电子科技大学

软件安全分析技术

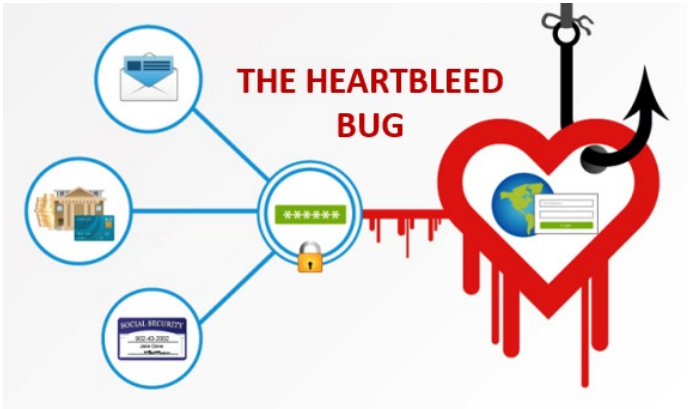
汇报人：李俊强

日期：2021. 7. 12

背景知识

研究背景

熊猫烧香



心脏出血

震网病毒

勒索病毒

病毒无处不在，导致病毒的根源其实就是漏洞(Bug)

漏洞说明

- 漏洞：硬件、软件、协议的具体实现或系统安全策略上存在的缺陷，从而使攻击者能够在未授权的情况下访问或破坏系统。
- 0day漏洞：未公开或未发补丁的漏洞，称为危害最大的漏洞或最有价值的漏洞。
- CVE (Common Vulnerabilities & Exposures) 通用漏洞披露

  Web URL: https://cve.mitre.org/

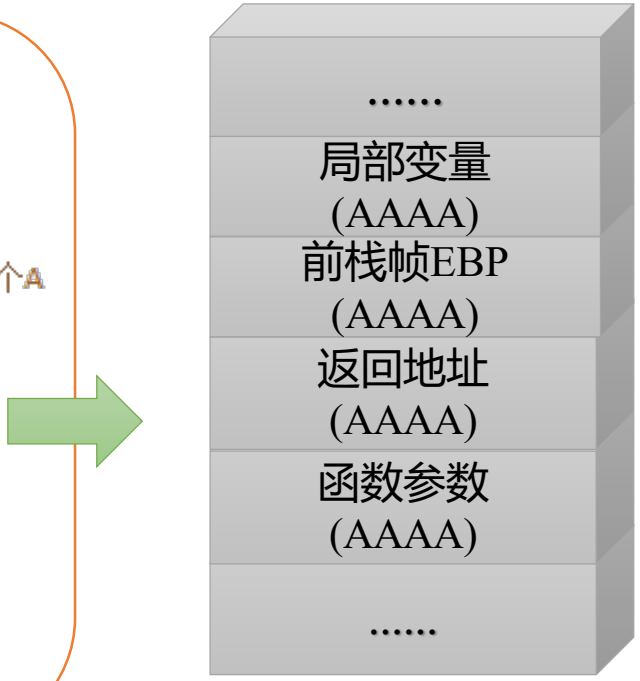
常见漏洞

- 栈溢出
- 堆溢出
- 整数溢出
- 格式化字符串
- 双重释放（Double Free）
- 释放重引用（Use after Free）……

栈溢出

```
1  #include <stdio.h>
2  #include <string.h>
3  int main()
4  {
5      char *str = "AAAAAAAAAAAAAAAAAAAAAAAAA";  //25个A
6      vulnfun(str);
7      return;
8  }
9  int vulnfun(char *str)
10 {
11      char stack[10];
12      strcpy(stack, str);
13 }
```

栈溢出实例



栈溢出原理图

电子科技大学
信息与通信工程学院

堆溢出

```
1 #include <windows.h>
2 #include <stdio.h>
3 int main()
4  {
5    HANDLE hHeap;
6    char *heap;
7    char str[] = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA";    // 0x20大小
8    hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x1000, 0xffff);  //创建一个堆
9    heap = HeapAlloc(hHeap, 0, 0x10);                              //为堆分配空间，大小为0x10
10   strcpy(heap, str);                                             //导致堆溢出
11   HeapFree(hHeap, 0, heap);                                      //释放堆内存
12   HeapDestroy(hHeap);                                            //销毁堆
13 }
```

整数溢出

```
1  #include <stdio.h>
2  int main()
3  {
4     unsigned short int a;
5     signed short int b;
6     a = 50000;
7     b = 50000;
8     printf("\n a = %d \t b = %d \n", a, b);
9     return 0;
10 }
```

```
ljq@ubuntu:~/Desktop/Ctest$ gcc in.c
ljq@ubuntu:~/Desktop/Ctest$ ./a.out
a = 50000        b = -15536
ljq@ubuntu:~/Desktop/Ctest$
```

类型	取值范围
unsigned short int	0~65535
signed short int	-32768~32767

格式化字符串

```
1 #include <stdio.h>
2 int main()
3 {
4     int a = 2;
5     double b =2.5;
6     char *str = "hello";
7     printf("a=%d,str=%s,b=%ld\n",a,str);
8     return 0;
9 }
```

电子科技大学
信息与通信工程学院

双重释放 Double Free

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    void *p1;
    p1 = malloc(100);
    free(p1);
    free(p1);
    return 0;
}
```

释放重引用 Use after Free

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *buf1;
    buf1 = (char *) malloc(100);
    free(buf1);
    strncpy(buf1, "hack", 5);
    printf("buf1:%s\n",buf1);
    return 0;
}
```

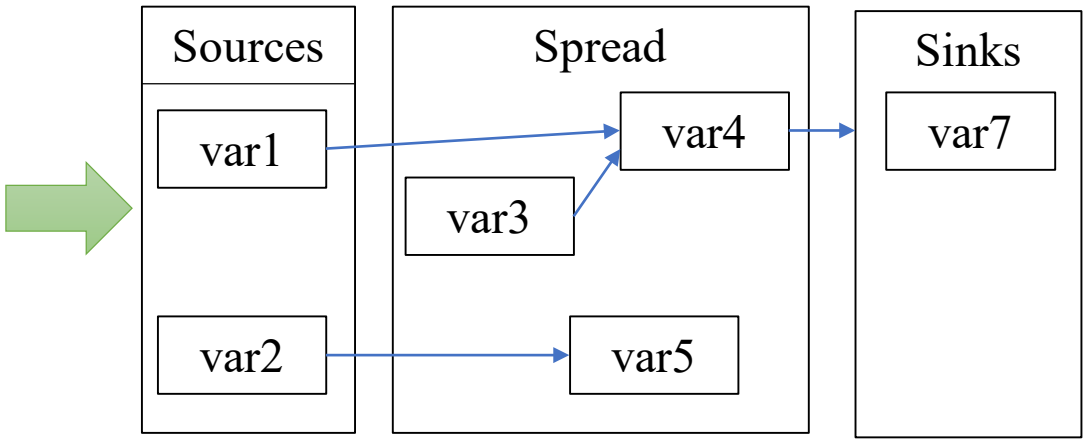
漏洞挖掘技术

技术	特点	局限性
静态分析 Static Analysis	• 无需运行待测程序 • 速度快	• 存在大量误报
模型检测 Model Checking	• 对待测系统进行有限状态机建模验证	• 状态空间爆炸
符号执行 Symbolic Execution	• 程序路径转化为符号表达，进而求解符号约束 • 覆盖深层次路径	• 路径爆炸，速度慢 • 拓展性差
污点分析 Taint Analysis	• 面向数据	• 效率低，耗时长 • 开发难度大
模糊测试 Fuzzing	• 输入随机或半随机的数据监控待测试程序 • 准确性高、速度快	• 无法识别访问控制漏洞、糟糕的设计逻辑等问题

漏洞挖掘技术

- 静态分析：Static Analysis
- 模型检测：Model Checking
- 符号执行：Symbolic Execution
- 污点分析：Taint Analysis
- 模糊测试：Fuzzing

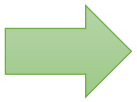
```
1 def rectangle():
2     var1 = input('Enter the first number:')
3     var3 = 5
4     var4 = var1 * var3
5     var7 = int(var4)
6     return var7
7 def reset():
8     var2 = input('Enter the second number:')
9     var5 = var2 * 0
10    return var5
```



Taint Analysis

- **Data-oriented**

```
1 void example(int x, int y, int z)
2 {
3     int cnt = 0;
4     if (x == 1) cnt++;
5     if (y == 2) cnt++;
6     if (z == 3) cnt++;
7     if (cnt == 3) bug(1);
8 }
```



① Random: <x!=1, y!=2, z!=3>

② <x==1, y!=2, z!=3>

③ <x!=1, y==2, z!=3>

…

⑧ <x==1, y==2, z==3>

Symbolic Execution

- **High Coverage**
- **Low Speed**
- **Path Explosion**

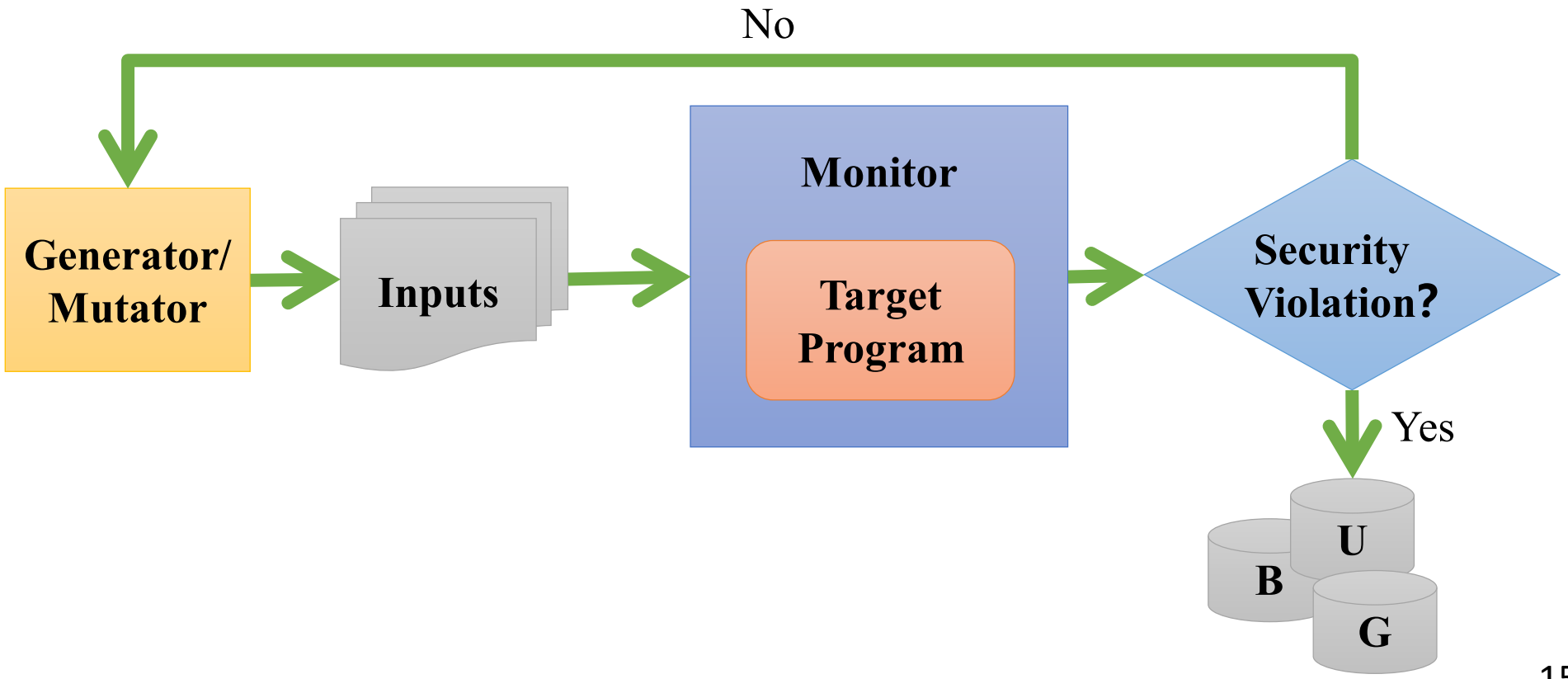
Fuzzing

- 学术界
  - 国内外顶尖大学都在研究Fuzzing技术
    - 国内：清华大学、中国科学技术大学、南京大学
    - 国外：帝国理工学院、宾夕法尼亚大学、普渡大学
  - 大量Fuzzing论文出现在安全顶会
    - S&P (IEEE Symposium on Security and Privacy)
    - CCS (ACM Conference on Computer and Communications Security)
    - USENIX Security (Usenix Security Symposium)
    - NDSS (ISOC Network and Distributed System Security Symposium)
- 工业界
  - 微软、谷歌、华为、阿里等大公司中80%都使用了模糊测试技术
  - 微软公司：采用模糊测试技术发现的漏洞数占总漏洞的20%~25%

Fuzzing

目标： 更多、更快地发现软件漏洞

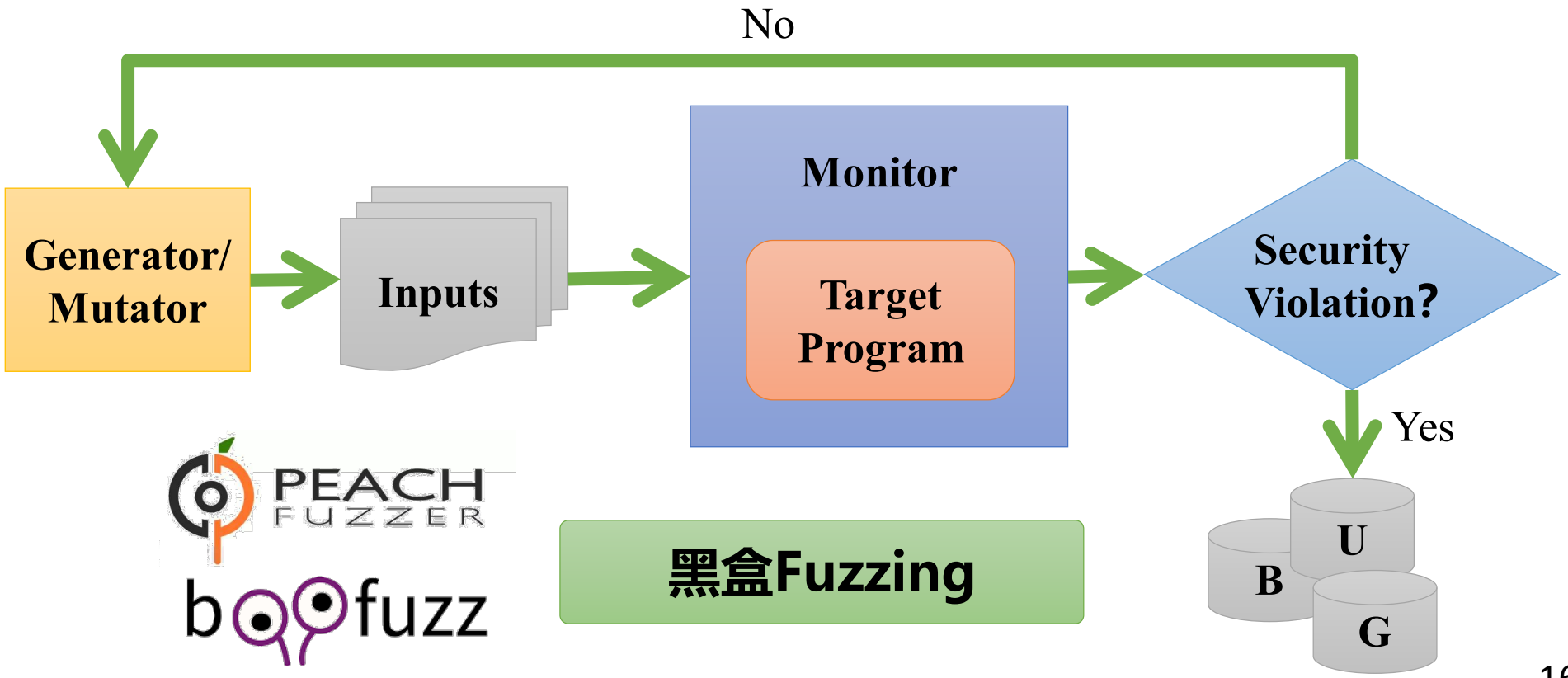
思路：



Fuzzing

目标： 更多、更快地发现软件漏洞

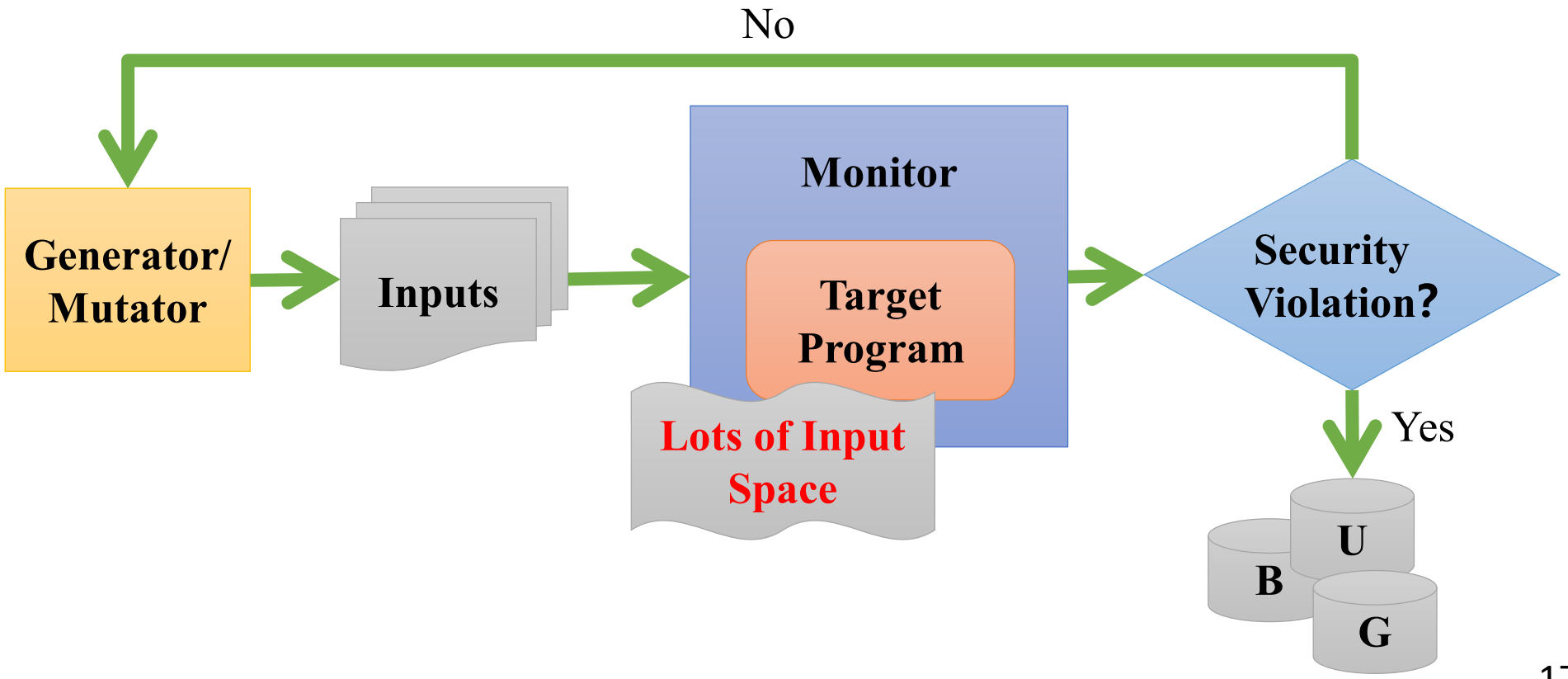
思路：



Fuzzing

目标： 更多、更快地发现软件漏洞

思路：



Fuzzing

例子：

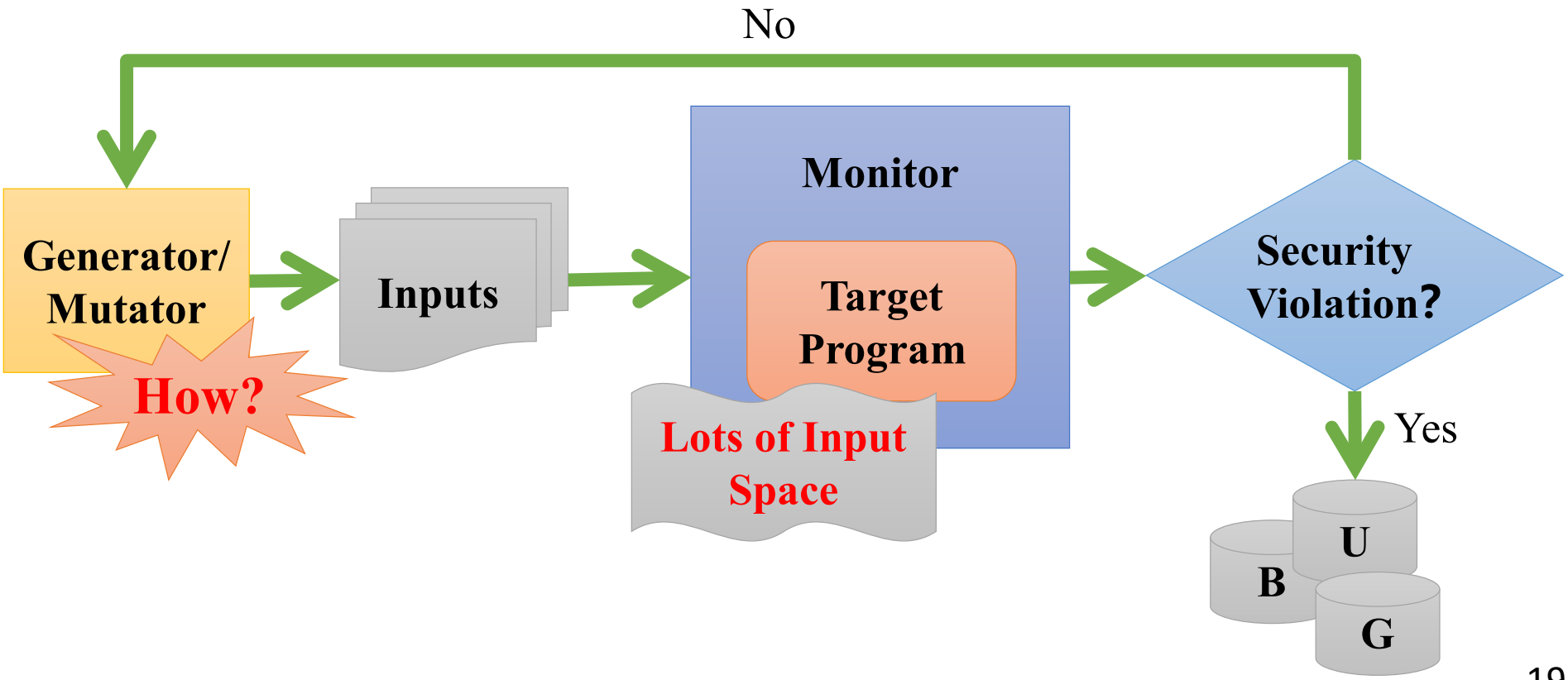
```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```

input = “oops” ⇒ $P = \frac{1}{2^8 * 2^8 * 2^8 * 2^8} = \frac{1}{2^{32}}$

Fuzzing

目标： 更多、更快地发现软件漏洞

思路：



Fuzzing

思路：遗传算法（Genetic Algorithm）+ 覆盖率（Coverage）统计

Fuzzing

是什么?

思路：遗传算法（Genetic Algorithm）+ 覆盖率（Coverage）统计

Fuzzing

思路：遗传算法（Genetic Algorithm）+ 覆盖率（Coverage）统计

覆盖率：

```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```

Source Code

Statement Coverage：

input = “oops”

Fuzzing

思路： 遗传算法（Genetic Algorithm）+ 覆盖率（Coverage）统计

覆盖率：

```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```

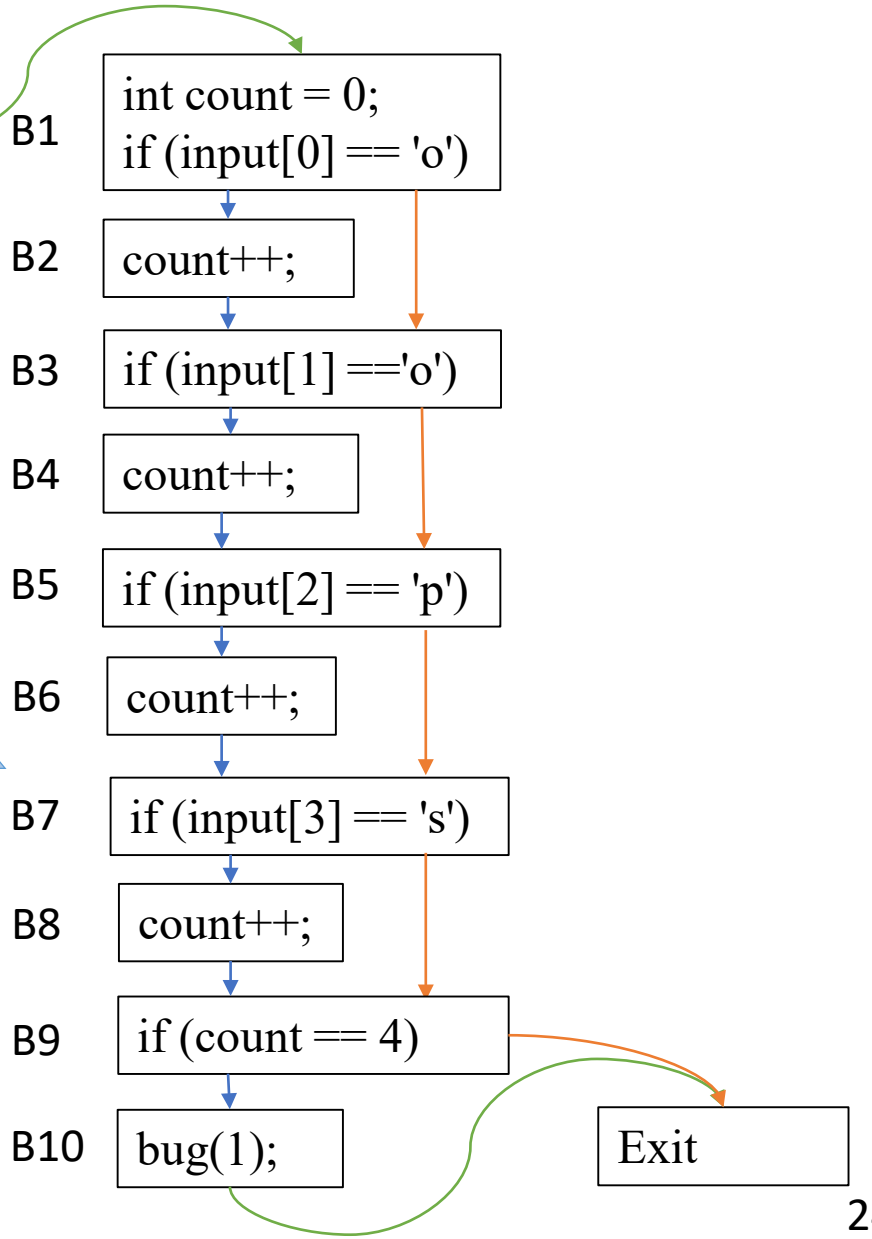
Source Code

Branch Coverage：

input = “oops”
input = “abcd”

Fuzzing

```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```



Edge Coverage: Control Flow Graph

input = "o*" ⇒ B1->B2

input = "a*" ⇒ B1->B3

电子科技大学
信息与通信工程学院

Fuzzing

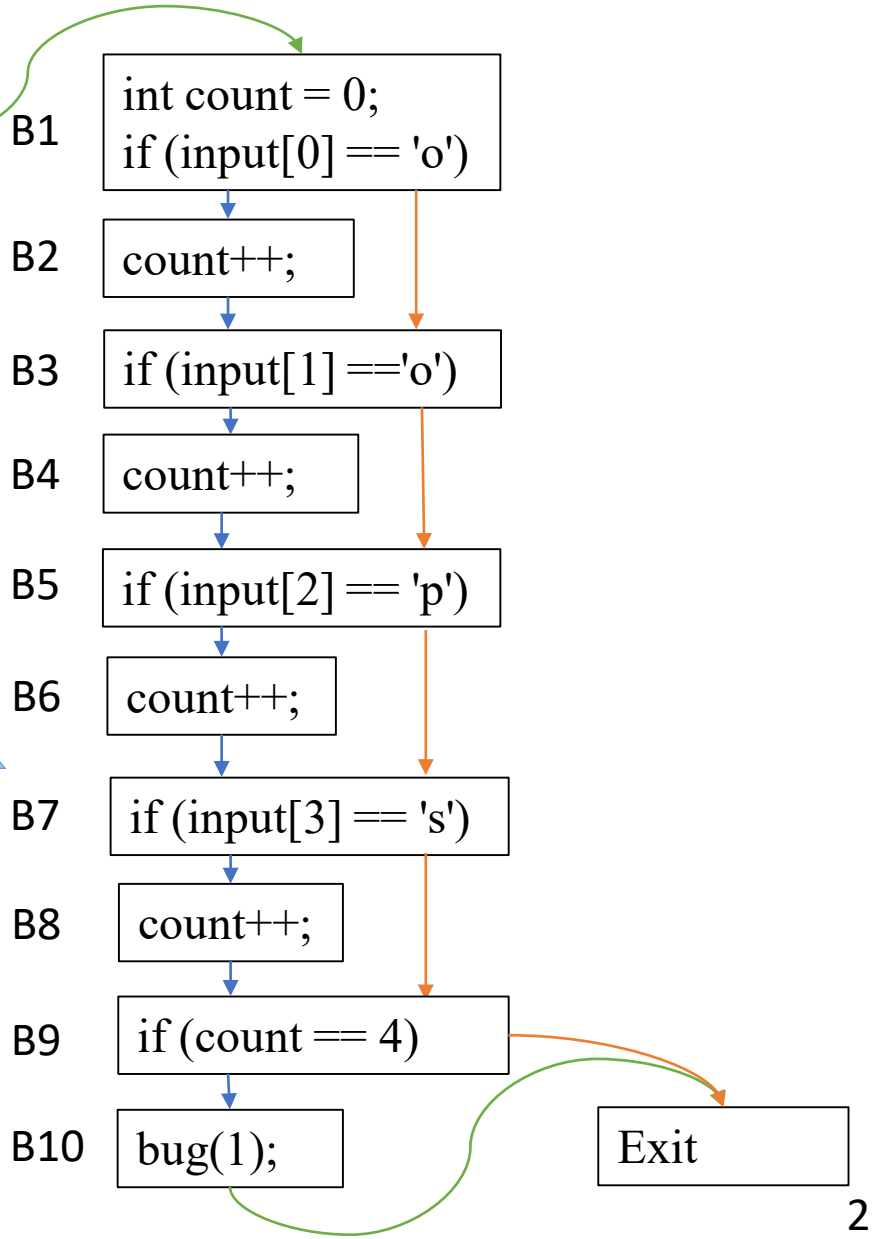
```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```

Path Coverage:

input = “oops” → B1->B2->…->B10

input = “oopx” → B1->…->B7->B9

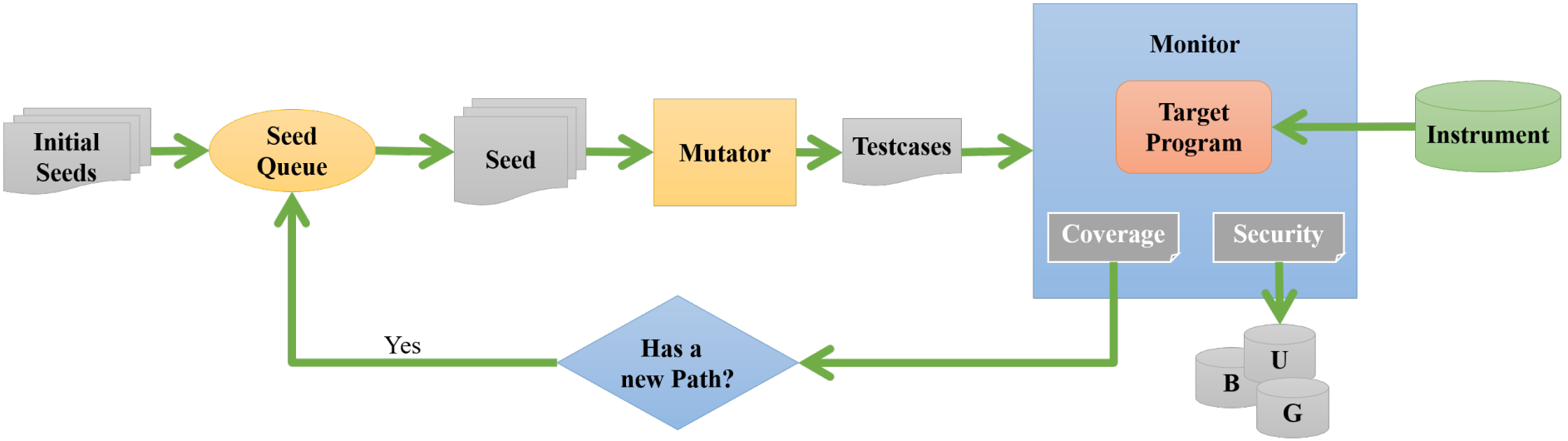
Control Flow Graph



Fuzzing

思路： 覆盖率（Coverage）统计 + 遗传算法（Genetic Algorithm）

流程：

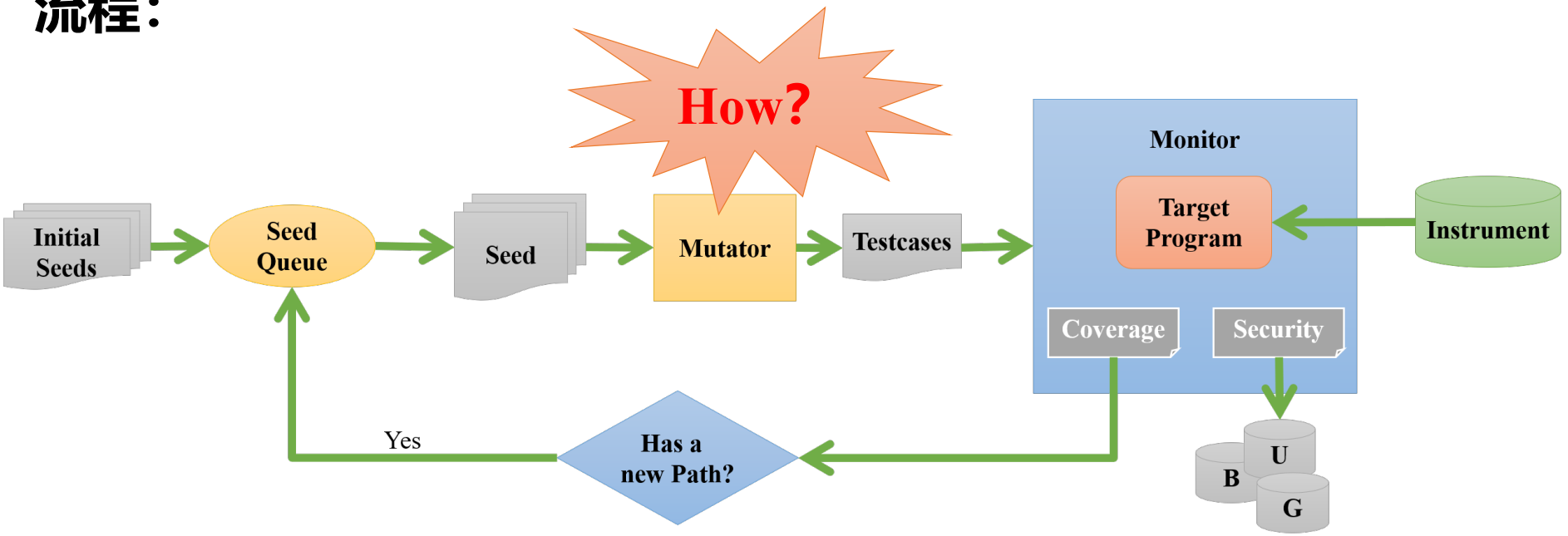


灰盒Fuzzing
AFL(American Fuzzy Lop)

Fuzzing

思路： 覆盖率（Coverage）统计 + 遗传算法（Genetic Algorithm）

流程：



灰盒Fuzzing
AFL(American Fuzzy Lop)

Fuzzing

Type	Meaning	Operators
bitflip	Invert one or several consecutive bits in a test case, where the stepover is 1 bit.	`bitflip 1/1`, `bitflip 2/1`, `bitflip 4/1`
byteflip	Invert one or several consecutive bytes in a test case, where the stepover is 8 bits.	`bitflip 8/8`, `bitflip 16/8`, `bitflip 32/8`
arithmetic inc/dec	Perform addition and subtraction operations on one byte or several consecutive bytes.	`arith 8/8`, `arith 16/8`, `arith 32/8`
interesting values	Replace bytes in the test cases with hard-coded interesting values.	`interest 8/8`, `interest 16/8`, `interest 32/8`
user extras	Overwrite or insert bytes in the test cases with user-provided tokens.	`user (over)`, `user (insert)`
auto extras	Overwrite bytes in the test cases with tokens recognized by AFL during `bitflip 1/1`.	`auto extras (over)`
random bytes	Randomly select one byte of the test case and set the byte to a random value.	`random byte`
delete bytes	Randomly select several consecutive bytes and delete them.	`delete bytes`
insert bytes	Randomly copy some bytes from a test case and insert them to another location in this test case.	`insert bytes`
overwrite bytes	Randomly overwrite several consecutive bytes in a test case.	`overwrite bytes`
cross over	Splice two parts from two different test cases to form a new test case.	`cross over`

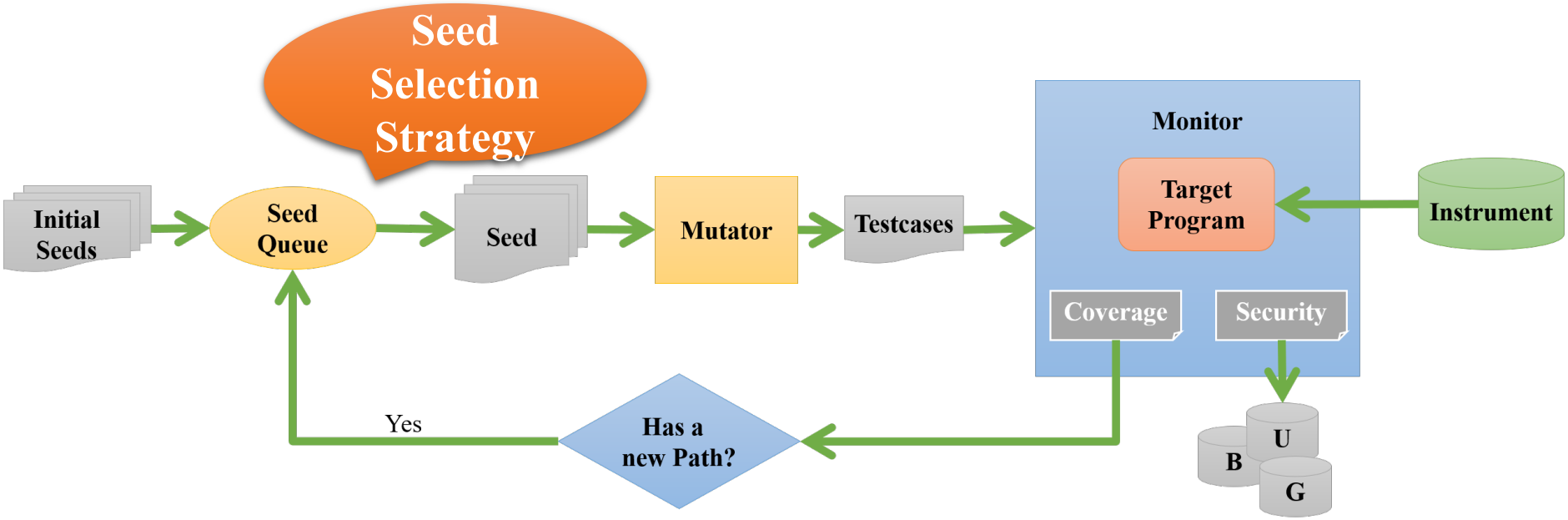
Fuzzing

例子：

```
1 void test(char input[4]) {
2     int count = 0;
3     if (input[0] == 'o') count++;
4     if (input[1] == 'o') count++;
5     if (input[2] == 'p') count++;
6     if (input[3] == 's') count++;
7     if (cout == 4) bug(1);  //bug is here
8 }
```

input = “oops” ⇒ $P \ll \frac{1}{2^{32}}$

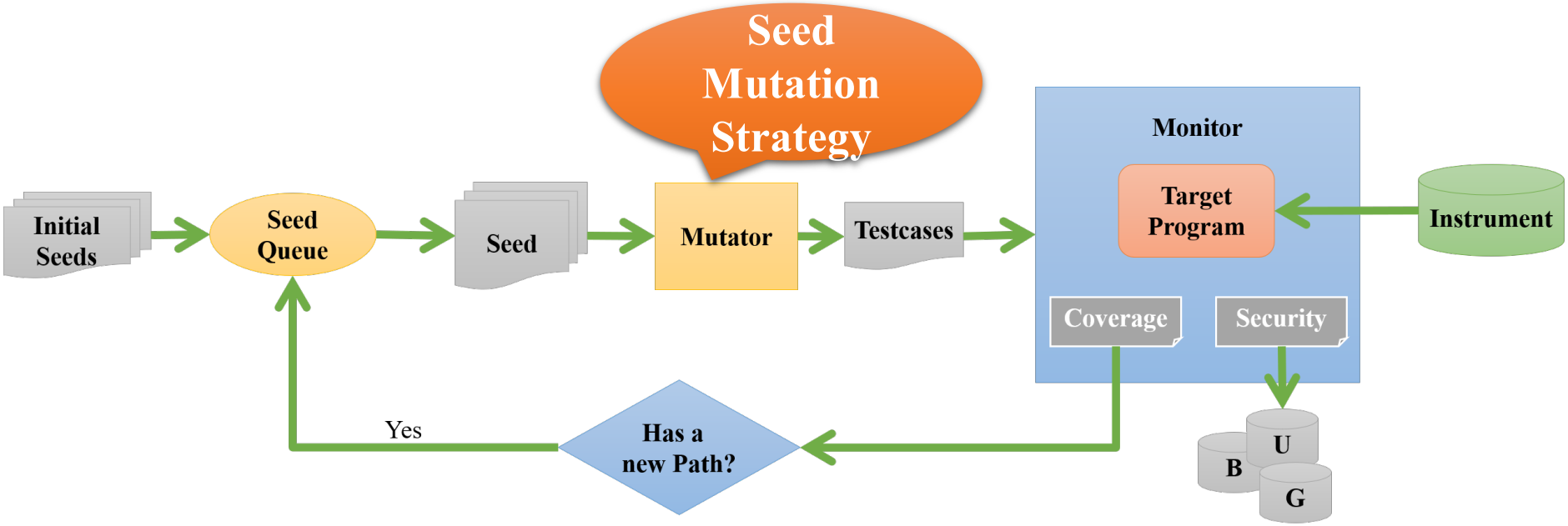
Fuzzing



Seed Selection Strategy

AFLFast (CCS16)	Low-frequency Paths	FairFuzz (ASE18)	Rare Branches
Vuzzer (NDSS17)	Deeper Paths	CollAFL (S&P18)	More unvisited children
AFLGo (CCS17)	Closer Paths	QTEP (FSE17)	More bugs
SlowFuzz (CCS17)	More Resources Consume	EcoFuzz (USENIX Security20)	Closer Paths

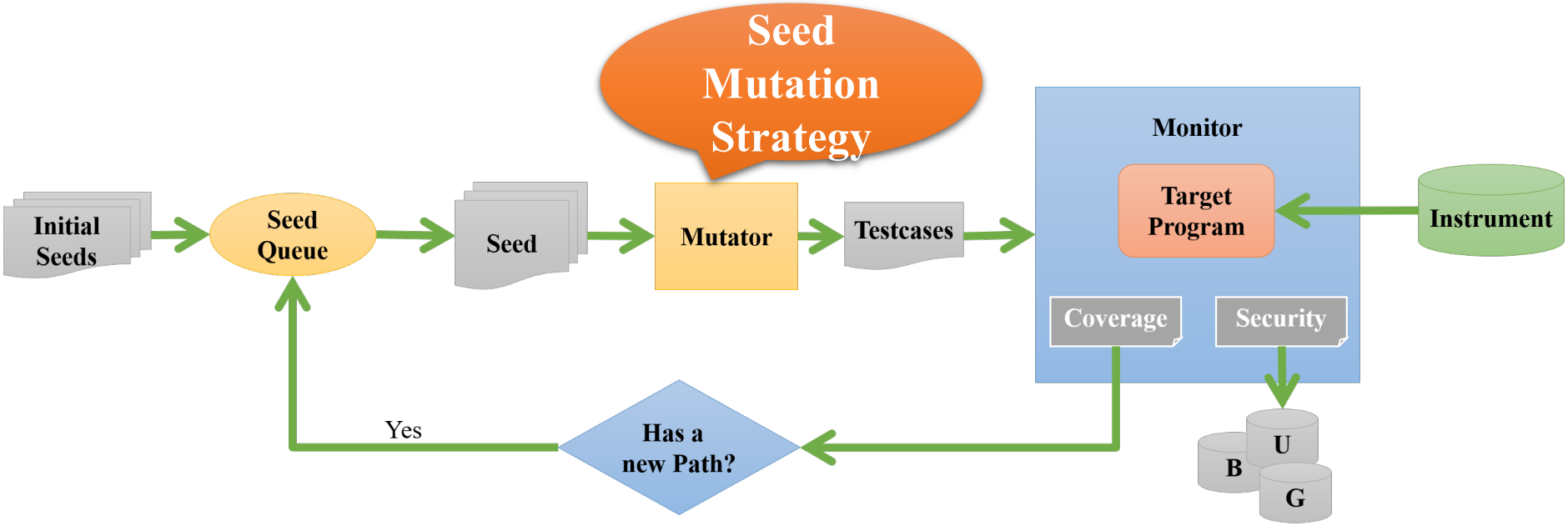
Fuzzing



Seed Mutation Strategy - AI

Skyfire (S&P17)	PCSG (Probabilistic Context-Sensitive Grammar)
Neuzz (S&P19)	Smooth NN Model + Gradient-guided Optimization
DeepFuzz (AAAI19)	Seq2Seq Model
MOPT (USENIX Security 19)	Particle Swarm Optimization
ILF (CCS19)	Neural Networks to Learn Fuzzing Policy

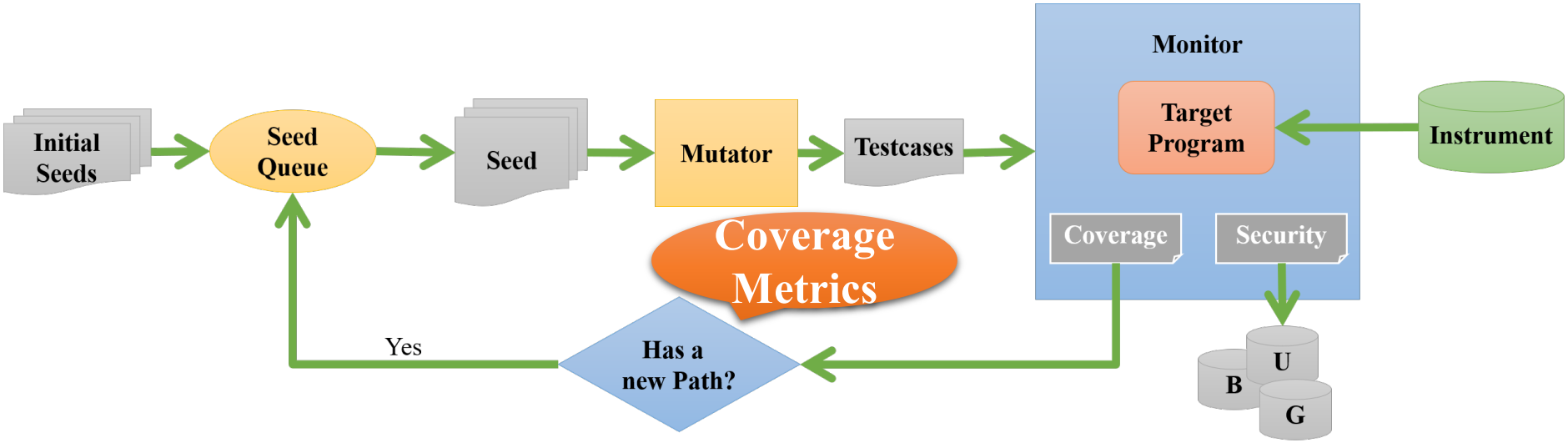
Fuzzing



Seed Mutation Strategy - Program Analysis

Vuzzer (NDSS17)、FANS (USENIX Security20)	Static Analysis and Fuzzing
QSYM (USENIX Security18) 、Intriguer (CCS19)	Symbolic Execution and Fuzzing
Angora (S&P18)、GreyOne (USENIX Security20)	Taint Analysis and Fuzzing
ParmeSan (USENIX Security20)	Sanitizer-guided Fuzzing

Fuzzing



Coverage Metrics

AFLGo (CCS17)

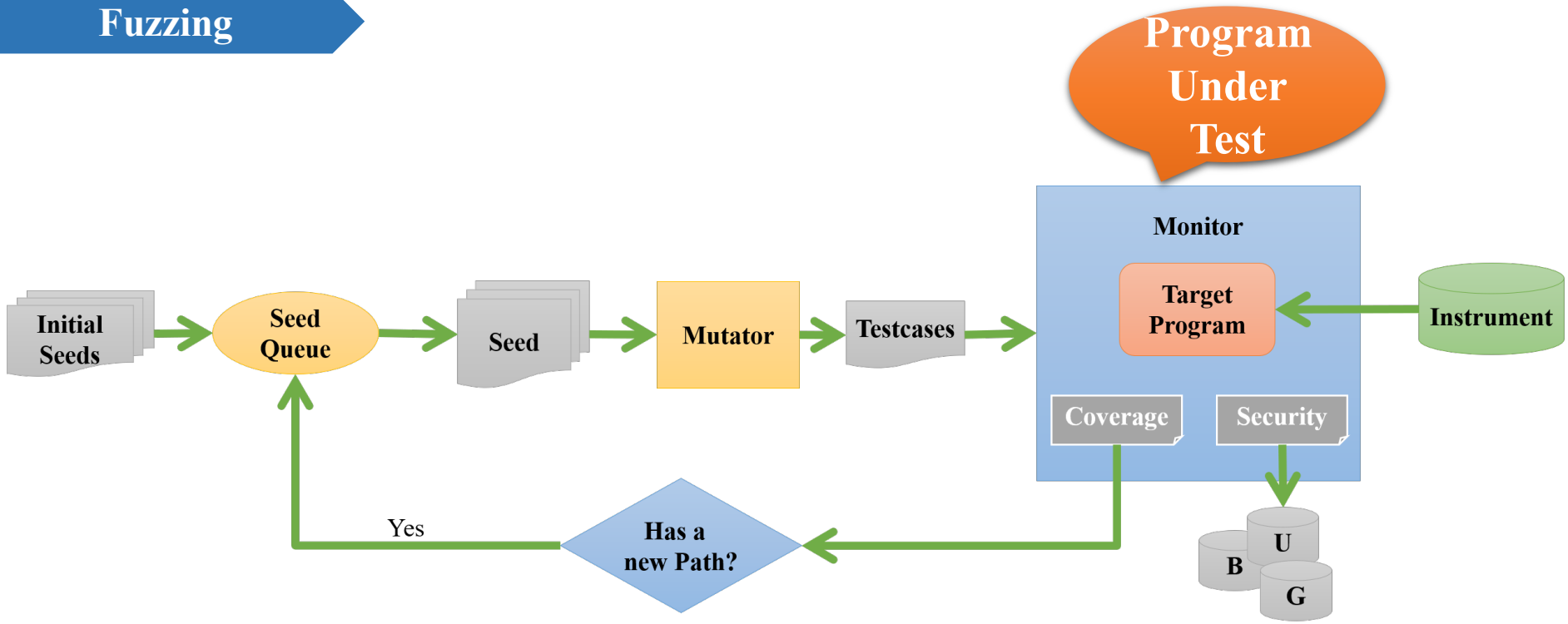
Hawkeye (CCS18) → Directed Fuzzing Target Specific Code

CAFL (USENIX Security21)

UnTracer (S&P19) Introduce to The Coverage Guidance Tracking

TortoiseFuzz (NDSS20) Introduce to The Coverage Accounting

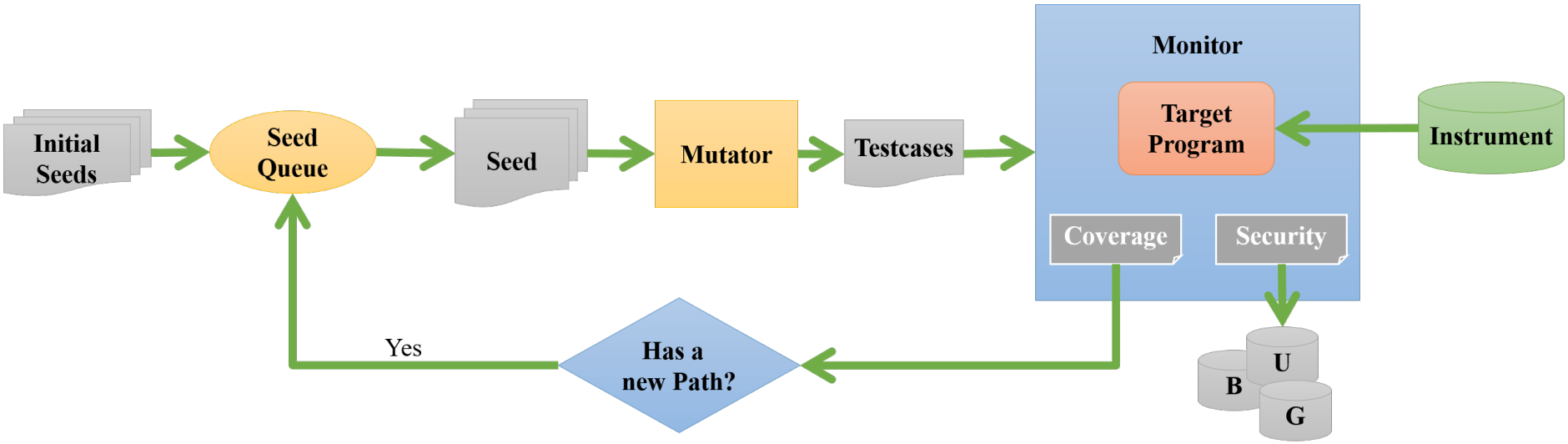
Fuzzing



Program Under Test

FuzzGen (USENIX Security20)	C/C++ Library	WINNIE (NDSS21)	Windows Program
HotFuzz (NDSS20)	Java Library	TLS-Attacker (CCS16)	Network Protocol
EVMFuzzer (FSE19)	Ethereum Virtual Machine	PGFuzz (NDSS21)	Robotic Vehicles
HFL (NDSS20)	Linux Kernel	Squirrel (CCS20)	DBMS

Fuzzing



Others

Evaluating Fuzz Testing (CCS18)

Fuzzing: On the Exponential Cost of Vulnerability Discovery (FSE20) → Evaluation

Seed Selection for Successful Fuzzing (ISSTA21)

IJON (S&P20) — Human and Fuzzing Cooperation

EnFuzz (USENIX Security19) — Mutiple Fuzzers Cooperation

电子科技大学

Thank You !