-by-the-numbers/

1.背景

- 软件漏洞常常难以检测
  - 漏洞类型多样
  - 软件规模增大，结构日趋复杂
  - 人工代码检查开销大，需要要专家知识
- 危害性：
  - 导致系统及其数据的保密性、完整性、可用性、访问控制受到威胁
  - 0 day漏洞危害尤其严重

1.背景

– 案例：log4j漏洞

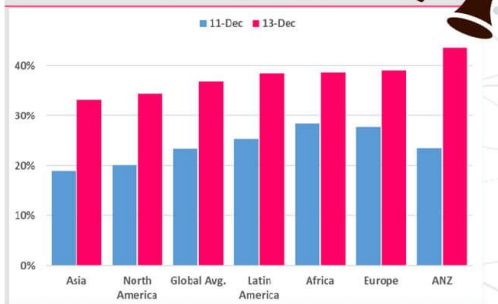
- Log4j 2.0(Log4J2)的一个零日远程代码执行漏洞，被定性为“过去十年来最大、最关键的漏洞”
- CVE-2021-44228
- 2021年11月24日发现并报告
- 允许用户通过输入日志数据执行任意代码
- 由于Log4j广泛应用于各种应用程序和服务，这个漏洞对许多组织和个人造成严重影响

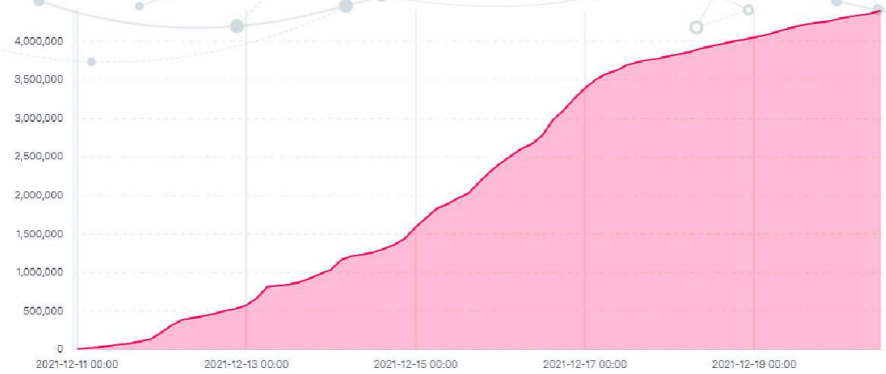
Check Point
SOFTWARE TECHNOLOGIES

Log4j - A TRUE CYBER PANDEMIC

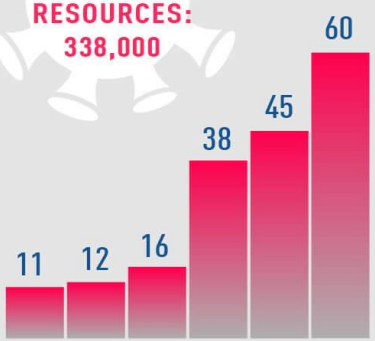
It is clearly one of the most serious vulnerabilities on the internet in recent years. When we discussed the Cyber pandemic, this is exactly what we meant – quickly spreading devastating attacks.

% Corporate Networks impacted per region





AMOUNT OF ATTACKED RESOURCES: 338,000



New variations of the original exploit being introduced rapidly - over 60 in less than 24 hours

Check Point prevented over 820.000 attack attempts since the outbreak

We have so far seen an attempted exploit on over 40% of corporate networks globally

The data used in this report was detected by Check Point Software's Threat Prevention technologies, stored and analyzed in Check Point ThreatCloud. ThreatCloud provides real-time threat intelligence derived from hundreds of millions of sensors worldwide, over networks, endpoints and mobiles. The intelligence is enriched with AI-based engines and exclusive research data from Check Point Research – The intelligence & research arm of Check Point Software Technologies.

图片来源：
https://blog.checkpoint.com/security/the-numbers-behind-a-cyber-pandemic-detailed-dive/

1.背景

- 模糊测试发展历史
  - 模糊测试是一种自动化的检测软件安全漏洞的方式
  - 由Barton Miller教授于1989年提出
    - 使用异常的输入测试目标程序
    - 发现Unix程序的可靠性问题
    - 25%-33%的被测应用程序在随机输入下崩溃

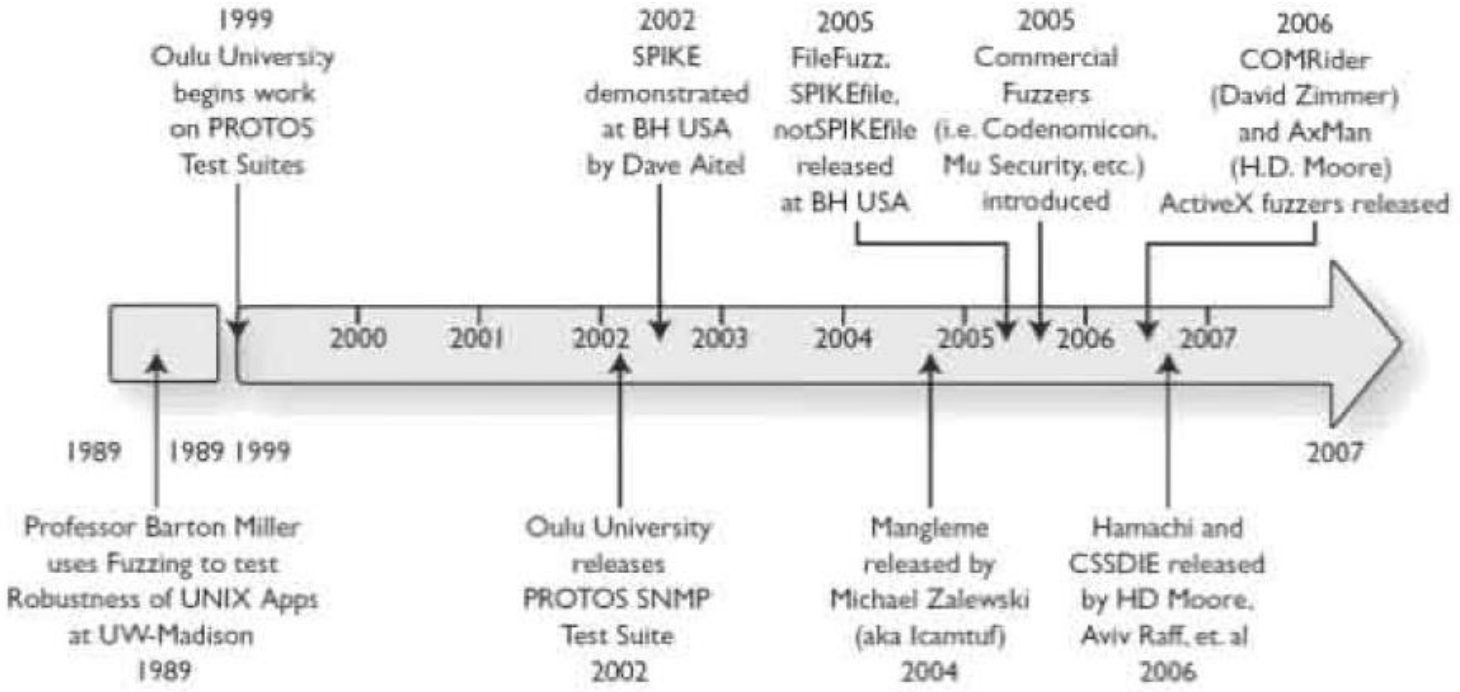
1.背景

A recent experience led us to believe that, while formal verification of a complete set of operating system utilities was too onerous a task, there was still a need for some form of more complete testing. It started on a dark and stormy night. One of the authors was logged on to his workstation on a dial-up line from home and the rain had affected the phone lines; there were frequent spurious characters on the line. It was a race to see if he could type a sensible sequence of characters before the noise scrambled the command. This line noise was not surprising; but we were surprised that these spurious characters were causing programs to crash. These programs included a significant number of basic operating system utilities. It is reasonable to expect that basic utilities should not crash (''core dump''); on receiving unusual input, they might exit with minimal error messages, but they should not crash. This experience led us believe that there might be serious bugs lurking in the systems that we regularly used.

B. P. Miller, et al, An empirical study of the reliability of UNIX utilities， 1990

1.背景

– 发展历程



Sutton M ,et al. Fuzzing: brute force vulnerability discovery. 2007.

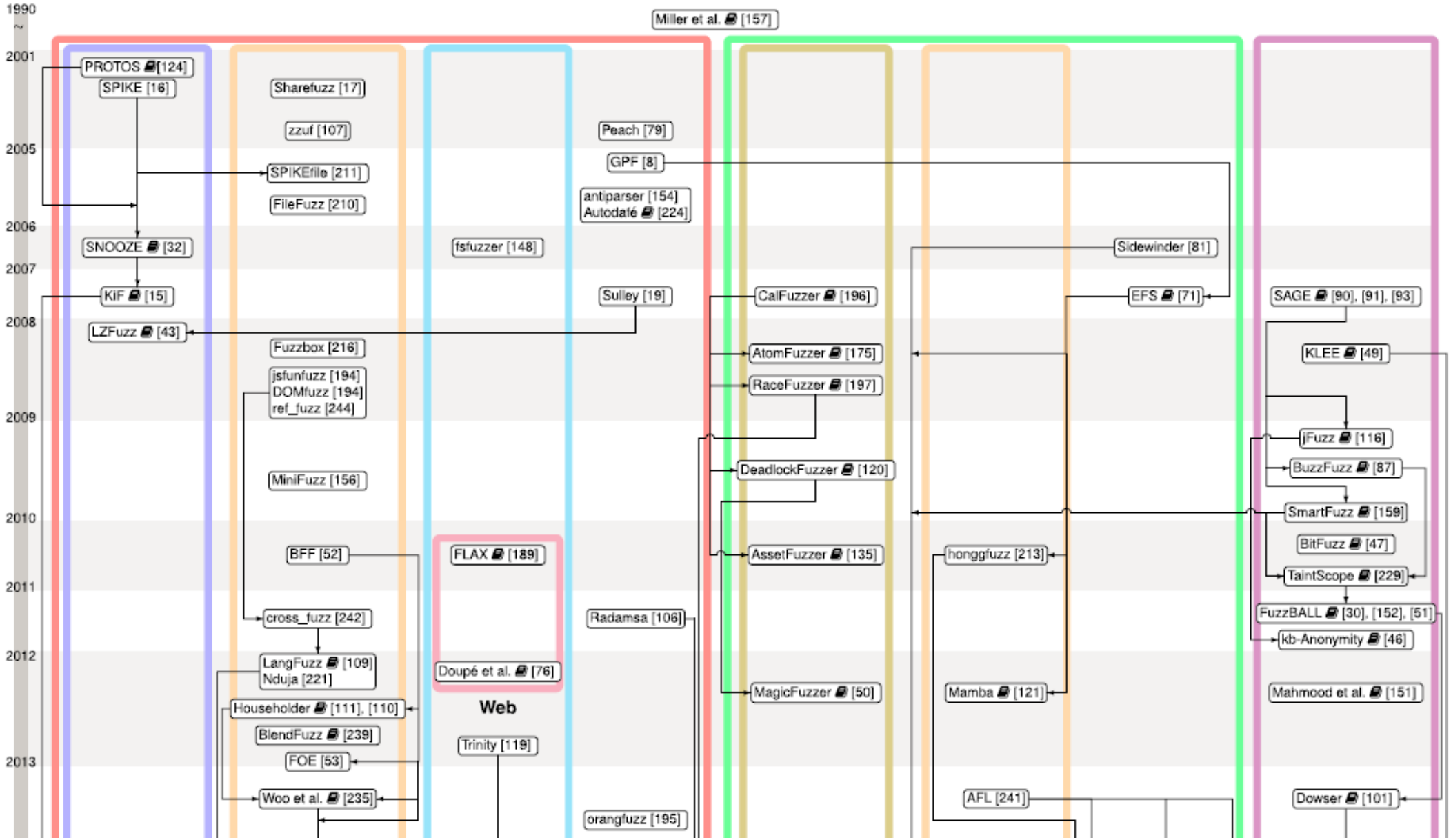
1.背景

– 发展历程

- Peach，2004：随机变异/生成的黑盒模糊测试
- Trinity，2005：基于知识的生成式黑盒模糊测试，针对OS kernel

- SAGE，2008：白盒模糊测试
- KLEE，2008：白盒模糊测试
- Sulley，2009：基于知识的生成式黑盒模糊测试，针对网络协议
- Csmith，2011：基于语法的生成式黑盒模糊测试，针对C编译器

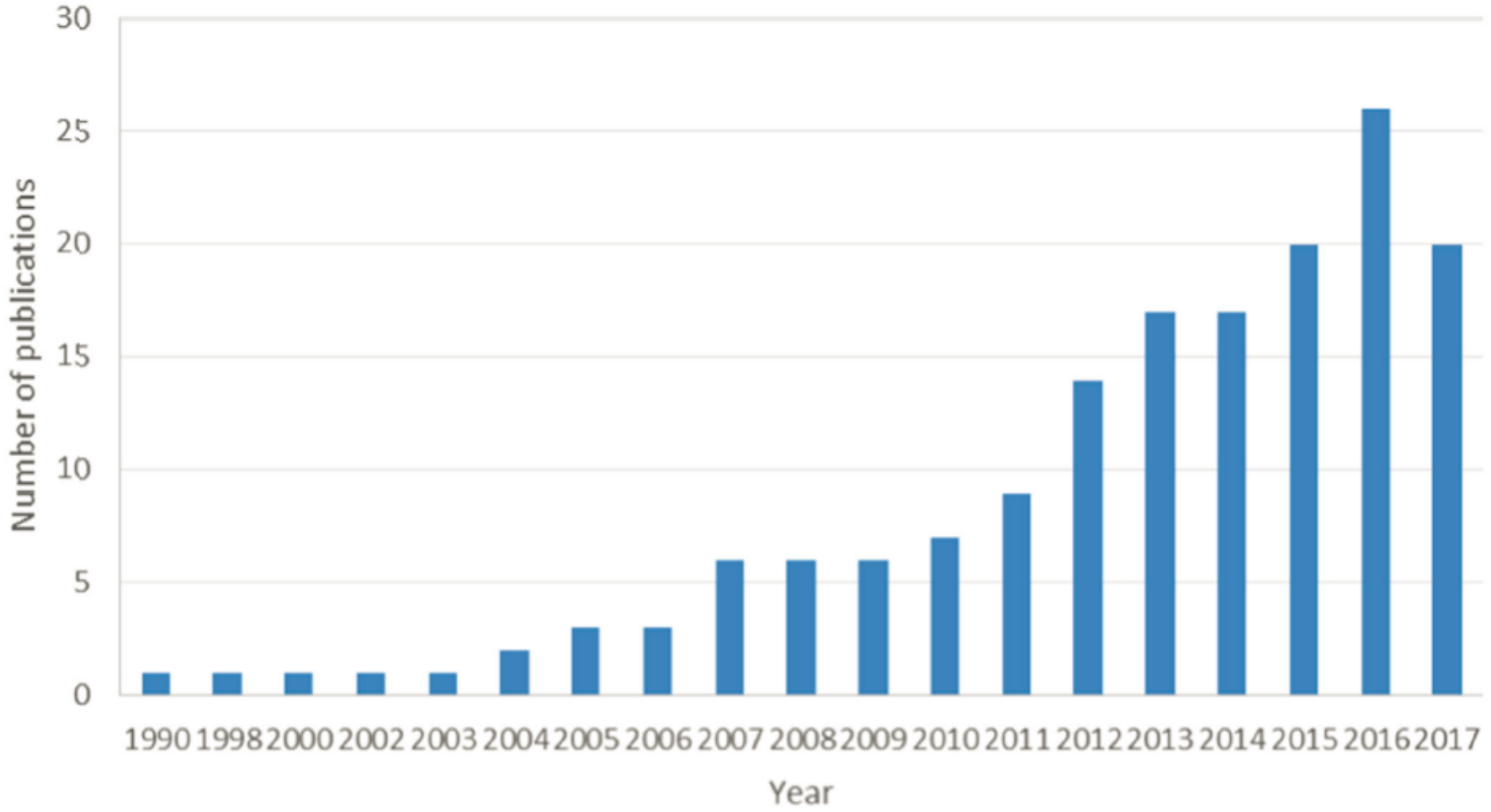
- AFL，2013：覆盖率导向的灰盒模糊测试
- ……



Manès, Valentin JM, et al. The art, science, and engineering of fuzzing: A survey. 2019

1.背景

– 论文发表



Liang et al, Fuzzing: State of the Art, 2018

2.技术原理

- 基本原理：
  - 模糊测试 （fuzz testing, fuzzing）是一种软件测试技术
  - 核心思想
    - 将自动或半自动生成的随机数据输入到一个程序中
    - 监视程序异常，如崩溃，断言（assertion）失败
    - 以发现可能的程序错误
    - 常常用于检测软件或计算机系统的安全漏洞

2.1 基本原理

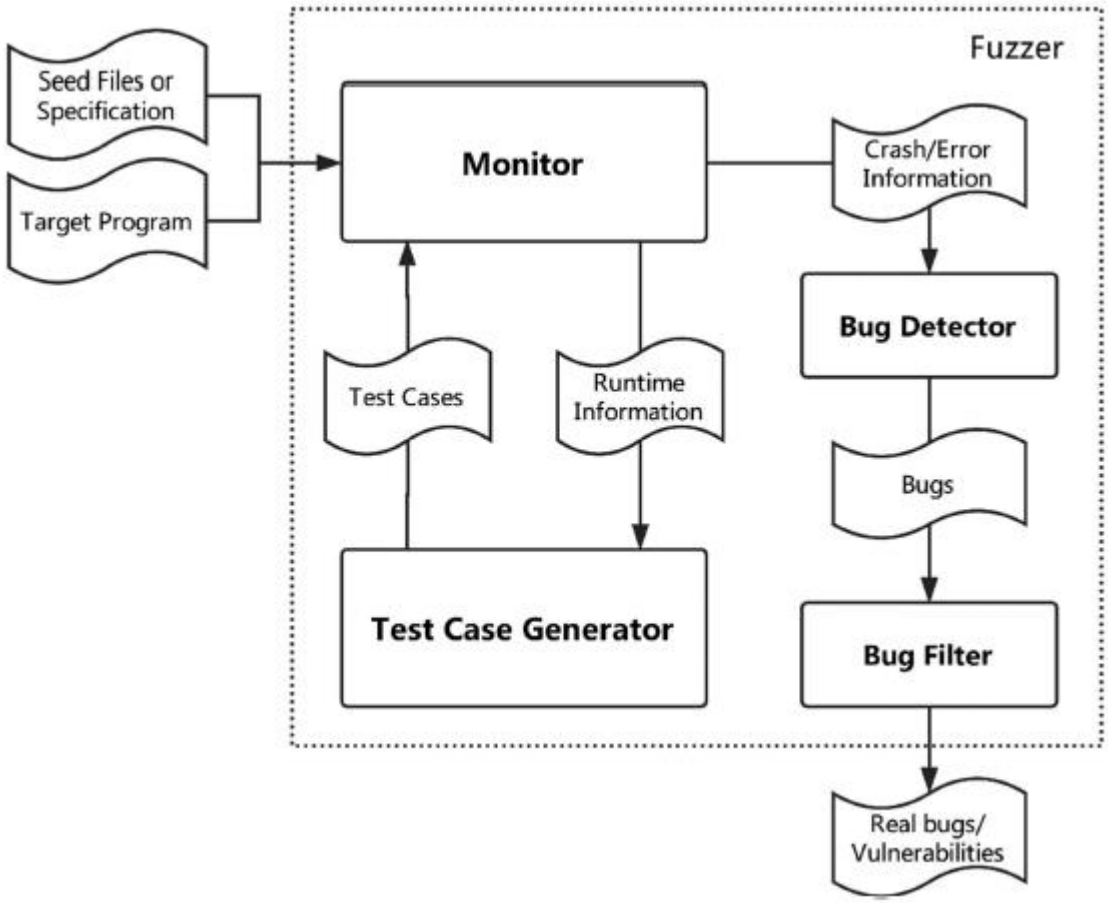
- 模糊测试的分类
  - 黑盒测试（Black-box）
    - 不关心程序内部逻辑，只观察程序行为（输入-输出对应关系）。将程序视为黑盒。
    - 代表工作：Peach 2004
  - 白盒测试
    - 通过分析程序内部逻辑，系统性地探索程序行为，通常要根据程序语义进行推理
    - 符号执行（Symbolic Execution）、动态符号执行（Concolic Testing）
    - 代表工作：KLEE 2008， SAGE 2008

2.1 基本原理

- 灰盒测试
  - 通过插桩等方式获取程序的部分内部信息，将其作为反馈来指导输入的生成
  - 不对程序的完整语义进行推理
  - 代表工作：AFL 2013， libfuzzer 2015

2.1 基本原理

- 模糊测试的基本过程：



Liang et al, Fuzzing: State of the Art, 2018

2.1 基本原理

- 一般的模糊测试主要包含如下组成部分：
  - 被测程序（PUT）
  - 输入生成
  - 监测机制
  - 调度机制
  - 漏洞输入分类

2.1 基本原理

- 被测程序（Program Under Test，PUT）
  - 模糊测试的被测程序通常有如下几类：
    - 文件处理程序
    - OS kernel
    - 网络协议
    - 编译器
    - 智能合约
    - 程序接口（API）
    - 其他
  - 不同类型的被测程序需要不同的模糊测试方法

2.1 基本原理

- 输入生成
  - 生成式(generation， model-based)
    - 通过目标程序输入的模型来生成输入
    - 1. 预定义的模型：
      - 用户给出目标程序合法输入的格式
      - 使用BNF文法或模板
    - 2. 推断模型：
      - 根据目标程序的行为推断合法输入的格式
      - 1). 推断目标程序输入的文法
      - 2). 使用神经网络生成输入

2.1 基本原理

- 变异式(mutation， model-free)
  - 通过对已有输入的局部修改（变异）来生成新的输入
  - 选取一些合法输入作为种子，不断进行变异
  - 常用变异策略：
    - 位翻转（bitflip）：翻转若干位的值
    - 算术操作（arithmetic）：对若干位组成的数字进行算术操作
    - 块级变异：插入、删除或替换若干连续位的内容
    - 基于字典的变异（dictionary）：预先给定一些有意义的值（如magic value，token等）称为字典。每次变异插入或将若干位替换为字典中的值。

2.1 基本原理

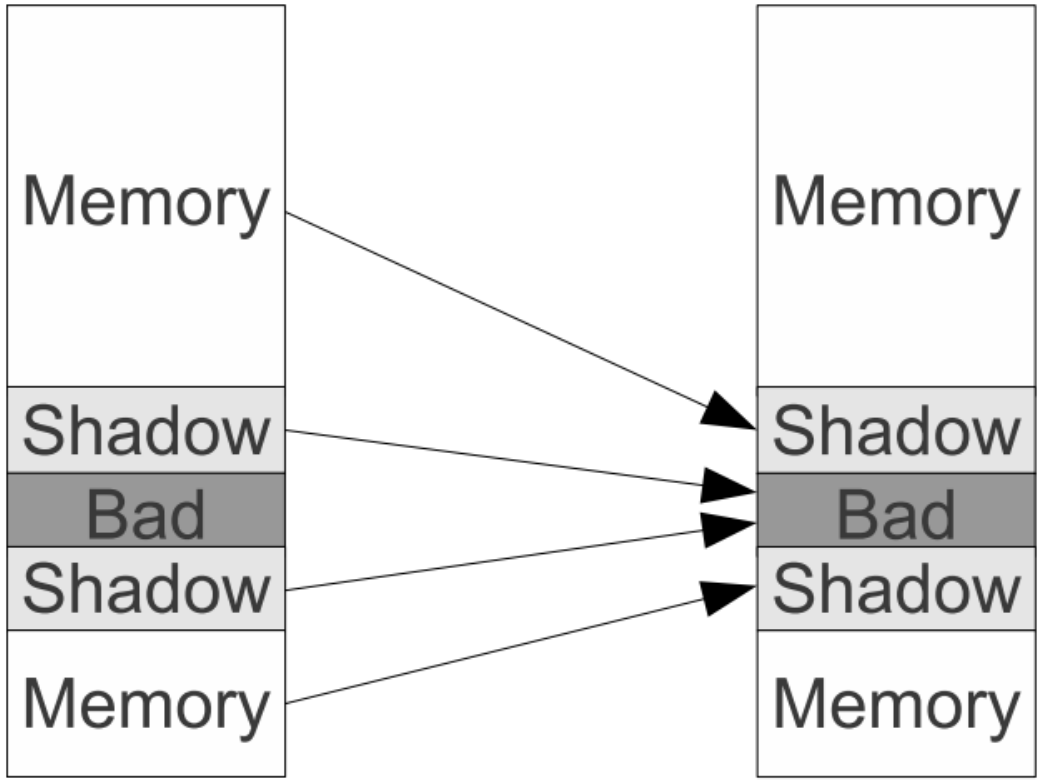
- 监测机制：
  - 用输入运行程序，根据程序行为判断何时出现bug
  - 又称为Bug Oracle

- 常用的Bug Oracle
  - 根据程序是否崩溃（如段错误）判断，简单但最常用
  - 差分测试（differential testing），比较相似程序的行为
  - 使用程序变换（插桩等）方式确定程序是否产生非预期行为，这类工具称为Sanitizer

2.1 基本原理

– Address Sanitizer（ASan）， 2012

- 通过插桩检测程序内存错误
- 使用影子内存标记内存位置的合法性
- 平均73%的减速



Serebryany, Konstantin, et al. AddressSanitizer: A fast address sanity checker. *2012*

2.1 基本原理

- Memory Sanitizer（MSan）， 2015
  - 通过插桩程序检测因C/C++未初始化内存产生的未定义行为
  - 使用影子内存
- Undefined Behavior Sanitizer（UBSan）， 2012
  - 通过编译时修改程序检测未定义行为
- Thread Sanitizer（TSan）， 2009
  - 通过编译时修改程序检测并发data race

2.1 基本原理

- 调度机制
  - 根据程序的运行时反馈，更新测试配置以更好地进行模糊测试
  - 黑盒模糊测试一般不进行配置更新
  - 灰盒模糊测试一般使用遗传算法进行配置更新

- 使用遗传算法进行配置更新
  - 维护种子队列，其中保存有希望或典型的输入
  - 定义适应度函数

2.1 基本原理

- 定义适应度函数：
  - 一般使用代码覆盖作为适应度
  - 常用：基本块覆盖、边覆盖
- 更新配置
  - 选取种子队列中元素进行变异
  - 确定一个种子生成几个新输入，每次生成变异几次（称为power schedule）
  - 若变异后的种子达到了更优的适应度，则将其加入种子队列

2.1 基本原理

- 漏洞输入分类
  - 多个触发漏洞输入可能只对应少数漏洞
  - 对输入进行分类，使每类对应一个漏洞，给开发者更好的反馈

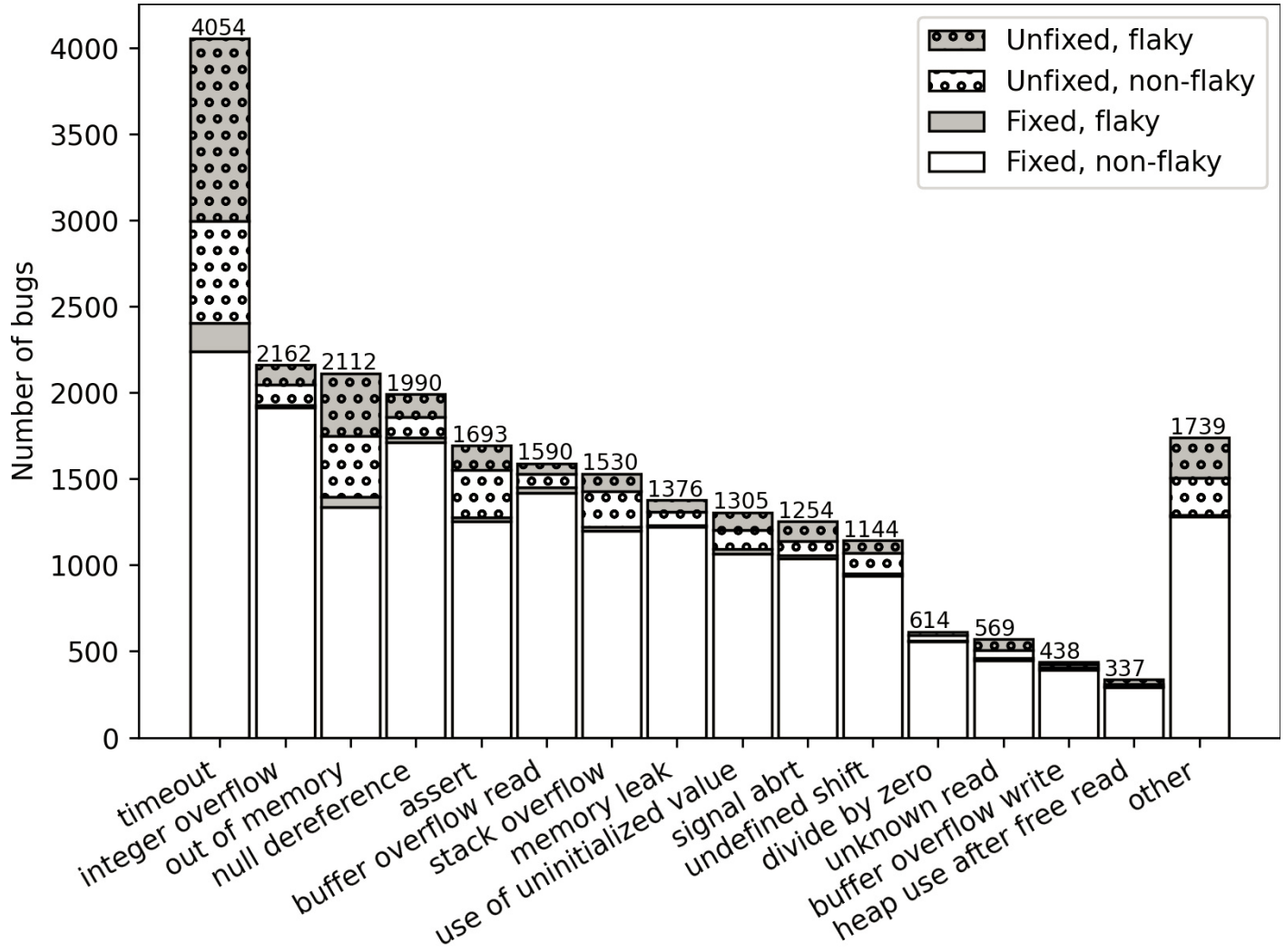
  - AFL（2013），使用执行路径进行漏洞输入分类
  - Semantic Crash Bucketing（2018），使用自动修复程序的方式进行漏洞输入分类

2.2 覆盖率导向的灰盒测试

- 覆盖率导向的灰盒模糊测试（Coverage Guided Fuzzing，CGF）
  - 近年来最为流行的模糊测试方法之一
  - 优势：
    - 相较于黑盒测试：通过覆盖率引导的方式减少盲目性
    - 相较于白盒测试：不用对程序的完整语义进行推理，有更高的效率
  - 广泛应用于真实程序，发现了大量漏洞
    - Google OSS-Fuzz项目

2.2 覆盖率导向的灰盒测试

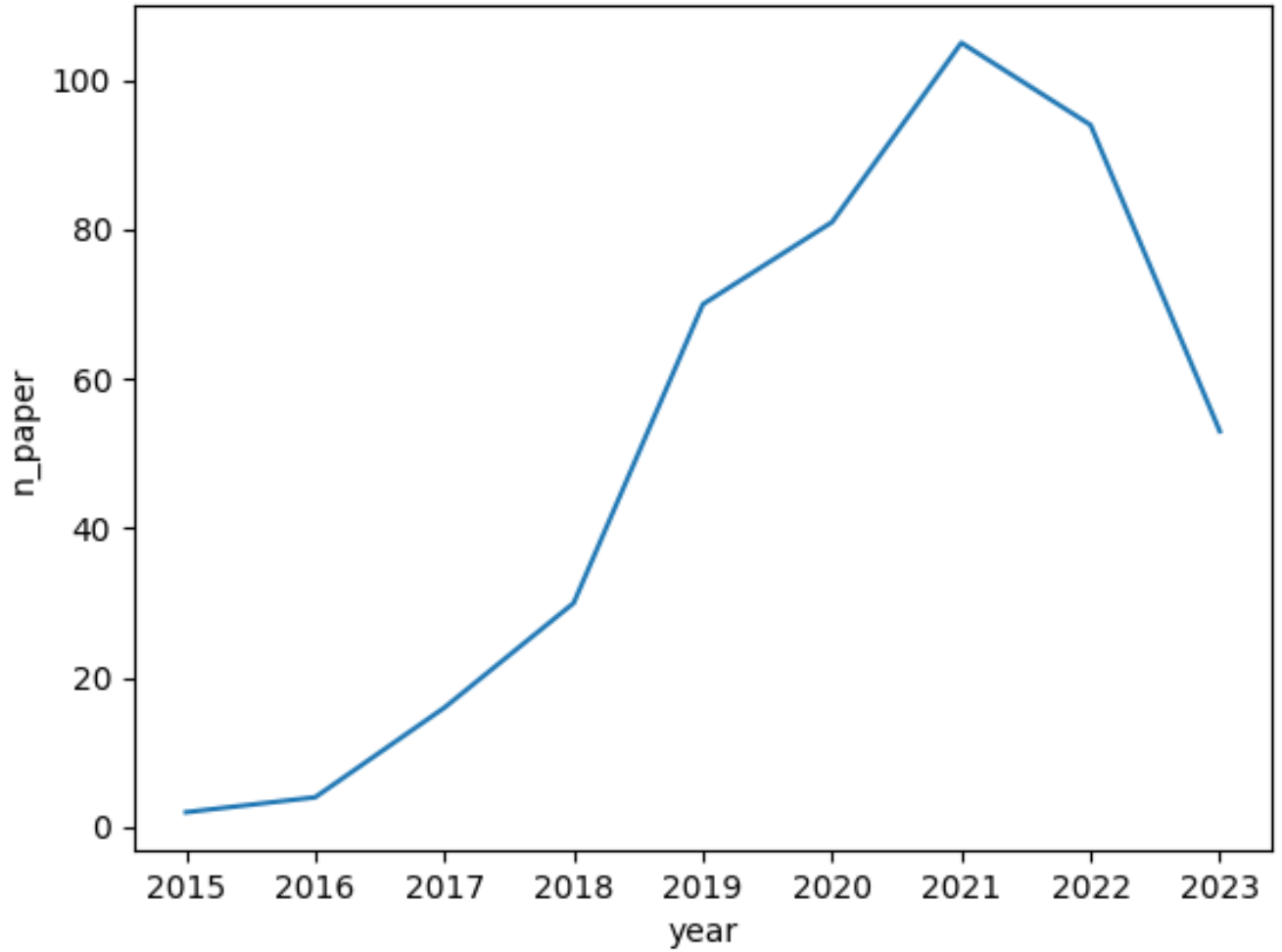
- OSS-Fuzz发现的漏洞类型统计：



Ding, Zhen Yu, and Claire Le Goues. An empirical study of oss-fuzz bugs. 2021

2.2 覆盖率导向的灰盒测试

– 近年Coverage Guided Fuzzing 顶会论文数量统计



数据来源：
https://github.com/wcventure/FuzzingPaper

2.2 覆盖率导向的灰盒测试

- 基本原理：
  - 基于遗传算法
  - 输入的覆盖作为适应度
  - 维护种子队列
  - 每次从种子队列中取一个元素进行变异
    - 用变异后的输入进行模糊测试
    - 根据适应度选择是否加入输入队列

2.2 覆盖率导向的灰盒测试

- 基本原理：

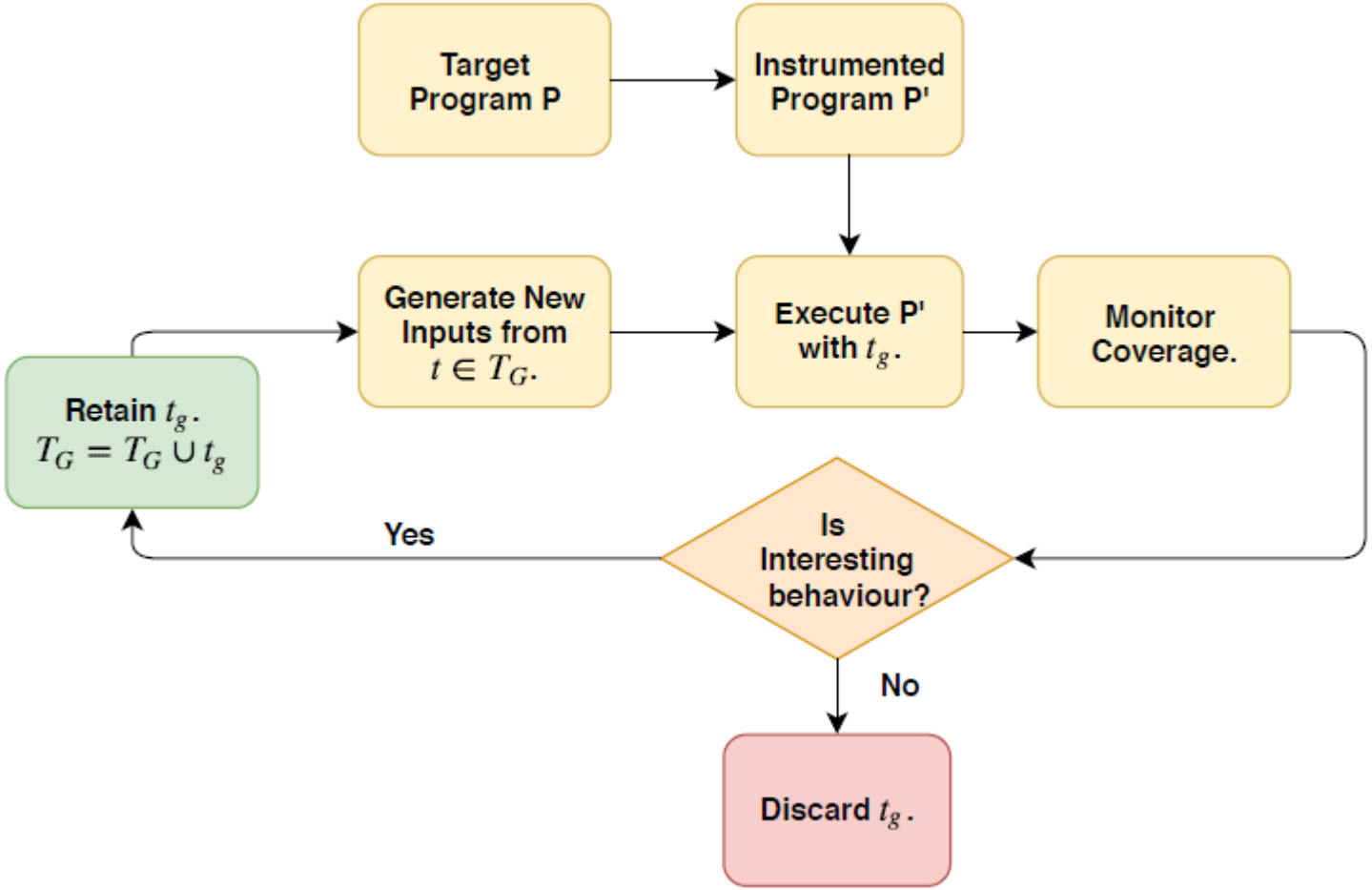
Algorithm 1 Coverage-guided fuzzing.

```
1:  procedure FUZZ(program p, set of seed inputs I_0)
2:      Inputs ← I_0
3:      TotalCoverage ← coverage of p on Inputs
4:      while within time budget do
5:          i ← pick from Inputs
6:          i' ← mutate i
7:          coverage, error ← execute p on i'
8:          if ∃ error then
9:              report error and faulty input i'
10:             optionally, exit
11:         else if coverage ⊈ TotalCoverage then
12:             add i' to Inputs
13:             TotalCoverage ← TotalCoverage ∪ coverage
```

Ding, Zhen Yu, and Claire Le Goues. An empirical study of oss-fuzz bugs. 2021

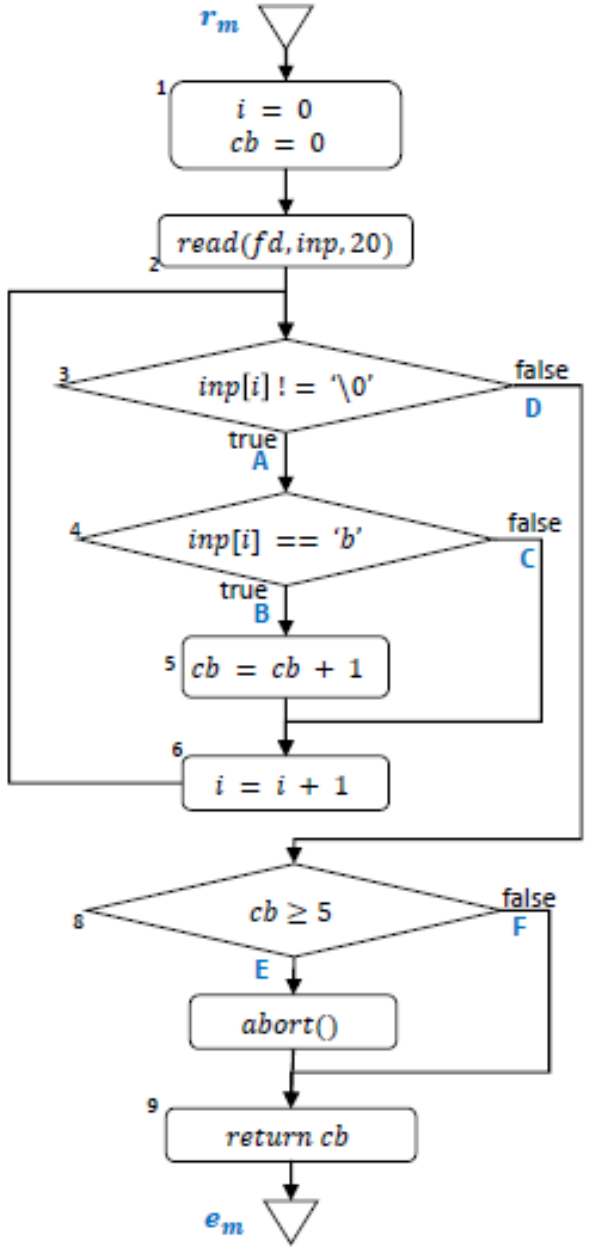
2.2 覆盖率导向的灰盒测试

- 基本原理：



Böhme, Marcel, et al. Coverage-based greybox fuzzing as markov chain. 2016

Grey-box fuzzing – Working example

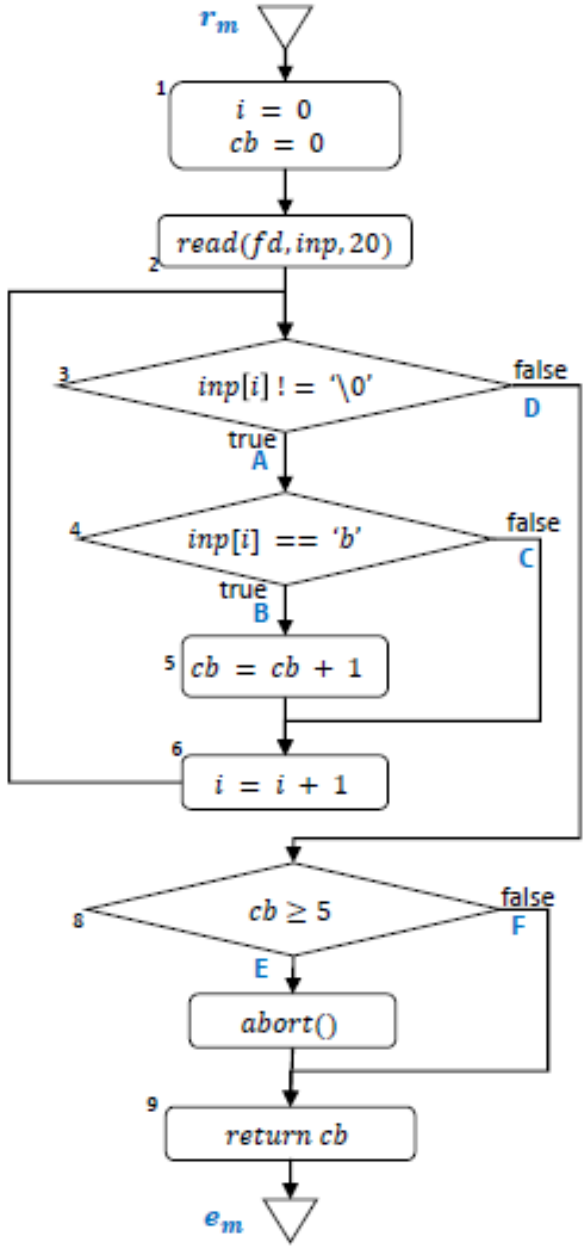


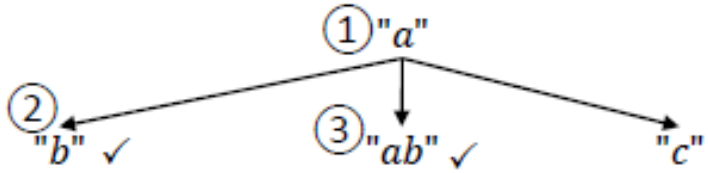
①"a"

Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1

来源：
https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example

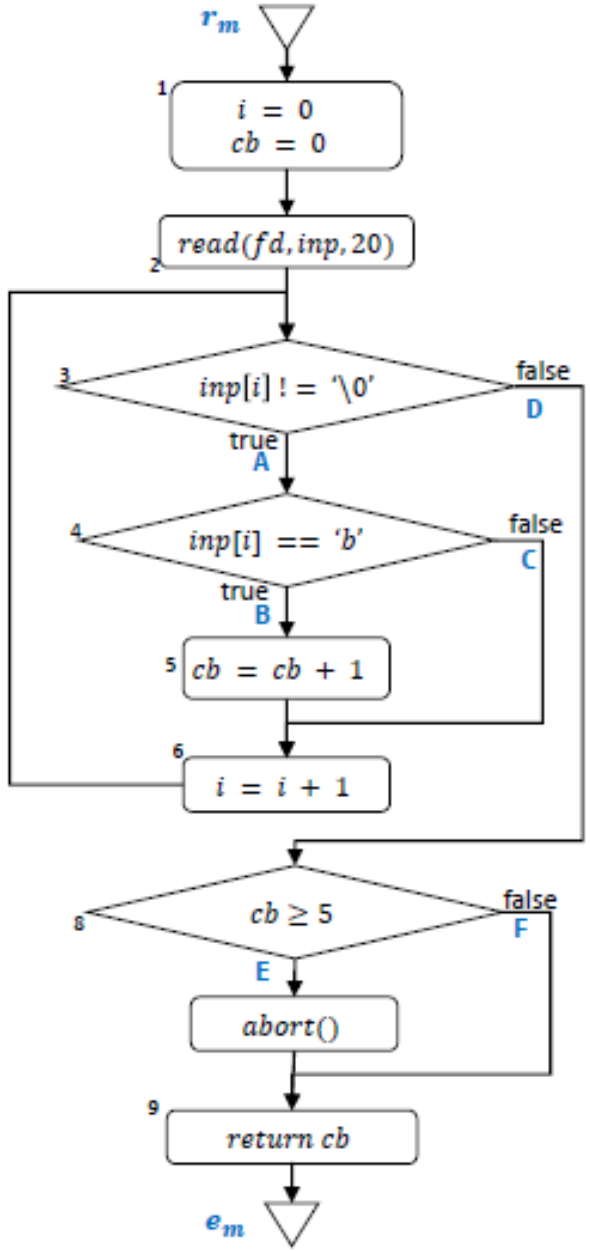


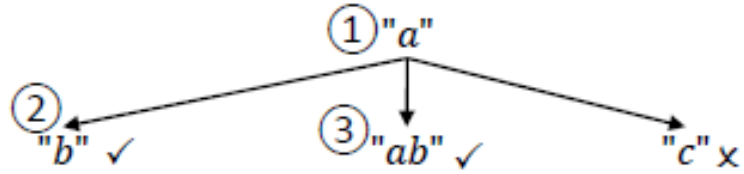


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

来源：
https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example

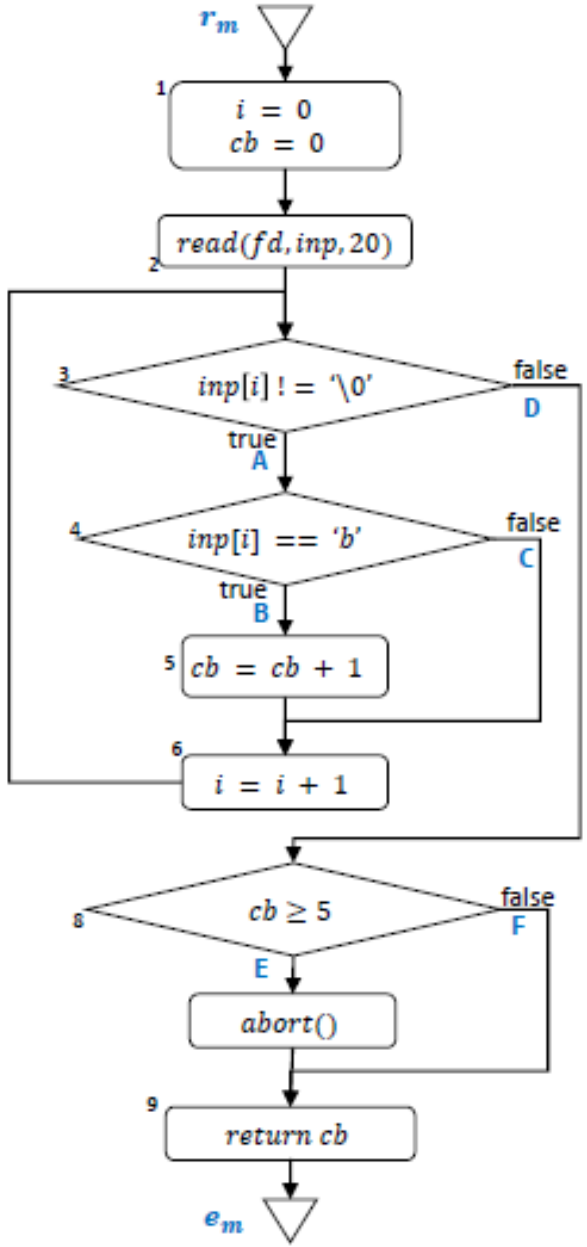


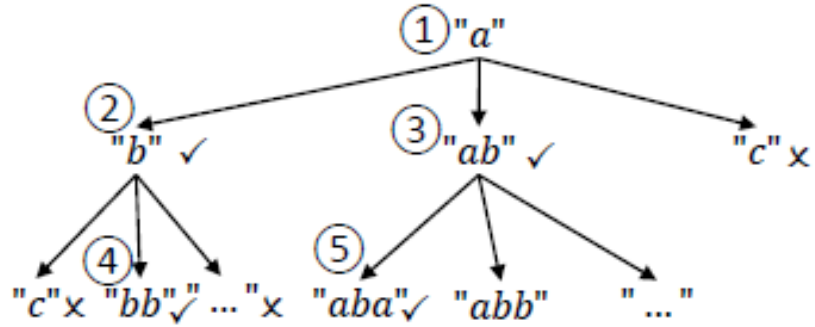


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
	"c"		1				1		1

来源：
https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

Grey-box fuzzing – Working example



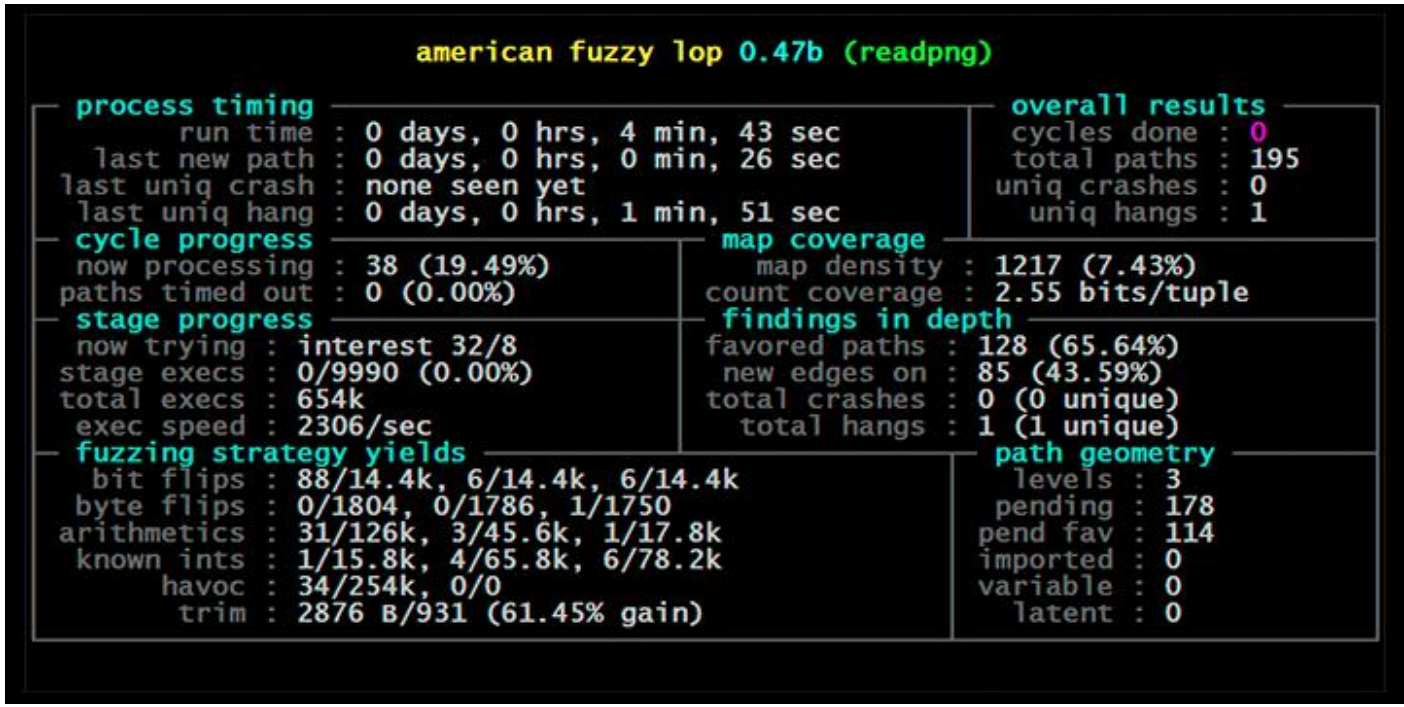


Id	input	AB	AC	BA	CA	BD	CD	DE	DF
1	"a"		1				1		1
2	"b"	1				1			1
3	"ab"	1	1		1	1			1
4	"bb"	2		1		1			1
5	"aba"	1	2	1	1		1		1
	"abb"	2	1	1	1	1			1

来源：
https://z14120902.github.io/teaching/ict_alg/fuzz_mc.pdf

2.2 覆盖率导向的灰盒测试

- 常用工具：
  - AFL
    - 通过插桩统计边覆盖率
    - 用共享内存保存覆盖情况
    - forkserver机制提升效率
    - 测试集裁剪
    - 支持binary-only fuzzing
    - 支持并行测试



2.2 覆盖率导向的灰盒测试

- AFL++
  - 在AFL基础上整合了大量新提出的技术，包括：
    - Power schedule 调度机制
    - MOPT 变异选择
    - cmplog 插桩
    - laf-intel 程序预处理
    - n-gram coverage

2.2 覆盖率导向的灰盒测试

- LibFuzzer
  - LLVM项目的一部分
  - in-process fuzzing
  - 支持cmplog插桩
  - 支持value-profile机制
  - entropy-based power schedule

2.2 覆盖率导向的灰盒测试

- 主要局限性
  - 更倾向于发现浅层漏洞
  - fuzz具有复杂输入格式的程序时存在覆盖率低的情况
  - 例如：
    - checksum、magicvalue
    - 需要加解密的输入

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

Stephens, Nick, et al. Driller: Augmenting fuzzing through selective symbolic execution. 2016

2.3 发展方向

- 模糊测试近年来几个发展方向：
  - 混合模糊测试
  - 领域特定模糊测试
  - 其他发展方向

2.3.1 混合模糊测试

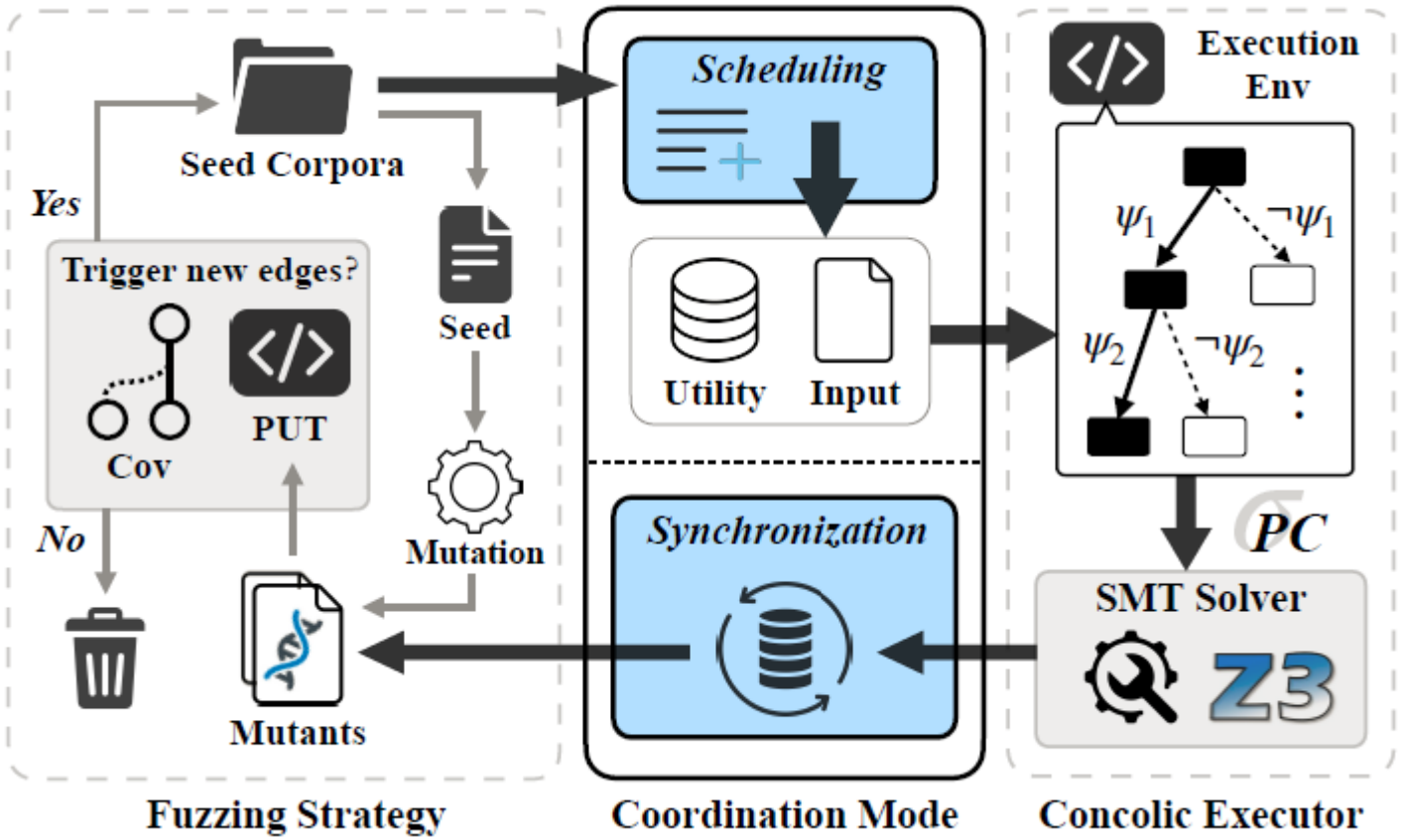
- 混合模糊测试（hybrid fuzzing）
  - 结合重量级程序分析技术（符号执行、污点分析等）和覆盖率导向的模糊测试
  - 解决模糊测试覆盖率低、倾向于发现浅层漏洞的问题
  - Motivating Example：

```
1 int main(void)
2 {
3     int x;
4     read(0, &x, sizeof(x));
5
6     if (x == 0x0123ABCD)
7         vulnerable();
8 }
```

Stephens, Nick, et al. Driller: Augmenting fuzzing through selective symbolic execution. 2016

2.3.1 混合模糊测试

- 基本过程：



Jiang, Ling, et al. Evaluating and improving hybrid fuzzing. 2023

2.3.1 混合模糊测试

- 代表性工作：
  - Driller 2016：
    - 结合符号执行和CGF
    - CGF遇到不能覆盖的情况时，调用符号执行工具进行求解
    - 将求解得到的结果加入种子队列

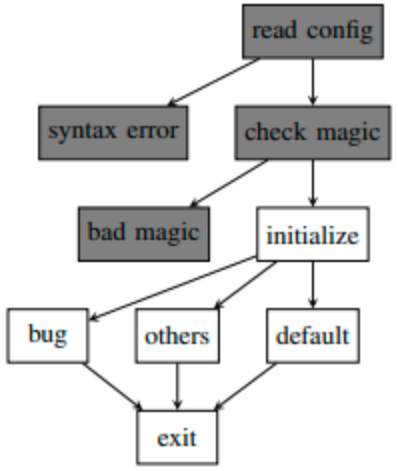


Fig. 1. The nodes initially found by the fuzzer.

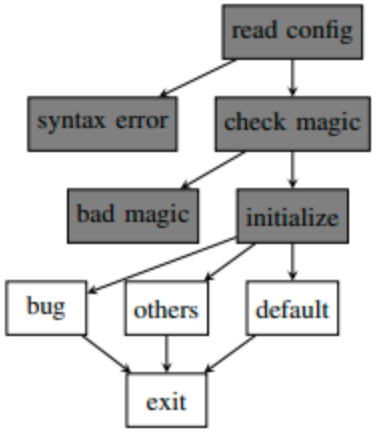


Fig. 2. The nodes found by the first invocation of concolic execution.

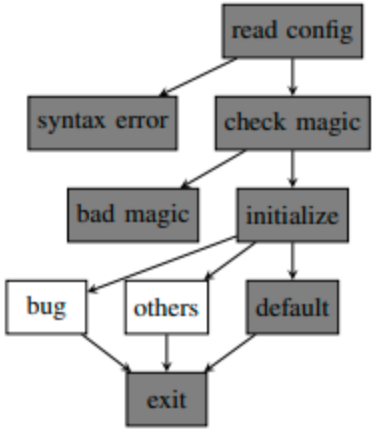


Fig. 3. The nodes found by the fuzzer, supplemented with the result of the first Driller run.

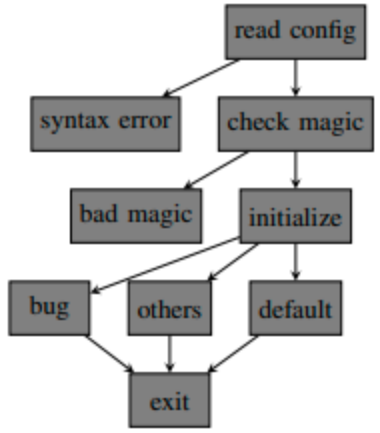
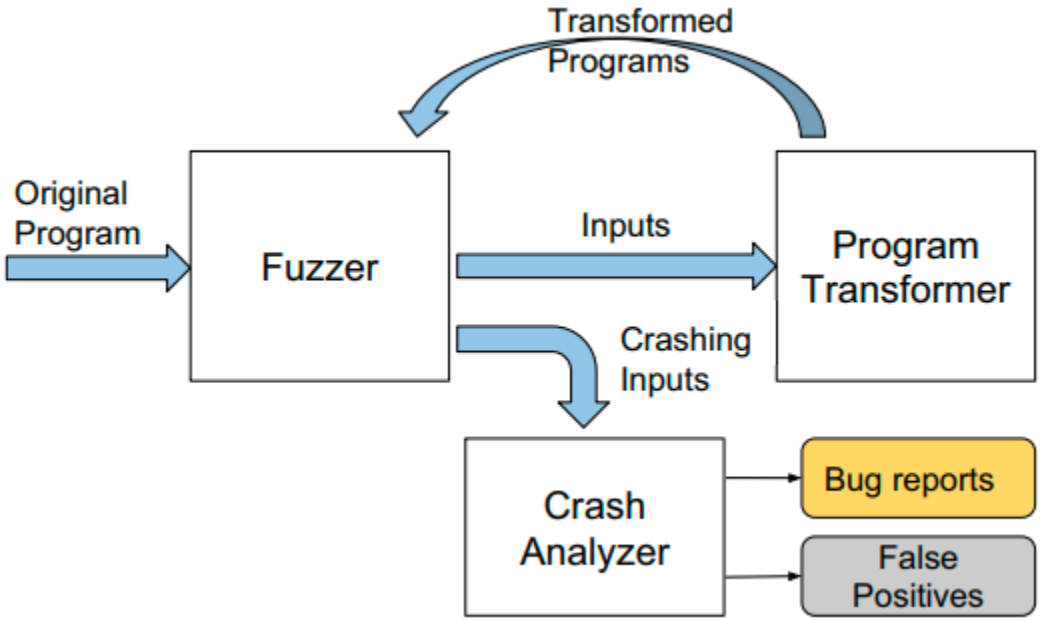


Fig. 4. The nodes found by the second invocation of concolic execution.

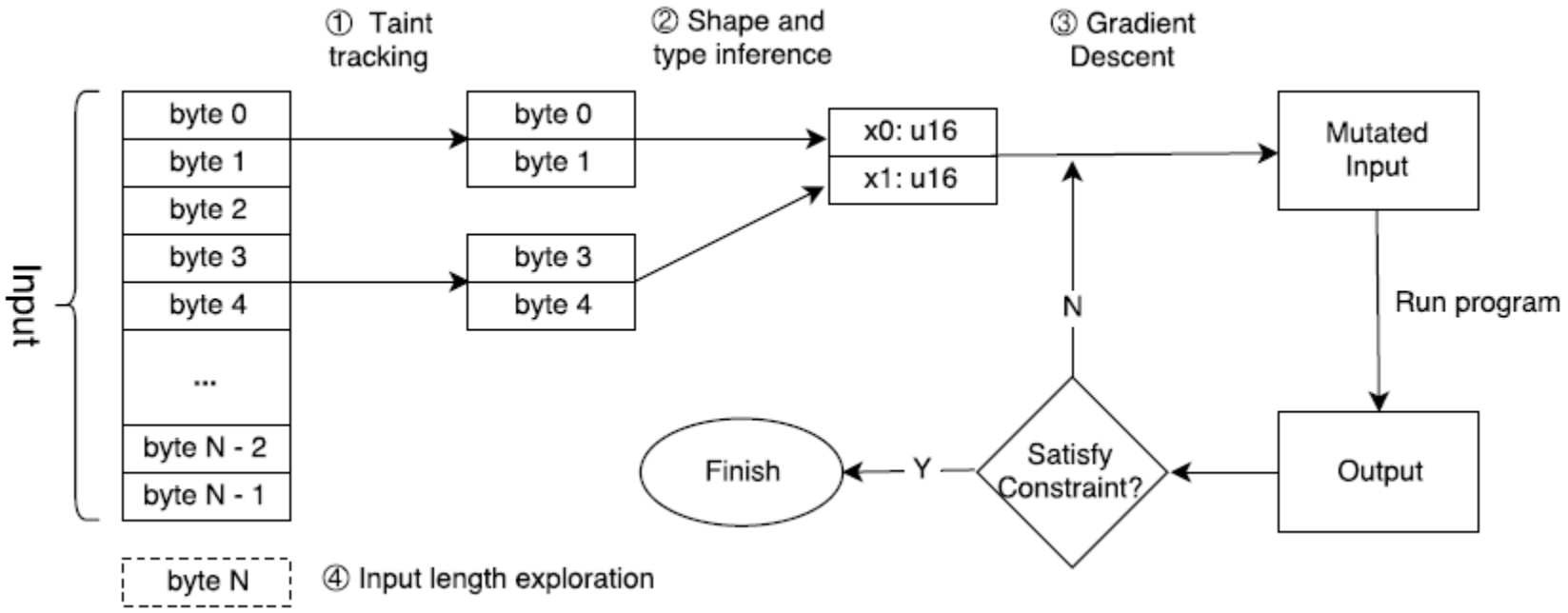
2.3.1 混合模糊测试

– TFuzz 2018：
  - 找到CGF不能通过的条件
  - 检测其中的NCC（Non-Critical Check）
  - 进行程序变换，去掉NCC后fuzz
  - 用符号执行确认漏洞输入



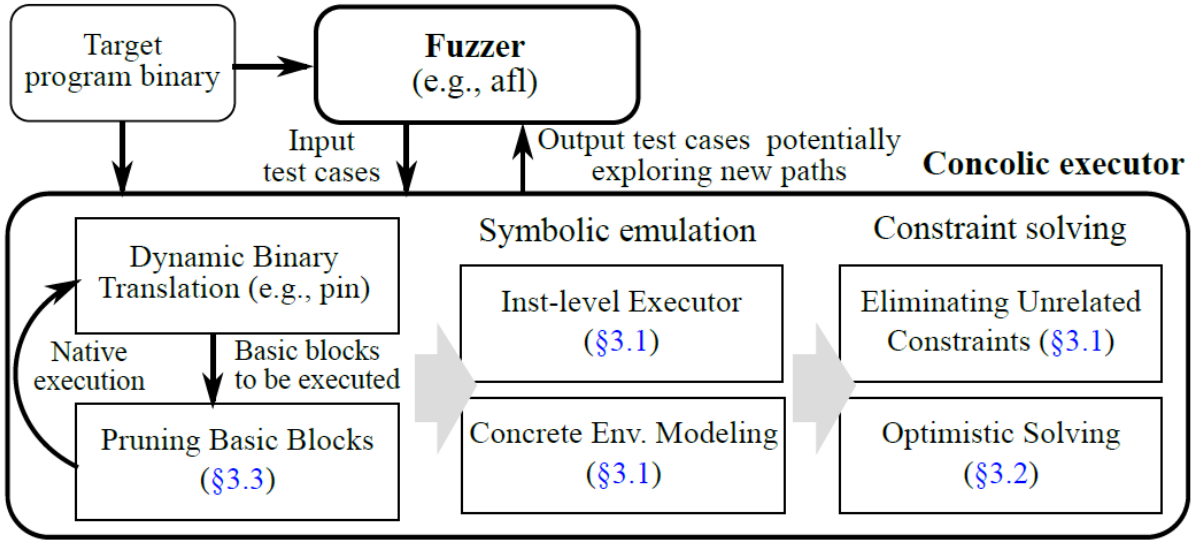
2.3.1 混合模糊测试

– Angora 2018 & Matryoshka 2019
  - 找到CGF不能通过的条件
  - 使用污点分析确定该条件对应的输入bit
  - 使用启发式搜索（梯度下降）方法求解
  - 处理嵌套条件



2.3.1 混合模糊测试

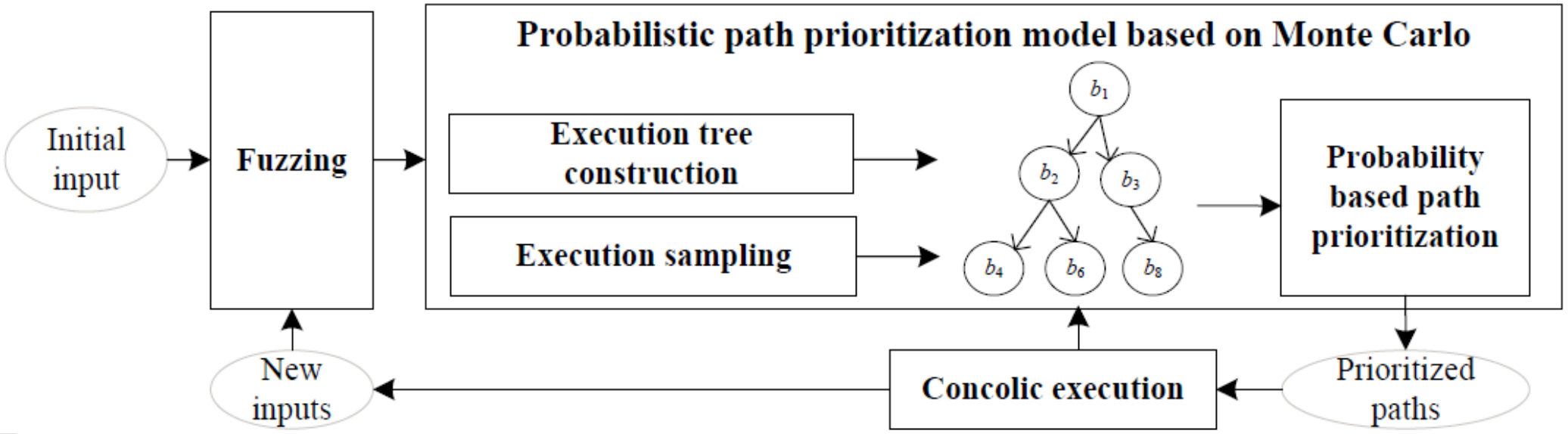
– QSYM 2018 & Symcc 2020
  - 优化混合模糊测试中的符号执行
  - 符号执行非常慢：
    – 符号模拟层使用IR，带来额外开销
    – 约束求解层要求严格解，常常超时
  - 改进：
    – 符号模拟层直接模拟CPU指令，减去IR的开销
    – 乐观求解：约束求解时去掉部分约束使求解更容易
  - 目前混合模糊测试的SOTA之一



2.3.1 混合模糊测试

– DigFuzz 2020

- 评估每条路径的“难度”
- 将路径按难度排序，困难的路径优先交给符号执行探索
- 建立基于蒙特卡洛方法的概率模型，计算每条路径的概率，概率越小则难度越高



2.3.1 混合模糊测试

- Pangolin 2020
  - 增量式约束求解，充分利用之前约束求解结果提供的信息
  - 将路径约束的解集建模为多面体（polyhedra）
  - 使用SMT-opt 求解路径约束对应的多面体模型
  - 利用多面体模型：
    - Fuzz该路径时从多面体中平均采样
    - 求解该路径上的约束时，用多面体模型加强约束，让求解更容易

2.3.2协议模糊测试

- 协议模糊测试的背景：
  - 在互联网飞速发展的今天，许多本地应用都是在B/S模式下转化为网络服务：服务器部署在网络上，用户使用客户端应用程序通过网络协议与服务器通信。
  - 协议中的安全问题可能会产生比本地应用更严重的损害，如拒绝服务、信息泄露等，如果不及时发现协议程序中的漏洞很有可能造成巨大的损失。

因此网络协议的安全性测试成为了一个非常重要的问题，协议模糊测试技术也随之逐渐发展。

2.3.2协议模糊测试

- 协议模糊测试面临的3个挑战：
  - 网络协议程序可能会定义自己的特有的通信协议，这些协议的标准/规范往往不公开
  - 协议模糊测试建立通信的过程和发送测试用例的过程会带来额外的开销，从而导致协议模糊测试的吞吐量往往低于本地文件模糊测试的吞吐量
  - 网络协议程序网络具有很大的状态空间，需要精心构造特定顺序的消息序列才能到达某一状态

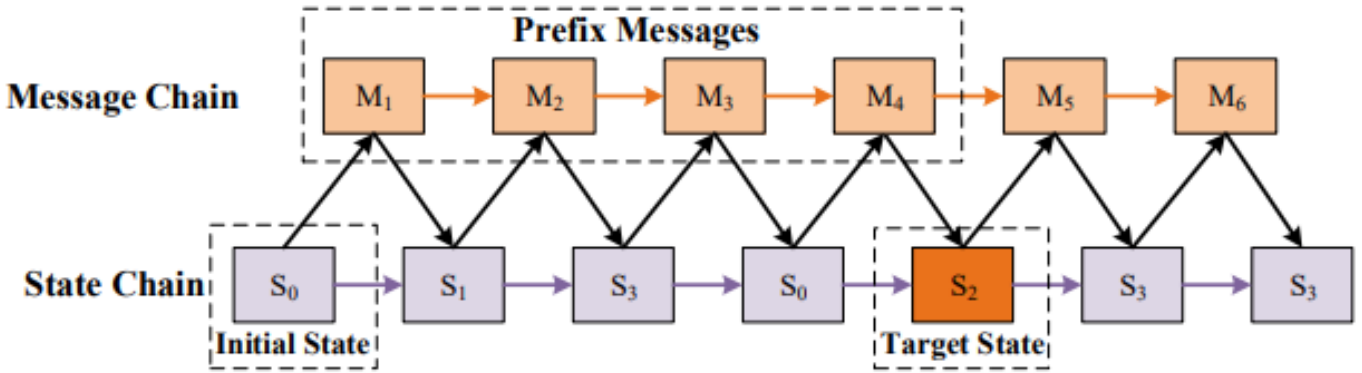
2.3.2协议模糊测试

- 协议模糊测试研究现状：
  - 早期的网络协议模糊器主要以黑盒的方式运行，且常常是无状态的，缺乏SUT的内部反馈，因此漏洞发现效率较低。随着灰盒模糊测试工具AFL的流行，研究者开始将AFL中基于程序反馈与遗传算法的思想引入到协议模糊测试中，有效提高了协议模糊测试的性能。

  - 2020年，Pham等人在AFL的基础上提出了首个有状态灰盒协议模糊测试工具AFLNET。AFLNET使用服务器的响应码来识别服务器的状态，并使用状态反馈结和覆盖路信息来指导模糊测试过程。随后针对AFLNET的缺点又出现了许多研究，包括StateAFL，SnapFuzz, SNPSFuzzer, NSFuzz等等。

2.3.2协议模糊测试

- AFLNET的基本原理：
  - AFLNET使用服务器的响应报文中的响应码来识别服务器的状态，为了针对特定的状态fuzz，AFLNET在模糊测试的过程中动态地构建信息链和状态链，通过多种状态选择算法选择合适的状态，并从种子库中搜寻并选择可以到达该状态的消息序列，并将该序列拆分为前缀M1，中缀M2和后缀M3，针对中缀信息进行变异，最终将其组成新的报文序列，然后将报文序列发往待测程序，如果该序列能够抵达新的状态或者实现更高的覆盖率，则将该序列视为感兴趣的，并存入种子库。



2.3.2协议模糊测试

- AFLNET的缺点：
  - AFLNET使用粗粒度的响应报文中的**响应码**作为网络应用程序的状态，但其很多时候不能反映协议的真实状态，从而造成状态冗余，影响模糊测试的性能
  - 为了到达某一状态，AFLNET需要发送较长的**前缀消息**，由此带来的额外开销使模糊测试的吞吐量极低
  - AFLNET与SUT之间缺少**同步机制**，导致模糊器无法得知SUT是否已经处理完发送的消息，因此模糊器需要设置一定的时间延迟等待SUT的响应报文
  - AFLNET更倾向于**更快到达目标状态**的消息序列，而这样的消息序列通常较短，很难覆盖更深层次的状态

2.3.2协议模糊测试

- 针对AFLNET的改进：
  - StateAFL：通过插桩的方式转储网络协议程序长生命周期的变量计算其散列值作为状态
  - SnapFuzz：将AFLNet中的异步网络通信转为基于UNIX套接字的通信，将所有文件操作重定向到内存中的文件系统，提高了协议模糊测试的吞吐量
  - SNPSFuzzer：使用快照机制，存储某一特定状态下的网络协议程序，从而避免了发送大量重复的前缀信息，提高了协议模糊测试的吞吐量
  - NSFuzz：利用网络应用程序的网络事件循环入口作为与模糊器之间的同步点，避免模糊器的等待时间

2.3.3其他

- 模糊测试的更多发展方向：
  - 基于AI模型的测试用例生成与变异

    从获取到的数据训练AI模型，使用训练好的模型生成新的测试用例
  - 基于静态与动态分析指导测试用例的生成

    基于静态、动态分析手段分析程序接受什么样的输入再去指导测试用例的生成
  - 针对特定目标程序的模糊测试

    针对特定类型的目标程序模糊测试，如内核程序、驱动程序、数据库程序、物联网程序等
  - 针对种子选择算法策略的优化

    根据不同的模糊需求设计更高效的种子选择算法从而优化模糊测试的效率

2.4实例

- AFL的实现细节
- 使用AFL进行模糊测试的例子

2.4.1AFL的实现细节

- AFL的实现细节-插桩
  - 使用AFL，首先要通过afl-gcc/afl-clang等工具对目标进行编译，在这个过程中会对目标进行插桩，具体插入代码的部分如下：

```
while (fgets(line, MAX_LINE, inf)) {

  /* In some cases, we want to defer writing the instrumentation trampoline
     until after all the labels, macros, comments, etc. If we're in this
     mode, and if the line starts with a tab followed by a character, dump
     the trampoline now. */

  if (!pass_thru && !skip_intel && !skip_app && !skip_csect && instr_ok &&
      instrument_next && line[0] == '\t' && isalpha(line[1])) {

    fprintf(outf, use_64bit ? trampoline_fmt_64 : trampoline_fmt_32,
            R(MAP_SIZE));

    instrument_next = 0;
    ins_lines++;
```

  - 这里通过fprintf()将格式化字符串添加到汇编文件的相应位置

2.4.1AFL的实现细节

- AFL的实现细节-插桩

其中**trampoline_fmt_32**如右图所示：

```
static const u8* trampoline_fmt_32 =

  "\n"
  "/* --- AFL TRAMPOLINE (32-BIT) --- */\n"
  "\n"
  ".align 4\n"
  "\n"
  "leal -16(%%esp), %%esp\n"
  "movl %%edi,  0(%%esp)\n"
  "movl %%edx,  4(%%esp)\n"
  "movl %%ecx,  8(%%esp)\n"
  "movl %%eax, 12(%%esp)\n"
  "movl $0x%08x, %%ecx\n"
  "call __afl_maybe_log\n"
  "movl 12(%%esp), %%eax\n"
  "movl  8(%%esp), %%ecx\n"
  "movl  4(%%esp), %%edx\n"
  "movl  0(%%esp), %%edi\n"
  "leal 16(%%esp), %%esp\n"
  "\n"
  "/* --- END --- */\n"
```

这段汇编代码，主要的操作是：

- 保存edi等寄存器
- 将ecx的值设置为fprintf()所要打印的变量内容
- 调用方法__afl_maybe_log()
- 恢复寄存器

其中movl $0x%08x, %%ecx\n将R(MAP_SIZE)赋值给ecx，R(MAP_SIZE)为0到MAP_SIZE之间的一个随机数

因此，在处理到某个分支需要插入桩代码时会生成一个随机数，保存在ecx中，用于标识这个代码块的key。

2.4.1AFL的实现细节

- AFL的实现细节-覆盖率的获取
  - 在__afl_maybe_log()检查共享内存映射后，执行右图代码
  - 变量__afl_prev_loc保存的是前一次跳转的”位置”，
  - 其值与ecx做异或后，保存在edi中
  - ecx的值右移1位后，保存在了变量__afl_prev_loc中

具体来说AFL为每个代码块生成一个随机数作为其“位置”的记录，随后，对分支处的”源位置“和”目标位置“进行异或，并将异或的结果作为该分支的key，并通过一个哈希表保存每个分支的执行次数。

```
  "__afl_maybe_log:\n"
  "\n"
  "  lahf\n"
  "  seto %al\n"
  "\n"
  "  /* Check if SHM region is already mapped. */\n"
  "\n"
  "  movl  __afl_area_ptr, %edx\n"
  "  testl %edx, %edx\n"
  "  je    __afl_setup\n"
  "\n"
  "__afl_store:\n"
  "\n"
  "  /* Calculate and store hit for the code location spe
  "     is a double-XOR way of doing this without taintir
  "     and we use it on 64-bit systems; but it's slower
  "\n"
#ifndef COVERAGE_ONLY
  "  movl __afl_prev_loc, %edi\n"
  "  xorl %ecx, %edi\n"
  "  shrl $1, %ecx\n"
  "  movl %ecx, __afl_prev_loc\n"
#else
  "  movl %ecx, %edi\n"
#endif /* ^!COVERAGE_ONLY */
  "\n"
#ifdef SKIP_COUNTS
  "  orb  $1, (%edx, %edi, 1)\n"
#else
  "  incb (%edx, %edi, 1)\n"
#endif /* ^SKIP_COUNTS */
  "\n"
```

2.4.1AFL的实现细节

- AFL的实现细节-fork server

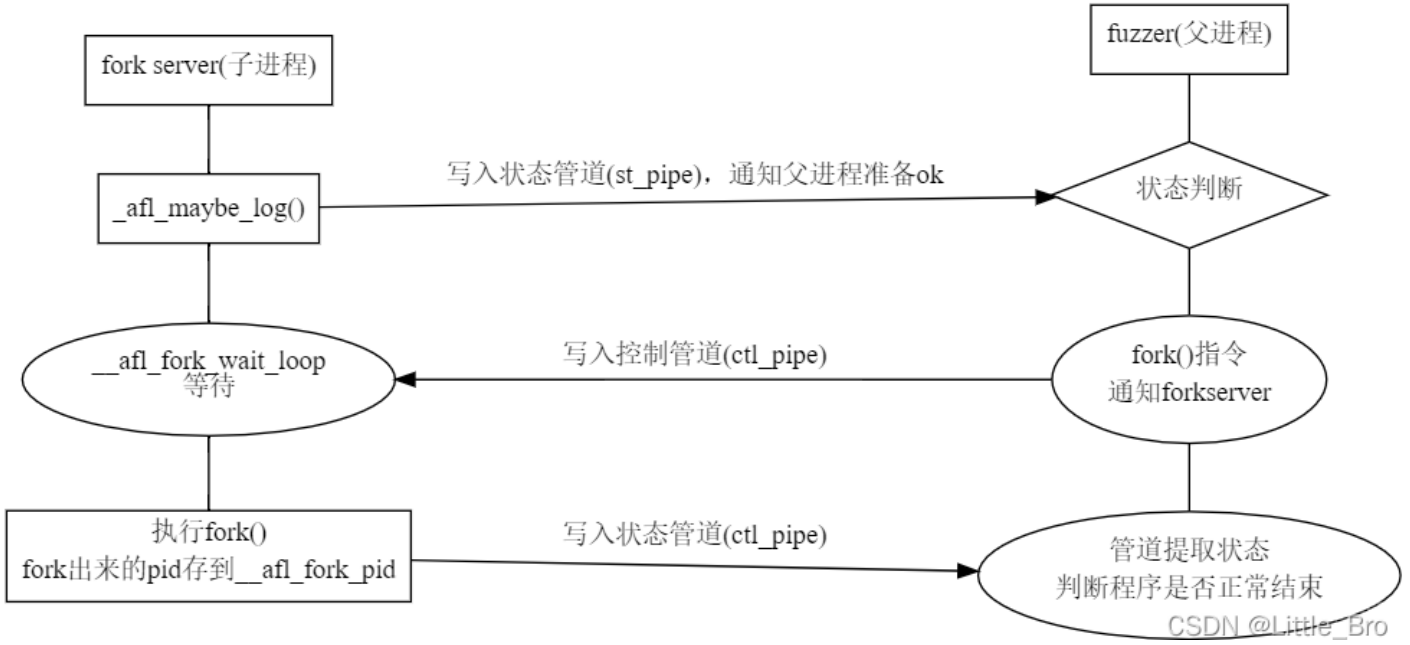
目标程序编译插桩之后afl通过调用afl-fuzz开始fuzzing。为了更高效地fork和执行target，AFL实现了一套fork server机制，其基本流程如左图所示：

- fuzzer执行fork()得到父进程和子进程，父进程为fuzzer，子进程为target进程，即fork server
- 父子进程之间通过2个管道进行通信，分别用于传递状态和命令，代码如右图所示
- 子进程会进行一系列设置，包括将两个管道分配到预先指定的fd，并最终执行target，代码如右图所示
- 对于父进程会读取状态管道的信息，如果一切正常，则说明fork server创建完成。


```
if (!forksrv_pid) {
    ...
        if (dup2(ctl_pipe[0], FORKSRV_FD) < 0) PFATAL("dup2() failed");
    if (dup2(st_pipe[1], FORKSRV_FD + 1) < 0) PFATAL("dup2() failed");
    ...
        execv(target_path, argv);
}
```

2.4.1AFL的实现细节

- AFL的实现细节-fork server
  - Fork server 与 fuzzer之间的通信具体可以如下图所示：



2.4.1AFL的实现细节

- 使用AFL进行模糊测试可以简单分为3步骤：
  - 准备语料库
    - 自己手动编写
    - 从网络上找已有的样本
  - 编译测试目标
    - export CC=afl-clang
    - export CXX=afl-clang++
  - 对目标进行fuzz
    - afl-fuzz -i $HOME/testcases/ -o $HOME/out/ – target @@

2.4.2使用AFL进行模糊测试

- 复现CVE-2019-13288
  - CVE-2019-13288：Xpdf是Foo实验室的一款开源的PDF阅读器。该产品支持解码LZW压缩格式的文件以及阅读加密的PDF文件。Xpdf 4.01.01版本中的Parser.cc文件的‘Parser::getObj()’函数存在安全漏洞。攻击者可借助特制的文件利用该漏洞造成拒绝服务。

  - 环境搭建：

```
sudo apt install build-essential
wget https://dl.xpdfreader.com/old/xpdf-3.02.tar.gz
tar -xvzf xpdf-3.02.tar.gz
mkdir pdf_examples && cd pdf_examples
wget https://github.com/mozilla/pdf.js-sample-files/raw/master/helloworld.pdf
wget http://www.africau.edu/images/default/sample.pdf
wget https://www.melbpc.org.au/wp-content/uploads/2017/10/small-example-pdf-file.pdf
```

2.4.2使用AFL进行模糊测试

- 编译插桩：

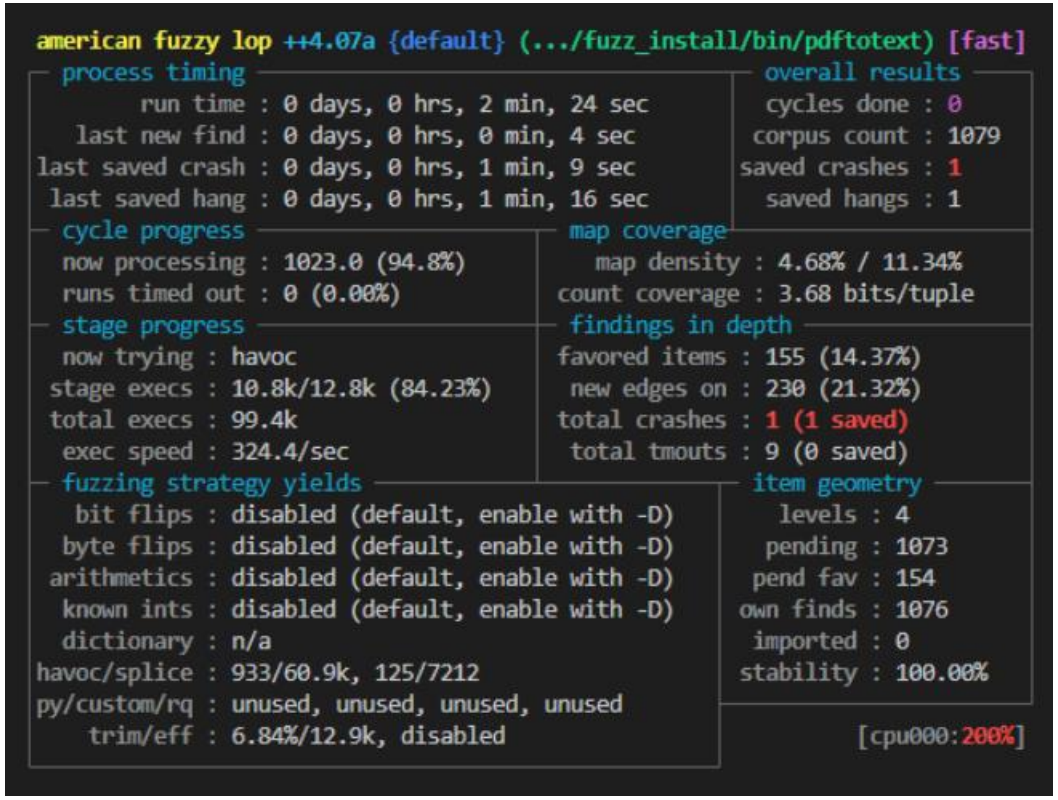
```
CC=afl-clang-fast CXX=afl-clang-fast++ ./configure
make -j4
make install
```

- Fuzz：

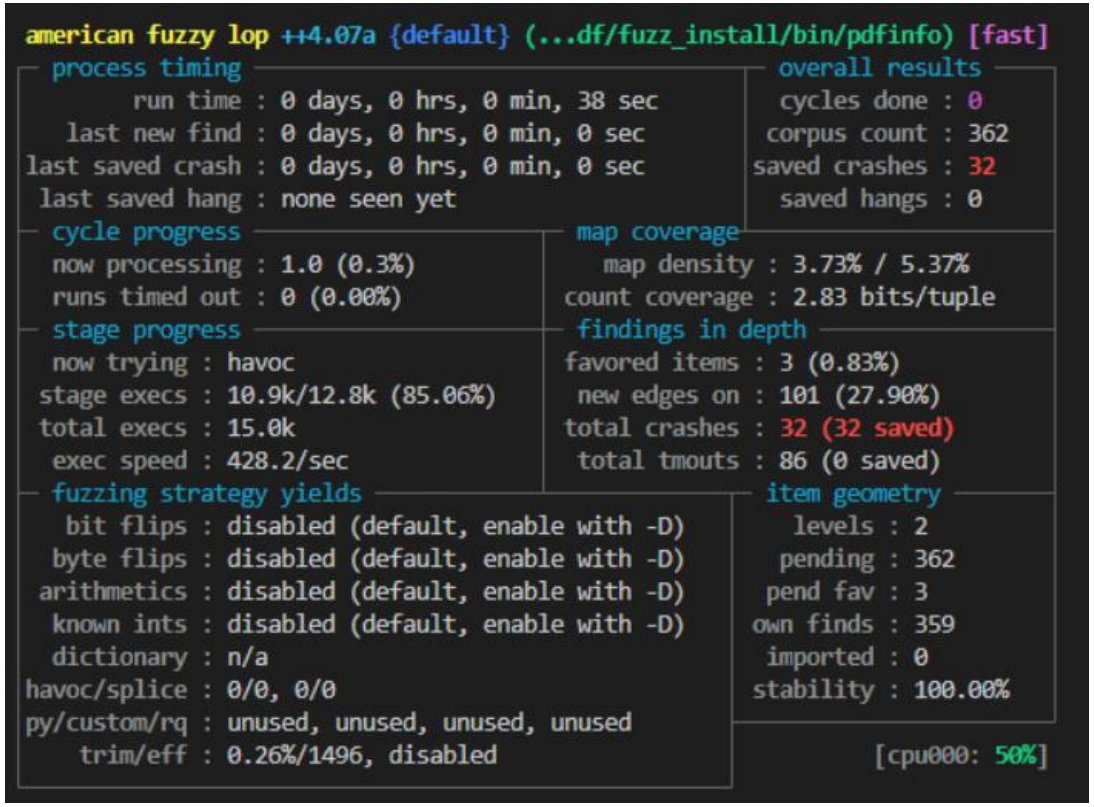
```
afl-fuzz -i $fuzz_in -o fuzz_out -- $fuzz_bin/pdftotext  @@ ./output
afl-fuzz -i $fuzz_in -o afl_out -- $fuzz_bin/pdfinfo -box @@
```

2.4.2使用AFL进行模糊测试

- Pdftotext测试结果：



- Pdfinfo测试结果



2.4.2使用AFL进行模糊测试

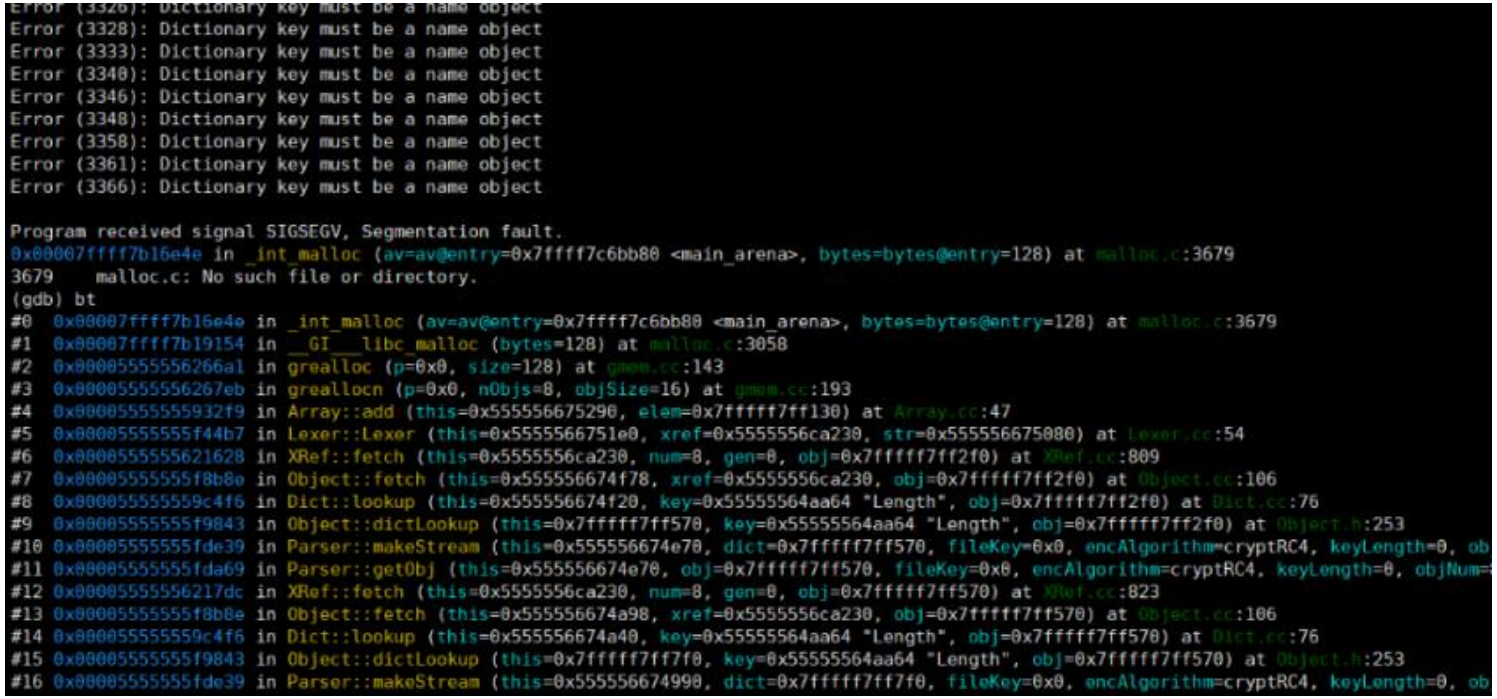
- 编译出带调试符号的文件:

```
make clean
CFLAGS="-g -O0" CXXFLAGS="-g -O0" ./configure --
prefix=/home/xpdf/dbg_install
make j4 && make install
```

- 调试:

```
export dbg_bin=/home/xpdf/dbg_install
gdb --args $dbg_bin/pdftotext fuzz_out/default/crashes/ ./output
```

2.4.2使用AFL进行模糊测试



- 报错信息 Program received signal SIGSEGV, Segmentation fault，存在内存泄漏。报错位置_int_malloc(av=av@entry=0x7ffff7c6bb80,bytes=bytes@entry=128)at malloc.c:3679，glibc 报了个错，显然是堆内存出了问题。执行流信息，分析一下可以看出调用过程是循环的，判断为无限循环漏洞。从 xpdf/Parse.cc 94 行的 makeStream 调用，一路跟着报错往下翻就会找到漏洞位置。

Q & A

问答