Fuzzing: the way to vulnerabilities

Chao Zhang
Tsinghua University

About Me

Experience

- Tsinghua University, Assoc. Prof., 2016/11-present
- UC Berkeley, Postdoc, 2013/9-2016/9, Advisor: Dawn Song
- Peking University, Ph.D., 2008/9-2013/7, Advisors: 邹维，韦韬
- Peking University, Undergraduate, 2004/9-2008/7, Math

Honors

- Young Elite Scientists Sponsorship Program by CAST
- DARPA CGC, Captain of Team CodeJitsu
  - Defense #1 in 2015 CQE, Attack #2 in 2016 CFE
- Microsoft BlueHat Prize Contest 2012
  - Special Recognition Award
- DEFCON CTF 2015 (#5), 2016 (#2), 2017 (#5)
- GeekPwn 2017/5/12

What are vulnerabilities?

Errors introduced in the design or implementations of software/system/hardware/protocol/algorithms, that could be exploited to break victims' CIA attributes.

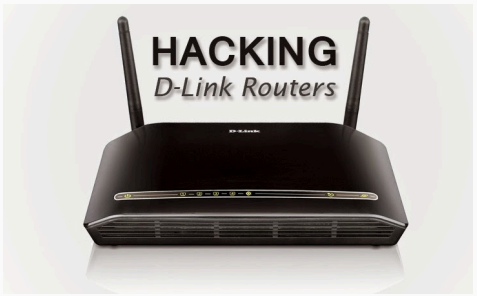
```
int main(int argc, char** argv){
  if(argc<2)
  {
    printf("please tell me the working mode ID.\n");
    return 2;
  }
  int mode = atoi(argv[1]);
  printf("working mode: %x\n", mode);

  if(mode==12345678){
    char buf[256];
    printf("ping: ");
    gets(buf);
    printf("pong: %s\n", buf);
    return 1;
  }
  else{
    printf("try again.\n");
    return 0;
  }
}
```

栈溢出：输入超过256字节

- 内存破坏漏洞
  - 栈溢出
  - 堆溢出
  - 整数溢出
  - 格式化字符串
  - 悬空指针
  - 未初始化变量
  - 竞态条件
  - 命令注入
  - …
- 逻辑漏洞
- 协议漏洞
- Web漏洞
- Android应用漏洞
- …

Consequences of vulnerabilities




Oops, your files have been encrypted!
中华人民共和国
警察
POLICE

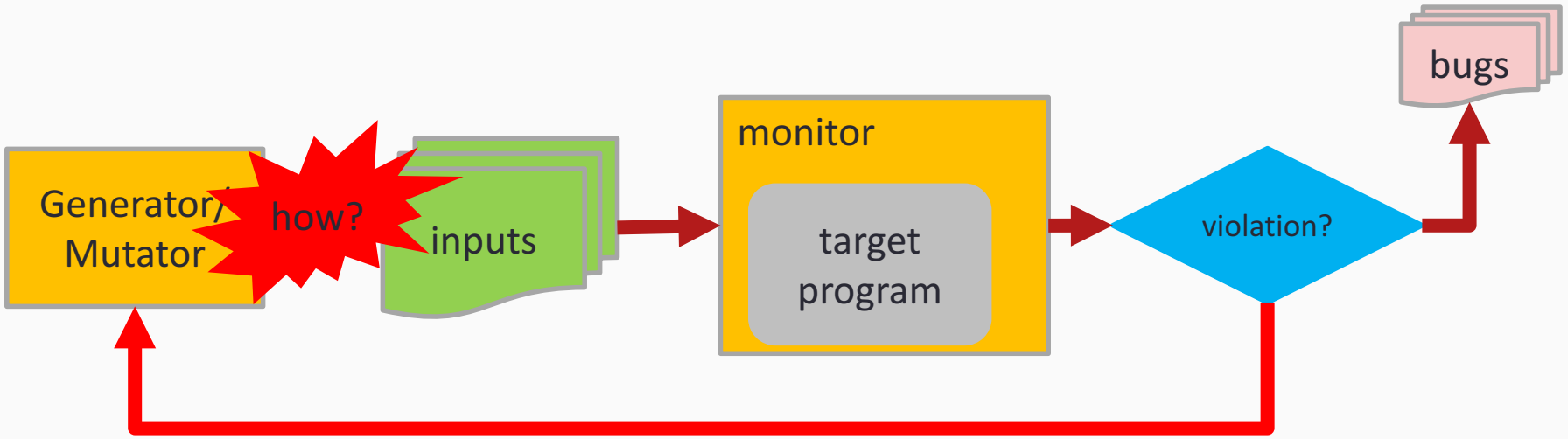
Finding vulnerabilities is a key to both defenses and attacks.

So, how to find vulnerabilities?

Vulnerability Detection Solutions

- Static Analysis
- Taint Analysis
- Fuzzing
  - mutation, generation
  - blackbox, greybox, whitebox
  - smart, dumb
- Symbolic Execution
- Dynamic Detection
  - online
  - offline

Fuzzing


Generator/
Mutator
how?
inputs
monitor
target
program
violation?
bugs

Random Fuzzing

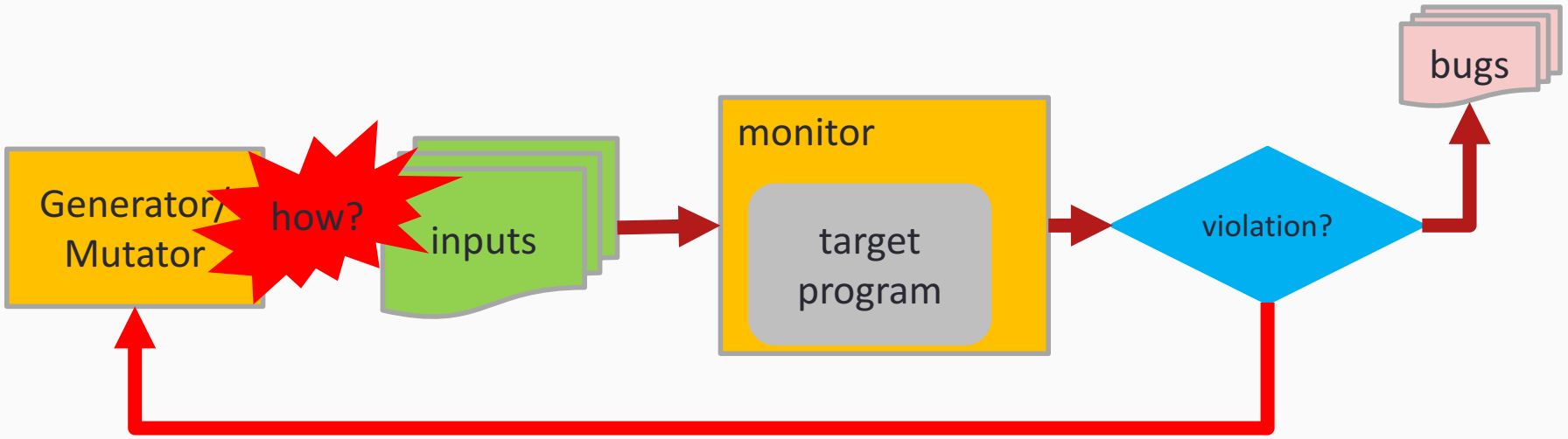
```
int main(int argc, char** argv){
  if(argc<2)
  {
    printf("please tell me the working mode ID.\n");
    return 2;
  }
  int mode = atoi(argv[1]);
  printf("working mode: %x\n", mode);

  if(mode==12345678){
    char buf[256];
    printf("ping: ");
    gets(buf);
    printf("pong: %s\n", buf);
    return 1;
  }
  else{
    printf("try again.\n");
    return 0;
  }
}
```

- ./test 1
- ./test 2
- ./test 3
- ./test 4
- …
- ./test 12345678
  - and the input buf...
- …
- …

Random fuzzing is very ineffective.

Fuzzing



- generation-based
  - generate inputs from templates (e.g., grammar, specification)
- mutation-based
  - mutate inputs from seed inputs

Generation-based Fuzzing

```
<!--  A.  Local file header -->
  <Block name="LocalFileHeader">
    <String name="lfh_Signature" valueType="hex" value="504b0304" token="true" mut
    <Number name="lfh_Ver" size="16" endian="little" signed="false"/>
    ...
    [truncated for space]
    ...
    <Number name="lfh_CompSize" size="32" endian="little" signed="false">
      <Relation type="size" of="lfh_CompData"/>
    </Number>
    <Number name="lfh_DecompSize" size="32" endian="little" signed="false"/>
    <Number name="lfh_FileNameLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FileName"/>
    </Number>
    <Number name="lfh_ExtraFldLen" size="16" endian="little" signed="false">
      <Relation type="size" of="lfh_FldName"/>
    </Number>
    <String name="lfh_FileName"/>
    <String name="lfh_FldName"/>
    <!--  B.  File data -->
    <Blob name="lfh_CompData"/>
  </Block>
```

Src: http://www.flinkd.org/2011/07/fuzzing-with-peach-part-1/

Mutation-based Fuzzing

- Mutate from seed input

```
D7 CD FD 9A 00 00 DB FE 0B 00 C5 00 00 01 E8 03 ;
            ×Íý š..Ûþ..Å...è.
D7 CD FE 9A 00 00 DB FE 0B 00 C5 00 00 01 E8 03 ;
            ×Íþ š..Ûþ..Å...è.
D7 CD FF 9A 00 00 DB FE 0B 00 C5 00 00 01 E8 03 ;
            ×Íÿ š..Ûþ..Å...è.
```

Comparison

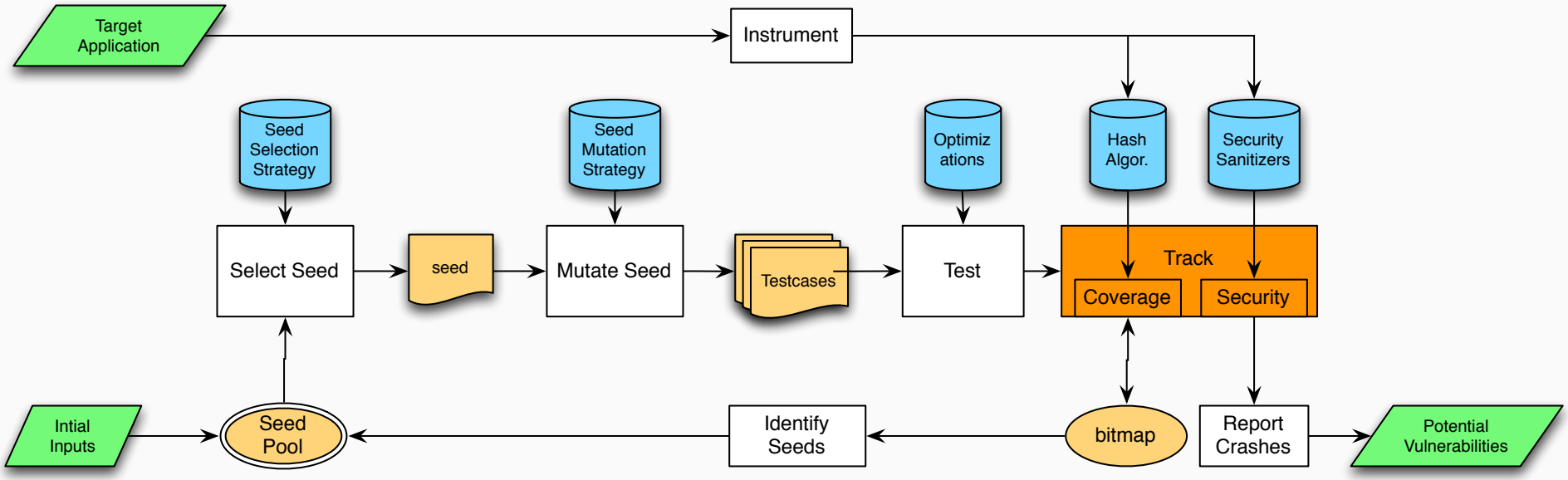
	setup	knowledge	coverage	checksum etc.
Mutation-based	Super easy to setup and automate +	Little to no protocol knowledge required +	Limited by initial corpus −	May fail for protocols with checksums, or other complexity −
Generation-based	Writing generator is labor intesive for complext protocols −	require spec of protocol −	Completeness +	Can deal with complex checksums and dependencies +

	blackbox	greybox	whitebox
mutation-based	Miller	AFL, Driller, Vuzzer, TaintScope, Mayhem	SAGE, Libfuzzer,
generation-based	SPIKE, Sulley, Peach		

Basics of AFL

AFL

- An open source coverage-guided mutation-based fuzzer, very successful and popular.

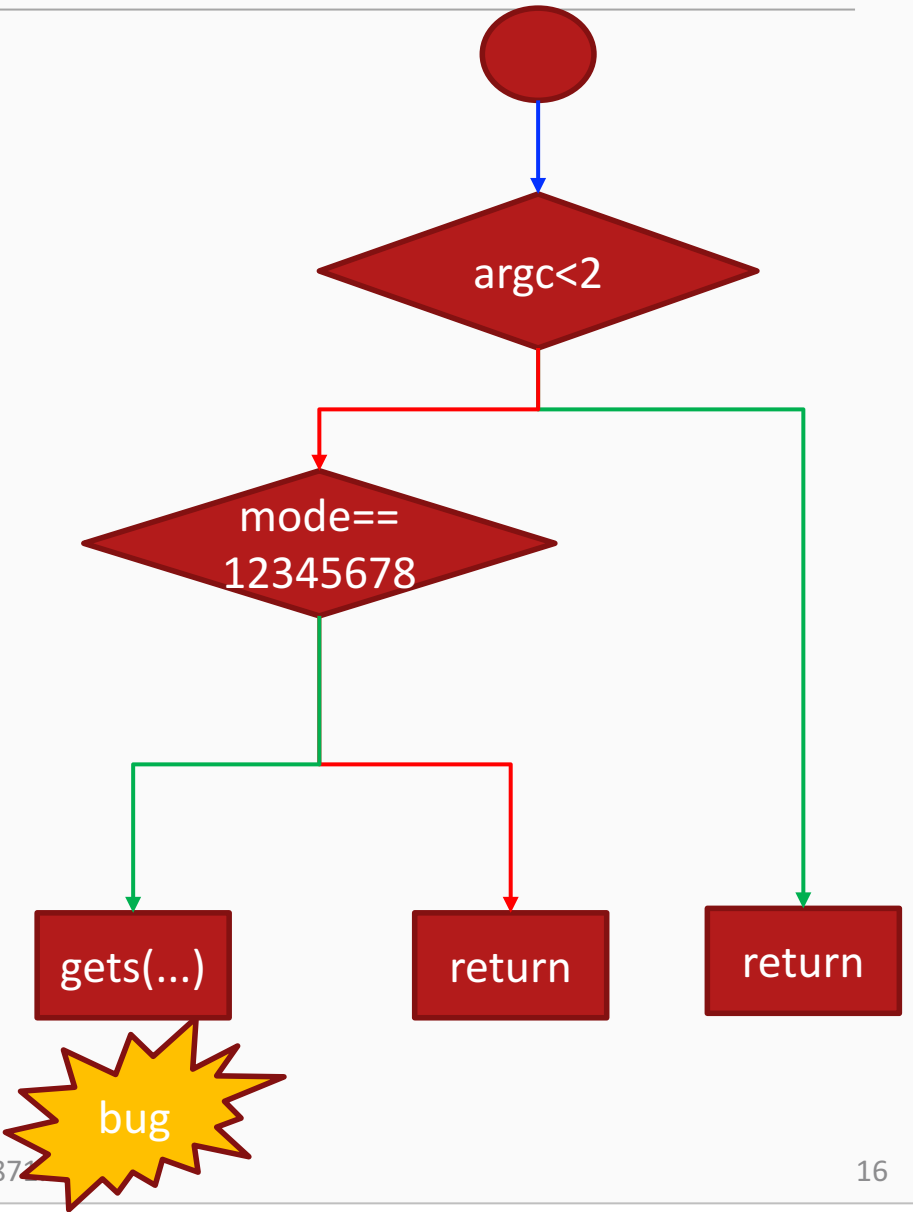


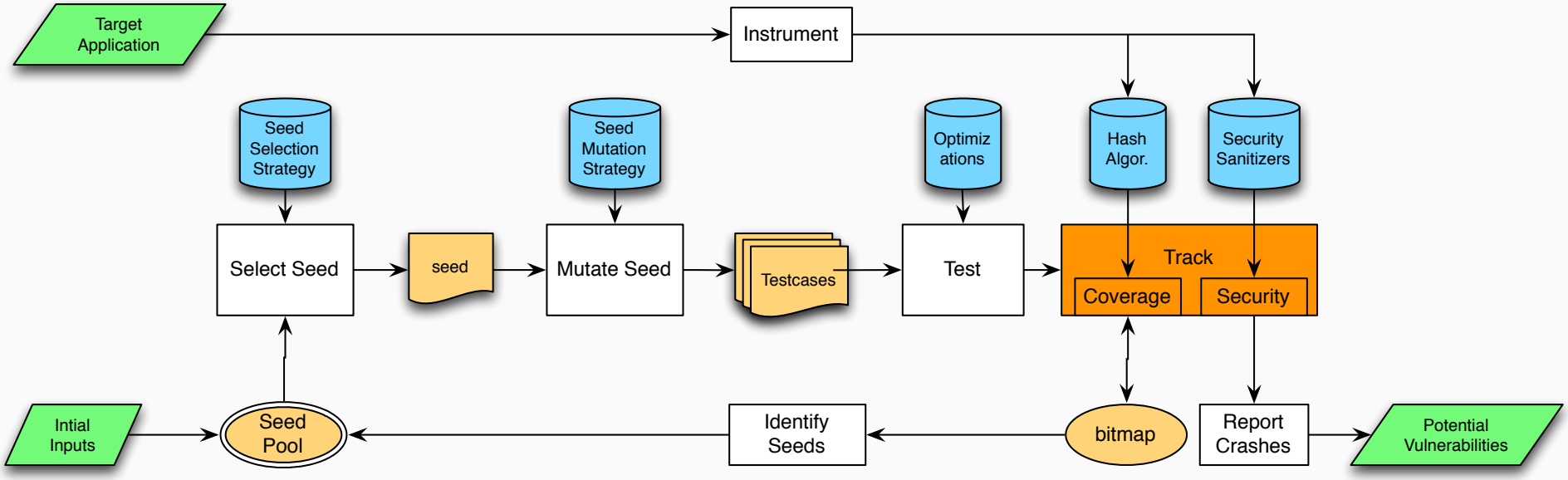
scalable, few knowledge is required
evolving, code coverage guided,
fast, throughput is high
sensitive, able to catch security violations

Evolving: Code Coverage

```c
int main(int argc, char** argv){
  if(argc<2)
  {
    printf("please tell me the working mode ID.\n");
    return 2;
  }
  int mode = atoi(argv[1]);
  printf("working mode: %x\n", mode);

  if(mode==12345678){
    char buf[256];
    printf("ping: ");
    gets(buf);
    printf("pong: %s\n", buf);
    return 1;
  }
  else{
    printf("try again.\n");
    return 0;
  }
}
```





Idea: evolve, discard useless testcases.

only keep GOOD seeds that contribute to the code coverage.

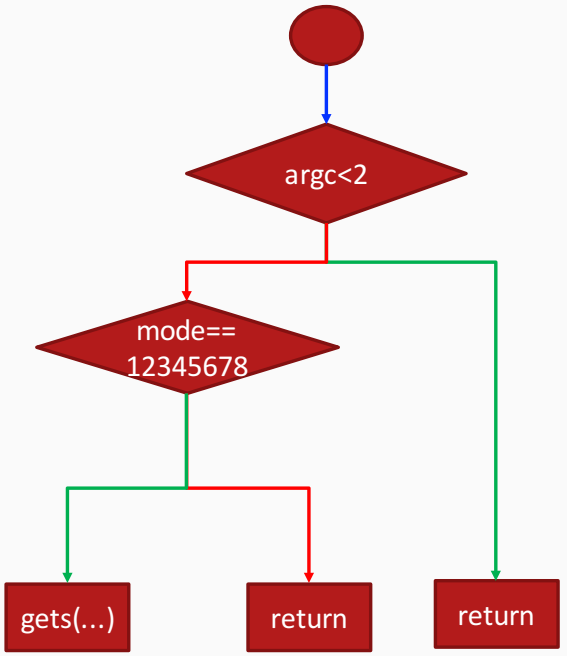
Idea: evolve, discard useless testcases.

only keep GOOD seeds that contribute to the code coverage.

- How to track code coverage?
  - give each basic block an ID
  - compute hash for edges based on block IDs
  - update the bitmap to store edge hit count
    - trace_bits

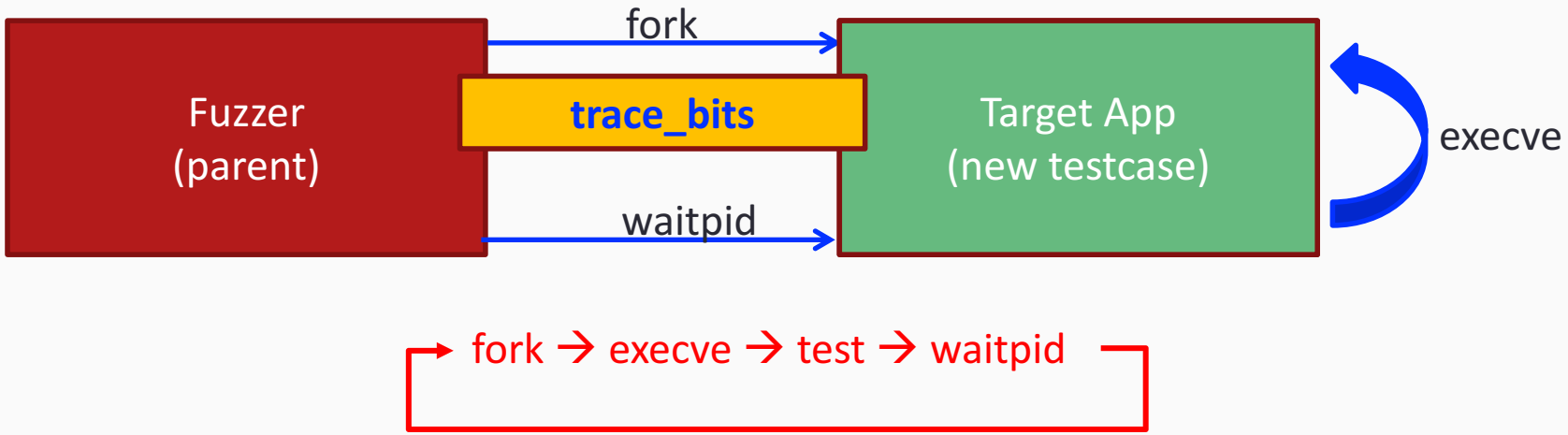
```
cur_loc = <compile_time_random>
hash = cur_loc ^ (prev_loc << 1)
trace_bits [hash] ++;
```

- Whether a new testcase is GOOD?
  - maintain a global bitmap tracking the hit count history of each edge
    - virgin_bits
  - compare trace_bits with virgin_bits,



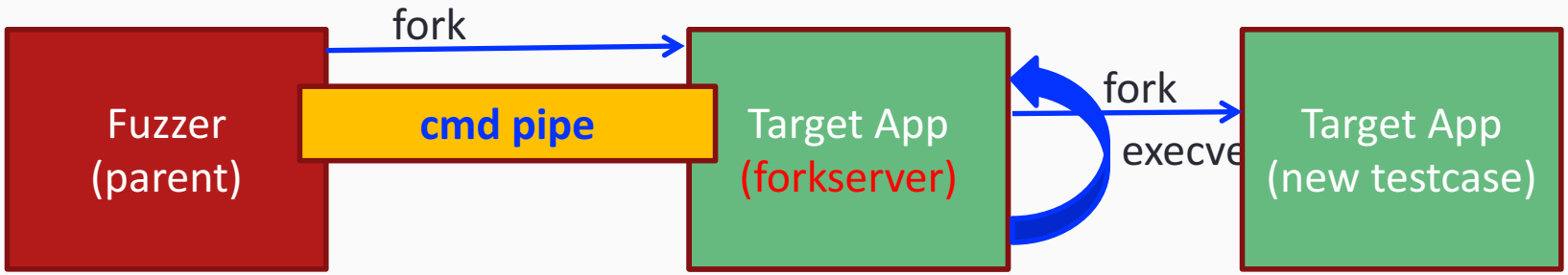
Fast: fork and execve

- dumb_mode



Faster: forkserver

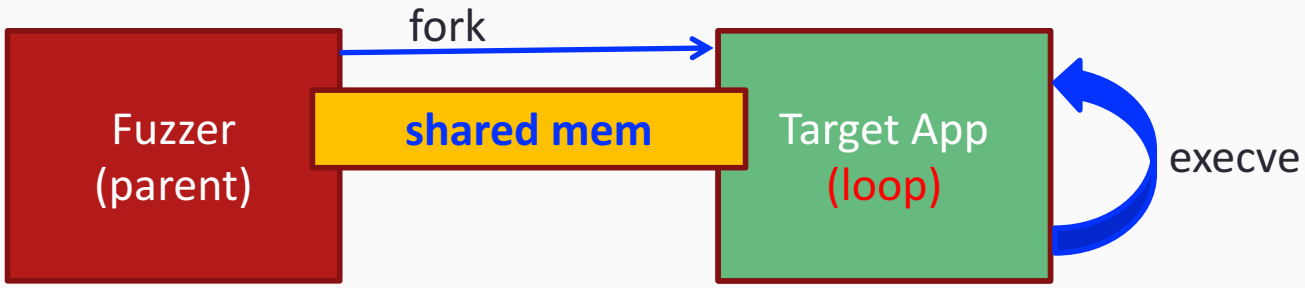
- execve is still slow
  - prepare the Target App, and fork it without execve



fork → execve → forkserver → fork → test

Fastest: persistent mode

- fork is still slow
  - keep only one copy of target app process



fork → execve → loop → test

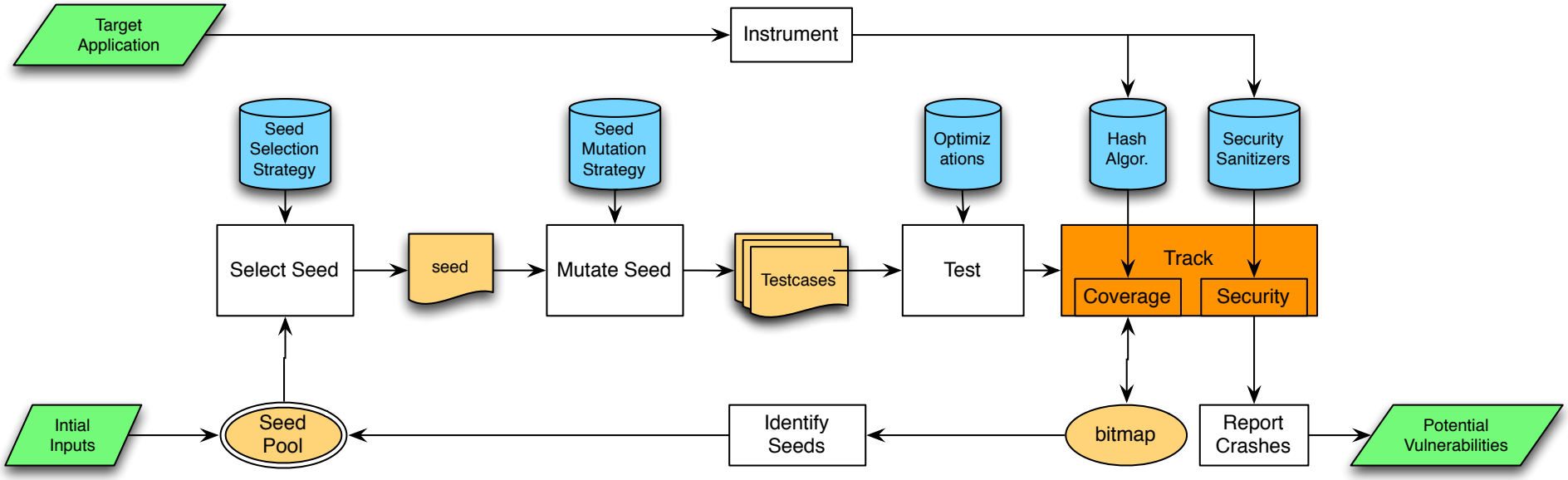
Sensitive: catch potential bugs

- Security violations:
  - AddressSanitizer
  - UBSan
  - MemorySanitizer
  - ThreadSanitizer
  - DataFlowsanitizer
  - LeakSanitizer
  - …

AddressSanitizer:

- buffer overflow
  - keeps track of whether memory is addressable
  - places un-addressable redzone around objects
  - check each object access
- Use after free
  - Quarantine for free-ed memory

Research Questions


Target Application
Instrument
Seed Selection Strategy
Seed Mutation Strategy
Optimiz ations
Hash Algor.
Security Sanitizers
Select Seed
seed
Mutate Seed
Testcases
Test
Track
Coverage
Security
Intial Inputs
Seed Pool
Identify Seeds
bitmap
Report Crashes
Potential Vulnerabilities

Key Questions of Fuzzing

- How to get initial inputs?
- How to select seed from the pool?
- How to generate new testcases?
  - How to mutate seeds? Location and value.
- How to efficiently test target application?
- How to track the testing?
  - Code coverage, Security violation, ...?
- How do we update the seed pool?
  - identify good testcases, shrink seed pool...

How to get initial inputs?

- Why is it important?
  - cpu time
  - complex data structure
  - hard-to-reach code
  - reusable between fuzzings
- Solutions
  - standard benchmarks
  - crawling from the Internet
- Extra step
  - distill the corpus

How to select seed from the pool?

- Why is it important?
  - prioritize seeds which are more helpful,
    - e.g., cover more code, more likely to trigger vulnerabilities
  - save computing resources
  - faster to identify hidden vulnerabilities
- Solutions
  - AFLFast (CCS’ 16): seeds being picked fewer or exercising less-frequent paths
  - Vuzzer (NDSS’ 17): seeds exercising deeper paths
  - QTEP (FSE’ 17): seeds covering more faulty code
  - AFLgo (CCS’ 17): seeds closer to target vulnerable paths
  - SlowFuzz (CCS’ 17): seeds consuming more resources

How to generate new testcases?

- Why is it important?
  - explore more code in a shorter time
  - target potential vulnerable locations
- Solutions
  - Vuzzer (NDSS’ 17):
    - where to mutate: bytes related to branches
    - what value to use: tokens used in the code.
  - Skyfire (Oakland’ 17):
    - learn Probabilistic Context-Sensitive Grammar from crawled inputs
  - Learn&Fuzz (Microsoft):
    - learn RNN from valid inputs

How to efficiently test application?

- Why is it important?
  - test more in a unit time
  - very important
- Solutions:
  - fork + execve
  - forkserver
  - persistent mode
  - Intel PT
  - ...

How to track the testing?

- Why is it important?
  - Code coverage: leading to thorough program states exploring
  - Security violations: capturing bugs that have no explicit results
- Solutions
  - Code coverage:
    - AFL bitmap, SanitizerCoverage
  - Security violations:
    - AddressSanitizer
    - UBSan
    - MemorySanitizer
    - ThreadSanitizer
    - DataFlowsanitizer
    - LeakSanitizer
    - ...

Fuzzing in real world

- Dumb enough, easy to use, but effective!
  - VERY popular in industry
- Key to find more vulnerabilities
  - domain knowledge
  - write your own mutation algorithm for your target application

纯干货：微软漏洞中国第一人黄正——如何用正确姿势挖掘浏览器漏洞（附完整 PPT）｜硬创公开课

Conclusions

- Fuzzing is the most popular vulnerability discovery solution.
- AFL is one of the most popular fuzzer, studied by researchers from academia and industry.
  - scalable, fast, evolving, sensitive
- We could improve AFL in many ways.
- Fuzzing is a hot research topic.

? &