模糊测试

安全性测试

南京大学 软件学院 iSE实验室

目录

01

起源与发展

模糊测试的诞生

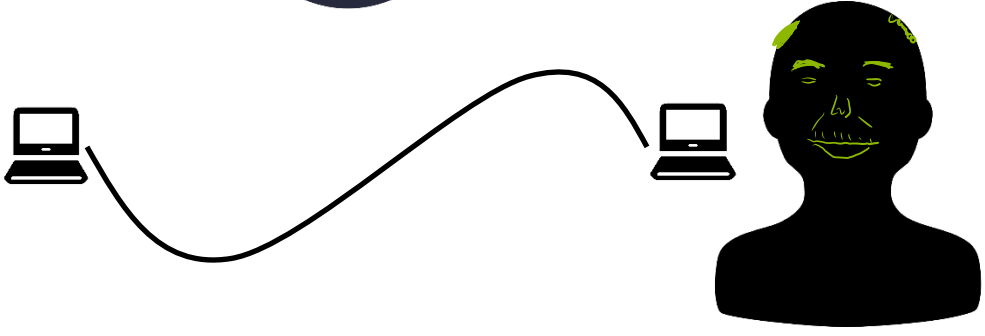
- Barton P. Miller，模糊测试之父
  - 1988年，Barton在自己的操作系统课上首次提出模糊测试
  - 1990年，*An Empirical Study of the Reliability of UNIX Utilities*[1]

Barton P. Miller

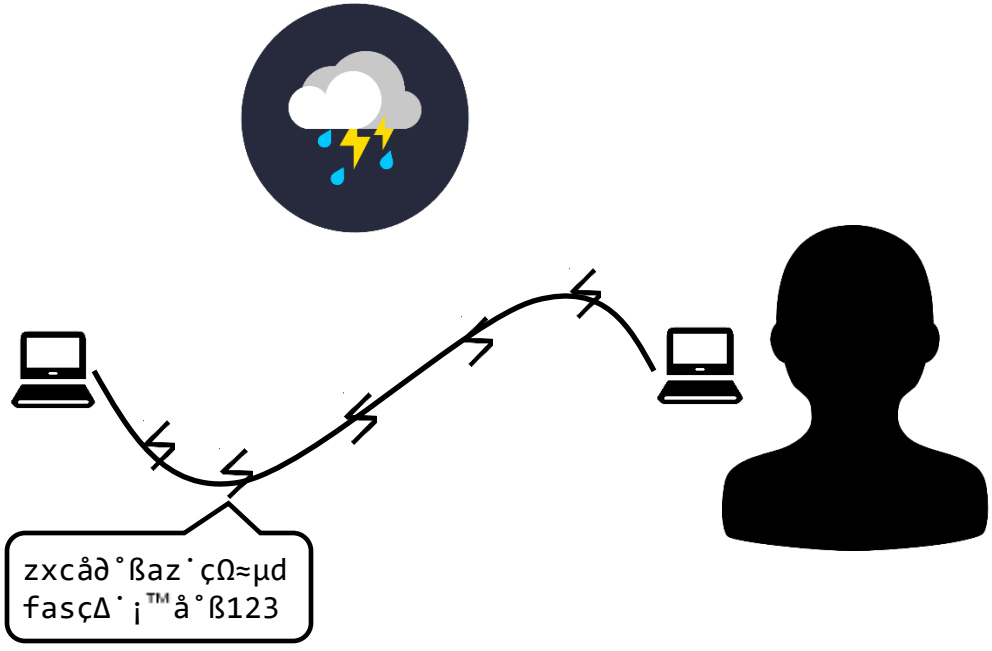
Vilas Distinguished Achievement Professor
Amar & Belinder Sohi Professor in Computer Sciences

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

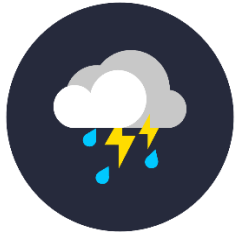
模糊测试的诞生

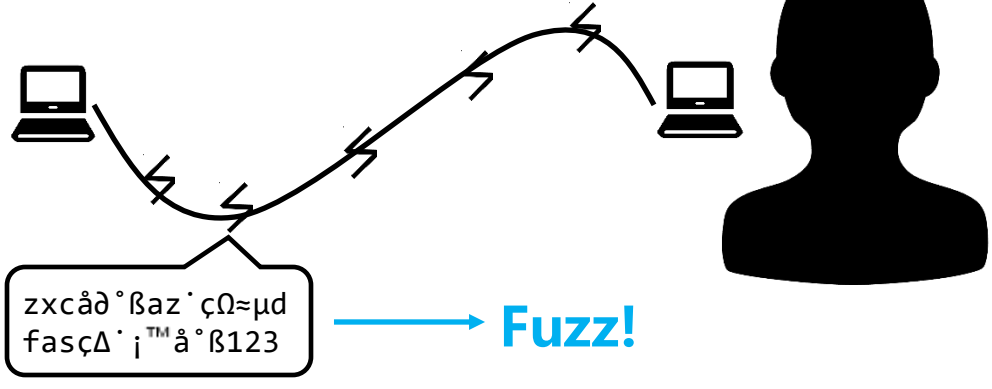


模糊测试的诞生

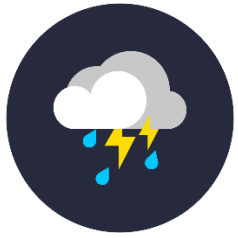


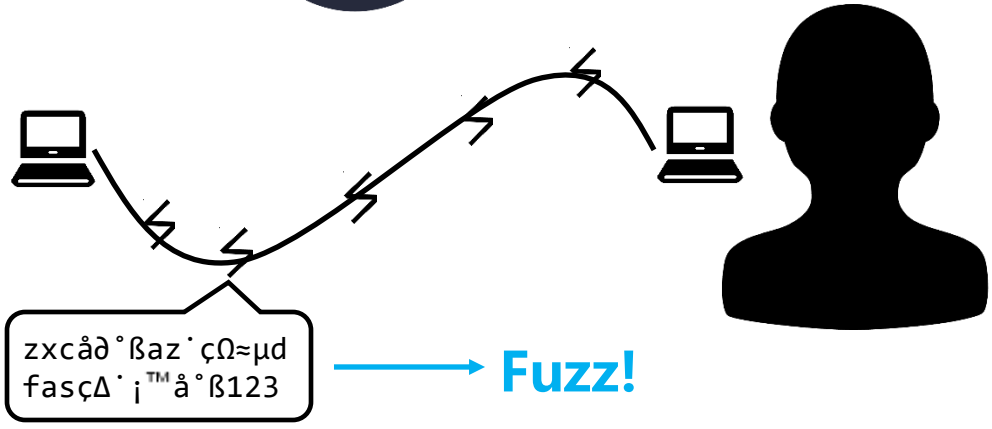
模糊测试的诞生





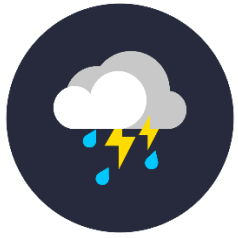
模糊测试的诞生

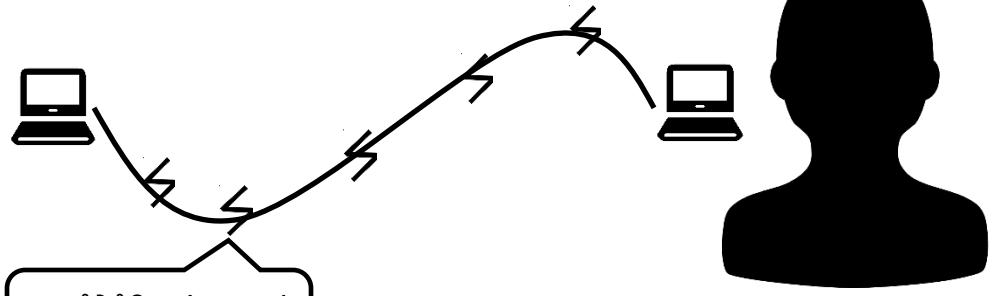




模糊测试的诞生

常见的字符乱码能够导致系统崩溃?



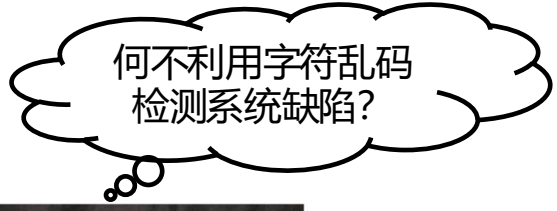


zxcåð˚ßaz˙çΩ≈µd
fasçΔ˙¡™å˚ß123

Fuzz!

Crash !

模糊测试的诞生



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

模糊测试的诞生

- 出发点：提升UNIX操作系统的可靠性
- 技术构想[1]
  - **核心组件**：一组用于产生随机字符的程序
  - **中心思想**：以随机字符串作为输入，运行操作系统组件（Utilities），观察是否崩溃
  - **最终结果**：保留能够产生崩溃的字符串输入，分析崩溃的类型，对崩溃进行分类
- 结果：在22个（总90个）组件程序（Utility Program）上触发崩溃

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

近期发展

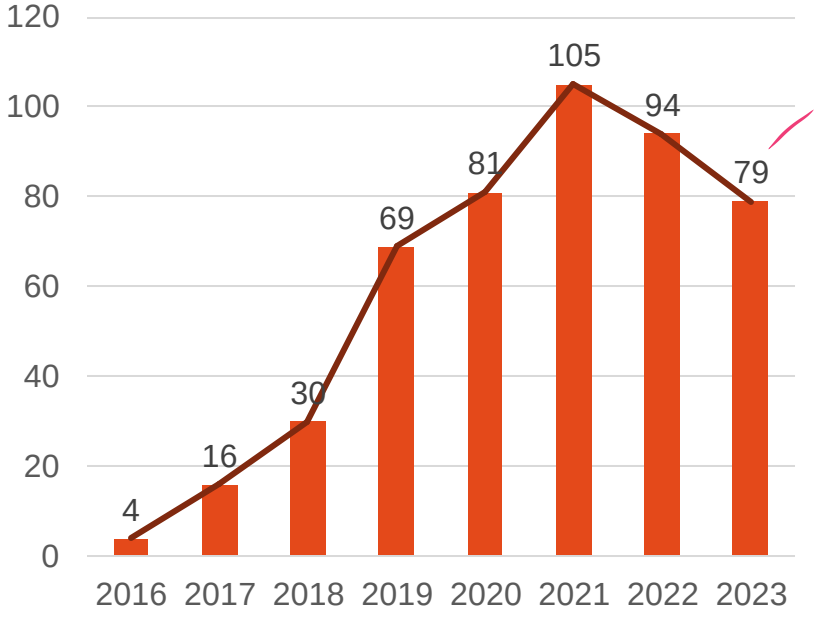
Year	Fuzzer	Solution(Process)	Fitness By	Target App/Bug	Input		Runtime Info.
					Muta. -based	Gene. -based	
2006	Sidewinder [62]	MC(seed.) + GA(rete.)	block transition	general	✓		◐
2007	RANDOOP [141]	GA(rete.)	legality	object-oriented	✓		●
2013	FuzzSim [192]	WCCP(seed.)	#bugs	general	✓		●
2014	COVERSET [158]	MSCP(set.)	code coverage	general		✓	◐
		ILP(seed.)	#bugs		✓		●+◐+○
2015	Joeri *et al.* [55]	GA(rete.)	state machine	protocol	✓		●
2016	AFLFast [23]	MC(seed.) + GA(rete.)	path transition	general	✓		◐
2016	classfuzz [42]	MH(mutation.) + GA(rete.)	code coverage	JVM	✓		◐
2017	VUzzer [157]	MC(seed.) + GA(rete.)	block transition	general	✓		◐
2017	AFLGo [22]	SA(seed.) + GA(rete.)	path transition	general	✓		◐
2017	NEZHA [151]	GA(rete.)	asymmetry	semantic bugs	✓		●+◐
2017	DeepXplore [148]	GA(rete.)	neuron coverage	deep learning	✓		◐
2018	STADS [17]	Species(seed.)*	state discovery	general	✓		◐
2018	CollAFL [66]	GA(rete.)	△code coverage	general	✓		◐
2018	Angora [37]	GD(byte.) + GA(rete.)	△code coverage	general	✓		◐
2019	DigFuzz [210]	MC(seed.) + GA(rete.)	block transition	general	✓		○
2019	MOPT [116]	PSO(mutation.) + GA(rete.)	code coverage	general	✓		◐
2019	NEUZZ [172]	NN(byte.) + GA(rete.)	branch behavior	general	✓		◐
2019	Cerebro [105]	MOO(seed.) + GA(rete.)	△code coverage	general	✓		◐
2019	DiffFuzz [138]	GA(rete.)	asymmetry	side-channel	✓		●
2020	AFLNET [154]	GA(rete.)	state machine	protocol	✓		◐
2020	EcoFuzz [203]	VAMAB(seed.) + GA(rete.)	path transition	general	✓		◐
2020	Entropic [21]	Shannon(seed.) + GA(rete.)	state discovery	general	✓		◐
2020	MTFuzz [171]	MTNN(byte.) + GA(rete.)	△branch behavior	general	✓		◐
2020	Ankou [118]	GA(rete.)	△code coverage	general	✓		◐
2020	FIFUZZ [87]	GA(rete.)	△code coverage	error-handling	✓		◐
2020	IJON [6]	GA(rete.)	△code coverage	general	✓		◐
2020	Krace [194]	GA(rete.)	alias coverage	data race	✓		◐
2021	AFL-HIER [88]	UCB1(seed.) + GA(rete.)	△path transition	general	✓		◐
2021	PGFUZZ [82]	GA(rete.)	safety policy	robotic vehicle	✓		●
2021	Yousra *et al.* [1]	GA(rete.)	validation log	SmartTV	✓		●
2021	AFLChurn [214]	SA(seed.) + ACO(byte.) + GA(rete.)	path transition + commit history	general	✓		◐

2006~2021年间发表的主要模糊测试技术一览[1]

[1] Zhu X, Wen S, Camtepe S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys (CSUR), 2022.

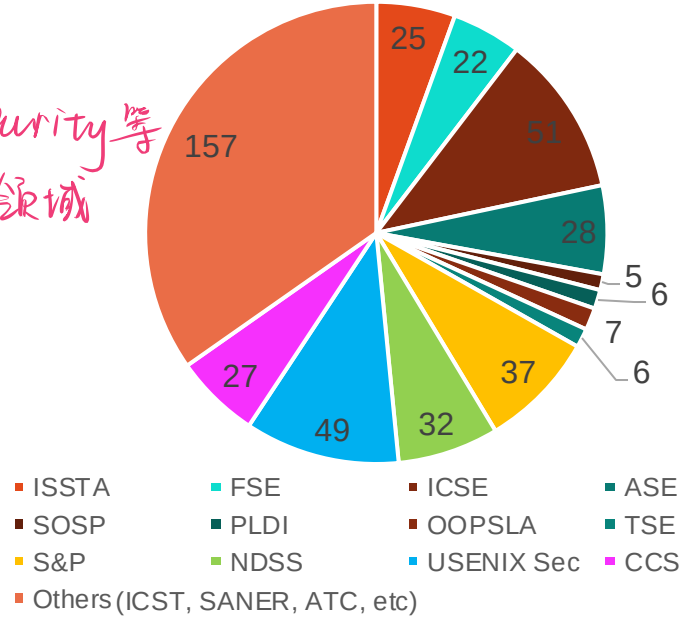
近期发展

简单的框架，多样的拓展



2016~2023年发表刊物数量趋势（不完全统计）

仅security等部分领域

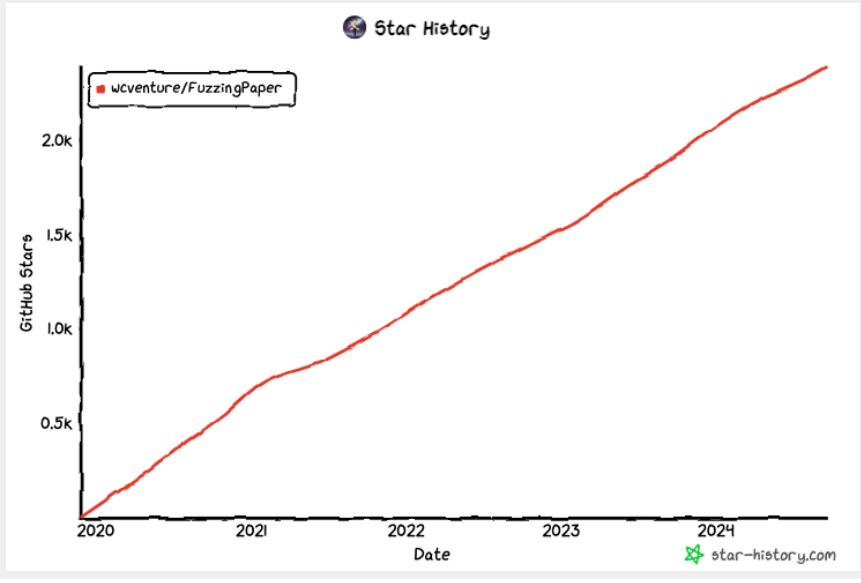


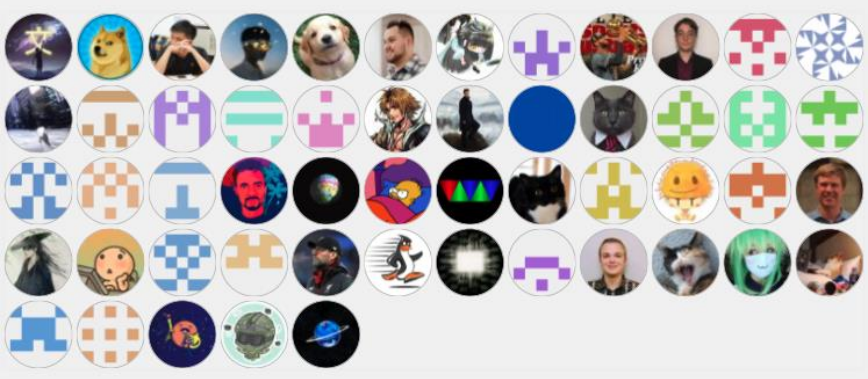
2016~2023年发表刊物分布（不完全统计）

[1] 数据来源：https://github.com/wcventure/FuzzingPaper/blob/master/README.md

近期发展

- https://wcventure.github.io/FuzzingPaper
  - **Paper Collection**





近期发展

- https://wcventure.github.io/FuzzingPaper
  - **Paper Collection**

All Papers (Classification according to Publication)

- **ESEC/FSE 2024**
  - BRF: Fuzzing the eBPF runtime
  - Evaluating Directed Fuzzers: Are We Heading in the Right Direction?
- **SP 2024**
  - Everything is Good for Something: Counterexample-Guided Directed Fuzzing via Likely Invariant Inference
  - LABRADOR: Response Guided Directed Fuzzing for Black-box IoT Devices
  - SoK: Prudent Evaluation Practices for Fuzzing
  - Titan: Efficient Multi-target Directed Greybox Fuzzing
  - Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay
  - DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic Protocol Fuzz Testing
  - SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing
  - Towards Smart Contract Fuzzing on GPU
  - LLMIF: Augmented Large Language Model for Fuzzing IoT Devices
  - SATURN: Host-Gadget Synergistic USB Driver Fuzzing
  - To Boldly Go Where No Fuzzer Has Gone Before: Finding Bugs in Linux' Wireless Stacks through VirtIO Devices
  - Predecessor-aware Directed Greybox Fuzzing
  - AFGen: Whole-Function Fuzzing for Applications and Libraries
  - SyzTrust: State-aware Fuzzing on Trusted OS Designed for IoT Devices
  - LLMIF: Augmented Large Language Model for Fuzzing IoT Devices
- **NDSS 2024**
  - Large Language Model guided Protocol Fuzzing
  - DeepGo: Predictive Directed Greybox Fuzzing
  - ShapFuzz: Efficient Fuzzing via Shapley-Guided Byte Selection
  - EnclaveFuzz: Finding Vulnerabilities in SGX Applications
  - MOCK: Optimizing Kernel Fuzzing Mutation with Context-aware Dependency
  - EnclaveFuzz: Finding Vulnerabilities in SGX Applications
  - Predictive Context-sensitive Fuzzing
  - ReqsMiner: Automated Discovery of CDN Forwarding Request Inconsistencies and DoS Attacks with Grammar-based Fuzzing
- **USENIX Security 2024**
  - Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities
  - ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing
  - Cascade: CPU Fuzzing via Intricate Program Generation
  - Towards Generic Database Management System Fuzzing
  - SHiFT: Semi-hosted Fuzz Testing for Embedded Applications
  - SDFuzz: Target States Driven Directed Fuzzing
  - Critical Code Guided Directed Greybox Fuzzing for Commits
  - EL3XIR: Fuzzing COTS Secure Monitors
  - MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware
  - WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors
  - HYPERPILL: Fuzzing for Hypervisor-bugs by leveraging the Hardware Virtualization Interface
  - Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing
  - From One Thousand Pages of Specification to Unveiling Hidden Bugs: Large Language Model Assisted Fuzzing of Matter IoT Devices
- **ASPLOS 2024**
  - [Greybox Fuzzing for Concurrency Testing]

02

概念与框架

定义

- **什么是Fuzzing?**
  - **Generate** some inputs, make some **run**s, and **see** what happens

合法但有害

初始构想

- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**
  - **工具**：模糊器（Fuzzer）
  - **目标**：待测程序（PUT）
  - **循环**：执行程序 ⇄ 崩溃分派（Crash Triage）

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**

工具Fuzzer：

- **三个组件：**输入生成组件（Input Generator）、测试执行组件（Test Executor）、输出分析组件（Output Analyzer）

Fuzz　　Analyser　　输入

$F = (G, E, A)$

Driver　　(CFI图)

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

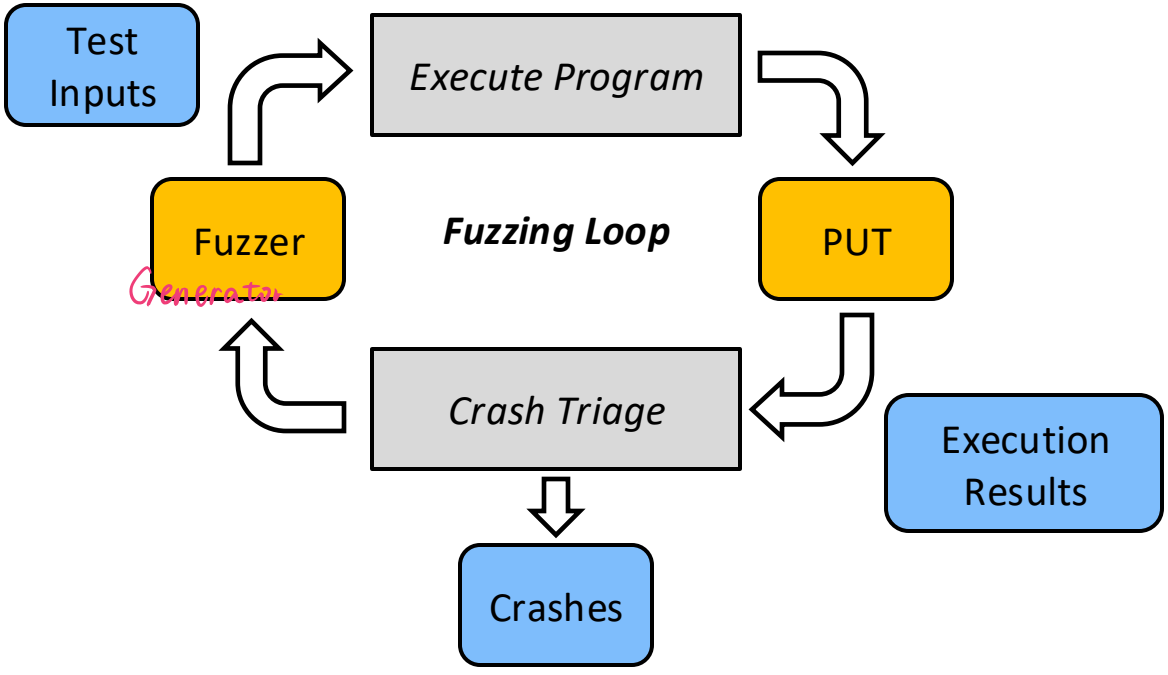
初始构想

- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**

[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

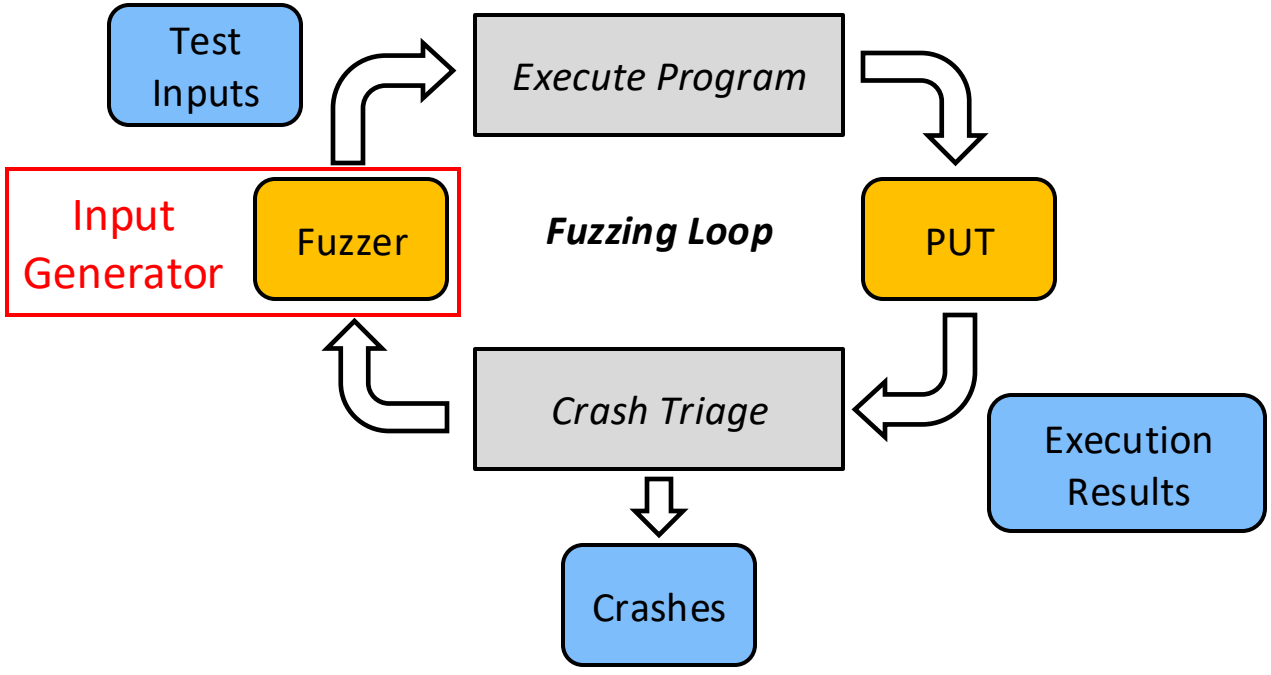
- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

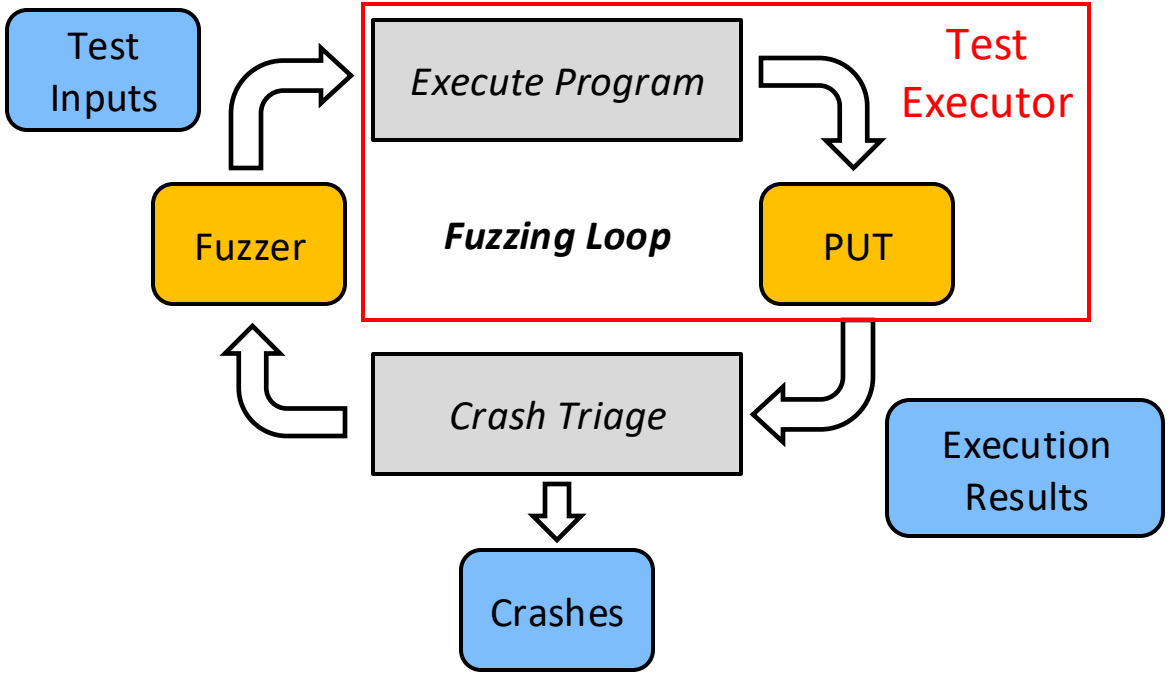
- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

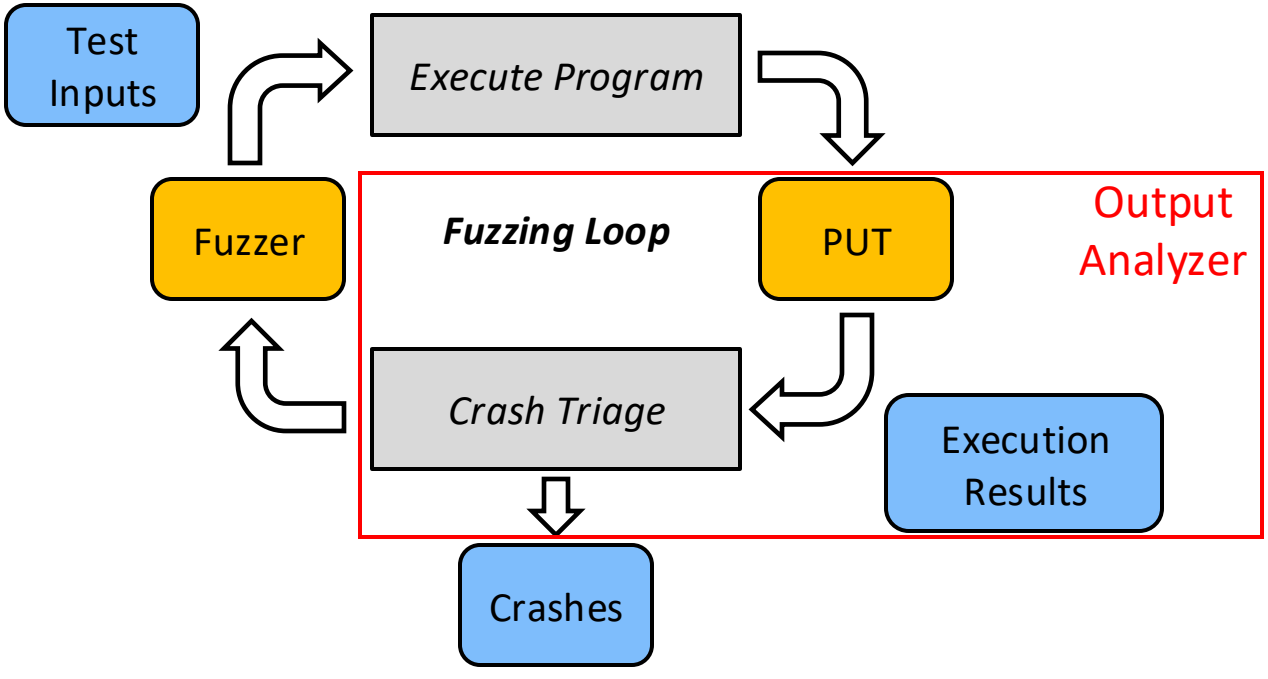
- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

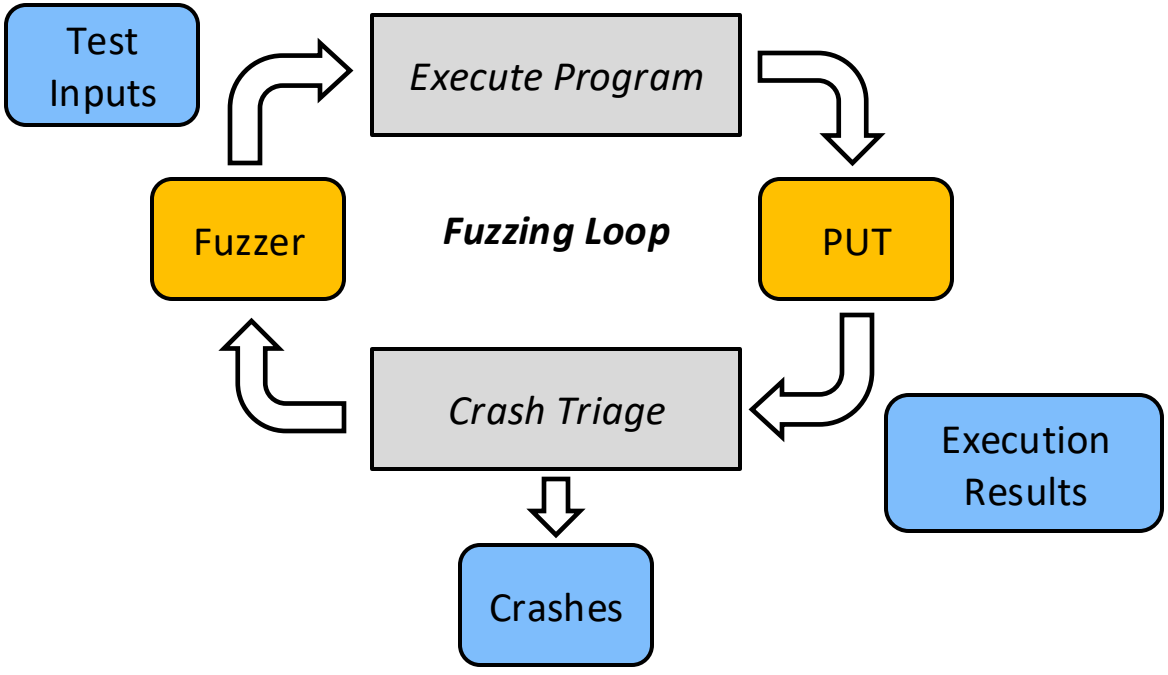
- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

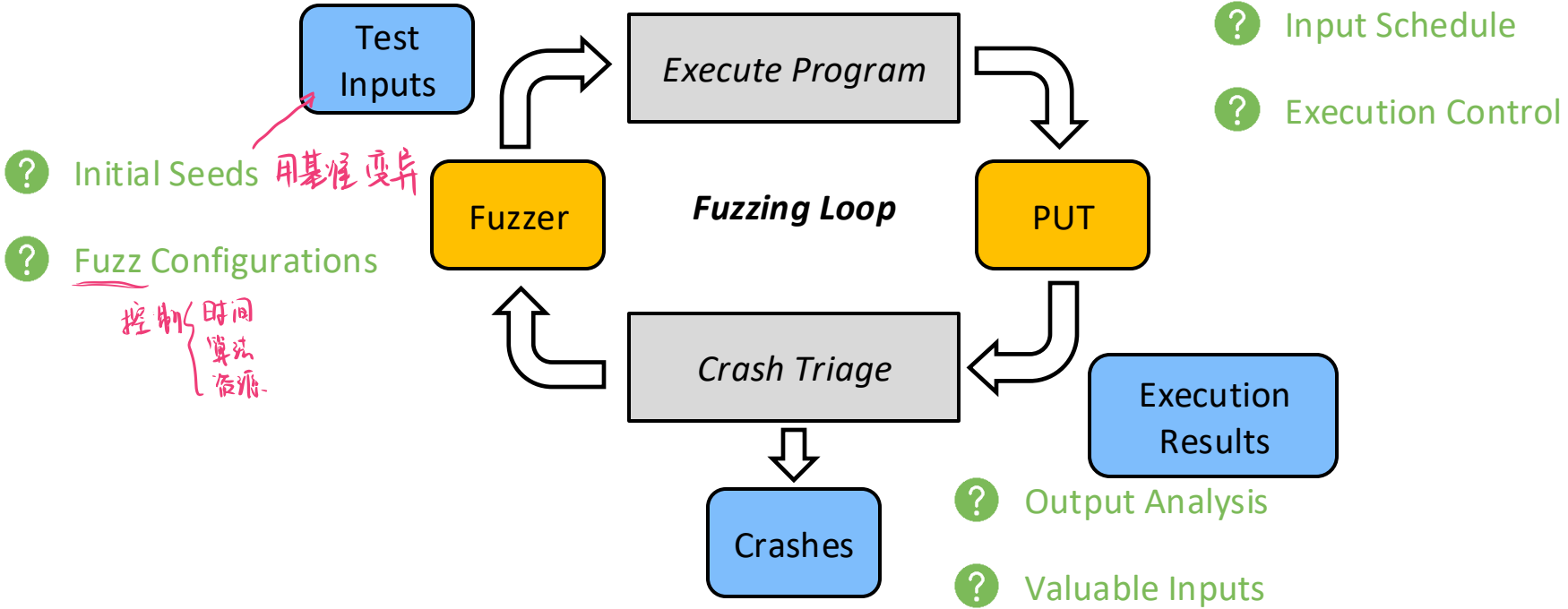
- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

初始构想

- 三要素[1]：一个（套）**工具**、一个**目标**、一个**循环**



[1] Miller B P, Fredriksen L, So B. An empirical study of the reliability of UNIX utilities[J]. Communications of the ACM, 1990, 33(12): 32-44.

相关术语

测试输入空间 → 输入

- 模糊（Fuzzing）与模糊测试（Fuzz Testing）
- 模糊器（Fuzzer）
- 模糊运动（Fuzzing Campaign）
- 缺陷预言（Bug Oracle）
- 模糊配置（Fuzz Configuration）
- 测试输入（Test Input）、测试用例（Test Case）与种子输入（Seed Input）

[1] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.
[2] Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
[3] 测试用例定义 · 维基百科：https://zh.wikipedia.org/zh-cn/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B

相关术语

- 模糊（Fuzzing）与模糊测试（Fuzz Testing）
  - **模糊**是指使用从输入空间（模糊输入空间）采样得到的输入来执行待测程序（PUT，Program Under Test）的过程。该模糊输入空间代表着测试人员针对待测程序定义的预期输入。
  - **模糊测试**是一种应用模糊（过程）来验证待测程序是否违反正确性策略（Correctness Policy）的测试技术。测试预言
  - 在文献中一般可互换

[1] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.
[2] Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
[3] 测试用例定义，维基百科：https://zh.wikipedia.org/zh-cn/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B

相关术语

- 模糊器（Fuzzer）
  - **模糊器**是一个（组）用于实现模糊测试的程序。→ 核心组件
- 模糊运动（Fuzz Campaign）
  - **模糊运动**是指一个**模糊器**按照一组特定的**正确性政策**在一个给定**待测程序**上的一次具体的**执行**。→ $< fuzzer,\ program >$ 二元组

[1] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.
[2] Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
[3] 测试用例定义，维基百科：https://zh.wikipedia.org/zh-cn/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B

相关术语

- 缺陷预言（Bug Oracle）
  - 一个用于确定一次给定**执行**是否违反具体**正确性策略**的程序，通常作为**模糊器**的一部分出现。
- 模糊配置（Fuzz Configuration）
  - 一组控制和描述模糊（测试）算法的数据和约束

[1] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.
[2] Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
[3] 测试用例定义，维基百科，https://zh.wikipedia.org/zh-cn/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B

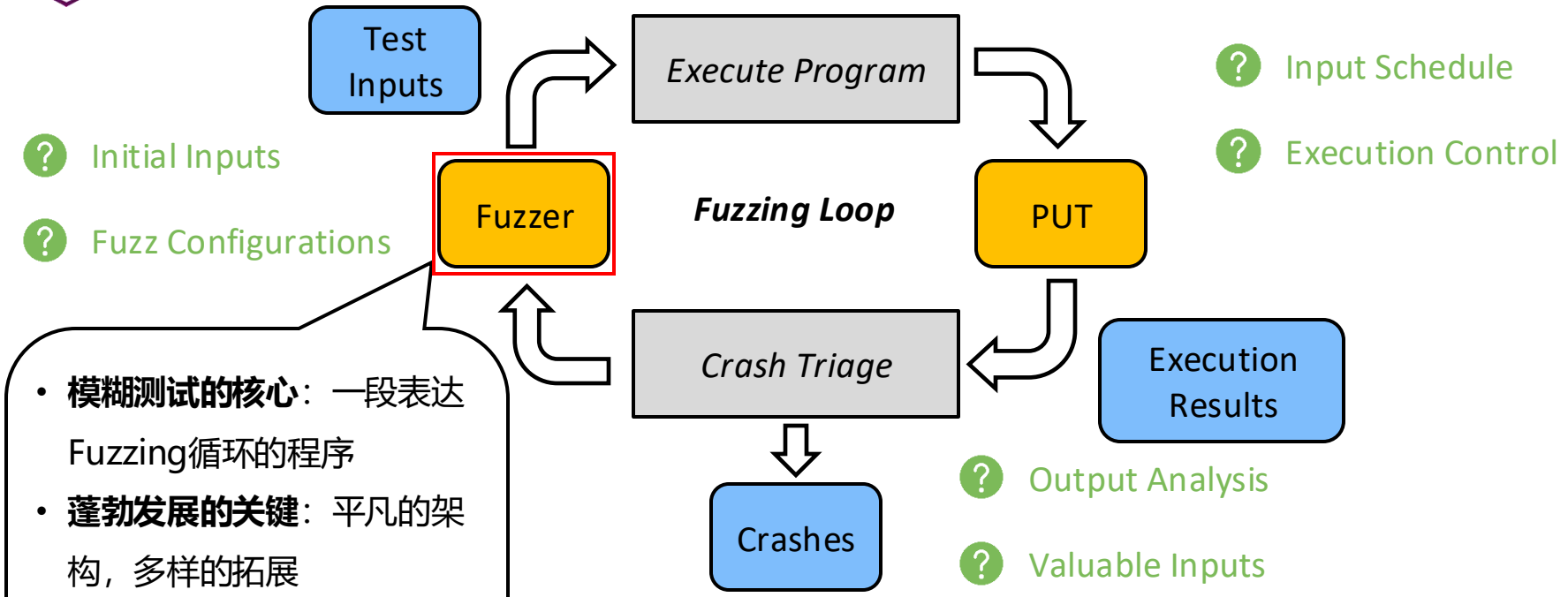
相关术语

- 测试输入（Test Input）、测试用例（Test Case）与种子输入（Seed Input）
  - **测试输入**是一组用于驱动待测程序执行的数据
  - **测试用例**是一组用于确定应用软件或软件系统是否能够正确工作的条件或变量[3] → 输入 + 逻辑（调用序列） + 预言
  - **种子输入**是一个或一组在模糊测试过程中为输入生成（Input Generation）提供基准的测试输入，简称**种子**（Seed）。

如一个.xml一张图片

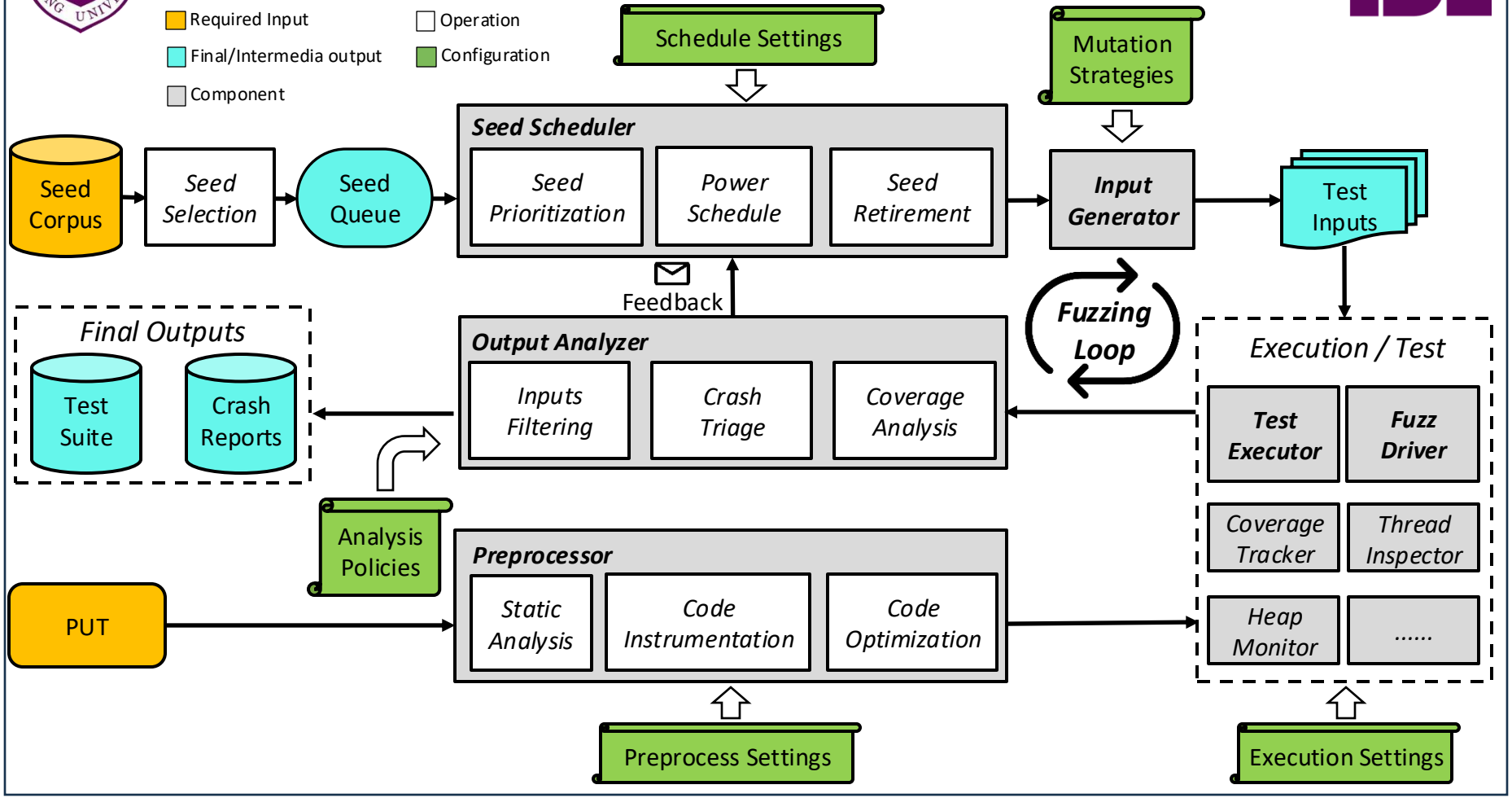
[1] Manès V J M, Han H S, Han C, et al. The art, science, and engineering of fuzzing: A survey[J]. IEEE Transactions on Software Engineering, 2019, 47(11): 2312-2331.
[2] Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software[C]//Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. 2011: 416-419.
[3] 测试用例定义 · 维基百科：https://zh.wikipedia.org/zh-cn/%E6%B5%8B%E8%AF%95%E7%94%A8%E4%BE%8B

模糊测试框架



- **模糊测试的核心**：一段表达Fuzzing循环的程序
- **蓬勃发展的关键**：平凡的架构，多样的拓展
- **主要构成**：输入生成、测试执行、输出分析

模糊测试框架


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
Seed Scheduler
Seed Corpus
Seed Selection
Seed Queue
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
PUT
Static Analysis
Code Instrumentation
Code Optimization
Preprocess Settings
Execution Settings

03

家族与分类

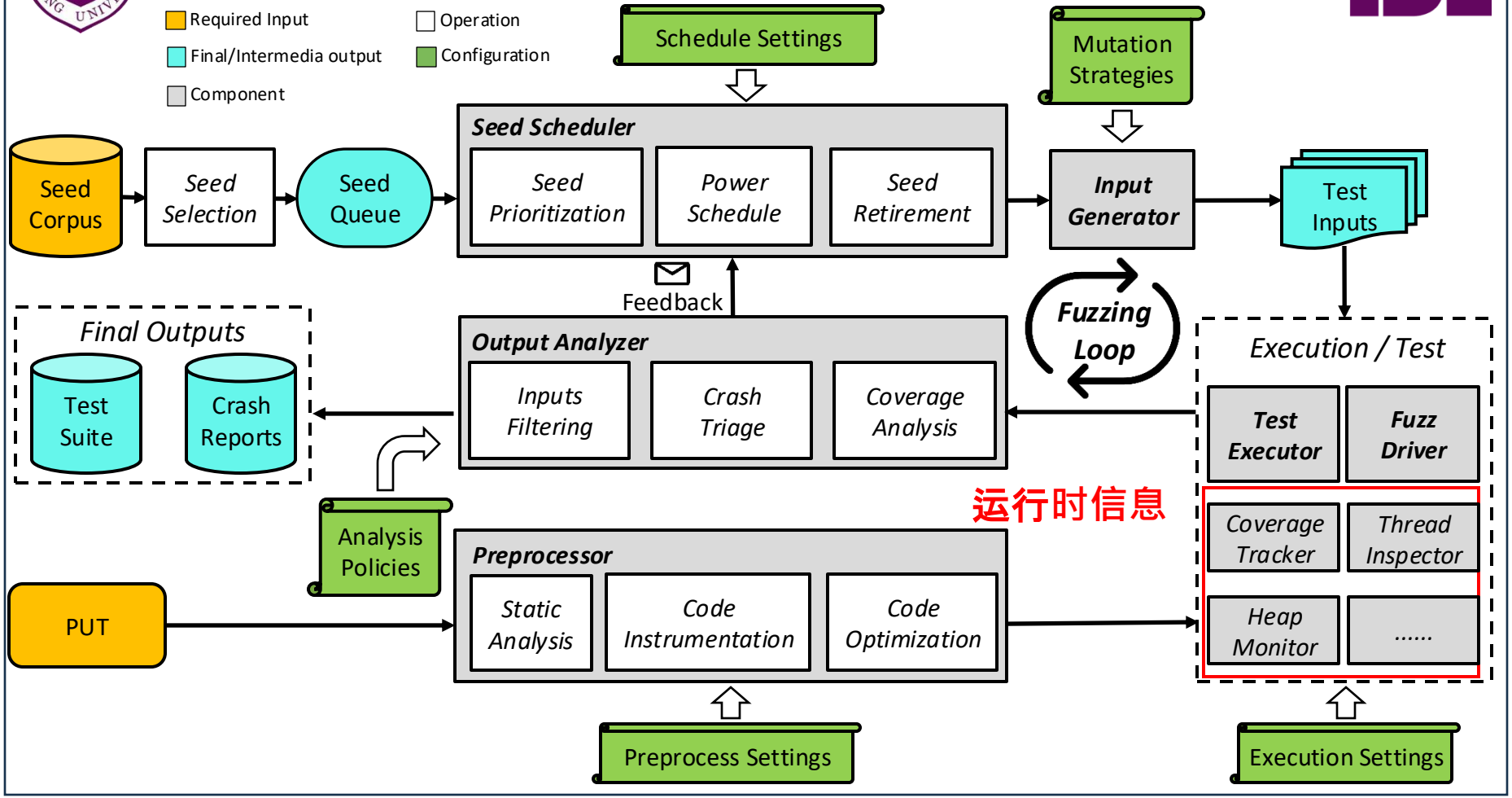
模糊测试家族

- 根据基础Fuzzer划分家族
  - AFL家族（C/C++）：AFL、AFLFast、AFLSmart、AFLNet、AFLGo、AFLIoT、FairFuzz、Mopt.、Neuzz
  - LibFuzzer家族（C/C++）：LibFuzzer、Entropic
  - JQF家族（Java）：JQF、BeDivFuzz、CONFETTI
  - 其他（Rust、Python等）：Angora、DeepXplore

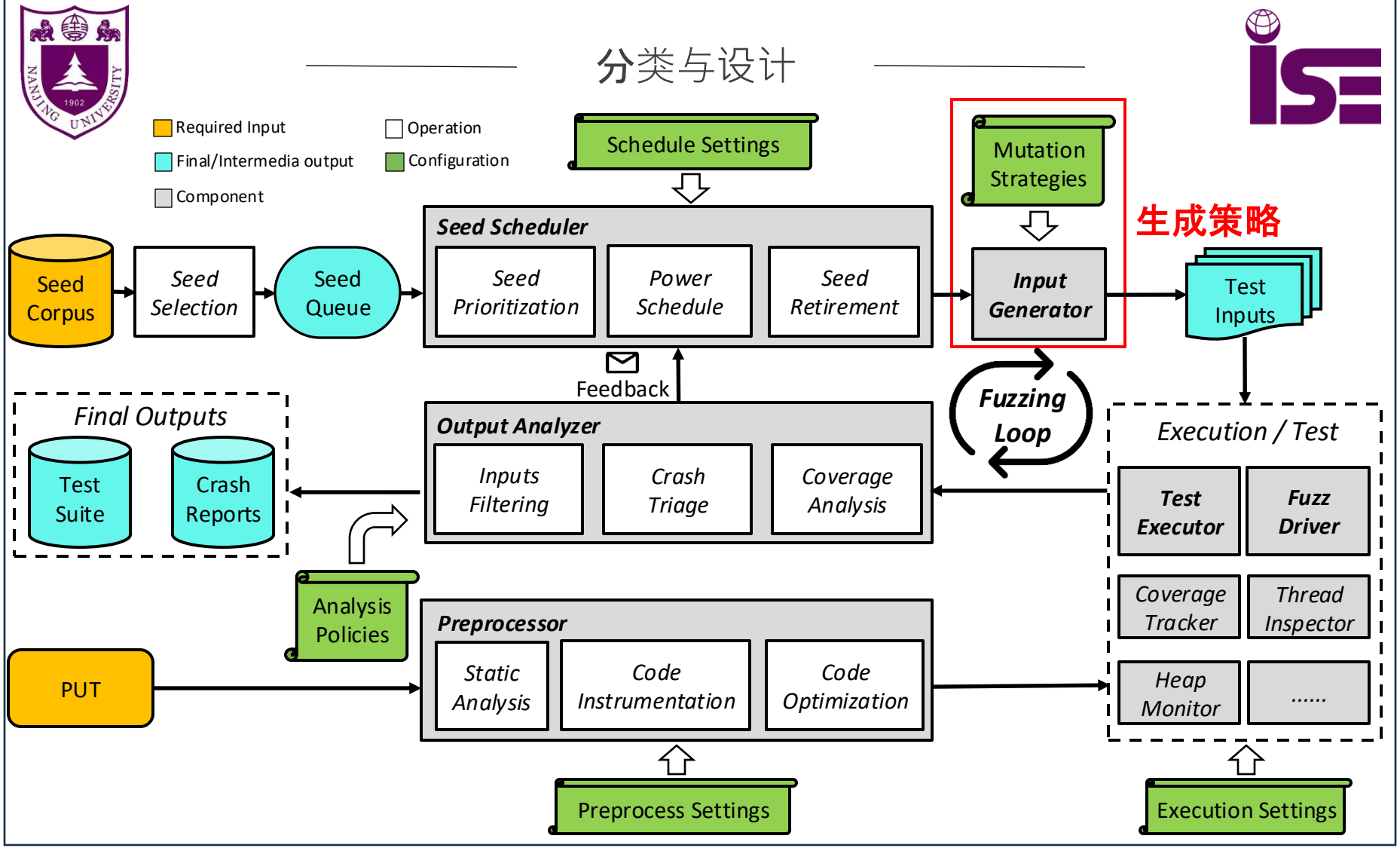
模糊测试分类

- 根据组件核心或技术贡献进行分类
  - 按照**采用的运行时信息**：黑盒、灰盒、白盒
  - 按照**输入生成的策略**：Mutation-based, Generation-based
  - 按照**引导过程**：Search-based（一些启发式算法），Gradient-based（梯度下降）
  - 按照**测试目标**：定向、非定向、某一类缺陷
  - 按照**应用领域**：网络协议、Compiler、DNN、IoT、内核
  - 按照**优化角度**：种子调度、变异策略、能量调度、过程建模

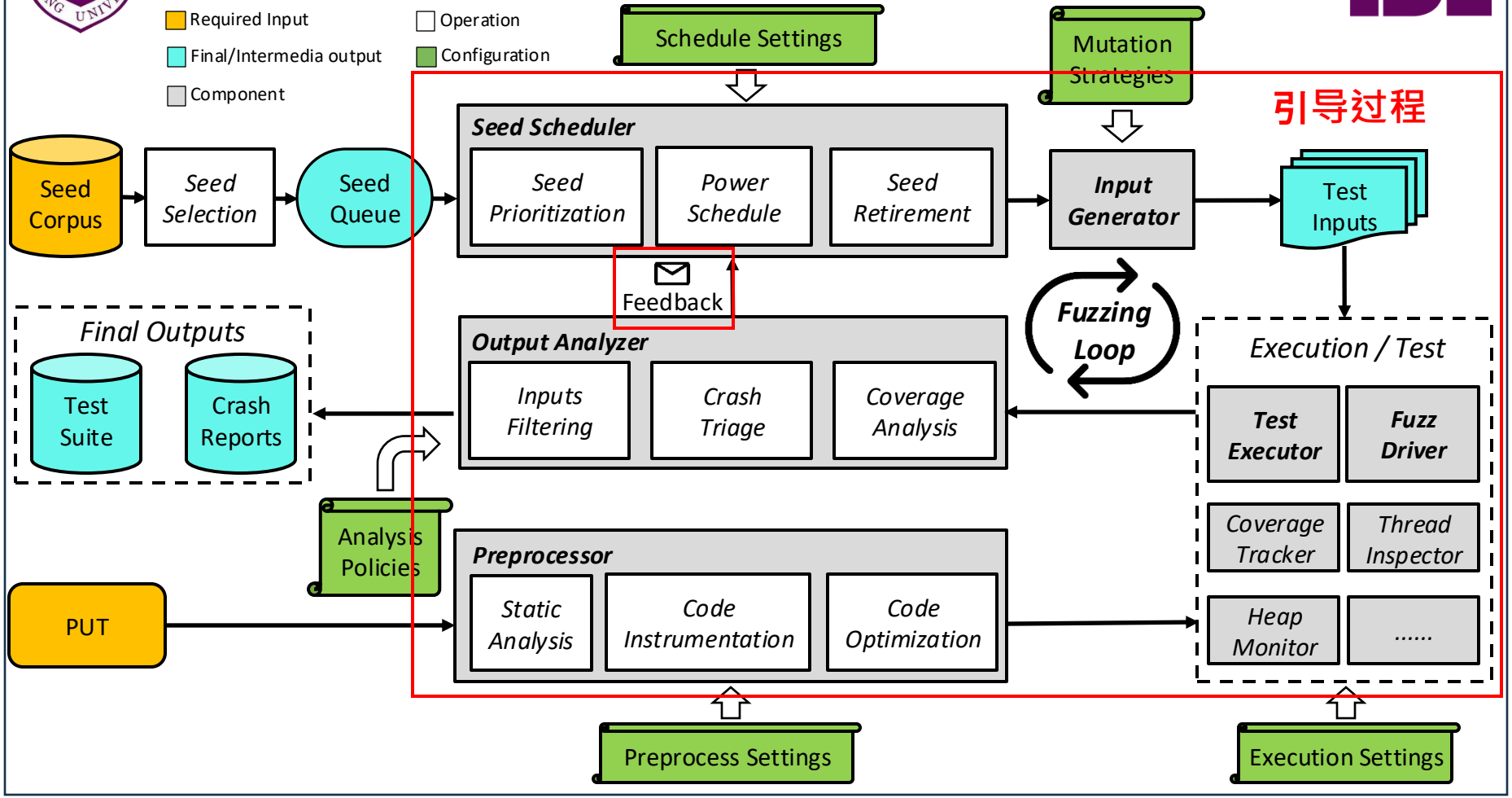
分类与设计


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
Seed Scheduler
Seed Corpus
Seed Selection
Seed Queue
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
运行时信息
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
PUT
Static Analysis
Code Instrumentation
Code Optimization
Preprocess Settings
Execution Settings

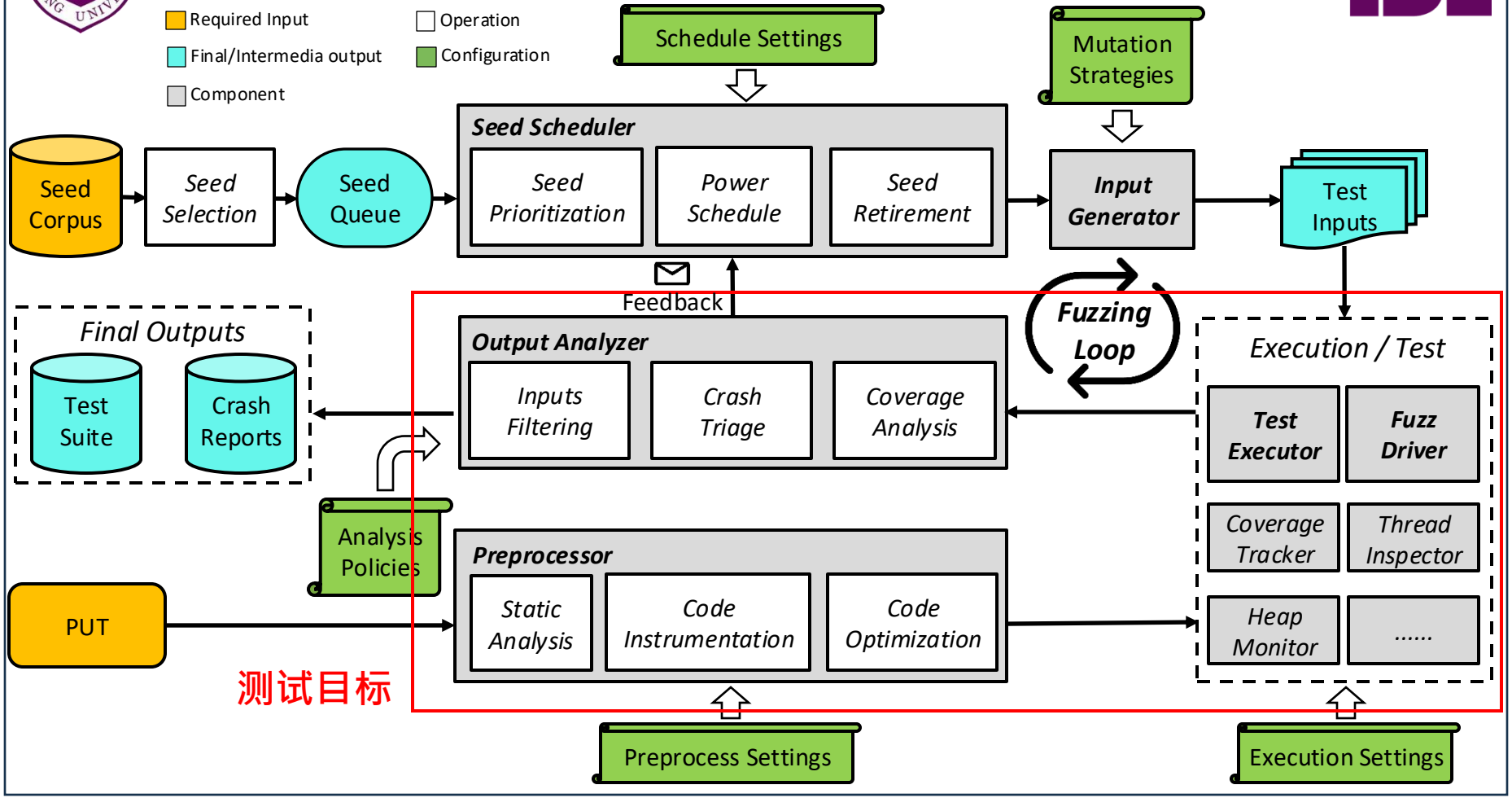
分类与设计


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
生成策略
Seed Scheduler
Seed Corpus
Seed Selection
Seed Queue
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
PUT
Static Analysis
Code Instrumentation
Code Optimization
Preprocess Settings
Execution Settings

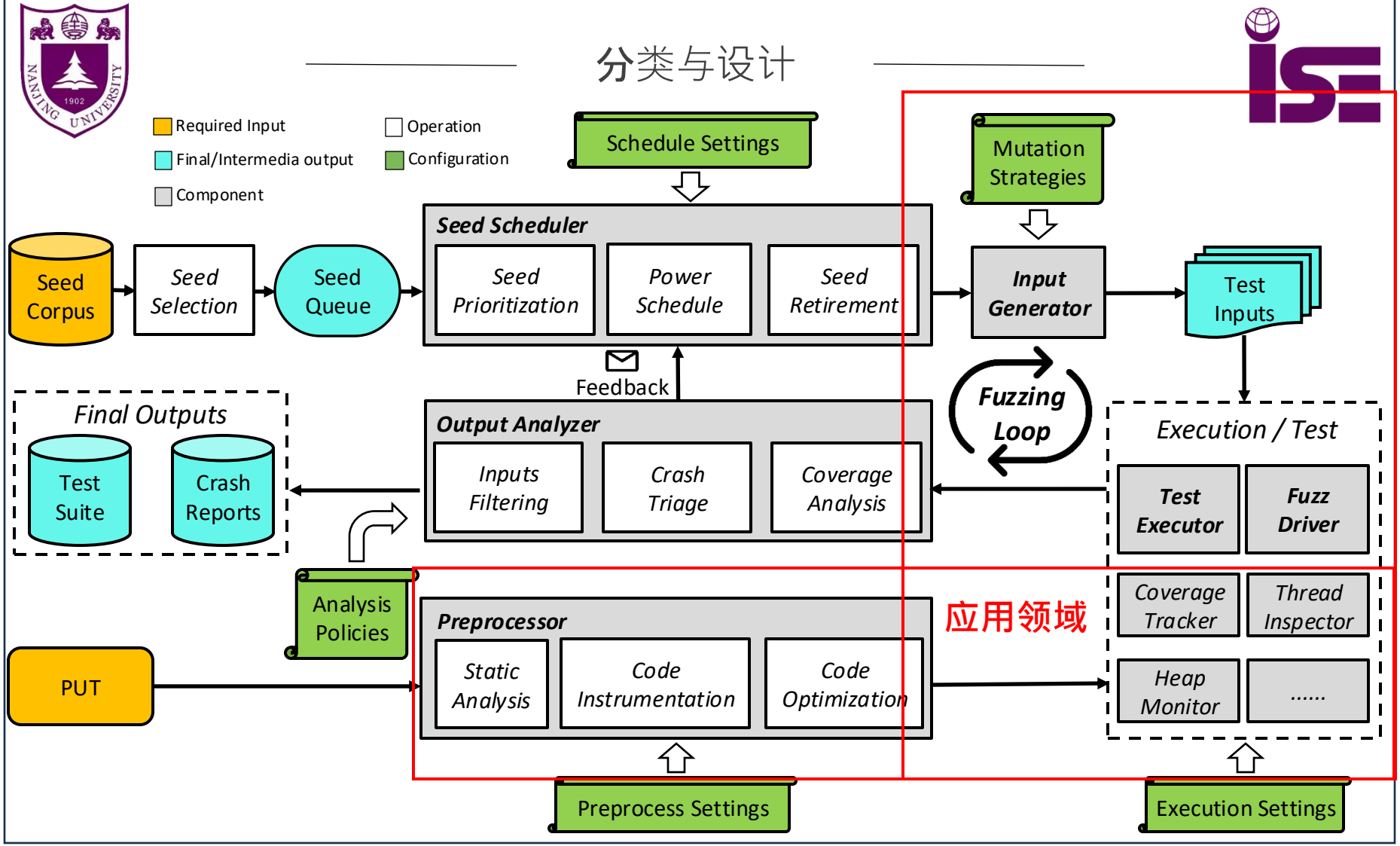
分类与设计


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
引导过程
Seed Corpus
Seed Selection
Seed Queue
Seed Scheduler
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
Static Analysis
Code Instrumentation
Code Optimization
PUT
Preprocess Settings
Execution Settings

分类与设计


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
Seed Scheduler
Seed Corpus
Seed Selection
Seed Queue
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
PUT
Static Analysis
Code Instrumentation
Code Optimization
测试目标
Preprocess Settings
Execution Settings

分类与设计


Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
Seed Scheduler
Seed Corpus
Seed Selection
Seed Queue
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
Preprocessor
Static Analysis
Code Instrumentation
Code Optimization
应用领域
PUT
Preprocess Settings
Execution Settings

按照运行时信息分类

- 按照**采用的运行时信息**[1]
  - 黑盒（Blackbox）模糊测试
    - 特点：不监控执行过程，也不使用执行过程中产生的任何信息，仅从输入和输出端入手优化模糊测试
    - 引导方式：利用输入格式或输出状态引导测试执行
    - 优缺点：效率高，但引导的有效性上面有所欠缺
    - 代表性工作：KIF[2]、IoTFuzzer[3]、CodeAlchemist[4]

有时不得不用

[1] Zhu X, Wen S, Camtepe S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys (CSUR), 2022.
[2] Abdelnur H J, State R, Festor O. KiF: a stateful SIP fuzzer[C]//Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications. 2007: 47-56.
[3] Chen J, Diao W, Zhao Q, et al. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing[C]//NDSS. 2018.
[4] Han H S, Oh D H, Cha S K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines[C]//NDSS. 2019.

按照运行时信息分类

- 按照**采用的运行时信息**[1]
  - 白盒（Whitebox）模糊测试
    - 特点：使用混合执行、污点分析（Taint Analysis）等比较昂贵的白盒分析技术优化模糊测试过程
    - 引导方式：利用详细的程序分析结果引导测试执行
    - 优缺点：反馈更加有效，但是效率不高、适配性较差
    - 代表性工作：Driller[2]、QSYM[3]、CONFETTI[4]

[1] Zhu X, Wen S, Camtepe S, et al. Fuzzing: a survey for roadmap[J]. ACM Computing Surveys (CSUR), 2022.
[2] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution[C]//NDSS. 2016, 16(2016): 1-16.
[3] Yun I, Lee S, Xu M, et al. QSYM: A practical concolic execution engine tailored for hybrid fuzzing[C]//27th USENIX Security Symposium (USENIX Security 18). 2018: 745-761.
[4] Kukucka J, Pina L, Ammann P, et al. CONFETTI: Amplifying Concolic Guidance for Fuzzers[C]//44th IEEE/ACM International Conference on Software Engineering, ser. ICSE. 2022, 22.

- 按照**采用的运行时信息**[1]
  - **灰盒**（Greybox）模糊测试
    - Coverage-based Greybox Fuzzing, CGF
    - 特点：采用轻量级插装对程序进行监控，在执行过程中收集各类信息，如分支覆盖、线程执行、堆栈状态等
    - 引导方式：利用收集到的执行信息（内部状态）引导测试执行
    - 代表性工作：AFL、AFLGo、EcoFuzz、Zest、BeDivFuzz

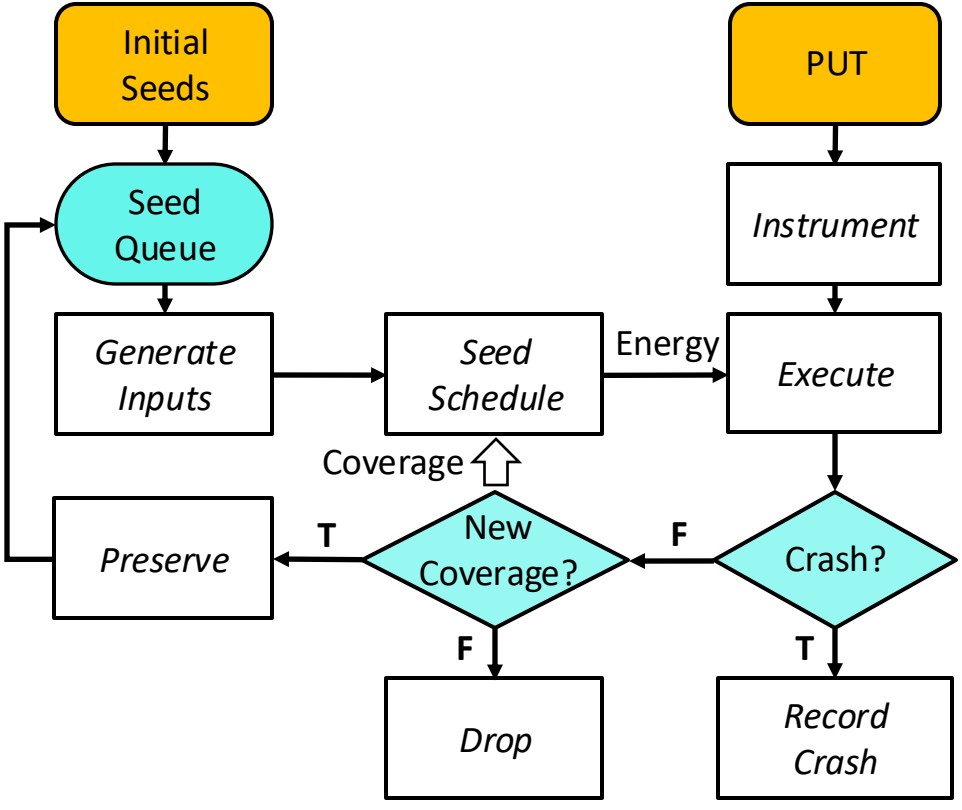
灰盒模糊测试框架

Algorithm 1 Greybox Fuzzing

Input: Initial Seeds S, Instrumented PUT p
Output: Seed Corpus S'

```
1:  S' ← S
2:  S_crash ← ∅
3:  repeat                                   ▷ Main fuzzing loop
4:      s ← CHOOSENEXT(S')
5:      e ← ASSIGNENERGY(s)                  ▷ Power scheduling
6:      for i from 1 to e do
7:          s' ← MUTATESEED(s)
8:          res ← EXECUTE(p, s')
9:          if ISCRASH(s', res) then
10:             add s' to S_crash
11:         else if ISINTERESTING(s', res) then
12:             add s' to S'
13:         end if
14:     end for
15: until resources exceeds or abort-signal comes
16: return S', S_crash
```

CGF伪代码



CGF一般流程

按照生成策略分类

- 按照**输入生成的策略**
  - Input Generator、Mutation Strategies
  - Mutation-based：基于随机或启发式变异策略
  - Generation-based：基于一定的文法规则/结构信息

- Mutation-based：基于随机变异或启发式变异策略
  - **本质**：将种子输入转换为比特串（Bits），对比特串进行变换
  - **优点**：可拓展性强，易于泛化，理论上可用于任意输入（图片、文本、音视频）
  - **缺点**：容易破坏输入的结构、产生无效输入（Invalid Input）；生成的大部分输入都无法通过语义检查（Semantic Checking），很难探测深层次的程序状态和程序元素（Program Elements）

基于变异的模糊测试

- Mutation-based：AFL[1]
  - AFL变异算子
    - bitflip L/S：以S为增量，每次翻转L位
    - arith L/8：加/减长度为L的小整数
    - interest L/8：翻转“有趣”字节位
    - **havoc**：对输入进行大肆破坏
    - splice：随机拼接两个种子输入

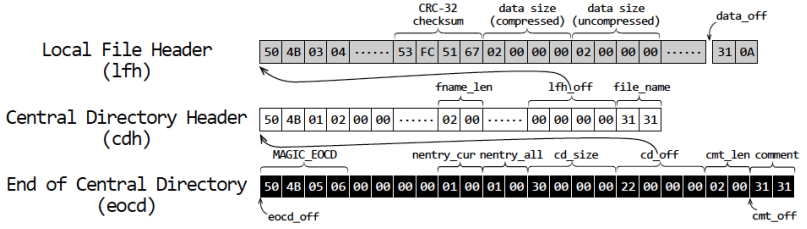
Fuzzing a Picture

[1] AFL博客：https://afl-1.readthedocs.io/en/latest/index.html

基于变异的模糊测试

- Mutation-based：SLF（ICSE'19）[1]
  - 思想：分析程序中的语义检查、识别比特串中与影响语义检查的域（Field）、根据两者之间的关系制定变异策略

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000h	50	4B	03	04	00	00	00	00	00	00	00	00	00	00	53	FC
00000010h	51	67	02	00	00	00	02	00	00	00	02	00	00	00	31	31
00000020h	31	0A	50	4B	01	02	00	00	00	00	00	00	00	00	00	00
00000030h	00	00	53	FC	51	67	02	00	00	00	02	00	00	00	02	00
00000040h	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050h	31	31	50	4B	05	06	00	00	00	00	01	00	01	00	30	00
00000060h	00	00	22	00	00	00	02	00	31	31						

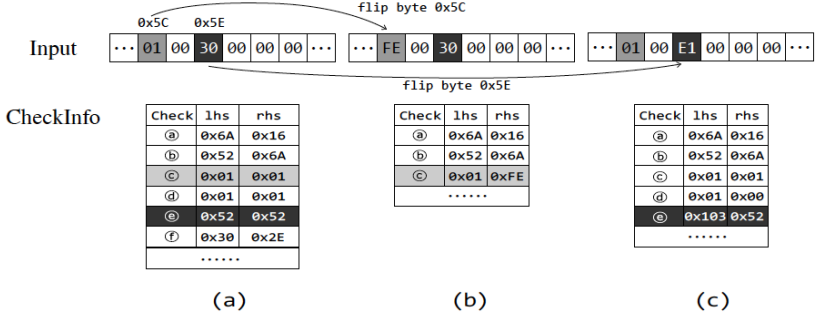


一个合法Zip文件的比特串

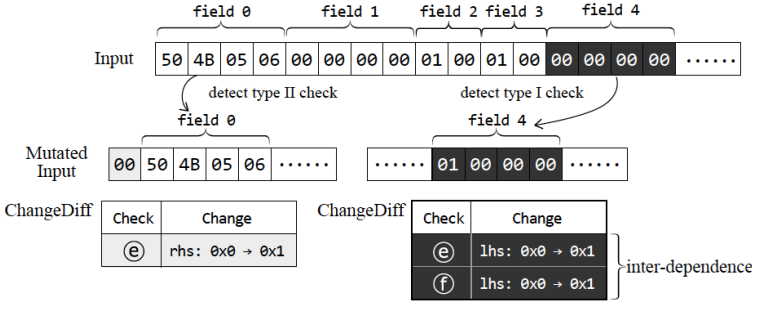
[1] You W, Liu X, Ma S, et al. SLF: Fuzzing without valid seed inputs[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 712-723.

基于变异的模糊测试

- Mutation-based：SLF（ICSE'19）[1]
  - 思想：分析程序中的语义检查、识别比特串中与影响语义检查的域（Field）、根据两者之间的关系制定变异策略



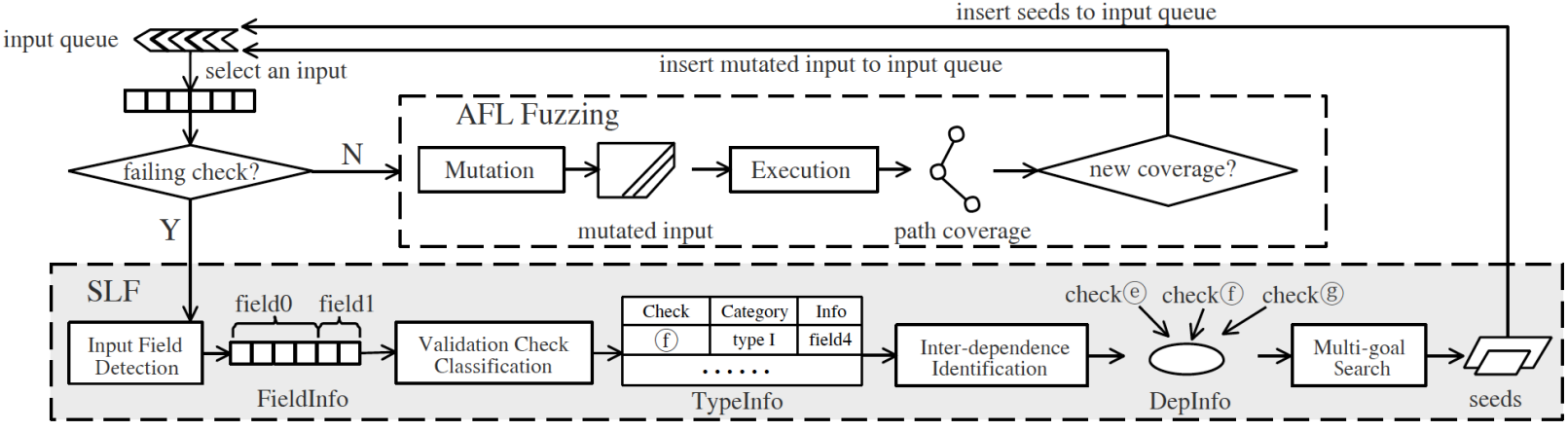
字节分组



类型推断与关系建立

[1] You W, Liu X, Ma S, et al. SLF: Fuzzing without valid seed inputs[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 712-723.

- Mutation-based：SLF（ICSE'19）[1]
  - 思想：分析程序中的语义检查、识别比特串中与影响语义检查的域（Field）、根据两者之间的关系制定变异策略

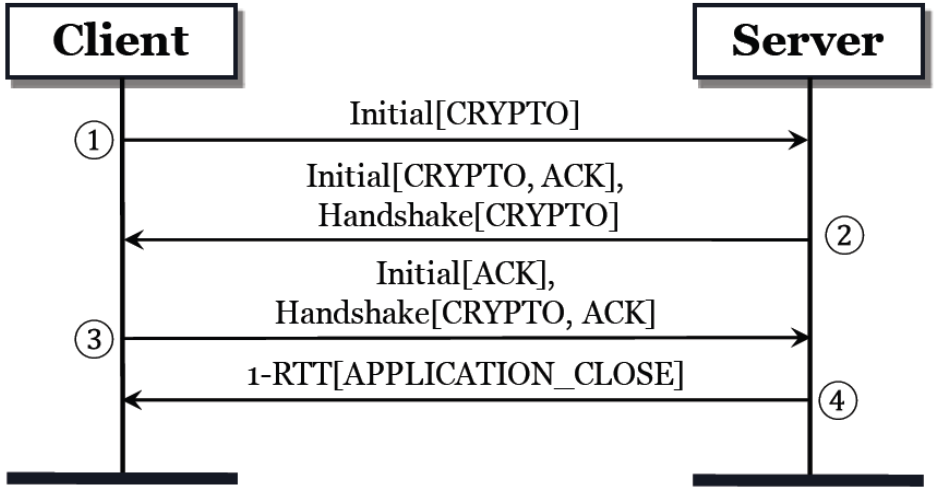


SLF技术流程概览

[1] You W, Liu X, Ma S, et al. SLF: Fuzzing without valid seed inputs[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 712-723.

基于生成的模糊测试

- Generation-based：基于一定的文法规则/结构信息
  - Data Model



QUIC服务器和客户端数据交换[1]

```
<StateModel name="QUIC" initialState="HANDSHAKE">
  <State name="HANDSHAKE">
    <Action type="output">
      <DataModel ref="Initial[CRYPTO]"/>
    </Action>
    <Action type="input">
      <DataModel ref="Initial[CRYPTO,ACK]+Handshake[CRYPTO]"/>
    </Action>
    <Action type="output">
      <DataModel ref="Initial[ACK]+Handshake[CRYPTO,ACK]"/>
    </Action>
  </State>
  ...
</StateModel>

<DataModel name="Initial[CRYPTO]">
  <Block name="Header">
    <Number name="info" size="8" value="cd" valueType="hex"/>
    <Number name="version" size="32" value="faceb002"
         valueType="hex" token="true"/>
    <Number name="DCID_length" size="8">
      <Relation type="size" of="DCID"/>
    </Number>
    <String name="DCID" nullTerminated="false"/>
    ...
  </Block>
  <Block name="CRYPTO"> ... </Block>
</DataModel>
```

Peach协议规格建模[1]

[1] Luo Z, Yu J, Zuo F, et al. Bleem: Packet sequence oriented fuzzing for protocol implementations[C]//32nd USENIX Security Symposium (USENIX Security 23). 2023: 4481-4498.

基于生成的模糊测试

- Generation-based：基于一定的文法规则/结构信息
  - Protocol（http, ftp, dtls, ...）

```
* Handshake Process/Protocol:
* -------------------------
*
* The protocol for the handshake involves the client sending an initial packet to the server,
* and the server responding with a unique 'Cookie' value, which the client has to respond with.
*
* Client - Initial Connect:
*
* [?:MagicHeader][2:SessionID][3:ClientID][HandshakeBit][RestartHandshakeBit]
* [8:MinVersion][8:CurVersion][8:HandshakePacketType][8:SentPacketCount][32:NetworkVersion]
* [16:NetworkFeatures][SecretIdBit][28:PacketSizeFiller][AlignPad][?:RandomData]
*                                                 --->
*                                                         Server - Stateless Handshake Challenge:
*
*                                                         [?:MagicHeader][2:SessionID][3:ClientID][HandshakeBit][RestartHandshakeBit]
*                                                         [8:MinVersion][8:CurVersion][8:HandshakePacketType][8:SentPacketCount][32:NetworkVersion]
*                                                         [16:NetworkFeatures][SecretIdBit][8:Timestamp][20:Cookie][AlignPad][?:RandomData]
*                                                 <---
* Client - Stateless Challenge Response:
*
* [?:MagicHeader][2:SessionID][3:ClientID][HandshakeBit][RestartHandshakeBit]
* [8:MinVersion][8:CurVersion][8:HandshakePacketType][8:SentPacketCount][32:NetworkVersion]
* [16:NetworkFeatures][SecretIdBit][8:Timestamp][20:Cookie][AlignPad][?:RandomData]
*                                                 --->
*                                                         Server:
*                                                         Ignore, or create UNetConnection.
*
*                                                         Server - Stateless Handshake Ack:
*
*                                                         [?:MagicHeader][2:SessionID][3:ClientID][HandshakeBit][RestartHandshakeBit]
*                                                         [8:MinVersion][8:CurVersion][8:HandshakePacketType][8:SentPacketCount][32:NetworkVersion]
*                                                         [16:NetworkFeatures][SecretIdBit][8:Timestamp][20:Cookie][AlignPad][?:RandomData]
*                                                 <---
* Client:
```

UE5专用服务器交互协议

- Generation-based：Inputs from Hell (TSE'19)[1]
  - Grammar
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入
    - 表示输入的分布：概率文法（Probabilistic Grammar）
    - 预期输入
      - 与概率文法相同：生成与文法表示的输入**相似**的测试输入
      - 与概率文法相反：生成与文法表示的输入**互补**的测试输入

[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

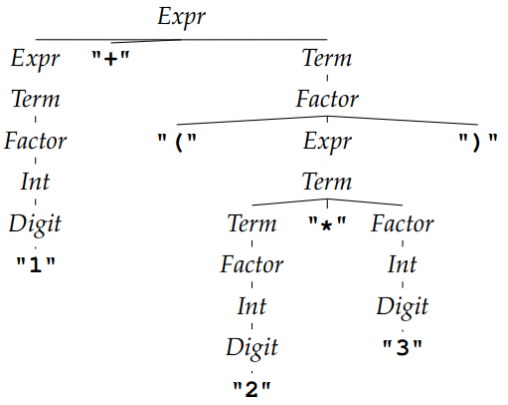
基于生成的模糊测试

- Generation-based：Inputs from Hell (TSE'19)[1]
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入

```
Expr → Term | Expr "+" Term | Expr "-" Term;
Term → Factor | Term "*" Factor
     | Term "/" Factor;
Factor → Int | "+" Factor
     | "-" Factor | "(" Expr ")";
Int → Digit Int | Digit;
Digit → "0" | "1" | "2" | "3" | ... | "9";
```

算数表达式的文法G

$I = 1 * (2 + 3)$

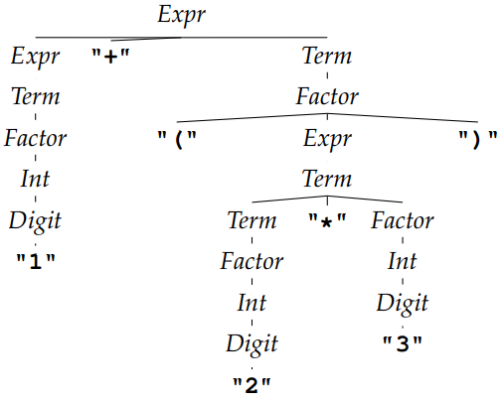


"$I = 1 + (2 * 3)$"的推导树（Derivation Tree）

[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

基于生成的模糊测试

- Generation-based：Inputs from Hell (TSE'19)[1]
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入



"$I = 1 + (2 * 3)$" 的推导树（Derivation Tree）

```
Expr   → 66.7% Term | 33.3% Expr "+" Term
       | 0% Expr "−" Term;
Term   → 75% Factor | 25% Term "*" Factor
       | 0% Term "/" Factor;
Factor → 75% Int | 0% "+" Factor
       | 0% "−" Factor | 25% "(" Expr ")";
Int    → 0% Digit Int | 100% Digit;
Digit  → 0% "0" | 33.3% "1" | 33.3% "2"
       | 33.3% "3" | 0% "4" | 0% "5"
       | 0% "6" | 0% "7" | 0% "8" | 0% "9";
```

用于表示I产生的输入分布的概率文法G_p

[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

- Generation-based：Inputs from Hell (TSE'19)[1]
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入

```
Expr → 66.7% Term | 33.3% Expr "+" Term
     | 0% Expr "-" Term;
Term → 75% Factor | 25% Term "*" Factor
     | 0% Term "/" Factor;
Factor → 75% Int | 0% "+" Factor
     | 0% "-" Factor | 25% "(" Expr ")";
Int → 0% Digit Int | 100% Digit;
Digit → 0% "0" | 33.3% "1" | 33.3% "2"
     | 33.3% "3" | 0% "4" | 0% "5"
     | 0% "6" | 0% "7" | 0% "8" | 0% "9";
```

Input Generation

```
(2 * 3)
2 + 2 + 1 * (1) + 2
((3 * 3))
3 * (((3 + 3 + 3) * (2 * 3 + 3))) * (3)
3 * 1 * 3
((3) + 2 + 2 * 1) * (1)
1
((2)) + 3
```

用于表示I产生的输入分布的概率文法G_p

概率文法G_p产生的相似输入

[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

基于生成的模糊测试

- Generation-based：Inputs from Hell (TSE'19)[1]
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入

```
Expr   → 66.7% Term | 33.3% Expr "+" Term
       | 0% Expr "−" Term;
Term   → 75% Factor | 25% Term "*" Factor
       | 0% Term "/" Factor;
Factor → 75% Int | 0% "+" Factor
       | 0% "−" Factor | 25% "(" Expr ")";
Int    → 0% Digit Int | 100% Digit;
Digit  → 0% "0" | 33.3% "1" | 33.3% "2"
       | 33.3% "3" | 0% "4" | 0% "5"
       | 0% "6" | 0% "7" | 0% "8" | 0% "9";
```

Grammar Inversion

```
Expr   → 0% Term | 0% Expr "+" Term
       | 100% Expr "−" Term;
Term   → 0% Factor | 0% Term "*" Factor
       | 100% Term "/" Factor;
Factor → 0% Int | 50% "+" Factor
       | 50% "−" Factor | 0% "(" Expr ")";
Int    → 100% Digit Int | 0% Digit;
Digit  → 14.3% "0" | 0% "1" | 0% "2" | 0% "3"
       | 14.3% "4" | 14.3% "5" | 14.3% "6"
       | 14.3% "7" | 14.3% "8" | 14.3% "9";
```

用于表示I产生的输入分布的概率文法G_p

表示与I相反分布的互补文法G_p^{-1}

[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

基于生成的模糊测试

- Generation-based：Inputs from Hell (TSE'19)[1]
  - 思想：挖掘已有的测试输入，得到现有的测试输入分布。根据该分布进行输入生成以得到预期的测试输入

```
Expr → 0% Term | 0% Expr "+" Term
     | 100% Expr "-" Term;
Term → 0% Factor | 0% Term "*" Factor
     | 100% Term "/" Factor;
Factor → 0% Int | 50% "+" Factor
       | 50% "-" Factor | 0% "(" Expr ")";
Int → 100% Digit Int | 0% Digit;
Digit → 14.3% "0" | 0% "1" | 0% "2" | 0% "3"
      | 14.3% "4" | 14.3% "5" | 14.3% "6"
      | 14.3% "7" | 14.3% "8" | 14.3% "9";
```

Input Generation

```
+5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
-6 / 9 / 5 / 8 - +7 / -9 / 6 - 4 - 4 - 6
+8 / ++8 / 5 / 4 / 0 - 5 - 4 / 8 - 8 - 8
-9 / -5 / 9 / 4 - -9 / 0 / 5 - 8 / 4 - 6
++7 / 9 / 5 - +8 / +9 / 7 / 7 - 6 - 8 - 4
-+6 / -8 / 9 / 6 - 5 / 0 - 5 - 8 - 0 - 5
```

表示与I相反分布的互补文法G_p^{-1}

互补文法G_p^{-1}产生的相反输入

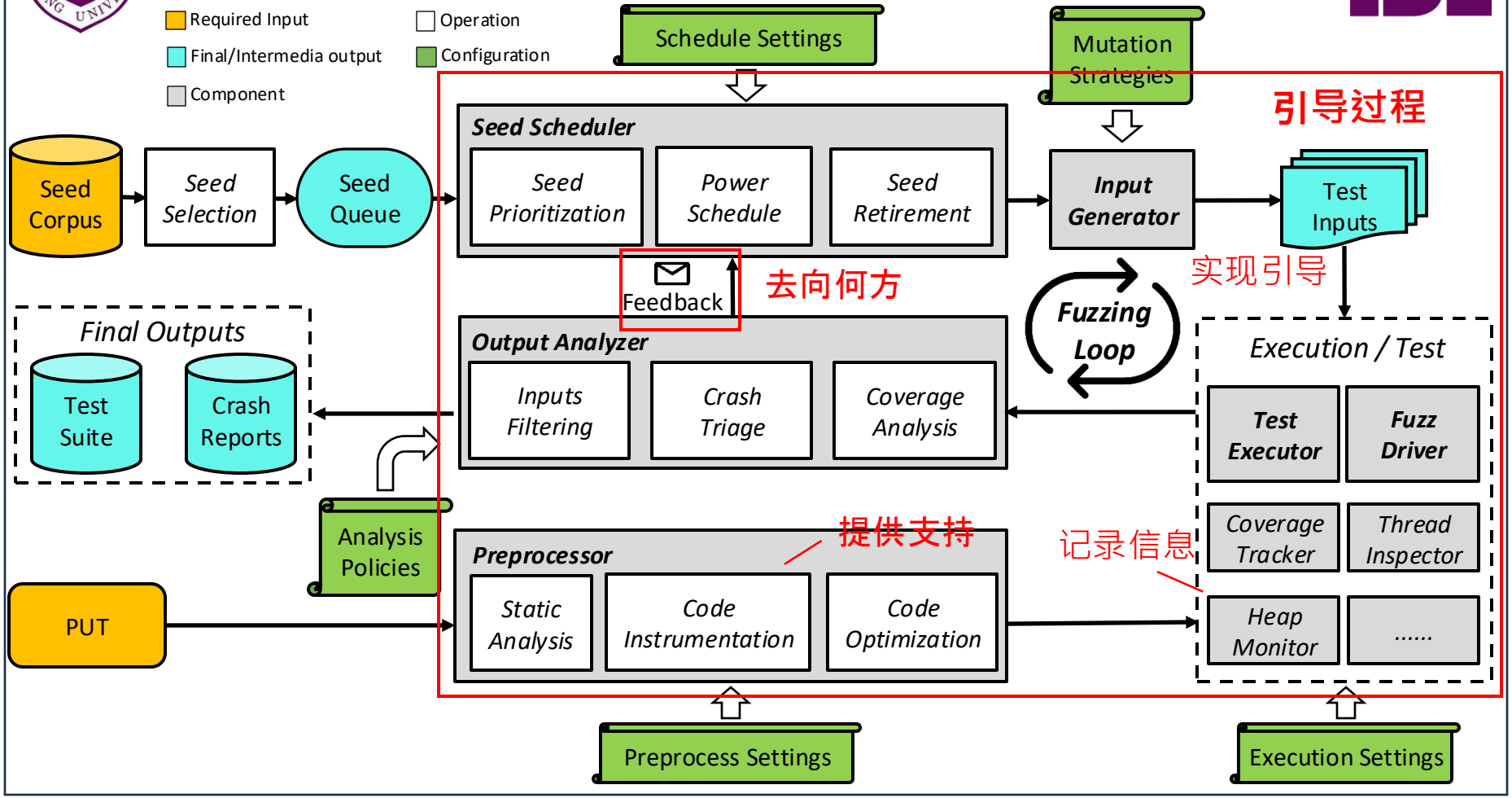
[1] Soremekun E, Pavese E, Havrikov N, et al. Inputs from hell learning input distributions for grammar-based test generation[J]. IEEE Transactions on Software Engineering, 2020.

按照引导方式分类

- 按照**引导方式**
  - Test Execution、Output Analysis
  - Search-based：将测试转化为**搜索问题**，以代码覆盖率作为指示器、以启发式算法（类遗传算法）为核心，将测试导向更高覆盖的方向 → CGF
  - Gradient-based：将测试转化为**优化问题**，以最大化缺陷发掘输入为目标构建目标函数，迭代求最优解 → 退而求覆盖率
  - 模糊测试中的**马太效应**（Matthew effect）[1]：好的种子输入能够演化产生品质良好的子代测试输入

[1] Chen H, Guo S, Xue Y, et al. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2325-2342.

按照引导方式分类

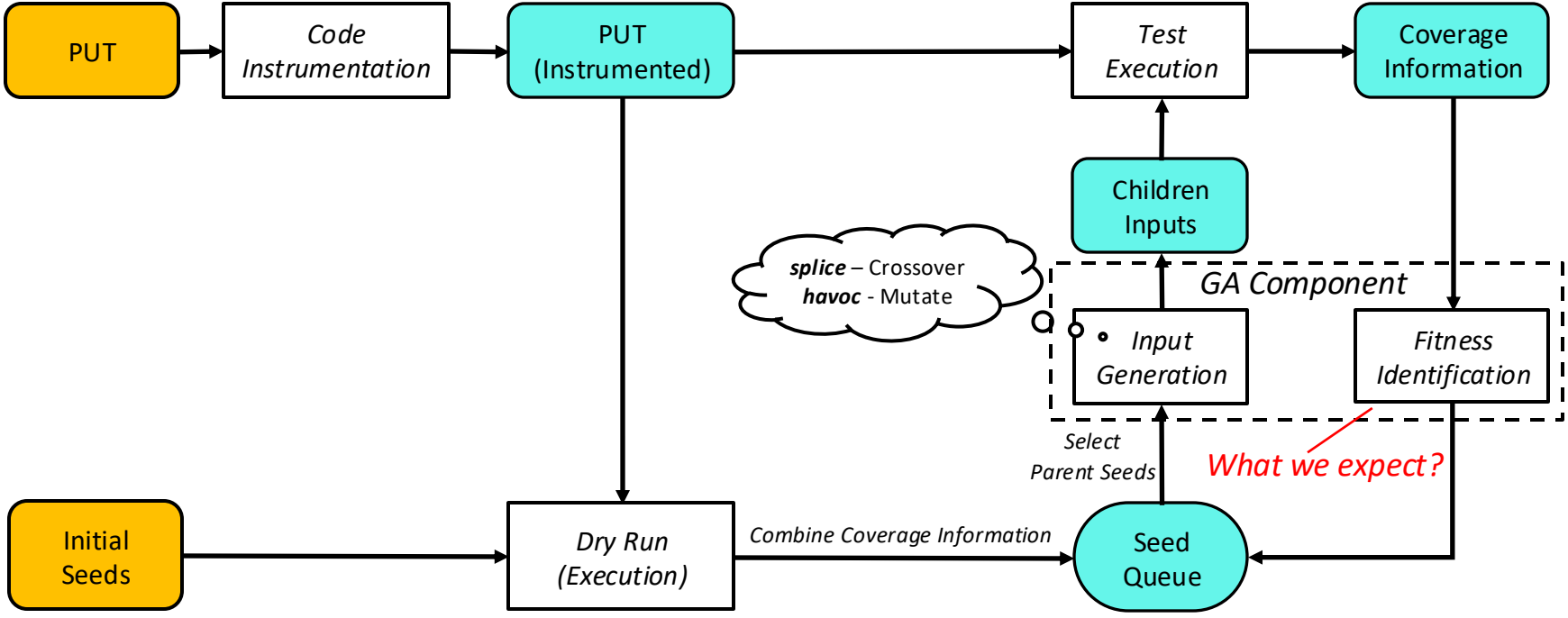

Required Input
Operation
Final/Intermedia output
Configuration
Component
Schedule Settings
Mutation Strategies
引导过程
Seed Corpus
Seed Selection
Seed Queue
Seed Scheduler
Seed Prioritization
Power Schedule
Seed Retirement
Input Generator
Test Inputs
Feedback
去向何方
实现引导
Fuzzing Loop
Final Outputs
Test Suite
Crash Reports
Output Analyzer
Inputs Filtering
Crash Triage
Coverage Analysis
Execution / Test
Test Executor
Fuzz Driver
Coverage Tracker
Thread Inspector
Heap Monitor
.....
Analysis Policies
提供支持
记录信息
Preprocessor
Static Analysis
Code Instrumentation
Code Optimization
PUT
Preprocess Settings
Execution Settings

基于搜索的模糊测试

- Search-based：Fuzzing as Search Problem
  - 核心：将模糊测试过程建模为搜索问题，根据模型构造启发式算法（Heuristic）解决问题
  - 启发式：遗传算法-AFL[1]、马尔科夫链-AFLFast[2]、信息熵-Entropic[3]、多臂老虎机问题-EcoFuzz[4]

[1] AFL README：https://github.com/google/AFL/blob/master/README.md
[2] Böhme M, Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as markov chain[C]//Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. 2016: 1032-1043.
[3] Böhme M, Manès V J M, Cha S K. Boosting fuzzer efficiency: An information theoretic perspective[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 678-689.
[4] Yue T, Wang P, Tang Y, et al. {EcoFuzz}: Adaptive {Energy-Saving} Greybox Fuzzing as a Variant of the Adversarial {Multi-Armed} Bandit[C]//29th USENIX Security Symposium (USENIX Security 20). 2020: 2307-2324.

- Search-based：AFL，基于插装的遗传算法



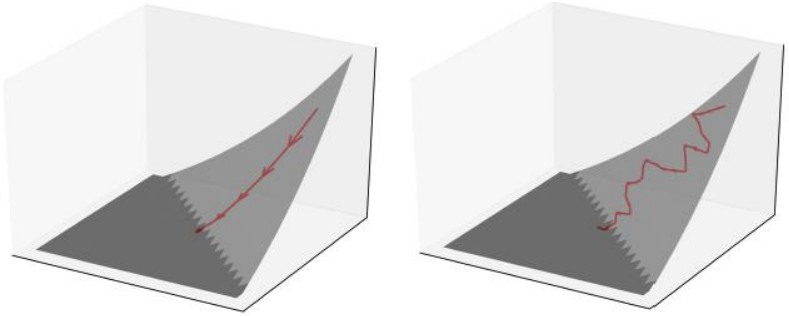
基于梯度的模糊测试

- Gradient-based：Fuzzing as Optimization Problem
  - 核心：将模糊测试过程建模为优化问题，问题的目标是最大化缺陷发掘数量，利用梯度下降算法持续求解最优解
  - 目标退阶：缺陷离散分布且无法预知 → 替换为代码覆盖
  - 应用DL技术-Neuzz[1] & MTFuzz[2]；利用梯度下降取代符号执行的约束求解过程-Angora[3]

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.
[2] She D, Krishna R, Yan L, et al. MTFuzz: fuzzing with a multi-task neural network[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 737-749.
[3] Chen P, Chen H. Angora: Efficient fuzzing by principled search[C]//2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018: 711-725.

基于梯度的模糊测试

- Gradient-based：Neuzz[1]
  - 核心思路：将模糊测试建模为无约束优化问题，利用梯度下降算法优化模糊测试过程
  - 动机：Gradient Descend > Evolutionary Algorithm



(a) gradient descent　　(b) evolutionary algorithm

梯度下降和演进算法在高阶优化问题上面的表现

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

基于梯度的模糊测试

- Gradient-based：Neuzz[1]
  - 挑战
    - 如何计算反馈 → 如何计算梯度
    - 优化目标（缺陷发现数目或覆盖率）是**不连续的**，无法直接适配梯度下降算法 → 引入程序平滑（Program Smoothing）技术

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

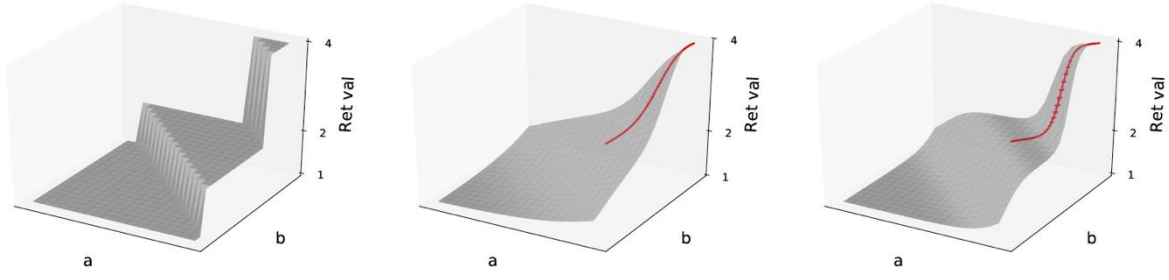
基于梯度的模糊测试

- Gradient-based：Neuzz[1]
  - 程序平滑（Neural Program Smoothing）：消除目标函数中的不连续性
    - 黑盒程序平滑：简单易用，但是会产生较大的近似错误
    - 白盒程序平滑：依赖符号执行（Symbolic Execution）和抽象解释技术（Abstract），会产生较大的额外开销

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

基于梯度的模糊测试

- Gradient-based：Neuzz[1]
  - 程序平滑（Neural Program Smoothing）：消除目标函数中的不连续性
    - **神经程序平滑**（Neural Program Smoothing）：利用NN模型模拟程序行为



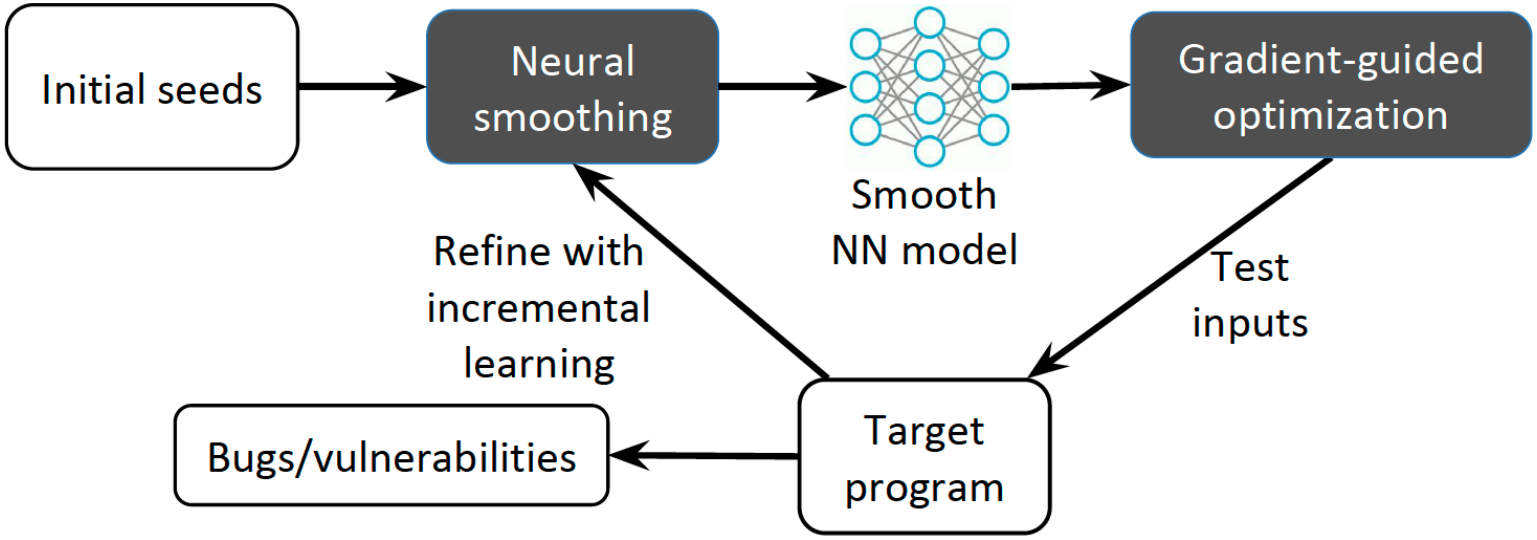
(a) Original (b) NN smoothing (c) NN smoothing + refining

Neuzz神经程序平滑

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

基于梯度的模糊测试

- Gradient-based：Neuzz[1]



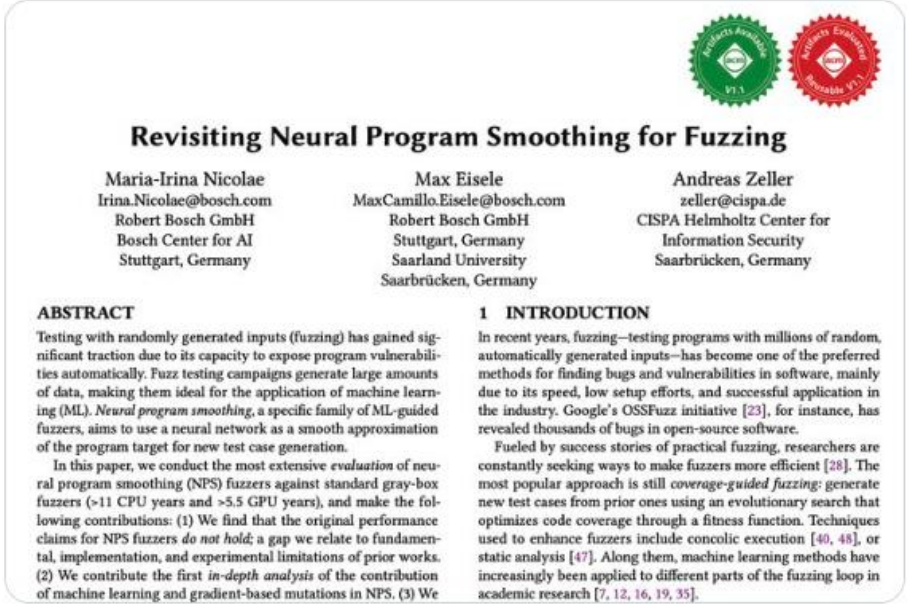
Neuzz工作流程概览

[1] She D, Pei K, Epstein D, et al. Neuzz: Efficient fuzzing with neural program smoothing[C]//2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019: 803-817.

关于Neuzz的争论

Andreas Zeller
@AndreasZeller

In one year of work, @ririnicolae, @MaxCamillo, and I have deconstructed the controversial NEUZZ ML-based fuzzer and found that its original performance claims do not hold. Looking forward to lively discussions today 12:15 at @FSEconf! dl.acm.org/doi/10.1145/36...



Revisiting Neural Program Smoothing for Fuzzing

Maria-Irina Nicolae
Irina.Nicolae@bosch.com
Robert Bosch GmbH
Bosch Center for AI
Stuttgart, Germany

Max Eisele
MaxCamillo.Eisele@bosch.com
Robert Bosch GmbH
Stuttgart, Germany
Saarland University
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

ABSTRACT

Testing with randomly generated inputs (fuzzing) has gained significant traction due to its capacity to expose program vulnerabilities automatically. Fuzz testing campaigns generate large amounts of data, making them ideal for the application of machine learning (ML). *Neural program smoothing*, a specific family of ML-guided fuzzers, aims to use a neural network as a smooth approximation of the program target for new test case generation.

In this paper, we conduct the most extensive *evaluation* of neural program smoothing (NPS) fuzzers against standard gray-box fuzzers (>11 CPU years and >5.5 GPU years), and make the following contributions: (1) We find that the original performance claims for NPS fuzzers *do not hold*; a gap we relate to fundamental, implementation, and experimental limitations of prior works. (2) We contribute the first *in-depth analysis* of the contribution of machine learning and gradient-based mutations in NPS. (3) We

1 INTRODUCTION

In recent years, fuzzing—testing programs with millions of random, automatically generated inputs—has become one of the preferred methods for finding bugs and vulnerabilities in software, mainly due to its speed, low setup efforts, and successful application in the industry. Google's OSSFuzz initiative [23], for instance, has revealed thousands of bugs in open-source software.

Fueled by success stories of practical fuzzing, researchers are constantly seeking ways to make fuzzers more efficient [28]. The most popular approach is still *coverage-guided fuzzing*: generate new test cases from prior ones using an evolutionary search that optimizes code coverage through a fitness function. Techniques used to enhance fuzzers include concolic execution [40, 48], or static analysis [47]. Along them, machine learning methods have increasingly been applied to different parts of the fuzzing loop in academic research [7, 12, 16, 19, 35].

引用次数 查看全部

	总计	2019 年至今
引用	25573	9099
h 指数	69	44
i10 指数	170	122

开放获取的出版物数量 查看全部

4 篇文章 59 篇文章

无法查看的文章 可查看的文章

根据资助方的强制性开放获取政策

[1] Nicolae M I, Eisele M, Zeller A. Revisiting neural program smoothing for fuzzing[C]//Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2023: 133-145.

- Replicability Studies

Replicability Studies

ISSTA would like to encourage researchers to replicate results from previous papers. A replicability study must go beyond simply re-implementing an algorithm and/or re-running the artifacts provided by the original paper. It should at the very least apply the approach to new, significantly broadened inputs. Particularly, replicability studies are encouraged to target techniques that previously were evaluated only on proprietary subject programs or inputs. A replicability study should clearly report on results that the authors were able to replicate as well as on aspects of the work that were not replicable. In the latter case, authors are encouraged to make an effort to communicate or collaborate with the original paper's authors to determine the cause for any observed discrepancies and, if possible, address them (e.g., through minor implementation changes). We explicitly encourage authors to not focus on a single paper/artifact only, but instead to perform a comparative experiment of multiple related approaches.

In particular, replicability studies should follow the ACM guidelines on replicability (different team, different experimental setup): The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently. This means that it is also insufficient to focus on reproducibility (i.e., different team, same experimental setup) alone. Replicability Studies will be evaluated according to the following standards:

- Depth and breadth of experiments
- Clarity of writing
- Appropriateness of conclusions
- Amount of useful, actionable insights
- Availability of artifacts

We expect replicability studies to clearly point out the artifacts the study is built on, and to submit those artifacts to the artifact evaluation. Artifacts evaluated positively will be eligible to obtain the prestigious Results Reproduced badge.

When Revisiting is Wrong! Rebuttal: Revisiting Neural Program Smoothing for Fuzzing

Dongdong She*, Kexin Pei†, Junfeng Yang*, Baishakhi Ray*, Suman Jana*
*Hong Kong University of Science and Technology
†The University of Chicago
*Columbia University

Abstract

MLFuzz, a work accepted at ACM FSE 2023, revisits the performance of a machine learning-based fuzzer, NEUZZ. We demonstrate that its main conclusion is entirely wrong due to several fatal bugs in the implementation and wrong evaluation setups, including an initialization bug in persistent mode, a program crash, an error in training dataset collection, and a mistake in fuzzing result collection. Additionally, MLFuzz uses noisy training datasets without sufficient data cleaning and preprocessing, which contributes to a drastic performance drop in NEUZZ. We address these issues and provide a corrected implementation and evaluation setup, showing that NEUZZ consistently performs well over AFL on the FuzzBench dataset. Finally, we reflect on the evaluation methods used in MLFuzz and offer practical advice on fair and scientific fuzzing evaluations.

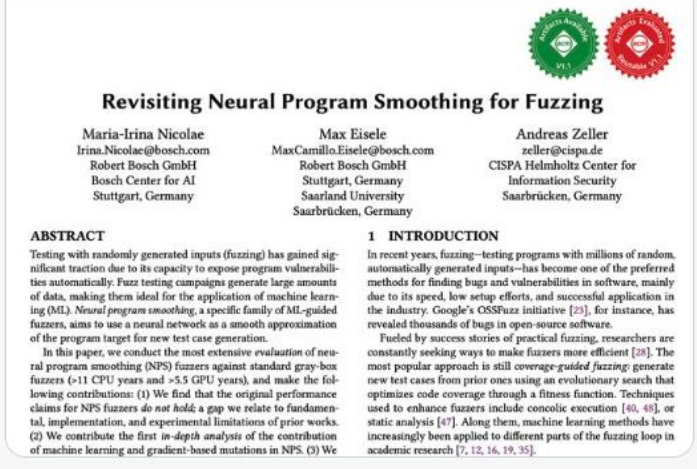
[1] She D, Pei K, Yang J, et al. Comment on Revisiting Neural Program Smoothing for Fuzzing[J]. arXiv preprint arXiv:2409.04504, 2024.

关于Neuzz的争论

Dongdong She @DongdongShe · Dec 6, 2023

Fuzzing throughput is a critical yet often ignored factor in fuzzing evaluation. Carelessly comparing a file-retrieval fuzzer with an in-memory fuzzer, even a well-known researcher would make such a mistake and draw a completely WRONG conclusion in a top-tier conference paper.

> **Andreas Zeller** @AndreasZeller · Dec 5, 2023
>
> In one year of work, @ririnicolae, @MaxCamillo, and I have deconstructed the controversial NEUZZ ML-based fuzzer and found that its original performance claims do not hold. Looking forward to lively discussions today 12:15 at @FSEconf! dl.acm.org/doi/10.1145/36...



Revisiting Neural Program Smoothing for Fuzzing

Maria-Irina Nicolae
Irina.Nicolae@bosch.com
Robert Bosch GmbH
Bosch Center for AI
Stuttgart, Germany

Max Eisele
MaxCamillo.Eisele@bosch.com
Robert Bosch GmbH
Stuttgart, Germany
Saarland University
Saarbrücken, Germany

Andreas Zeller
zeller@cispa.de
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

ABSTRACT

Testing with randomly generated inputs (fuzzing) has gained significant traction due to its capacity to expose program vulnerabilities automatically. Fuzz testing campaigns generate large amounts of data, making them ideal for the application of machine learning (ML). *Neural program smoothing*, a specific family of ML-guided fuzzers, aims to use a neural network as a smooth approximation of the program target for new test case generation.

In this paper, we conduct the most extensive *evaluation* of neural program smoothing (NPS) fuzzers against standard gray-box fuzzers (>11 CPU years and >5.5 GPU years), and make the following contributions: (1) We find that the original performance claims for NPS fuzzers *do not hold*; a gap we relate to fundamental, implementation, and experimental limitations of prior works. (2) We contribute the first *in-depth analysis* of the contribution of machine learning and gradient-based mutations in NPS. (3) We

1 INTRODUCTION

In recent years, fuzzing—testing programs with millions of random, automatically generated inputs—has become one of the preferred methods for finding bugs and vulnerabilities in software, mainly due to its speed, low setup efforts, and successful application in the industry. Google's OSSFuzz initiative [23], for instance, has revealed thousands of bugs in open-source software.

Fueled by success stories of practical fuzzing, researchers are constantly seeking ways to make fuzzers more efficient [28]. The most popular approach is still *coverage-guided fuzzing*: generate new test cases from prior ones using an evolutionary search that optimizes code coverage through a fitness function. Techniques used to enhance fuzzers include concolic execution [40, 48], or static analysis [47]. Along them, machine learning methods have increasingly been applied to different parts of the fuzzing loop in academic research [7, 12, 16, 19, 35].

3 · 16 · 4.5K

Dongdong She @DongdongShe · Dec 6, 2023

2. Report the fuzzing throughput (total number of mutations) along with code coverage. Ensure all fuzzer read testcase from the same approach (either throughput file or through memory). Comparing a file-retrieval fuzzer with an in-memory fuzzer can lead to invalid results.

1 · 3 · 2.7K

Dongdong She @DongdongShe · Dec 6, 2023

3. Open-source your seed corpus along with your source code. Fuzzing is a continuous optimization process. Without the same starting point (seed corpus), it's hard to reproduce the fuzzing result.

4 · 2.5K

Dongdong She @DongdongShe · Dec 6, 2023

How do you conduct a SCIENTIFIC evaluation for fuzzing research?
Blindly run different fuzzers regardless of their settings for 24 hours and compare their raw coverage number?
Here are a few tips I found that fuzzer practitioners and researchers often ignore.

2 · 1 · 15 · 4K

Dongdong She @DongdongShe · Dec 6, 2023

1. Ensure all fuzzers are testing the SAME binary. Otherwise, please do a coverage replay on a standard binary before comparing the raw number. Be careful with the discrepancy in AFL coverage, AFL++ coverage, LLVM sanitizer coverage, LLVM sanitizer coverage with no prune feature.

1 · 3 · 1.9K

关于Neuzz的争论

Dongdong She
@DongdongShe

What happens if you write buggy code and misconfigure the experimental setup when evaluating a fuzzer's performance? Wrong and misleading conclusion!

We found several fatal bugs and wrong experimental settings in MLFuzz (arxiv.org/pdf/2309.16618, a revisit work on NEUZZ published on a top tier software engineering conference ASE 2023, @AndreasZeller, @ASE_conf). These following bugs lead to wrong and misleading conclusions in MLFuzz.

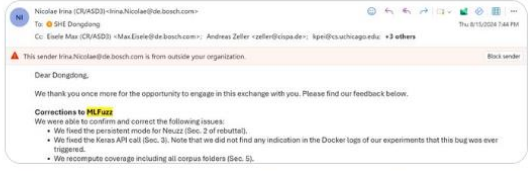
- An initialization bug ⇒ Failure setup of persistent mode fuzzing.
- A program crash ⇒ Unexpected early termination of NEUZZ.
- An error in training dataset collection ⇒ A poorly-trained neural network model.
- An error in result collection ⇒ Incomplete code coverage report

We confirmed these bugs with the MLFuzz's authors and write a rebuttal paper(arxiv.org/pdf/2409.04504) to explain the errors in MLFuzz and summarize the lessons on a fair and scientific fuzzing experiment/revisit.

1. Ensure the correctness of code implementation. Careful and rigorous debugging is needed. If you would like to patch a prior work, double-check your setting or patch is correct and seek help from original developer if needed. MLFuzz introduced 3 implementation bugs that led to wrong experimental results and conclusions.
2. Diverse benchmark selection. Try to evaluate your fuzzer on multiple benchmarks, like FuzzBench, Magma, UniFuzz.
3. Uniform code coverage metric. Covert different code coverage metrics like AFL XOR hash, LLVM coverage sanitizer (pruned), LLVM coverage sanitizer (no-pruned), AFL++ code coverage into a uniform one by replaying
4. Complete test case collection. Be sure to collect all the test cases

Dongdong She @DongdongShe · Sep 17
Ep5. Rebuttal MLFuzz

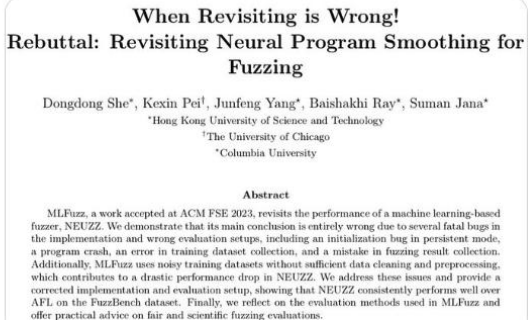
Thanks Irina's response. We never heard back from you and @AndreasZeller since last month when we sent the last email to ask if you guys were willing to write an errata of MLFuzz to acknowledge the bugs and wrong conclusion. So **I am happy to communicate**
Show more


Nicolae Irina (CR/ASD3)<irina.Nicolae@de.bosch.com>
To: SHE Dongdong
Cc: Eisele Max (CR/ASD3) <Max.Eisele@de.bosch.com>; Andreas Zeller <zeller@cispa.de>; kpei@cs.uchicago.edu; +3 others
Thu 8/15/2024 7:44 PM
This sender Irina.Nicolae@de.bosch.com is from outside your organization.
Block sender
Dear Dongdong,
We thank you once more for the opportunity to engage in this exchange with you. Please find our feedback below.
Corrections to MLFuzz
We were able to confirm and correct the following issues:
- We fixed the persistent mode for Neuzz (Sec. 2 of rebuttal).
- We fixed the Keras API call (Sec. 3). Note that we did not find any indication in the Docker logs of our experiments that this bug was ever triggered.
- We recompute coverage including all corpus folders (Sec. 5).

1 9 36 12K

Dongdong She @DongdongShe · Sep 14
What happens if you write buggy code and misconfigure the experimental setup when evaluating a fuzzer's performance? Wrong and misleading conclusion!

We found several fatal bugs and wrong experimental settings in MLFuzz (arxiv.org/pdf/2309.16618, a revisit work on NEUZZ published
Show more


When Revisiting is Wrong!
Rebuttal: Revisiting Neural Program Smoothing for Fuzzing

Dongdong She*, Kexin Pei†, Junfeng Yang*, Baishakhi Ray*, Suman Jana*
*Hong Kong University of Science and Technology
†The University of Chicago
*Columbia University

Abstract

MLFuzz, a work accepted at ACM FSE 2023, revisits the performance of a machine learning-based fuzzer, NEUZZ. We demonstrate that its main conclusion is entirely wrong due to several fatal bugs in the implementation and wrong evaluation setups, including an initialization bug in persistent mode, a program crash, an error in training dataset collection, and a mistake in fuzzing result collection. Additionally, MLFuzz uses noisy training datasets without sufficient data cleaning and preprocessing, which contributes to a drastic performance drop in NEUZZ. We address these issues and provide a corrected implementation and evaluation setup, showing that NEUZZ consistently performs well over AFL on the FuzzBench dataset. Finally, we reflect on the evaluation methods used in MLFuzz and offer practical advice on fair and scientific fuzzing evaluations.

复现与争论

PLDI 2022 (series) / PLDI Research Papers /

“Synthesizing Input Grammars”: A Replication Study

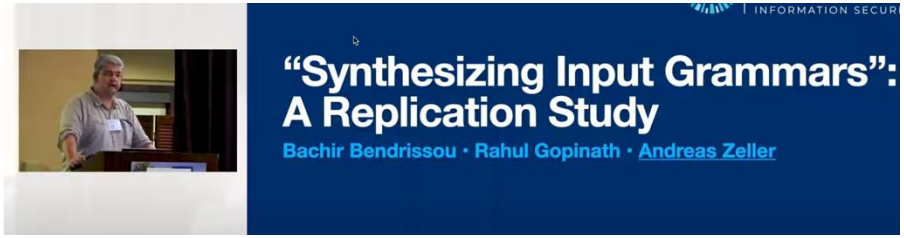
16:30 20m Talk ☆ **“Synthesizing Input Grammars”: A Replication Study**

Bachir Bendrissou CISPA Helmholtz Center for Information Security, Rahul Gopinath University of Sydney, Andreas Zeller CISPA Helmholtz Center for Information Security

DOI Pre-print

16:50 5m Talk ☆ **Response by authors of "Synthesizing Input Grammars"**

Osbert Bastani University of Pennsylvania



[1] https://pldi22.sigplan.org/details/pldi-2022-pldi/49/-Synthesizing-Input-Grammars-A-Replication-Study

复现与争论

Andreas Zeller

Andreas Zeller is faculty at the CISPA Helmholtz Center for Information Security and professor for Software Engineering at Saarland University. His research on automated debugging, mining software archives, specification mining, and security testing has proven highly influential. Zeller is one of the few researchers to have received two ERC Advanced Grants, most recently for his S3 project. Zeller is an ACM Fellow and holds an ACM SIGSOFT Outstanding Research Award.

10 October 2019

When Results Are All That Matters: The Case of the Angora Fuzzer

by Andreas Zeller and Sascha Just; with Kai Greshake

The Case

"Fuzzers" are programs that generate random inputs to trigger failures in tested programs. They are easily deployed and have found numerous vulnerabilities in various programs. The last decade has seen a tremendous increase in fuzzing research [4].

In 2018, Chen and Chen published the paper "Angora: Efficient Fuzzing by Principled Search" [1] featuring a new gray box, mutation-based fuzzer called Angora. Angora presented extraordinary results in its effectiveness and shines with its stellar performance on the Lava-M benchmark [3], dramatically outperforming all competition.

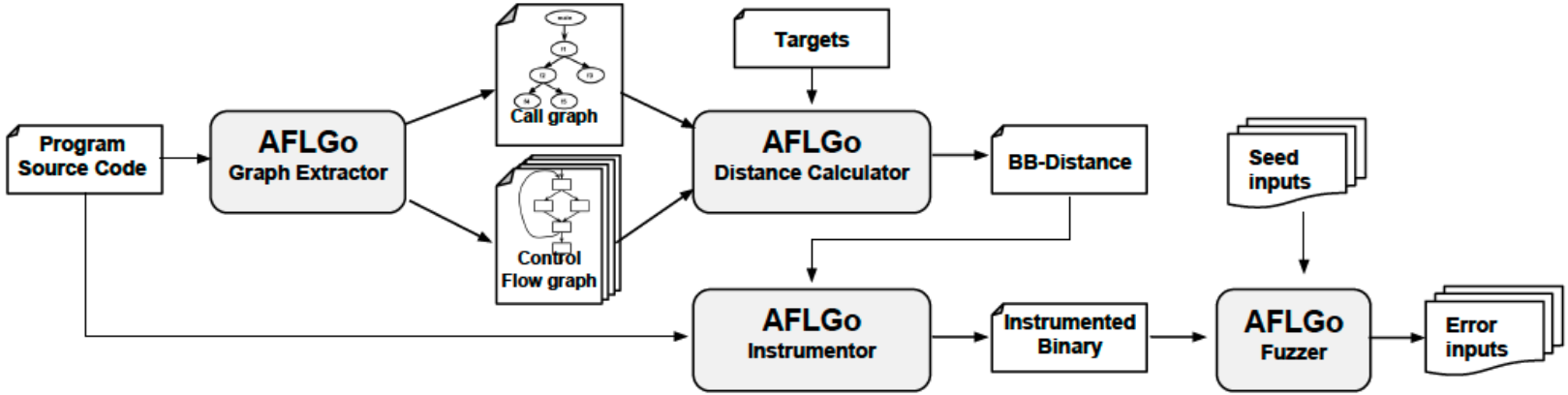
The reason behind this breakthrough and leap in effectiveness towards deeper penetration of target programs was cited as the combination of four techniques: scalable byte-level taint tracking, context-sensitive branch count, input length exploration, and search based on gradient descent. The former three techniques had already been explored in earlier work; but the novel key contribution, as also indicated by the paper title, was the *Gradient Descent* approach to solve constraints modeling branch conditions as functions and approximating their gradient.

[1] https://andreas-zeller.info/2019/10/10/when-results-are-all-that-matters-case.html

按照测试目的分类

- 按照**测试的目的**
  - 非定向模糊测试：**Wider and Deeper**
    - 目标：验证程序的正确性，检测程序中潜在的缺陷
  - 定向模糊测试：**Directed and Targeted**
    - 目标：针对程序中的某个目标位置进行快而有效的测试
    - 场景：缺陷复现、补丁检验、静态分析报告验证
    - 分类：白盒、**灰盒**

- AFLGo[1]：定向灰盒模糊测试
  - 基本思路：Distance + Annealing-based Scheduling，为更靠近目标位置的种子分配更多的能量



AFLGo整体流程

[1] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329-2344.

- AFLGo[1]：定向灰盒模糊测试
  - 基本思路：Distance + Annealing-based Scheduling，为更靠近目标位置的种子分配更多的能量

$$d_b(m, T_b) = \begin{cases} 0 & \text{if } m \in T_b \\ c \cdot \min_{n \in N(m)} (d_f(n, T_f)) & \text{if } m \in T \\ \left[\sum_{t \in T} (d_b(m,t) + d_b(t,T_b))^{-1} \right]^{-1} & \text{otherwise} \end{cases}$$

$$d(s, T_b) = \frac{\sum_{m \in \xi(s)} d_b(m, T_b)}{|\xi(s)|} \qquad \tilde{d}(s, T_b) = \frac{d(s, T_b) - \text{minD}}{\text{maxD} - \text{minD}}$$

AFLGo距离计算

$$T_{\text{exp}} = T_0 \cdot \alpha^k \tag{7}$$

$$0.05 = \alpha^{k_x} \quad \text{for } T_{\text{exp}} = 0.05; k = k_x \text{ in Eq. (7)} \tag{8}$$

$$k_x = \log(0.05)/\log(\alpha) \quad \text{solving for } k_x \text{ in Eq. (8)} \tag{9}$$

$$T_{\text{exp}} = \alpha^{\frac{t}{t_x}\frac{\log(0.05)}{\log(\alpha)}} \quad \text{for } k = \frac{t}{t_x} k_x \text{ in Eq. (7)} \tag{10}$$

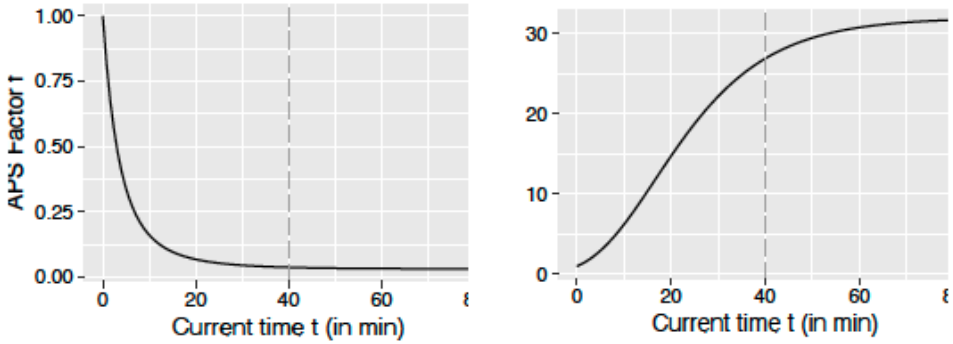
$$= 20^{-\frac{t}{t_x}} \quad \text{simplifying Eq. (10)} \tag{11}$$

$$p(s, T_b) = (1 - \tilde{d}(s, T_b)) \cdot (1 - T_{\text{exp}}) + 0.5 T_{\text{exp}} \tag{12}$$

AFLGo模拟退火调度

[1] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329-2344.

- AFLGo[1]：定向灰盒模糊测试
  - 基本思路：Distance + Annealing-based Scheduling，为更靠近目标位置的种子分配更多的能量



(a) Distance $\tilde{d}(s, T_b) = 1$ (b) Distance $\tilde{d}(s, T_b) = 0$

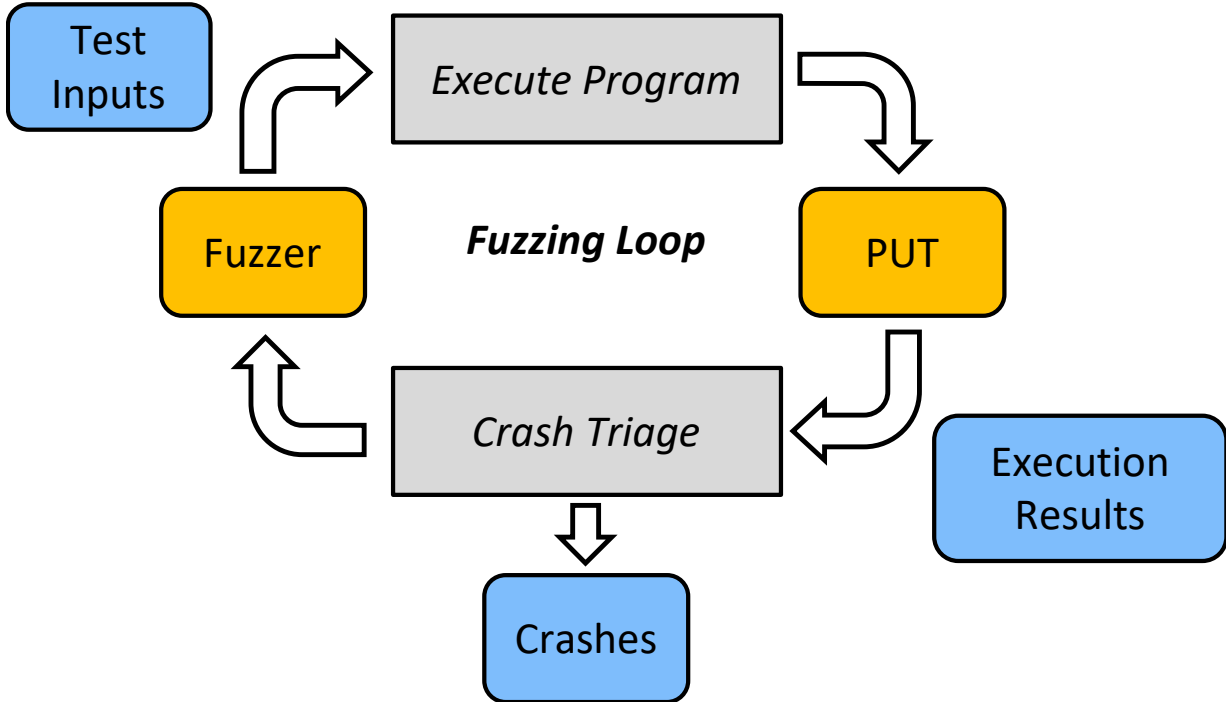
能量分配趋势

[1] Böhme M, Pham V T, Nguyen M D, et al. Directed greybox fuzzing[C]//Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017: 2329-2344.

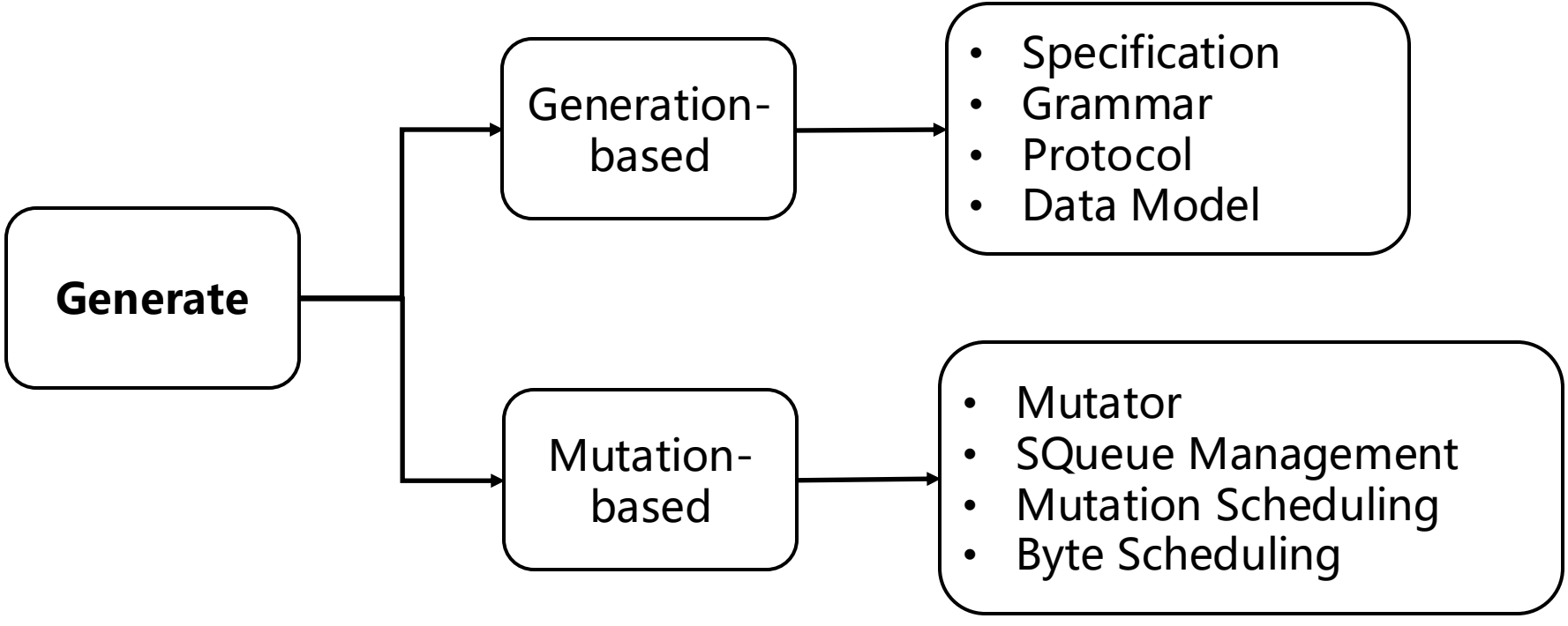
04

总结

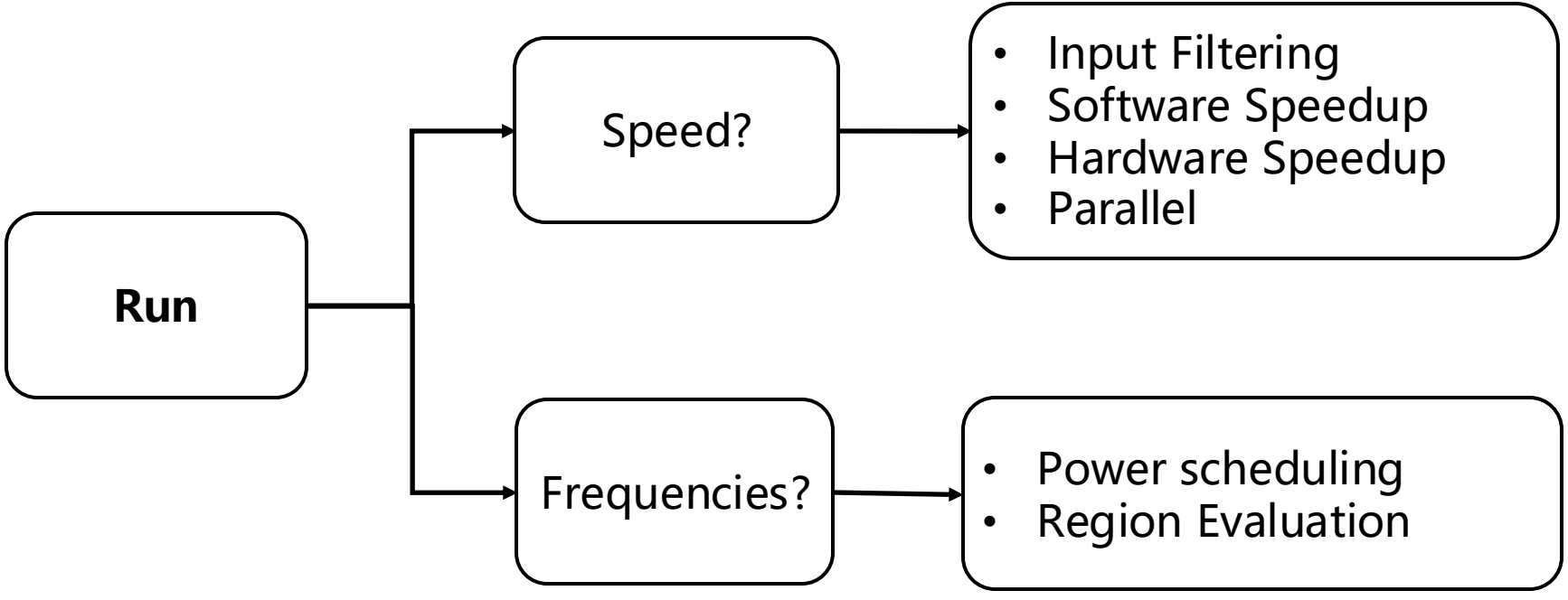
- **Generate** some inputs, make some **run**s, and **see** what happens



- **Generate** some inputs, make some runs, and see what happens

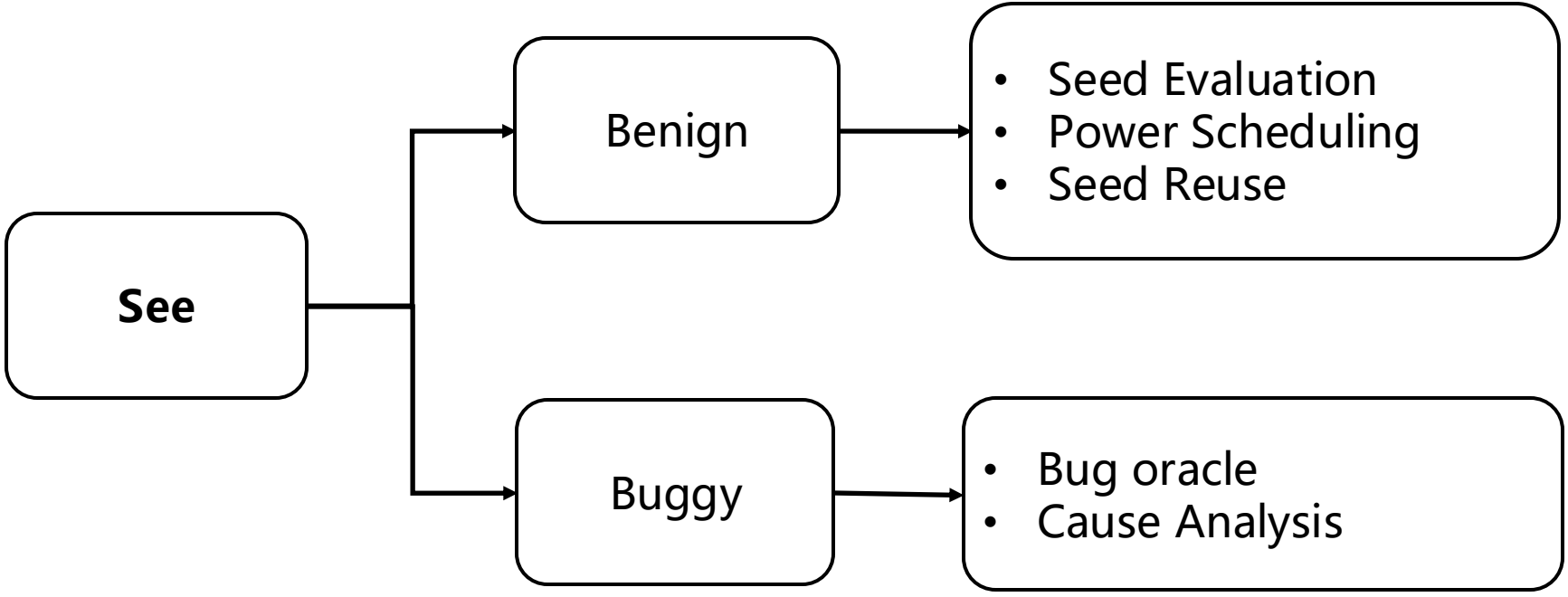


- Generate some inputs, make some **runs**, and see what happens



Takeaway

- Generate some inputs, make some runs, and **see** what happens



Takeaway

- Generate some inputs, make some runs, and **see** what happens
  - Fuzzing的难点不在Fuzzing本身！ → Fuzz Target
  - Performance要求 → Implementation

zychen@nju.edu.cn
fangchunrong@nju.edu.cn

Thank you!