FUZZIFICATION: Anti-Fuzzing Techniques

Jinho Jung, Hong Hu, David Solodukhin, Daniel Pagan,
Kyu Hyung Lee*, Taesoo Kim

Georgia Tech

UNIVERSITY OF GEORGIA *

Fuzzing Discovers Many Vulnerabilities

50 CVEs in 50 Days: Fuzzing Adobe Reader

December 12, 2018

Research By: Yoav Alon, Netanel Ben-Simon

Fuzzing Discovers Many Vulnerabilities

50 CVEs in 50 Days: Fuzzing Adobe Reader

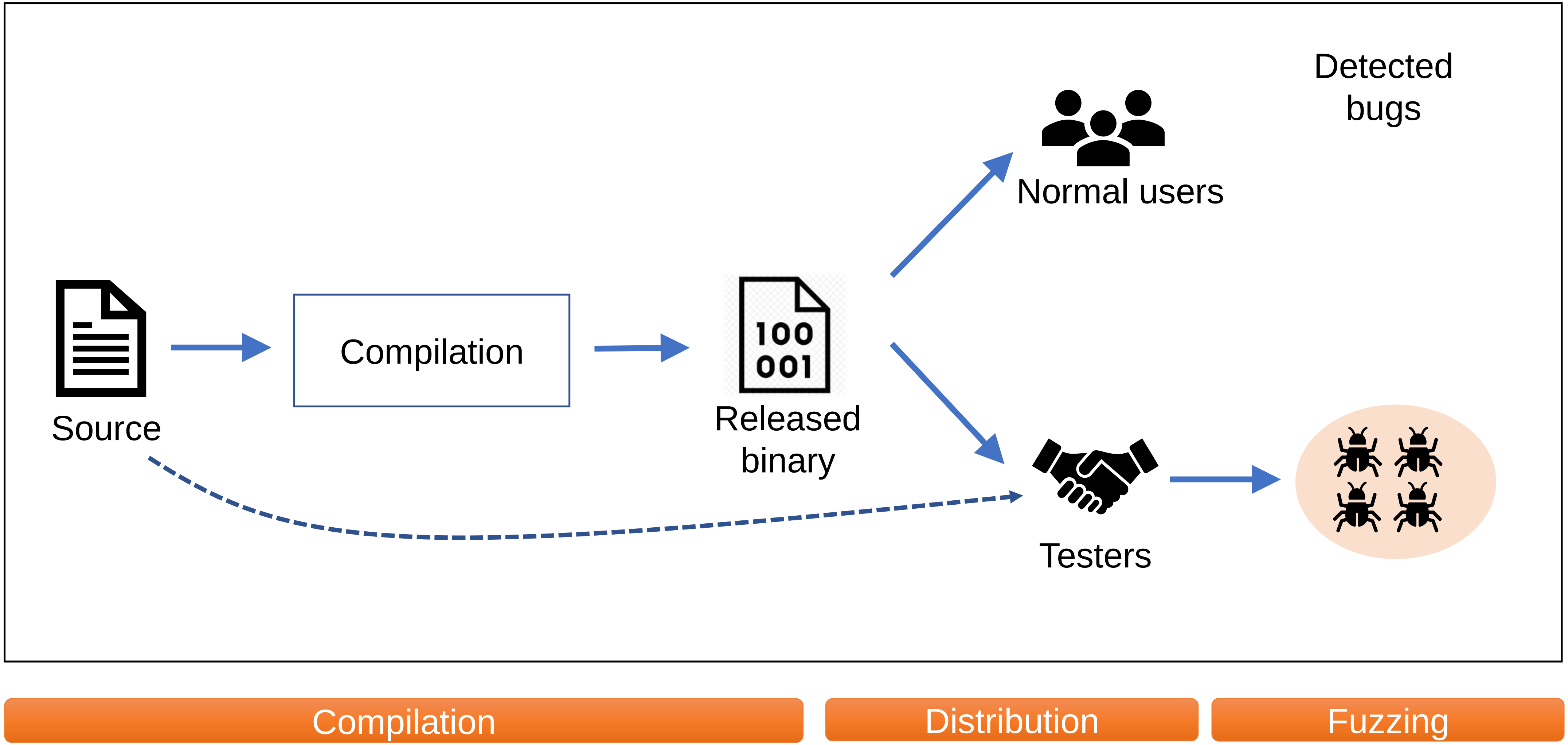
December 12, 2018

Research By: Yoav Al

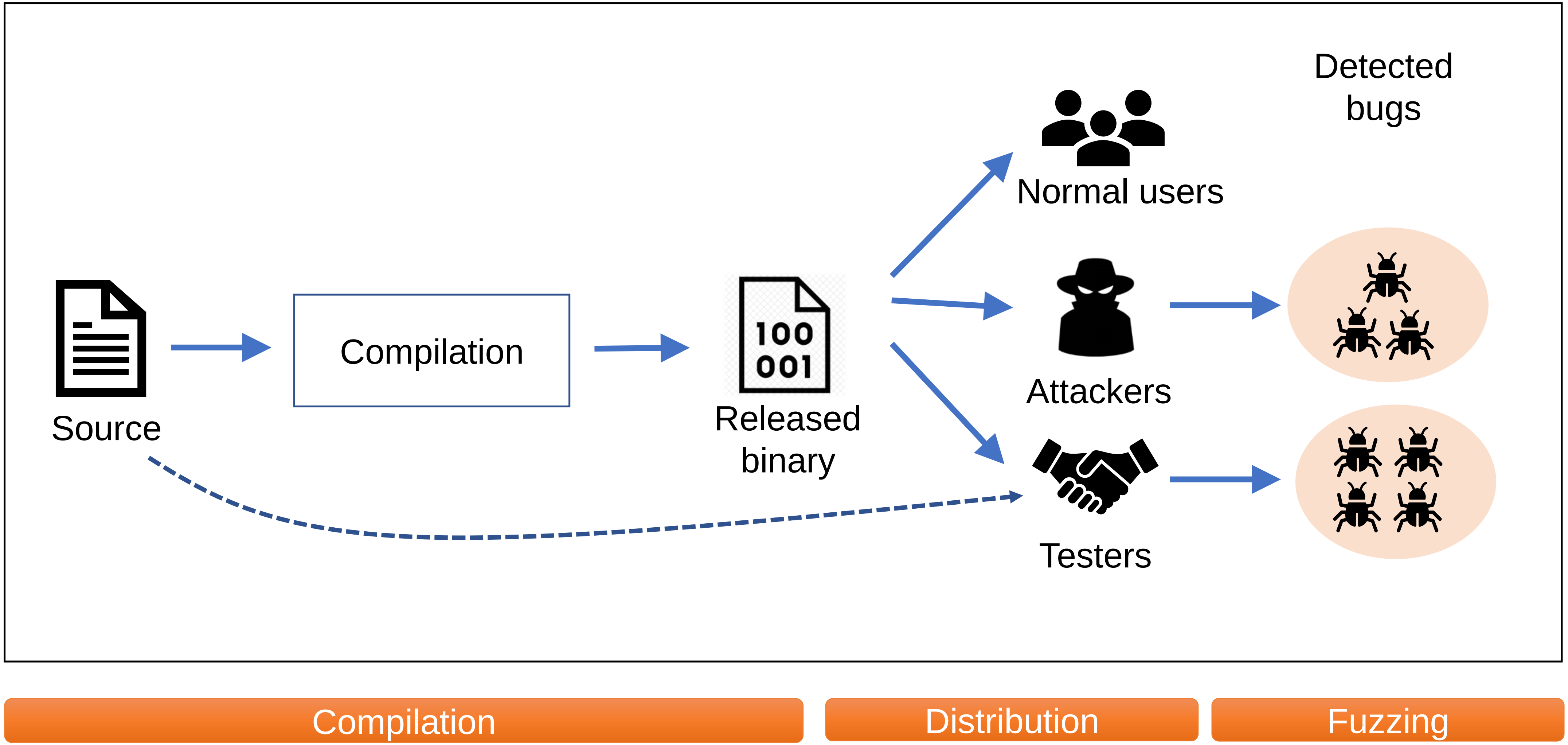
Google's automated fuzz bot has found over 9,000 bugs in the past two years

Google improves OSS-Fuzz service, plans to invite new op source projects to join.

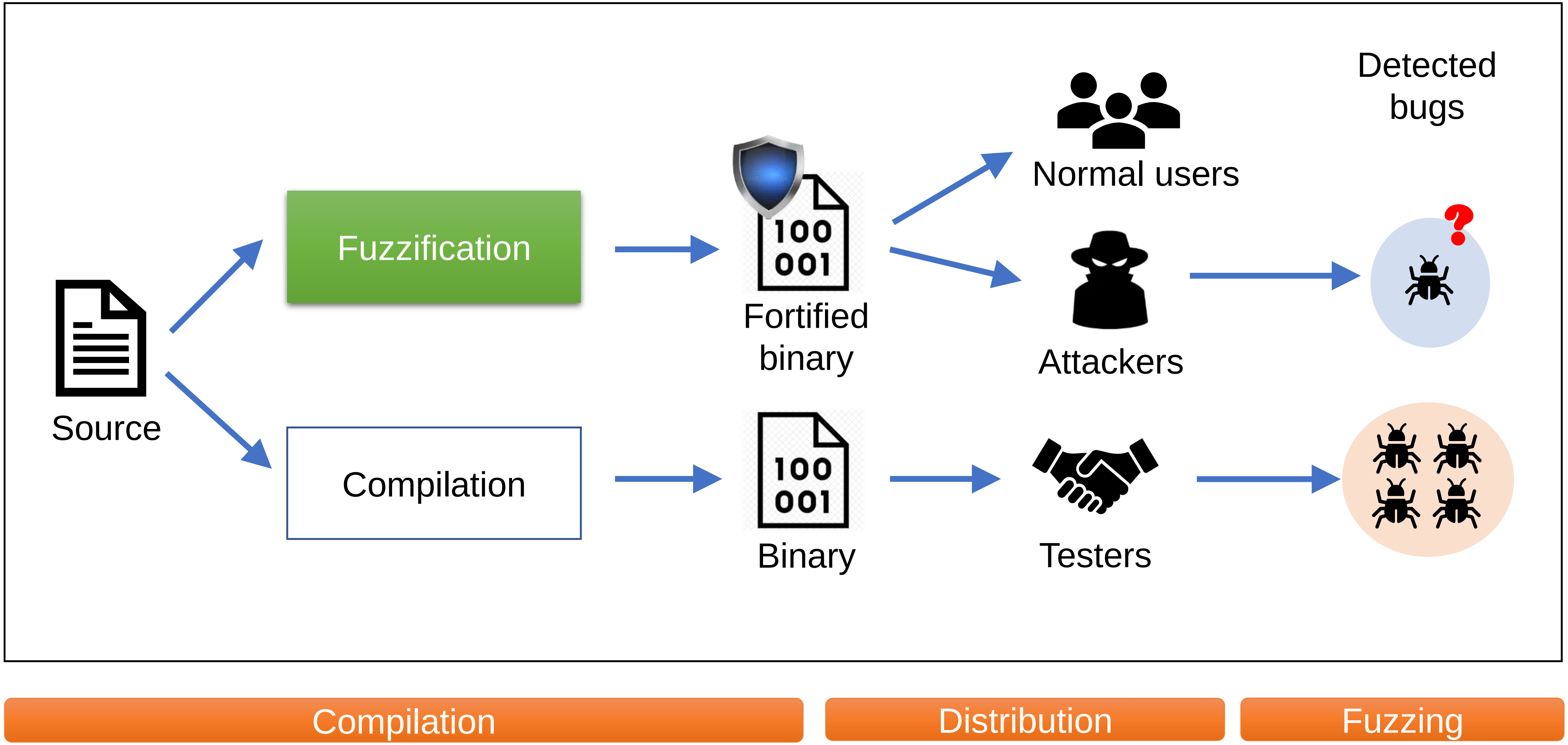
Testers Find Bugs with Fuzzing



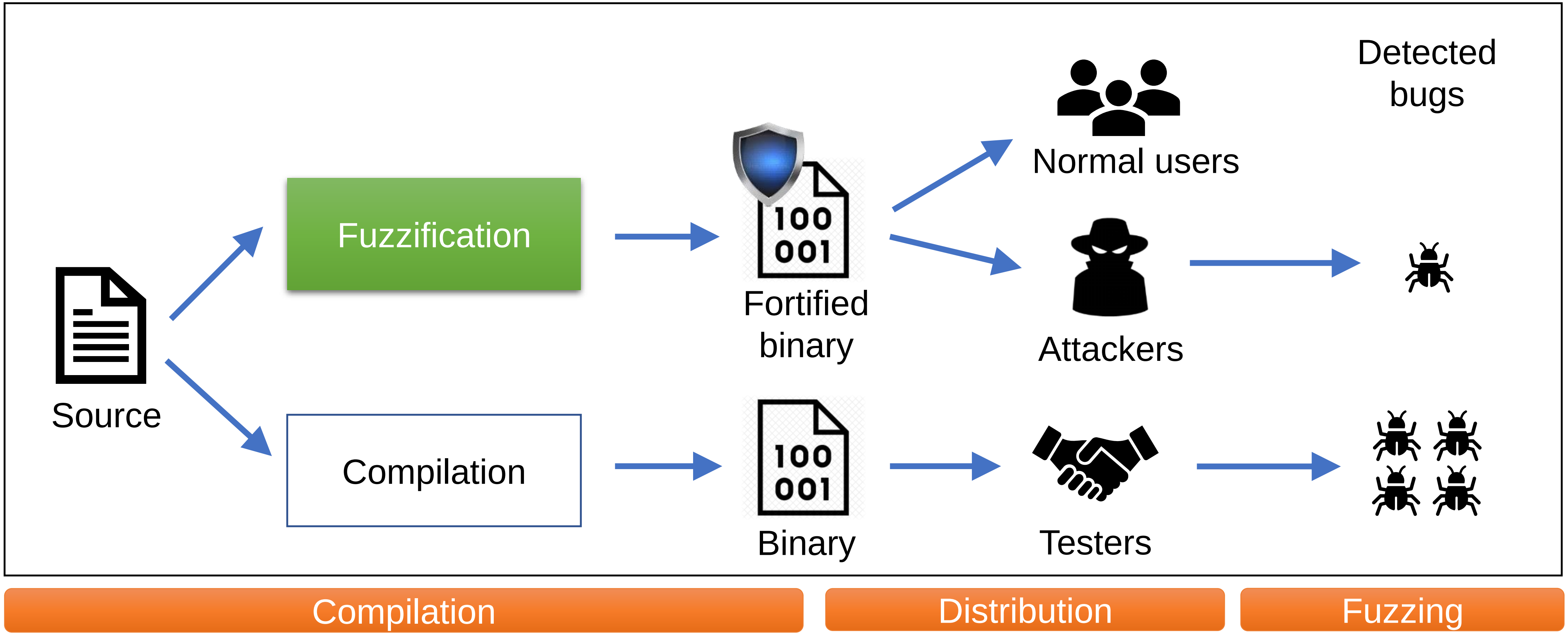
But Attackers Also Find Bugs 😔



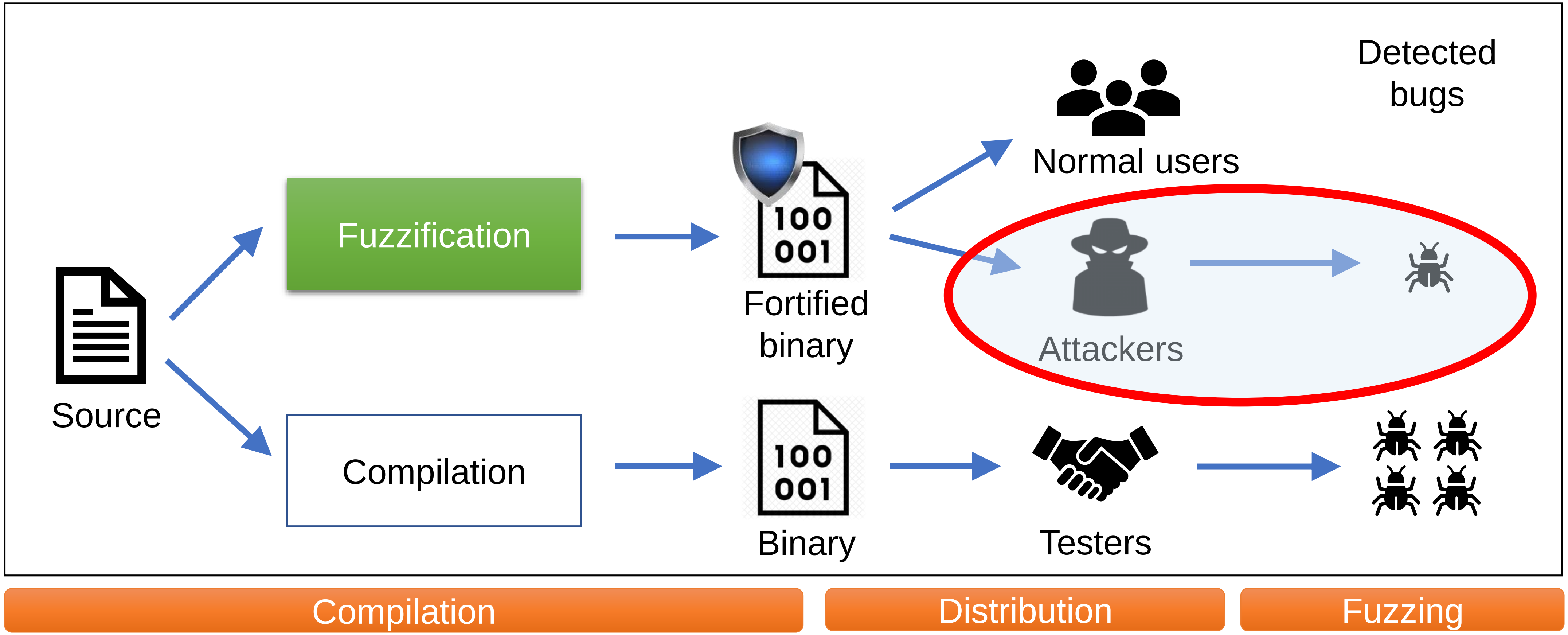
Our work: Make the Fuzzing Only Effective to the Testers



Threat Model

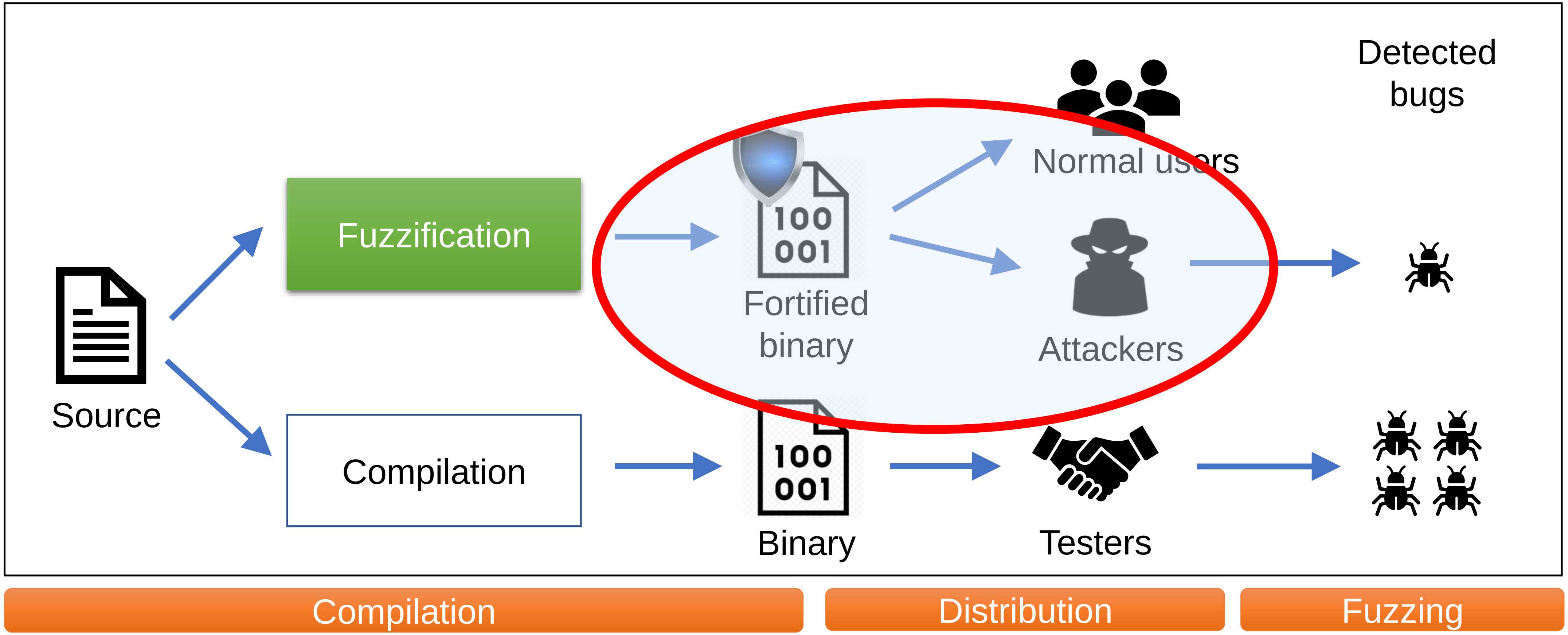


Threat Model



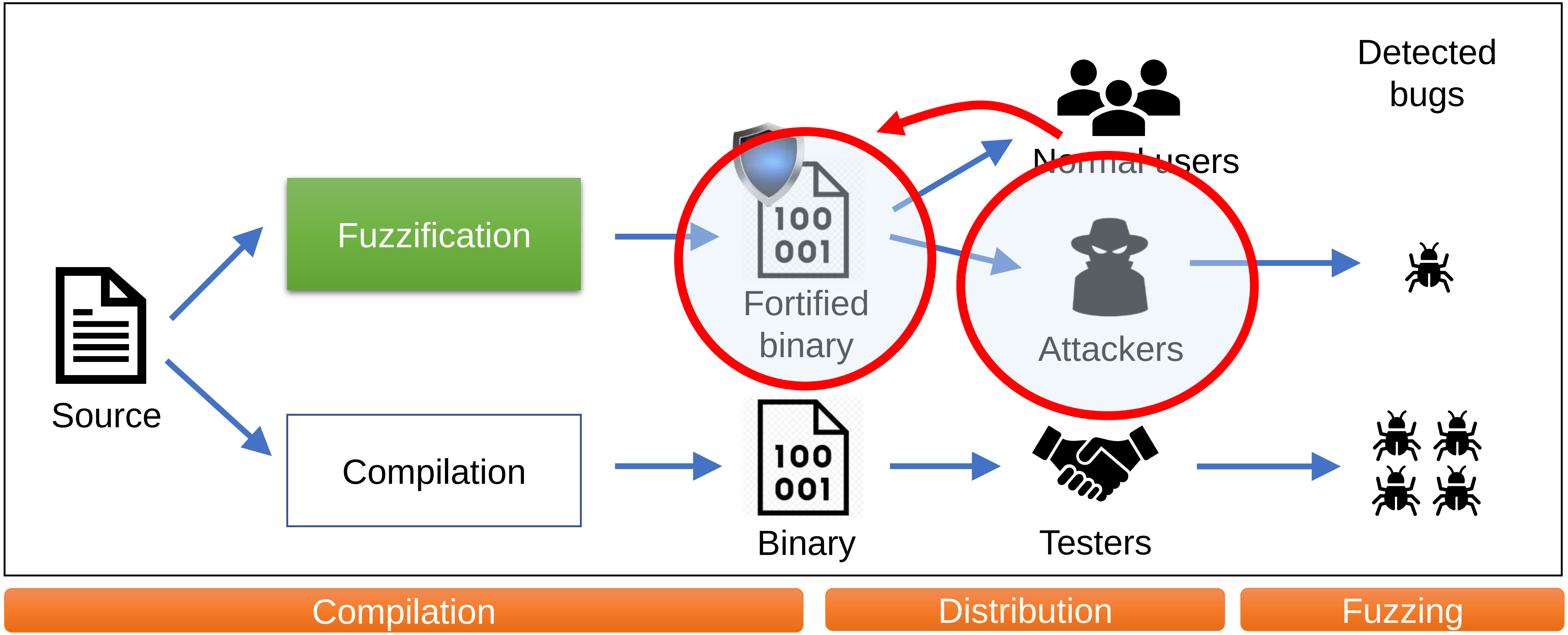
Adversaries try to find vulnerabilities from fuzzing

Threat Model



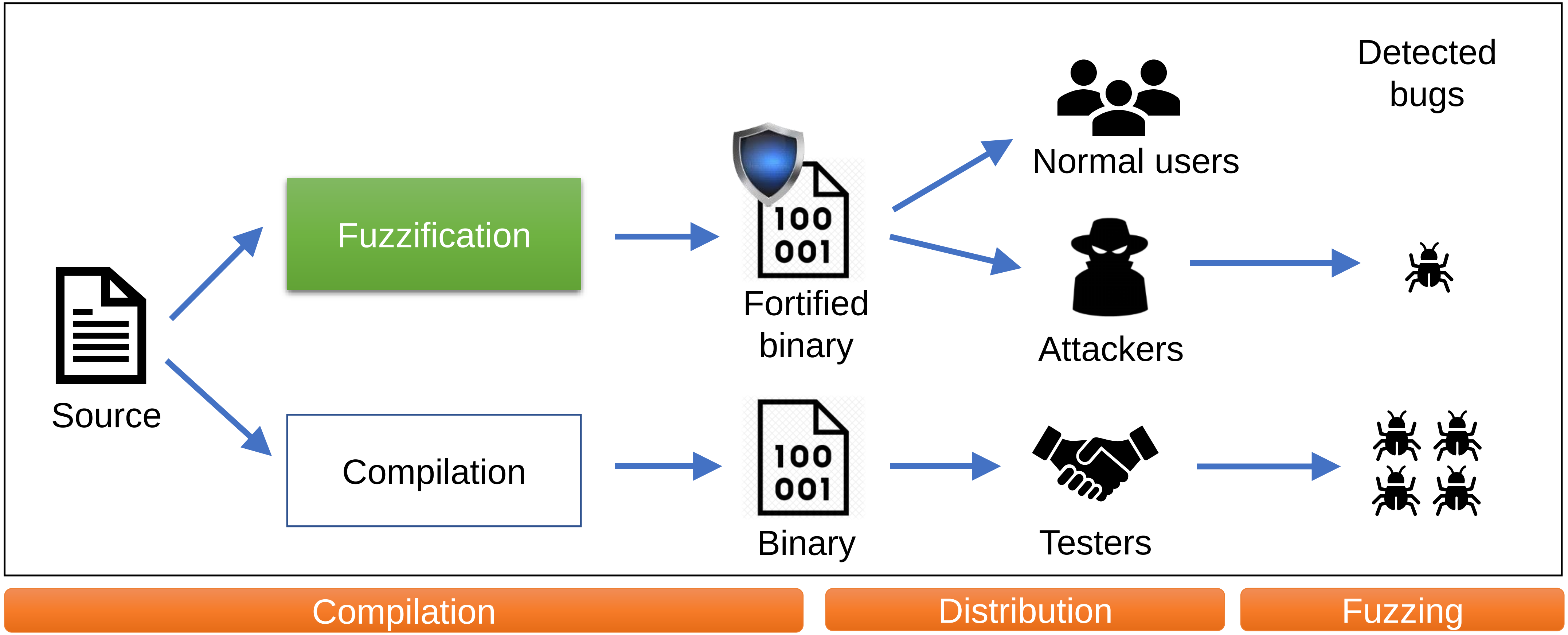
Adversaries only have a copy of fortified binary

Threat Model

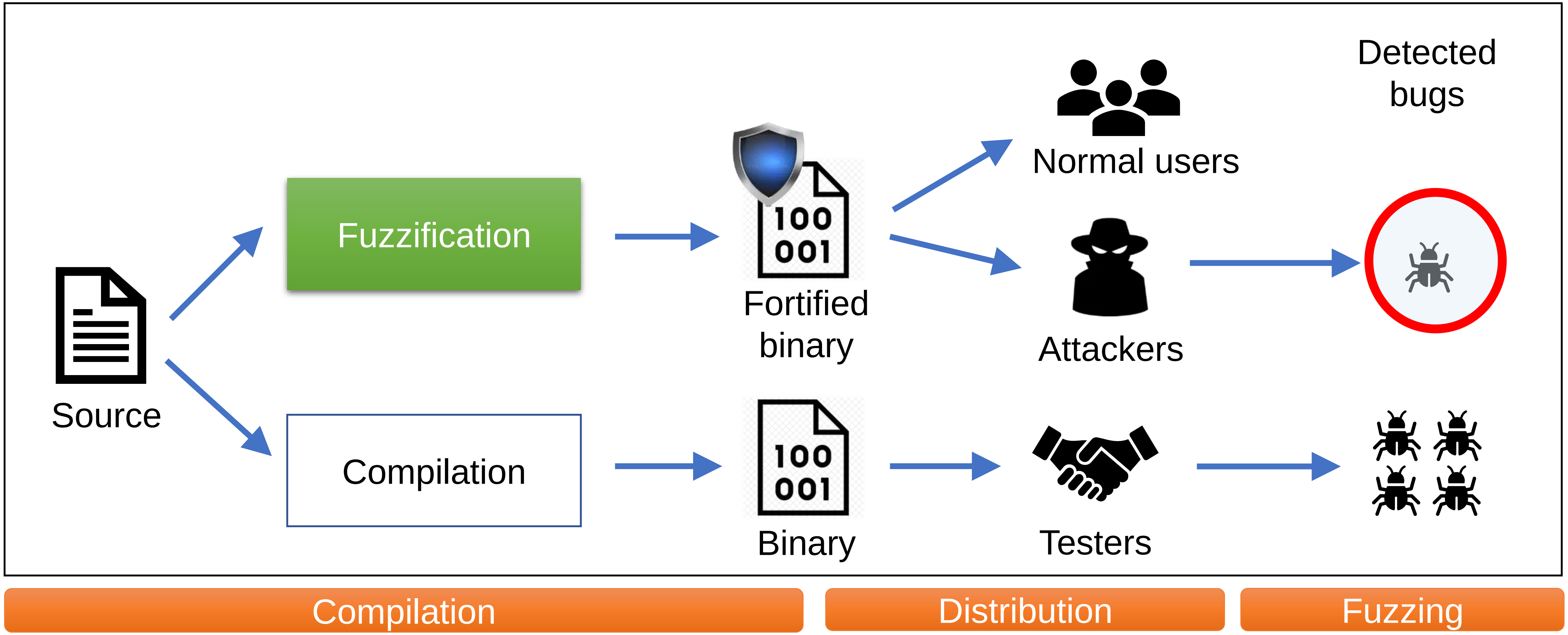


Adversaries know Fuzzification and try to nullify

Research Goals



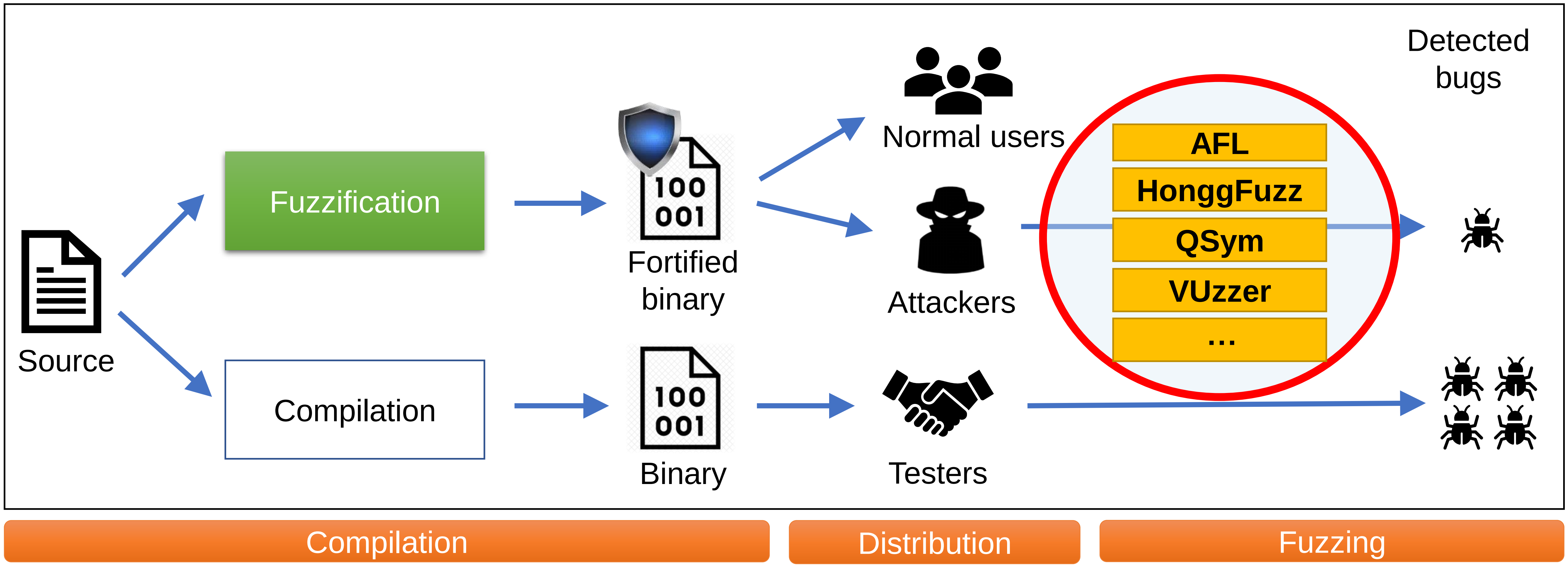
Research Goals



Hinder Fuzzing

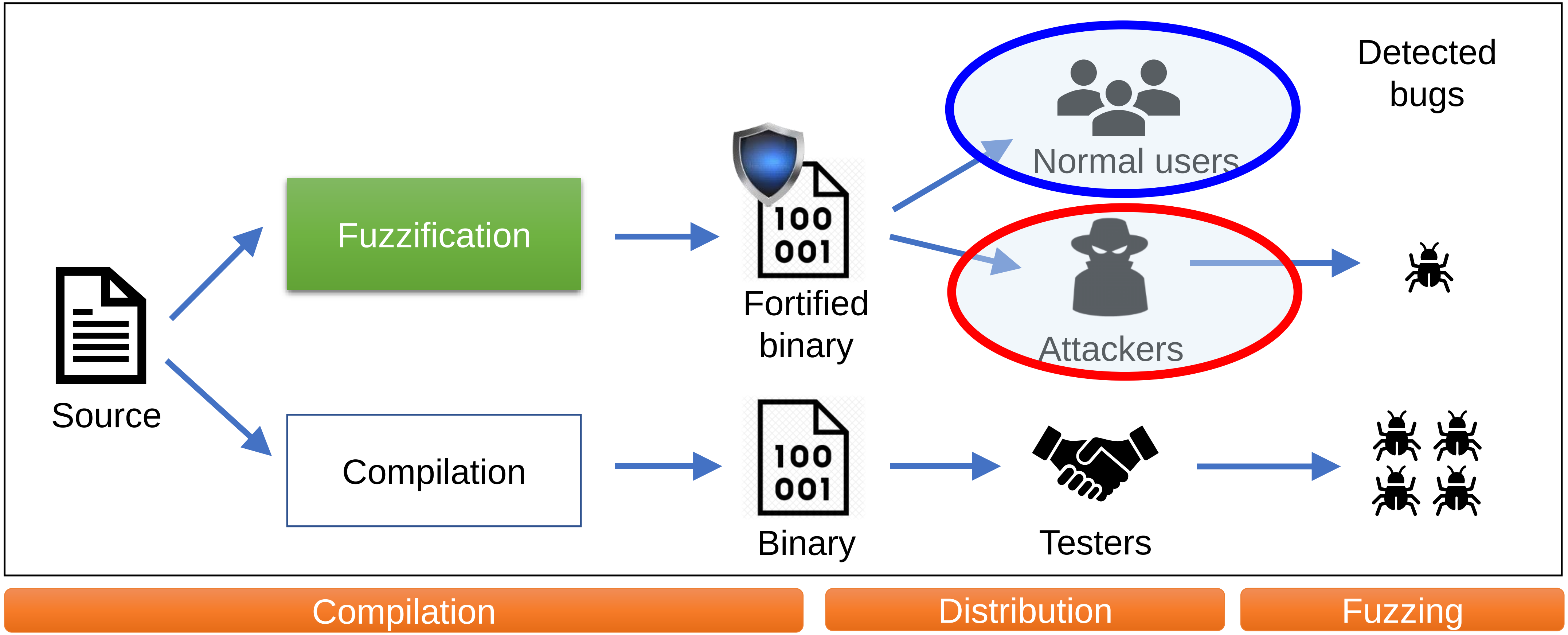
Reduce the number of detected bugs

Research Goals



Generic Affect most of the fuzzers

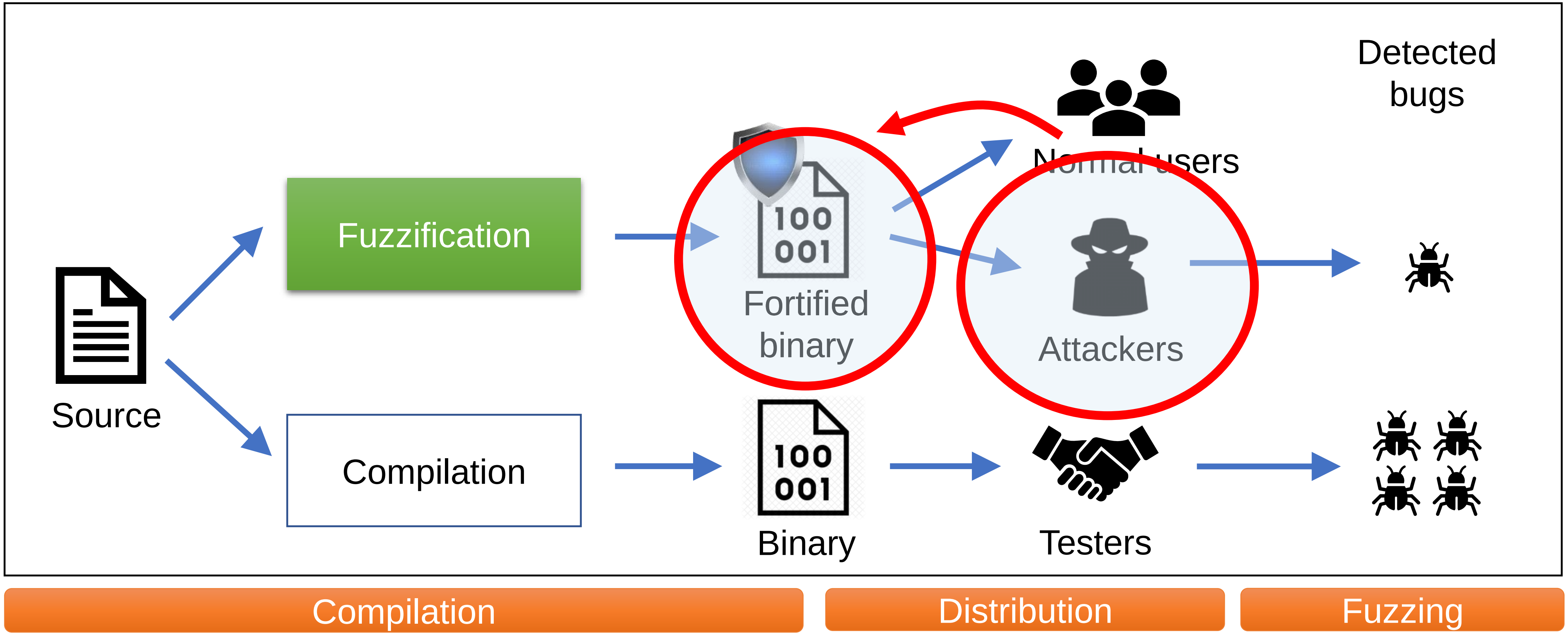
Research Goals



Overhead

Low overhead to normal user
High overhead to attackers

Research Goals



Resiliency Resilient to the adversarial analysis

Why Existing Methods Are Not Applicable?

Method	Generic to most fuzzers	Low overhead	Resilient to adversary
Packing or obfuscation	O	X	O

Why Existing Methods Are Not Applicable?

Method	Generic to most fuzzers	Low overhead	Resilient to adversary
Packing or obfuscation	O	X	O
Bug injection	O	O	X

Why Existing Methods Are Not Applicable?

Method	Generic to most fuzzers	Low overhead	Resilient to adversary
Packing or obfuscation	O	X	O
Bug injection	O	O	X
Fuzzer detection	X	O	X

Why Existing Methods Are Not Applicable?

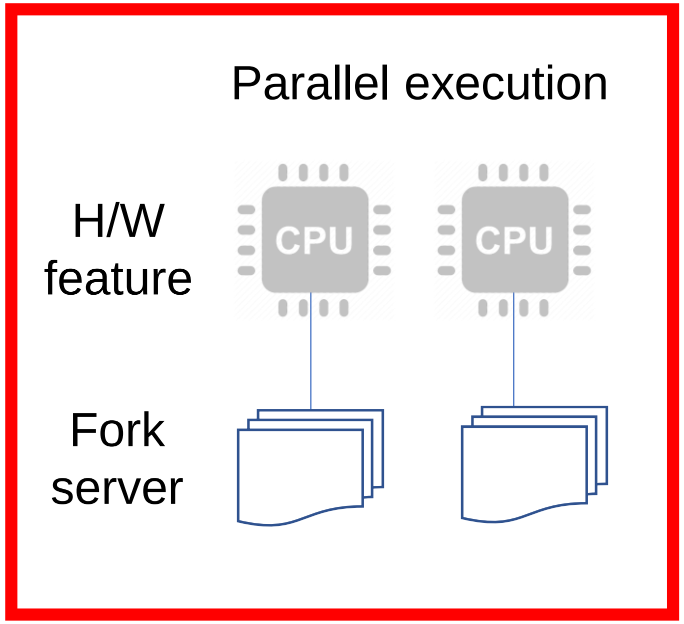
Method	Generic to most fuzzers	Low overhead	Resilient to adversary
Packing or obfuscation	O	X	O
Bug injection	O	O	X
Fuzzer detection	X	O	X
Emulator detection	$\boxed{X}$	O	$\boxed{X}$

Why Existing Methods Are Not Applicable?

Method	Generic to most fuzzers	Low overhead	Resilient to adversary
Packing or obfuscation	O	X	O
Bug injection	O	O	X
Fuzzer detection	X	O	X
Emulator detection	X	O	X
Fuzzification	O	O	O

Fuzzification Hinders Advanced Features

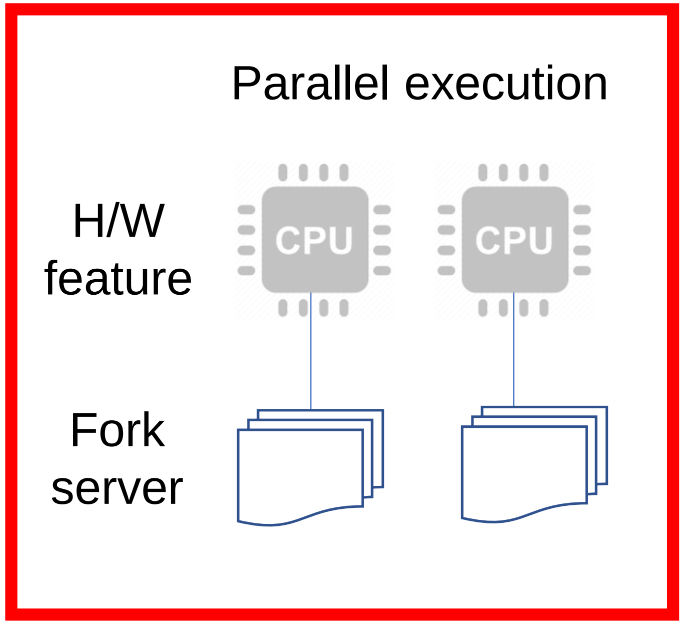
- **Fast execution**
- Coverage-guidance
- Hybrid approach



Fuzzification Hinders Advanced Features

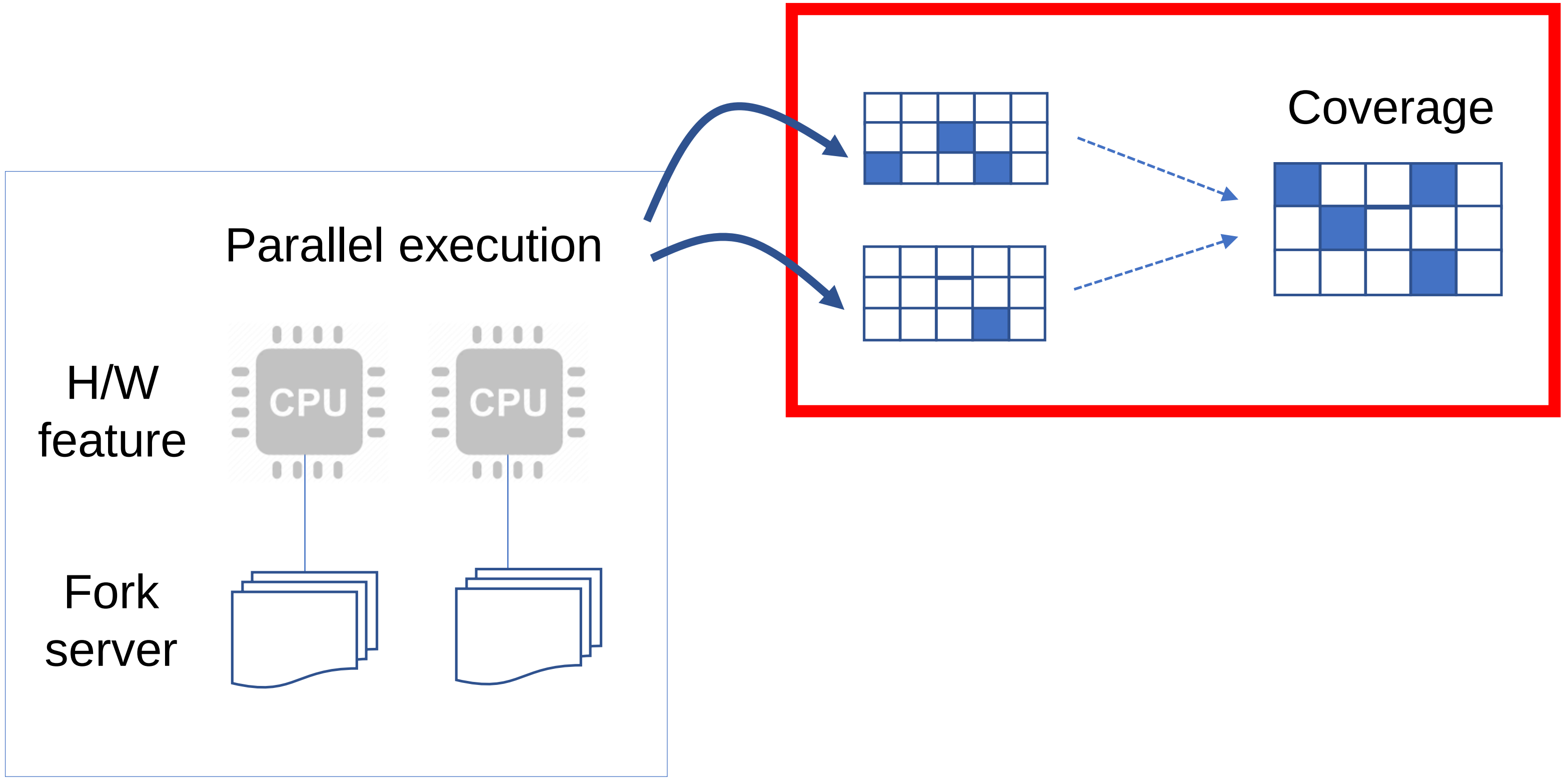
- ~~Fast execution~~
- Coverage-guidance
- Hybrid approach

SpeedBump



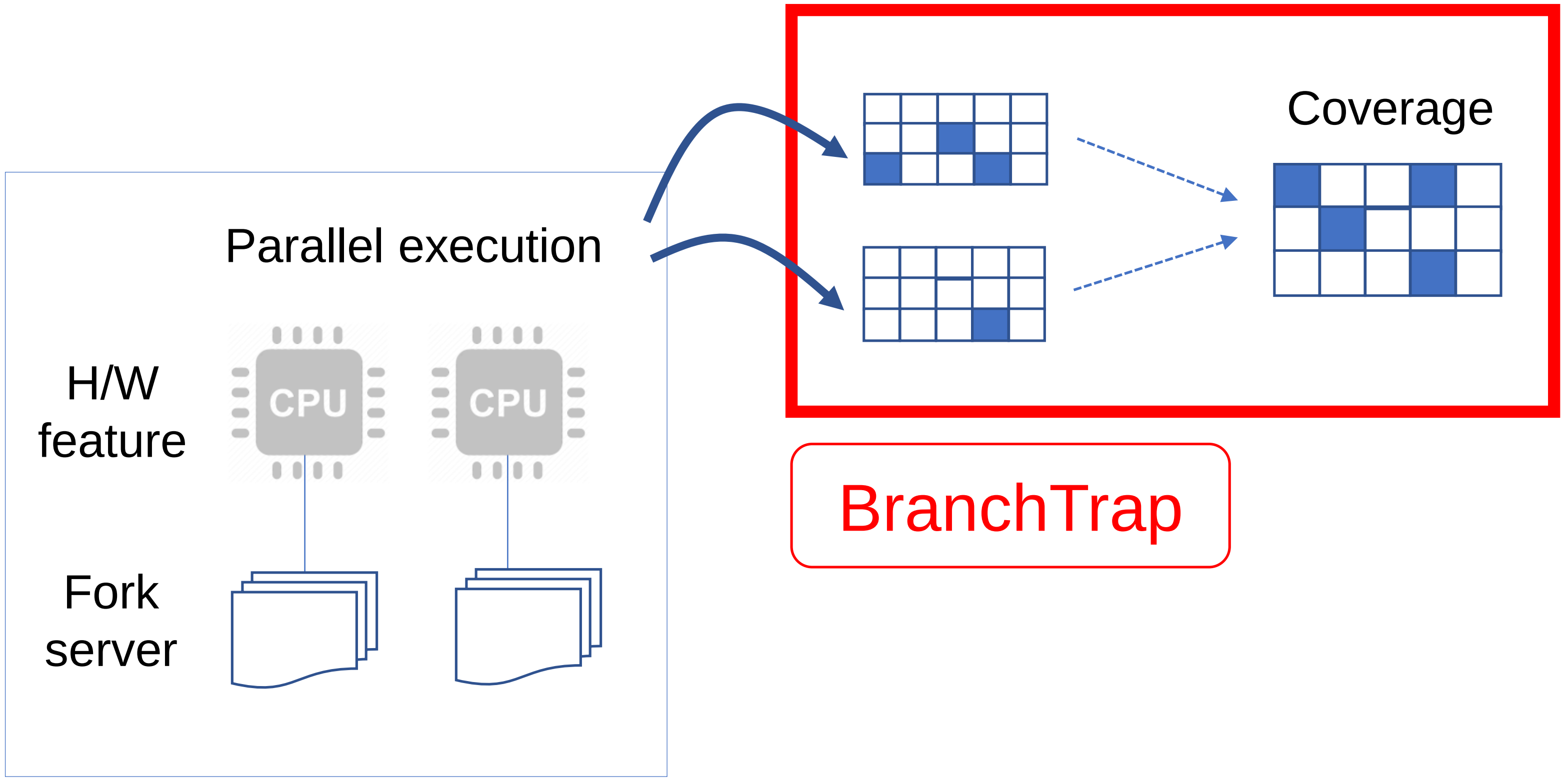
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach



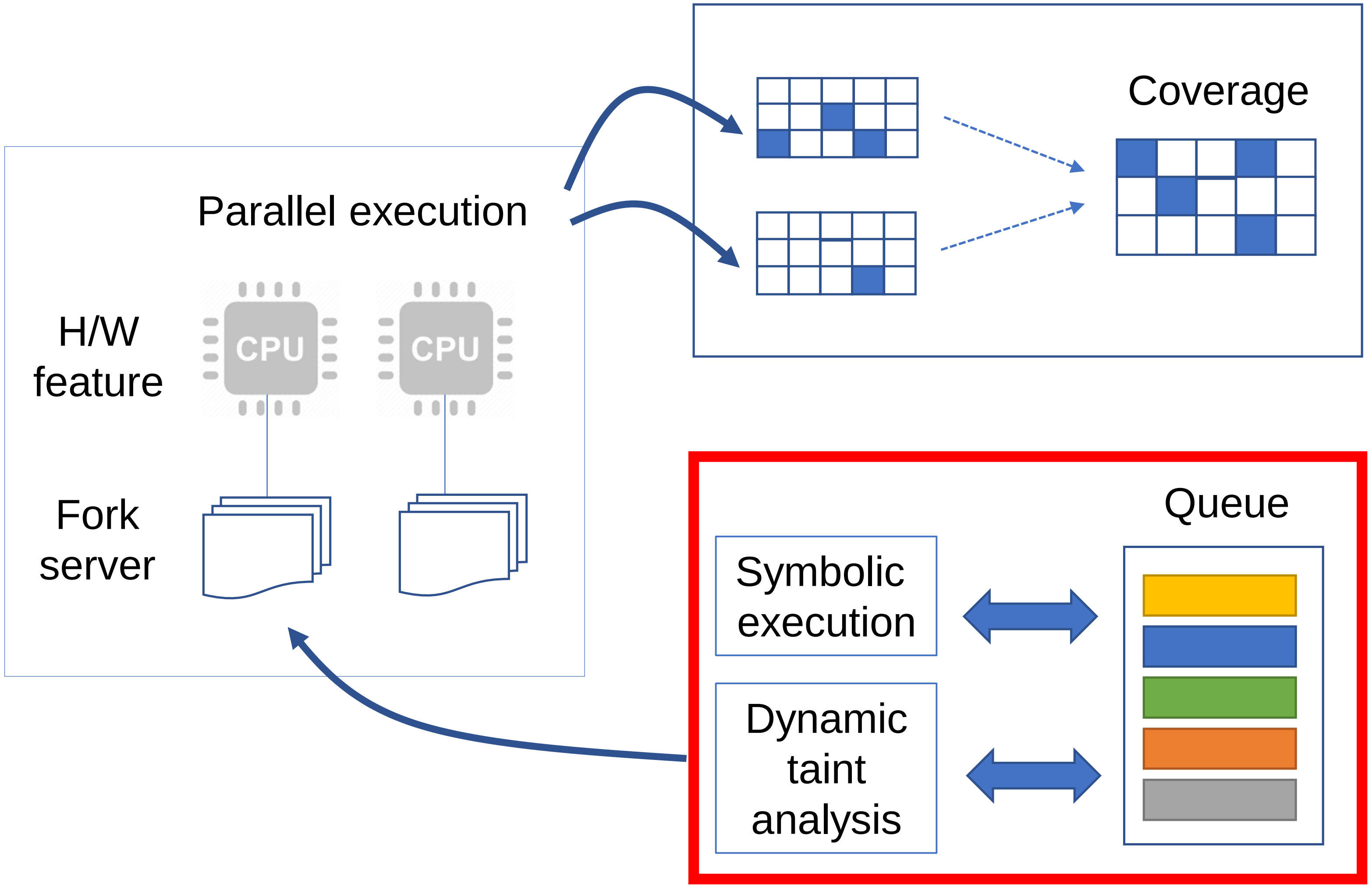
Fuzzification Hinders Advanced Features

- Fast execution
- ~~Coverage-guidance~~
- Hybrid approach



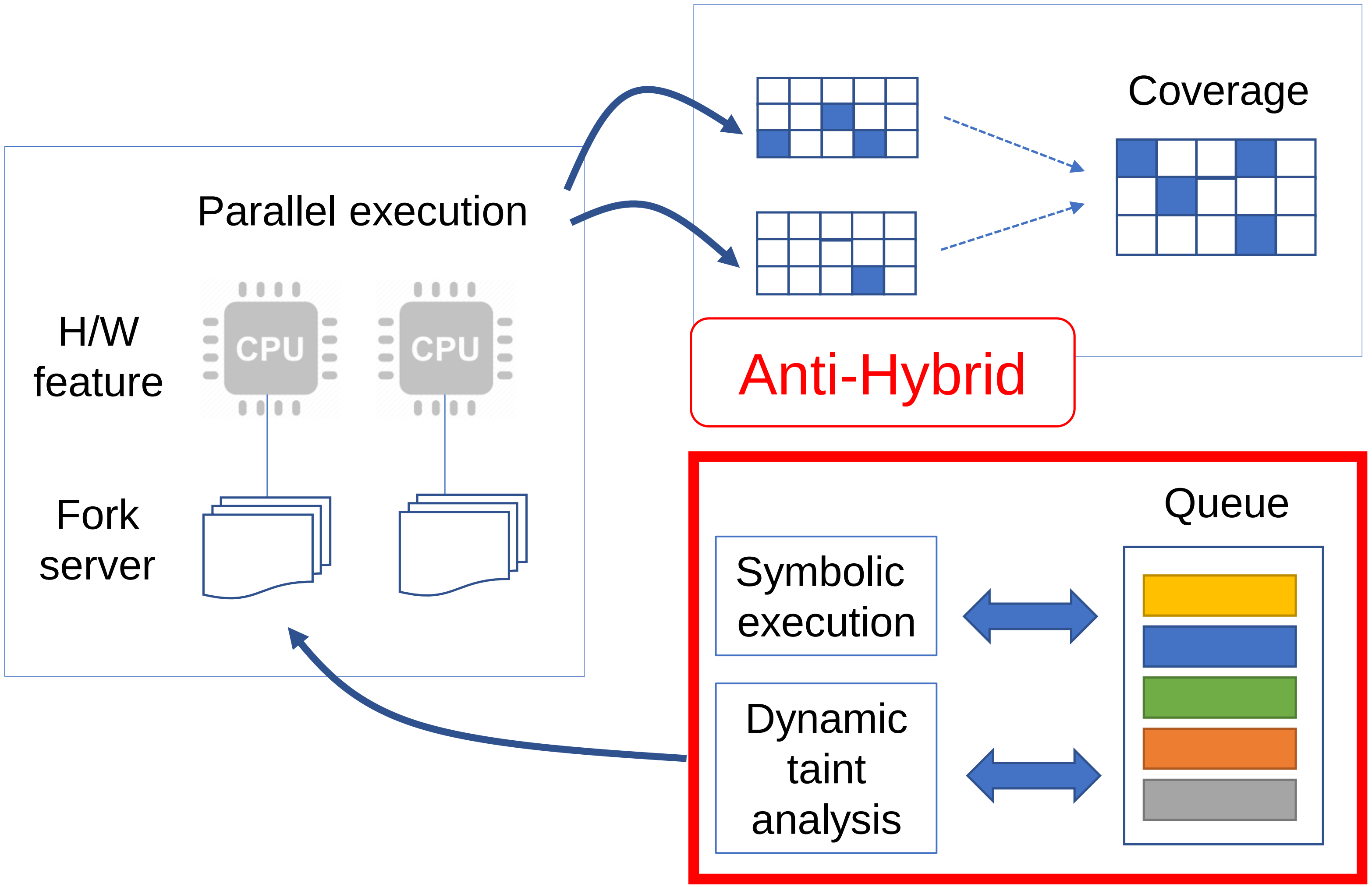
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- Hybrid approach

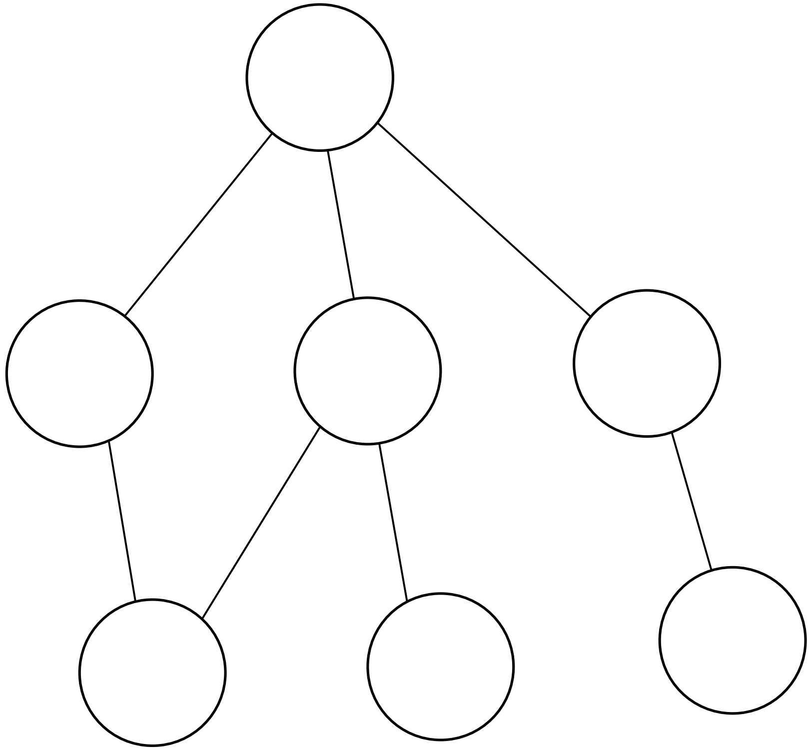


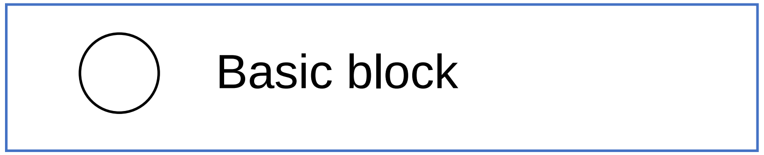
Fuzzification Hinders Advanced Features

- Fast execution
- Coverage-guidance
- ~~**Hybrid approach**~~

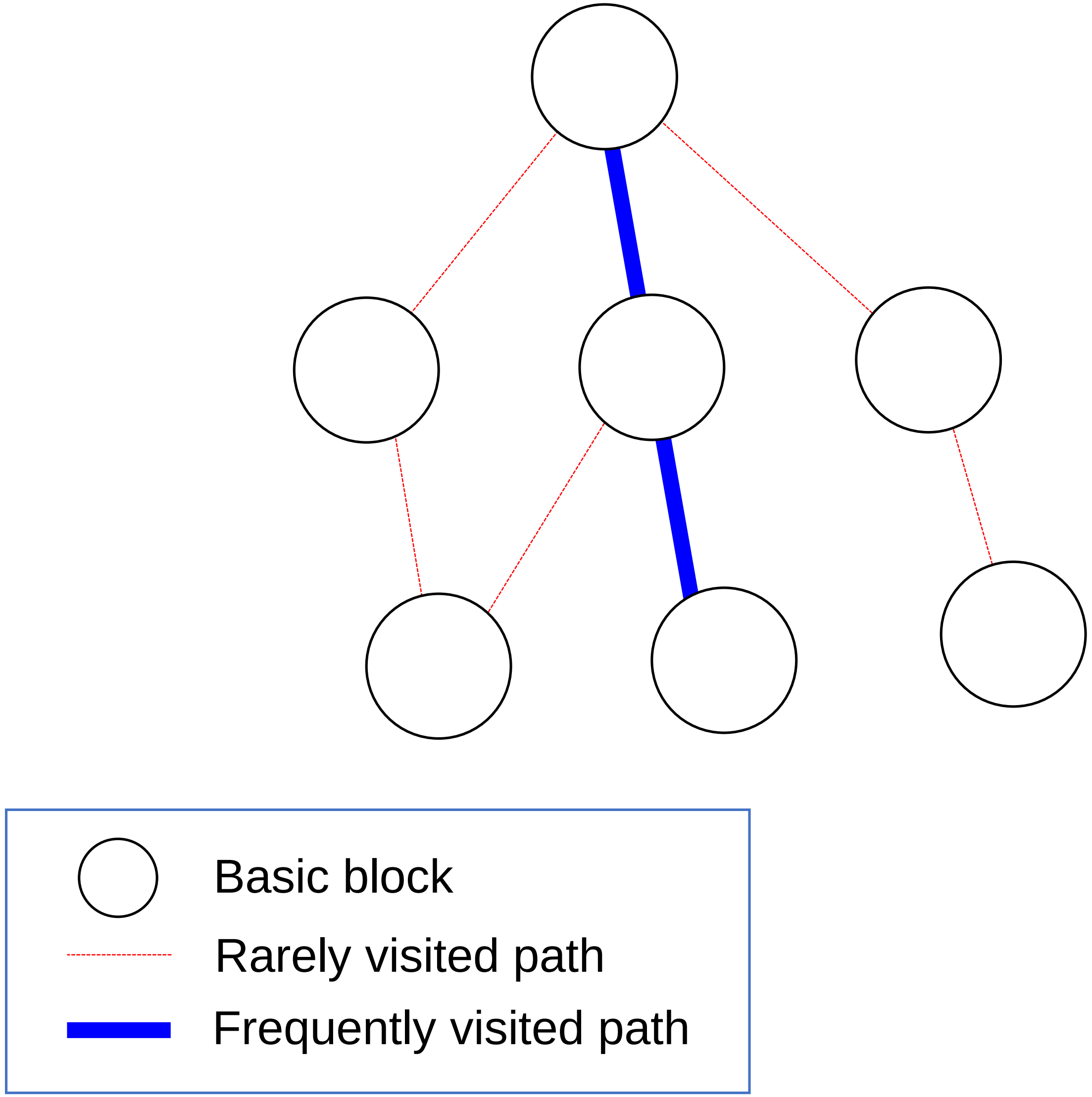


SpeedBump: Selective Delay Injection



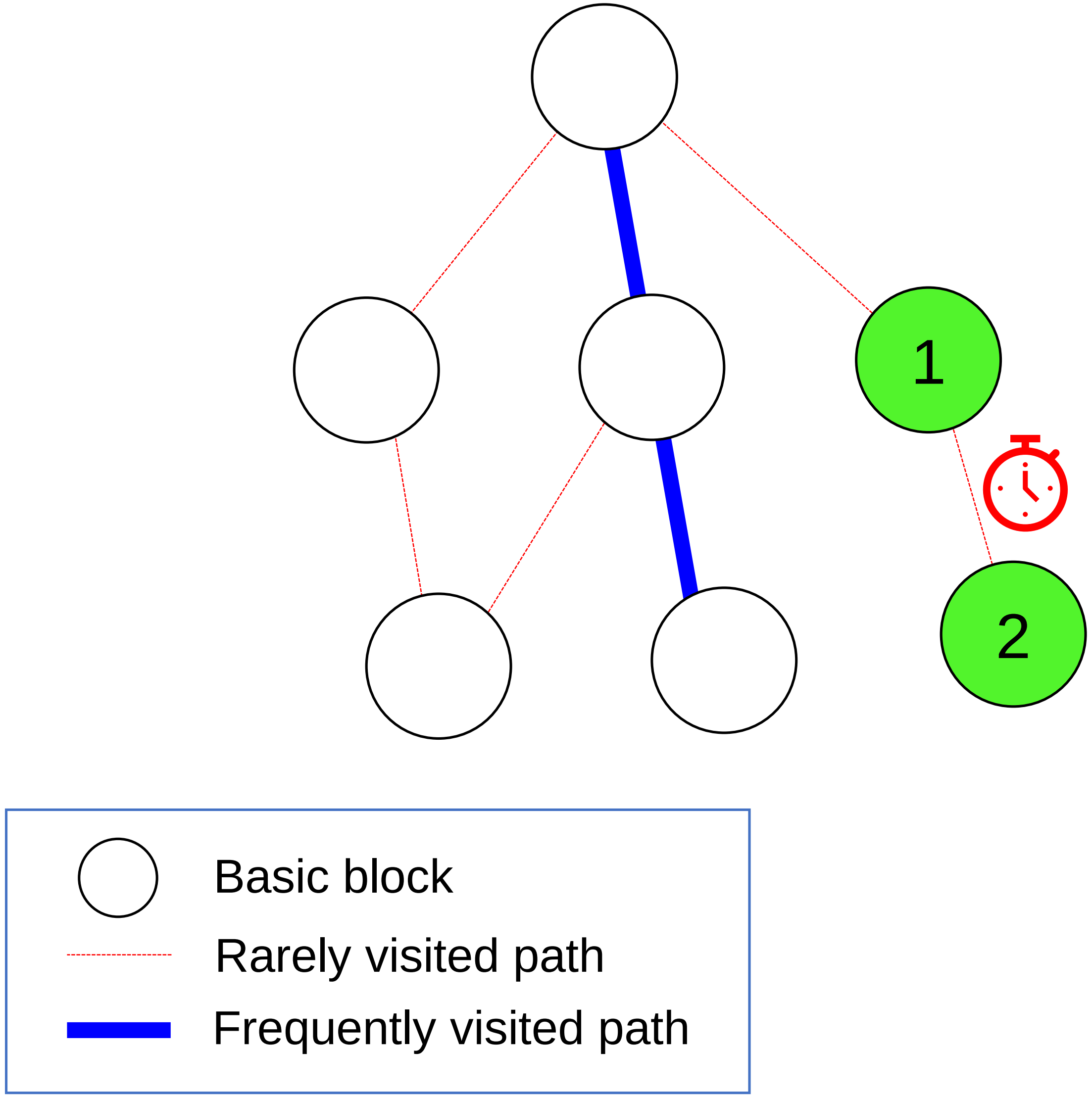


SpeedBump: Selective Delay Injection



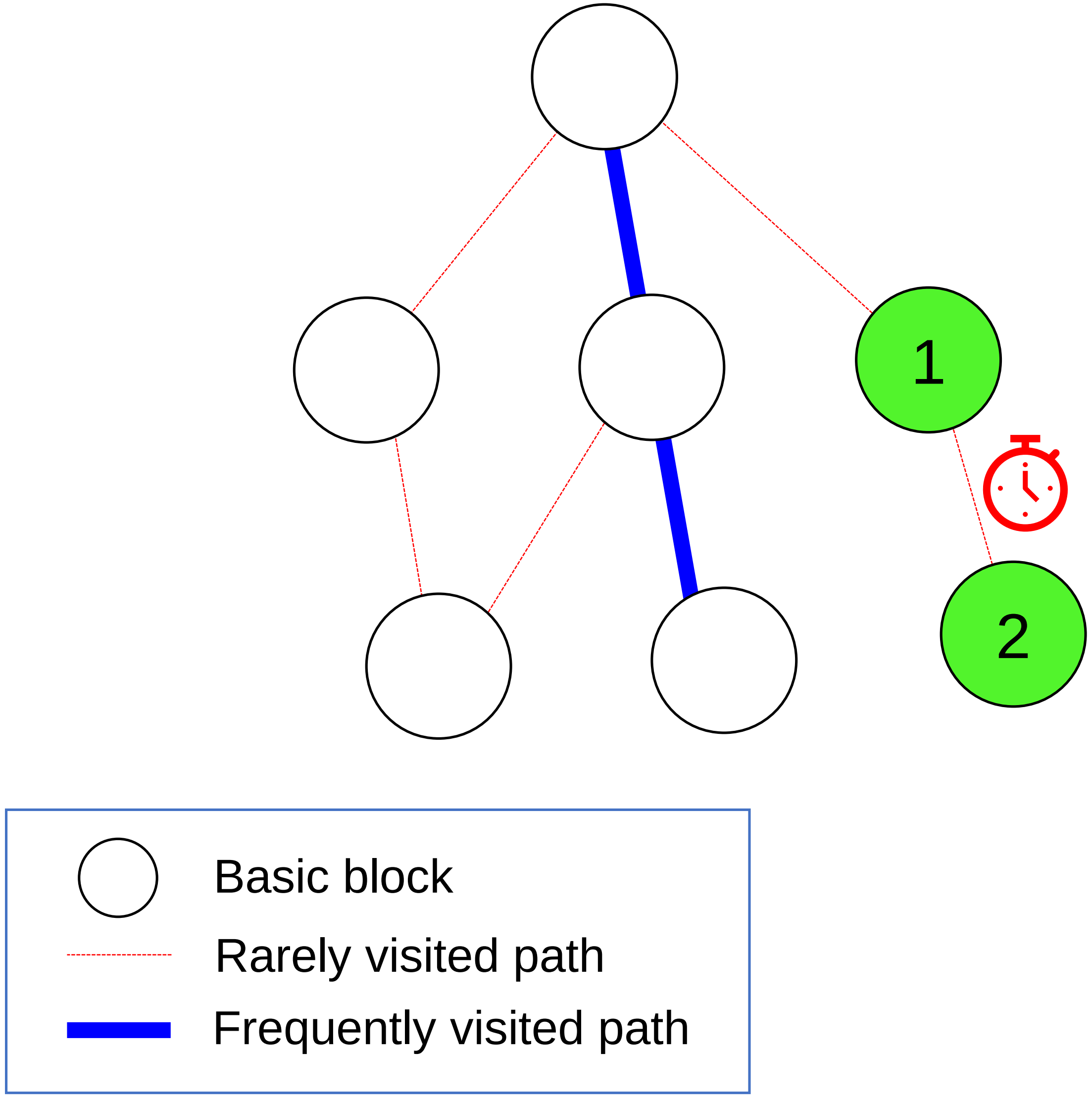
- Identify **frequently** and rarely visited paths

SpeedBump: Selective Delay Injection



- Identify **frequently** and rarely visited paths
- Inject delays from the most rarely visited edges

SpeedBump: Selective Delay Injection



- Why this is effective?
  - User: follows common paths
  - Attacker: searches for new paths

➔ Impact of delay is more significant to attackers

SpeedBump: How to delay?

- Strawman: using sleep()
  ➔ trivially removed by adversary

SpeedBump: How to delay?

- Strawman: using sleep()
  ➔ trivially removed by adversary

- Counter to advanced adversary
  - Use randomly generated code
    ➔ avoid static-pattern

SpeedBump: How to delay?

- Strawman: using sleep()
  ➔ trivially removed by adversary

- Counter to advanced adversary
  - Use randomly generated code
    ➔ avoid static-pattern
  - Impose control-flow and data-flow dependency
    ➔ avoid automated analysis

SpeedBump: Selective Delay Injection

```
int rarely_executed_code ( )
{
    return 0;
}
```

SpeedBump: Selective Delay Injection

```c
int rarely_executed_code ()
{
    return 0;
}
```

```c
//define global variables
int global1 = 1;
int global2 = 2;

int rarely_executed_code ()
{
    //inject delay function
    int pass = 20;
    global2 = func(pass);
    return 0;
}
```

SpeedBump: Selective Delay Injection

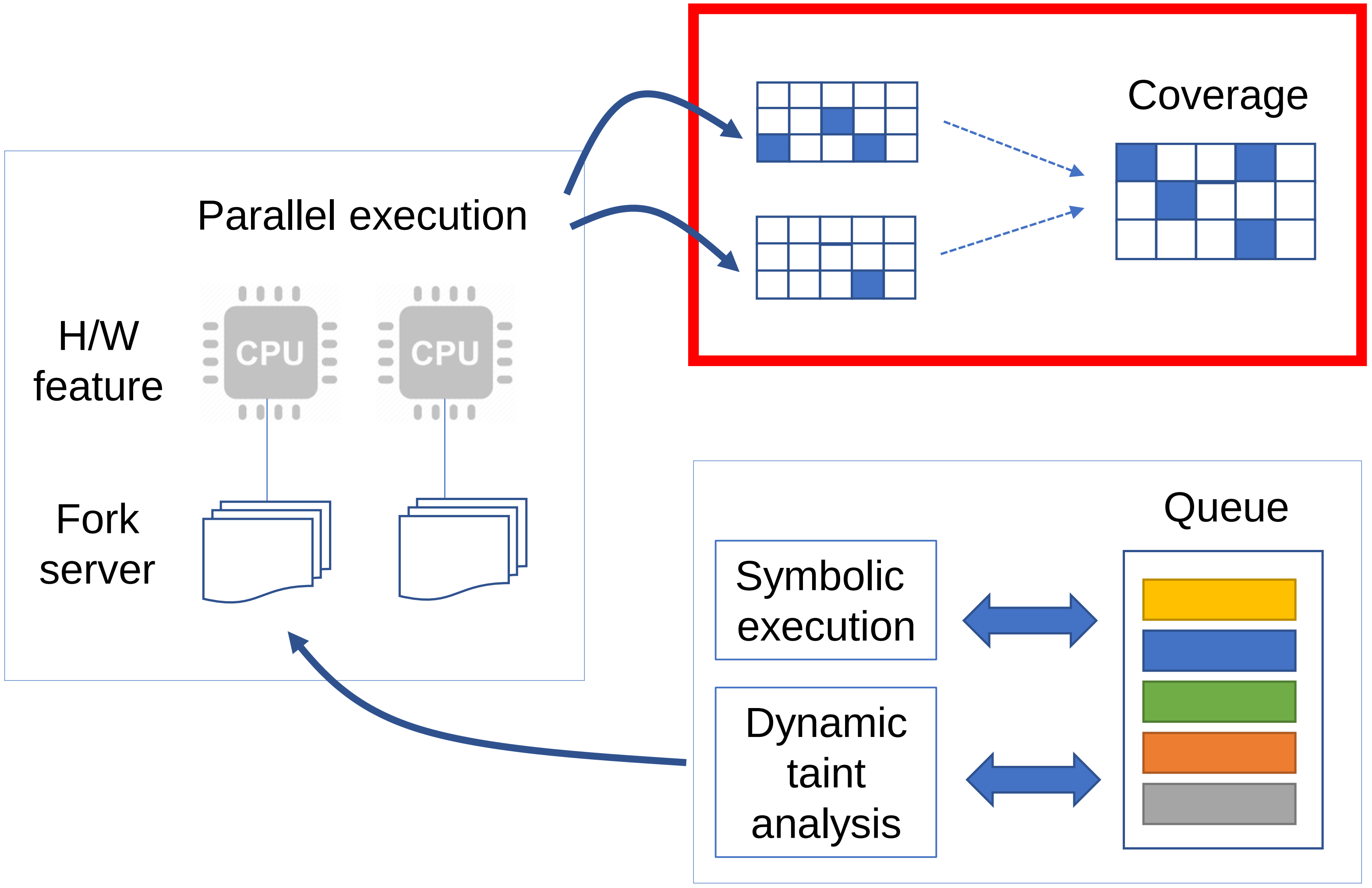
```c
int rarely_executed_code ()
{
    return 0;
}
```

```c
//define global variables
int global1 = 1;
int global2 = 2;

int rarely_executed_code ()
{
    //inject delay function
    int pass = 20;
    global2 = func(pass);
    return 0;
}
```

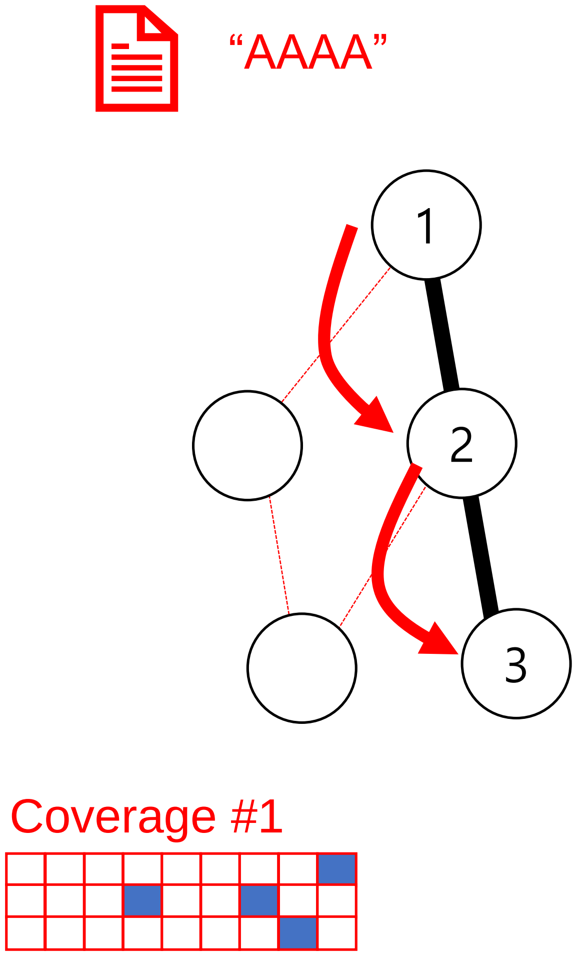
```c
int func(int p6) {
  int local1[10];

  // affect global1 variable
  global1 = 45;
  int local2 = global1;
  for (int i = 0; i < 1000; i++)
    // affect local1 variable
    local1[i] = p6 + local2 + i;

  // affect global2 variable
  return local1[5];
}
```

BranchTrap Hinders Coverage Management

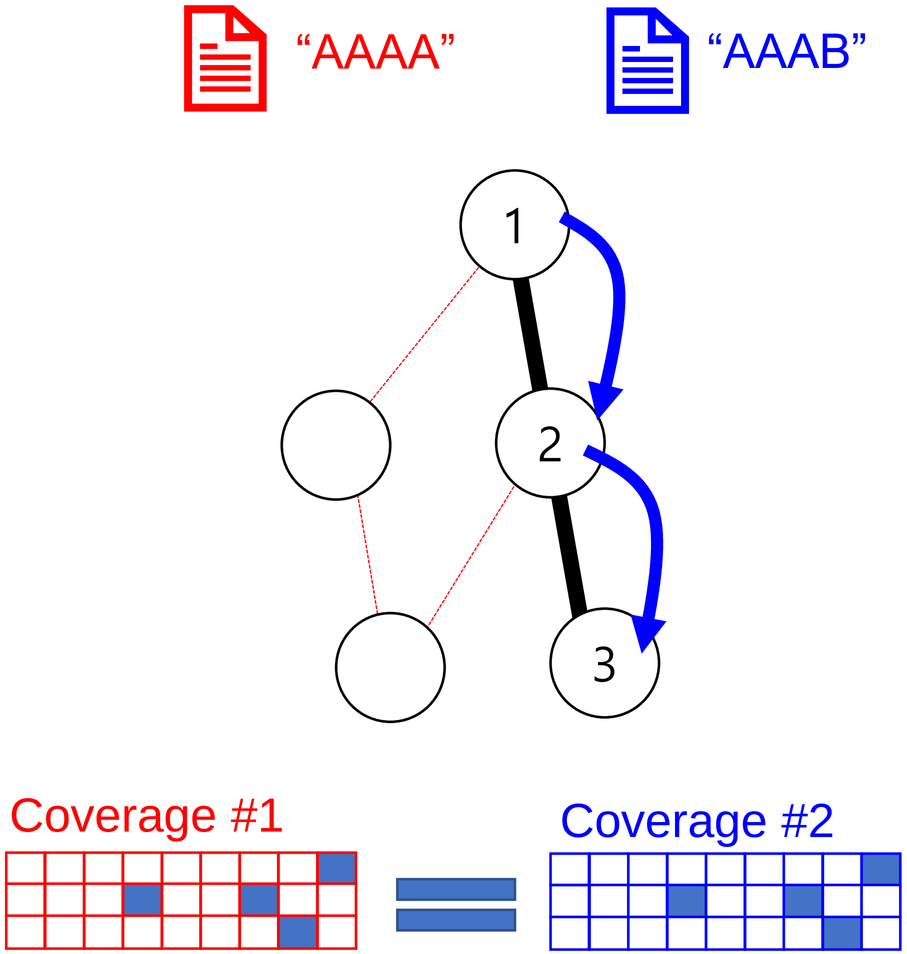
- Fast execution
- ~~Coverage-guidance~~
- Hybrid approach



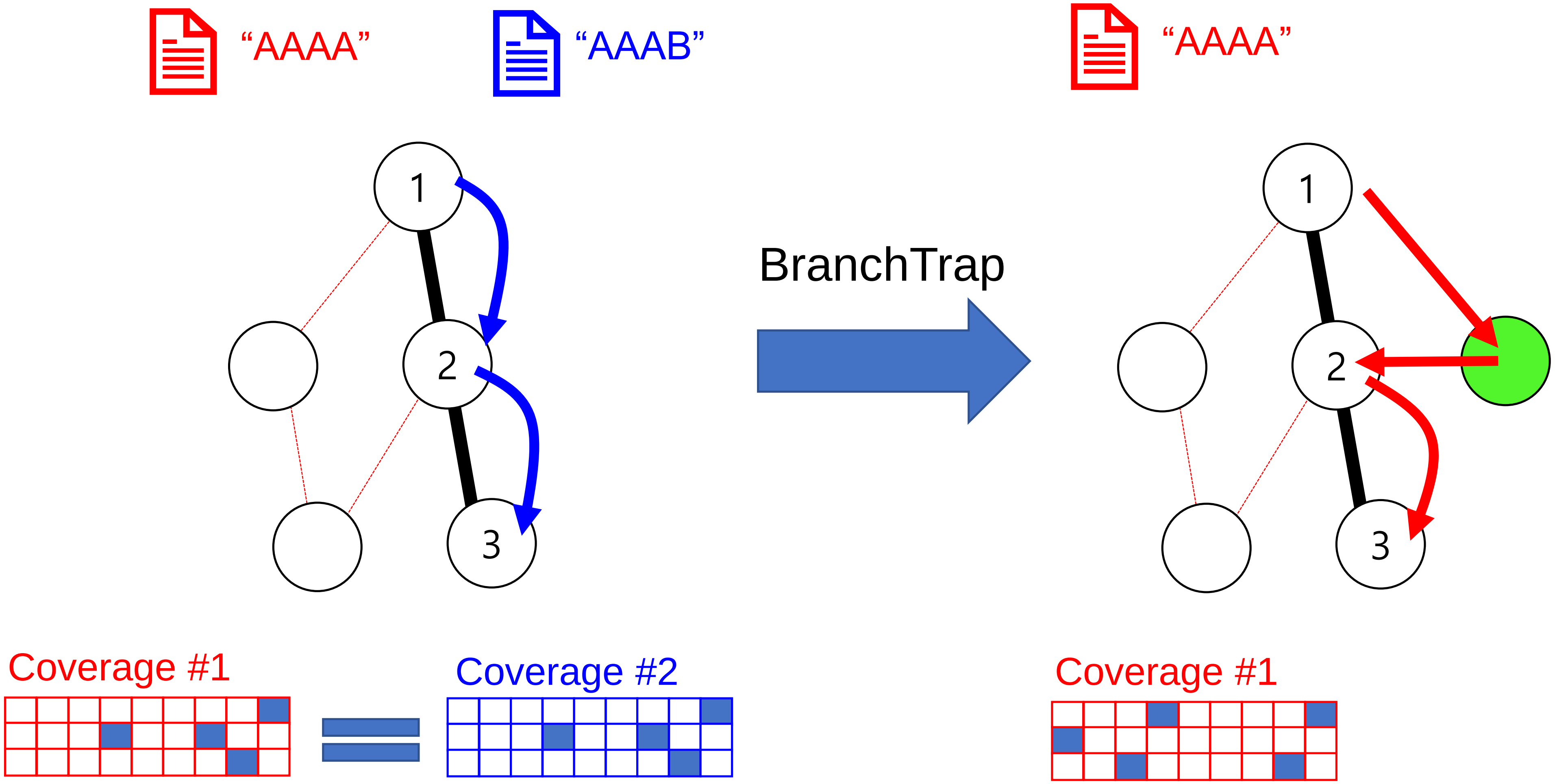
BranchTrap#1: Fabricates Input-sensitive Paths



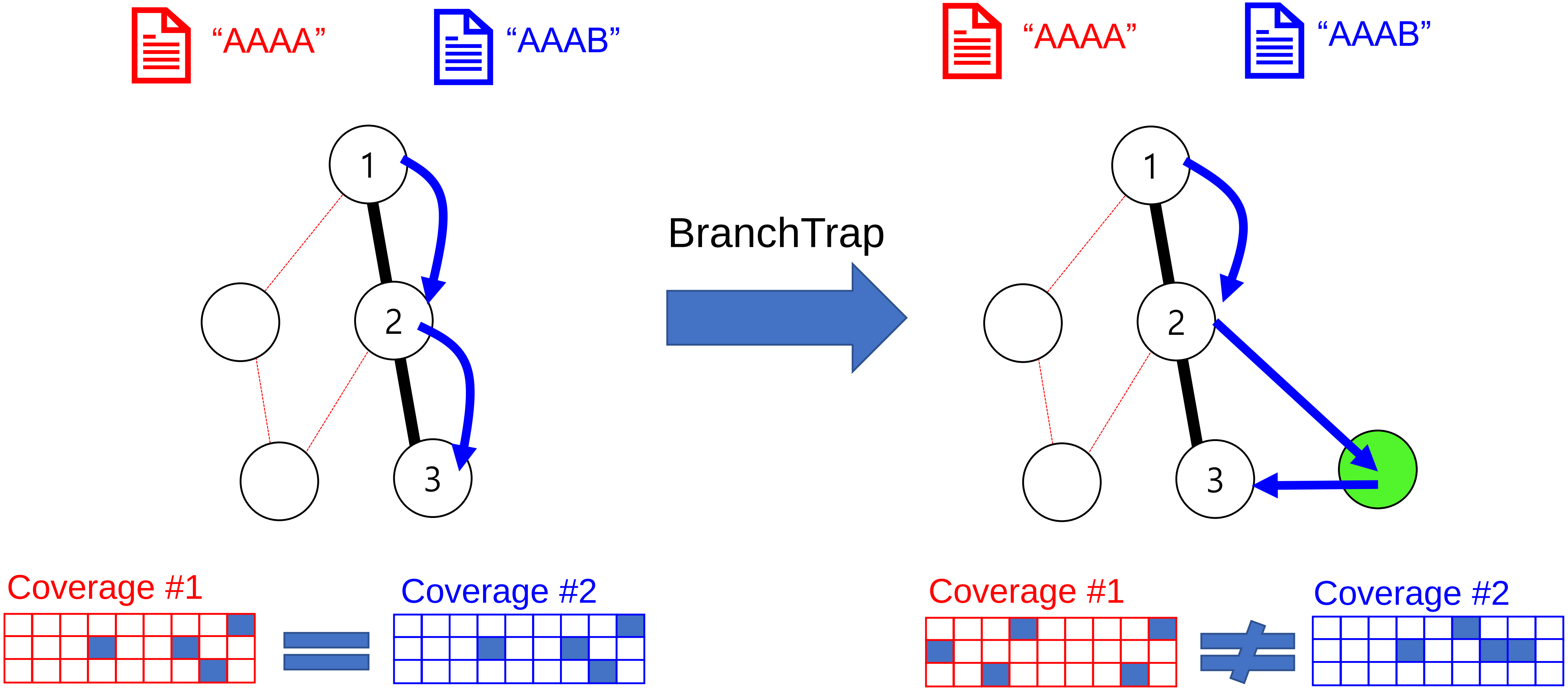
BranchTrap#1: Fabricates Input-sensitive Paths



BranchTrap#1: Fabricates Input-sensitive Paths



BranchTrap#1: Fabricates Input-sensitive Paths



BranchTrap#1: ROP-based Fake Paths Generation

Func1 **(arg1**, **arg2)**

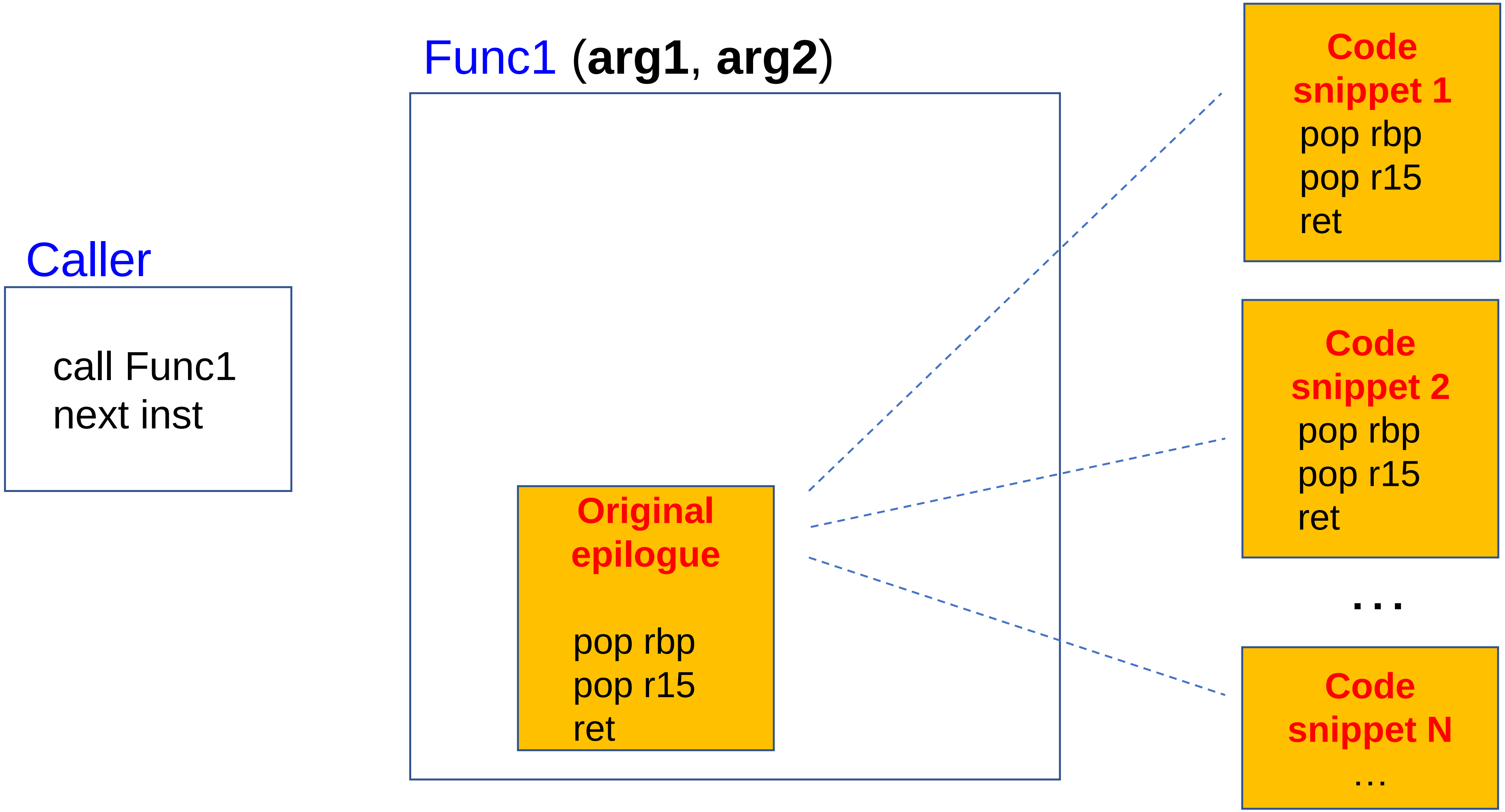
Caller

call Func1
next inst

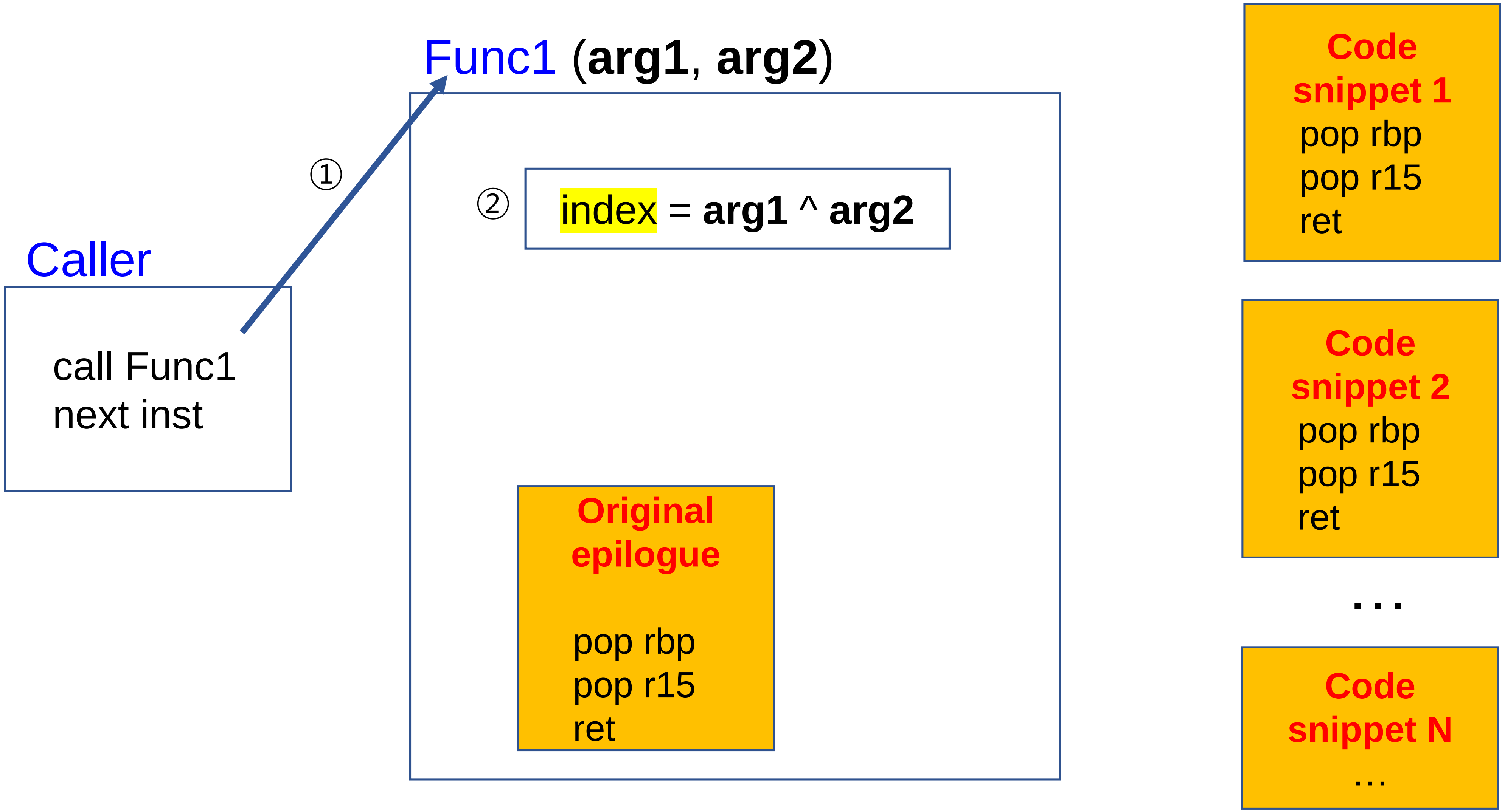
Original epilogue

pop rbp
pop r15
ret

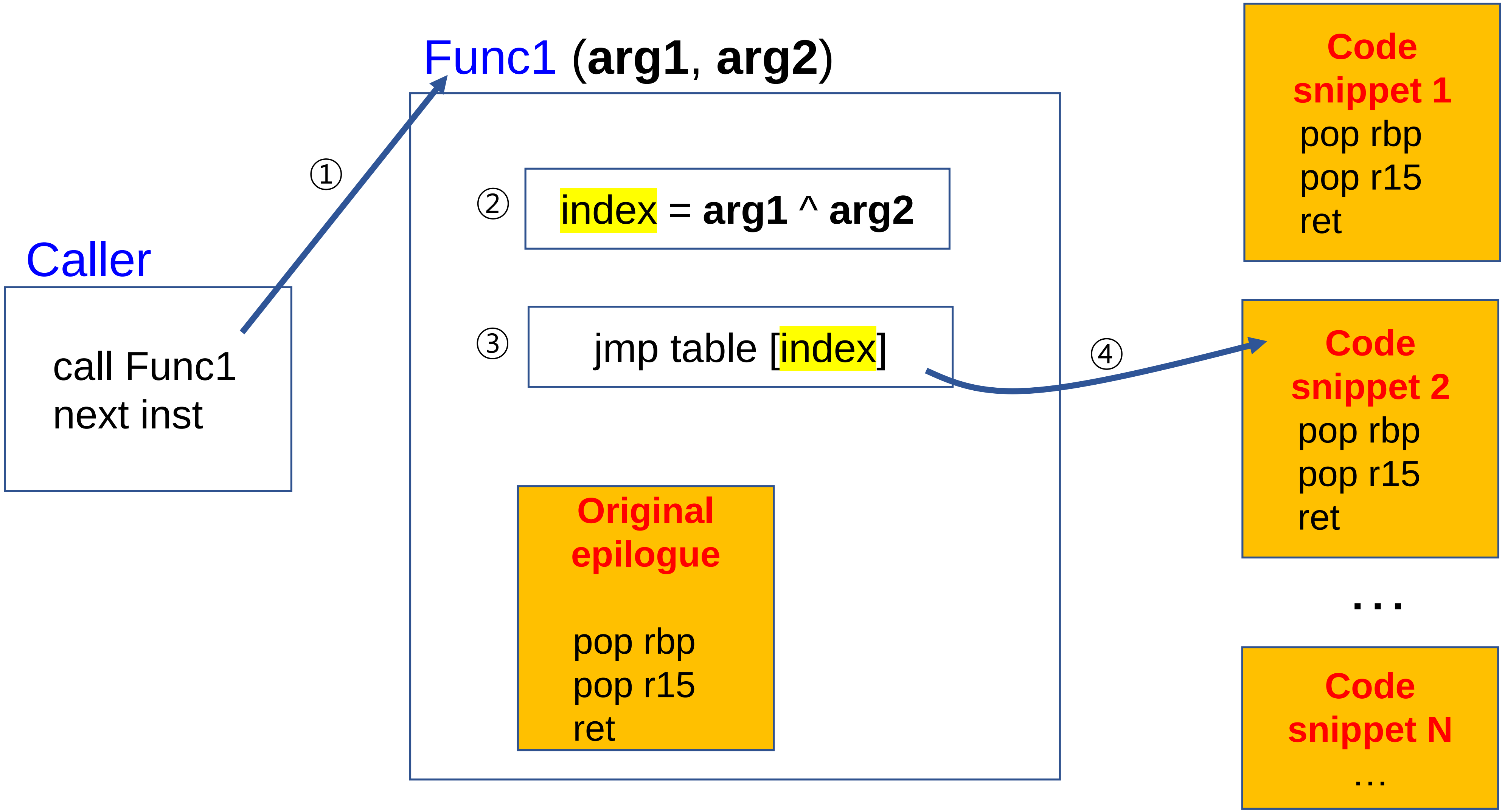
BranchTrap#1: ROP-based Fake Paths Generation



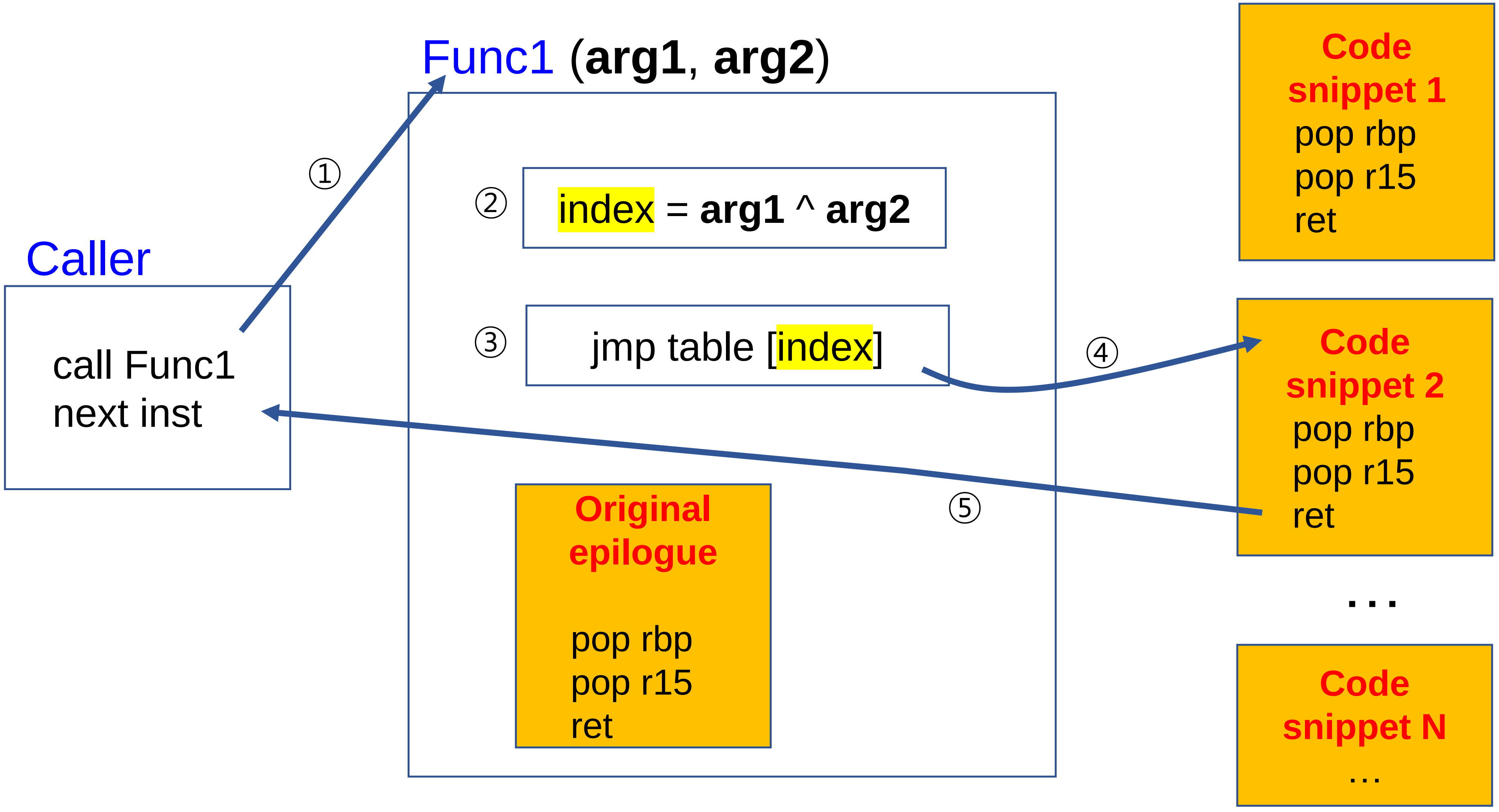
BranchTrap#1: ROP-based Fake Paths Generation



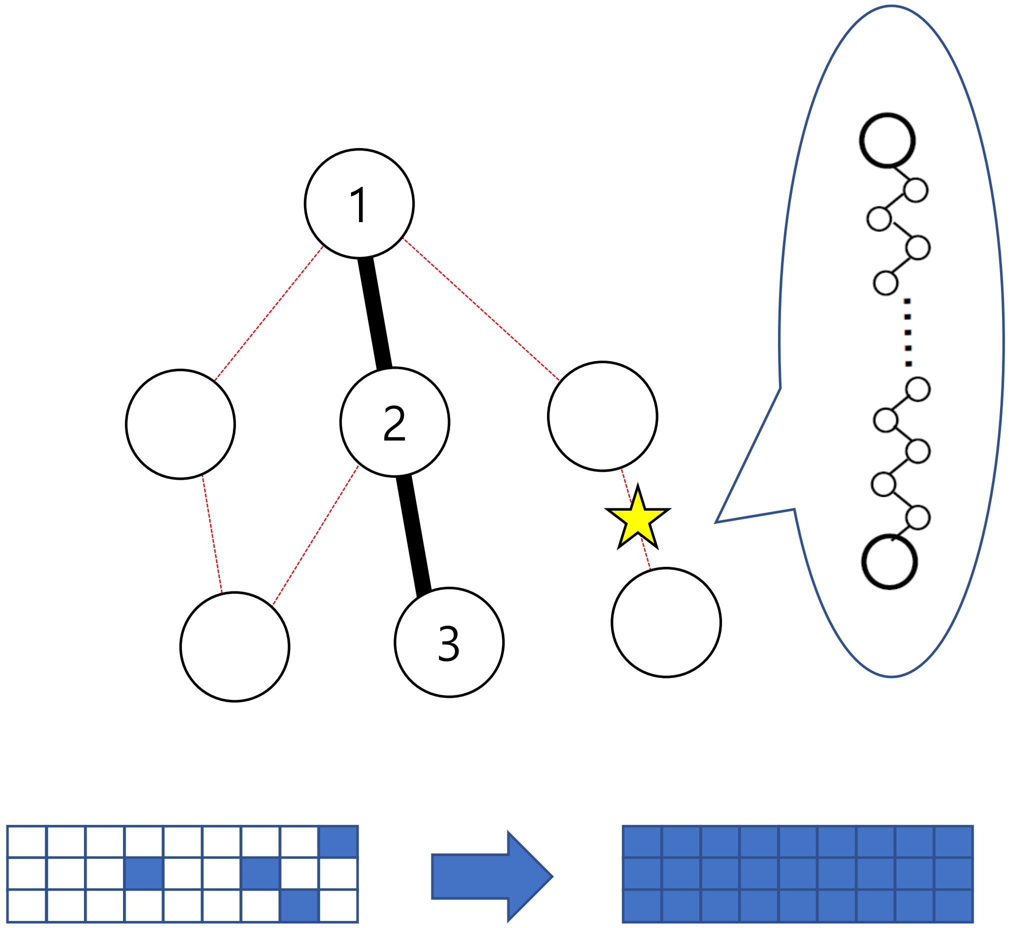
BranchTrap#1: ROP-based Fake Paths Generation



BranchTrap#1: ROP-based Fake Paths Generation



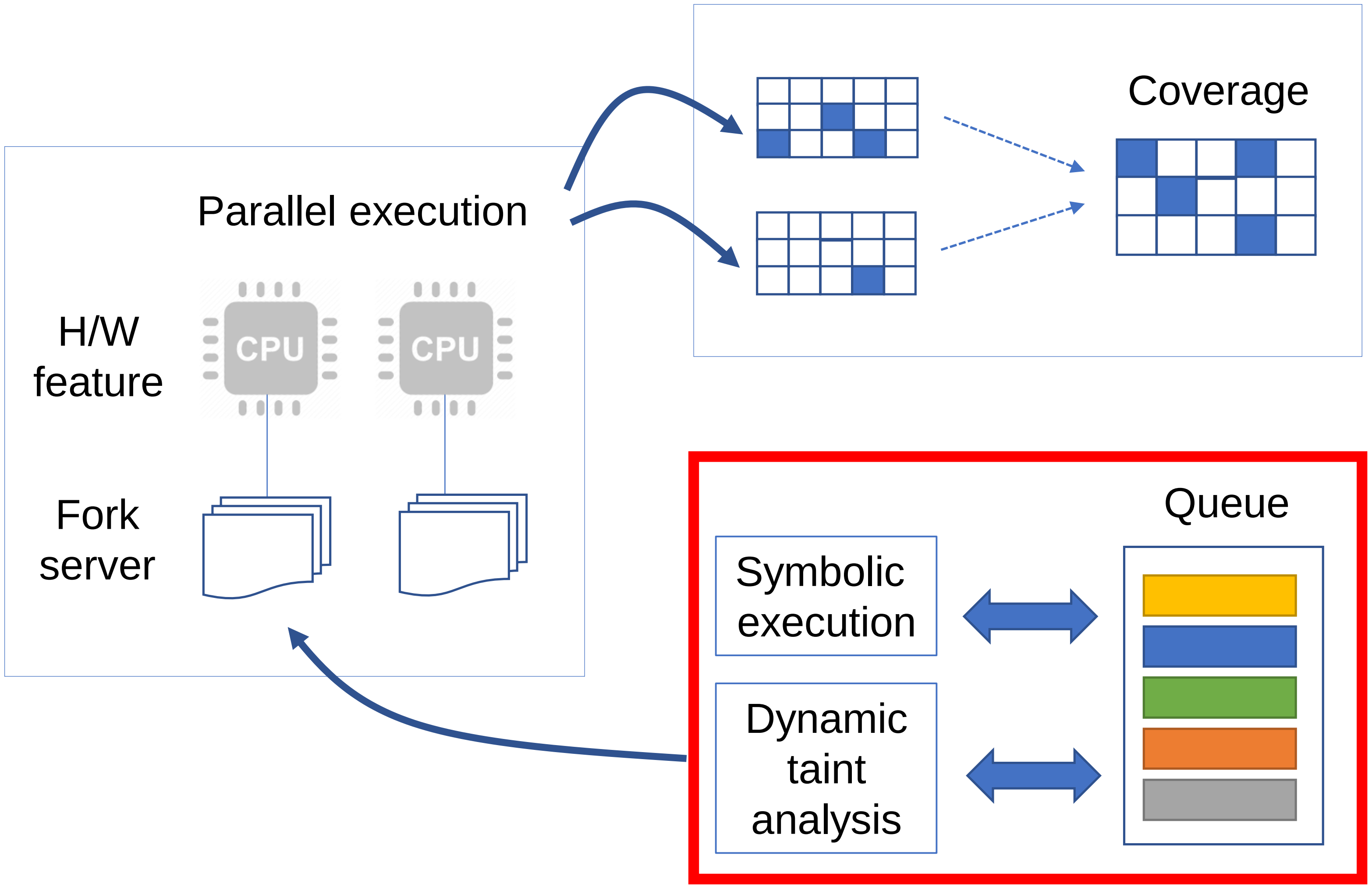
BranchTrap#2: Saturate Feedback State



- One-time visit makes effect
- BranchTrap:
  - Saturates bitmap data
  - Prevents coverage recording

AntiHybrid Hinders Hybrid Fuzzing

- Fast execution
- Coverage-guidance
- ~~Hybrid approach~~



Challenge of Hybrid Fuzzing

- Dynamic taint analysis
  - Expensive implicit flow

Transform *explicit* data-flow → *implicit* data-flow

Challenge of Hybrid Fuzzing

- Dynamic taint analysis
  - Expensive implicit flow

Transform *explicit* data-flow → *implicit* data-flow

- Symbolic execution
  - Path explosion

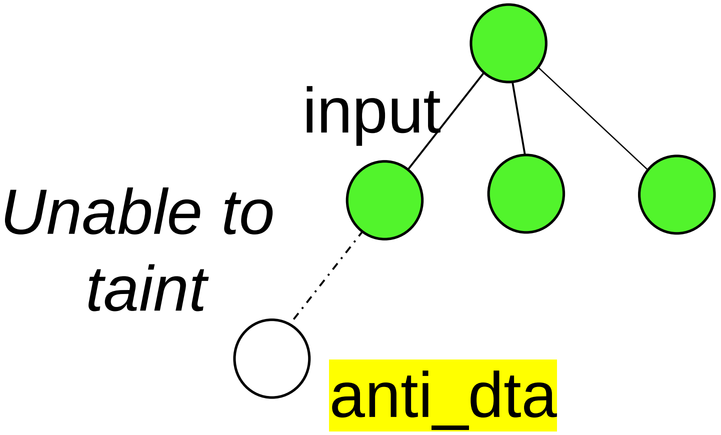
Introduce an arbitrary path explosions

AntiHybrid Avoids Dynamic Taint Analysis

- Transform explicit data-flow to implicit data-flow

```
char input = ‘a’;

if (!strcmp(input, ‘a’))
{ … }
```

```
char input = ‘a’;

char anti_dta;
if (input == 97)
  anti_dta = ‘a’;

if (!strcmp(anti_dta, ‘a’))
{ … }
```

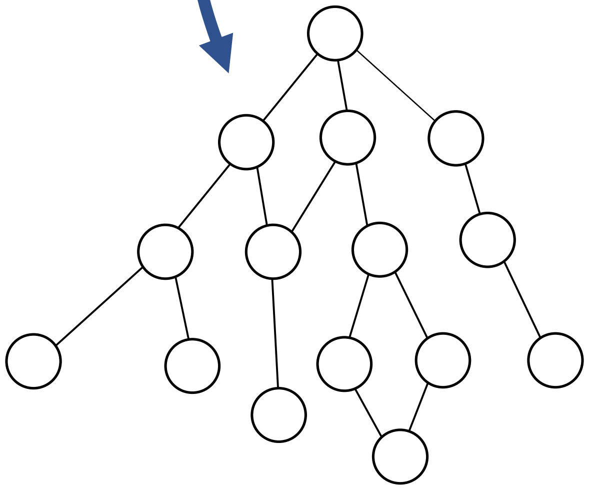


AntiHybrid Incurs Path Explosions

- Inject hash calculations into branches

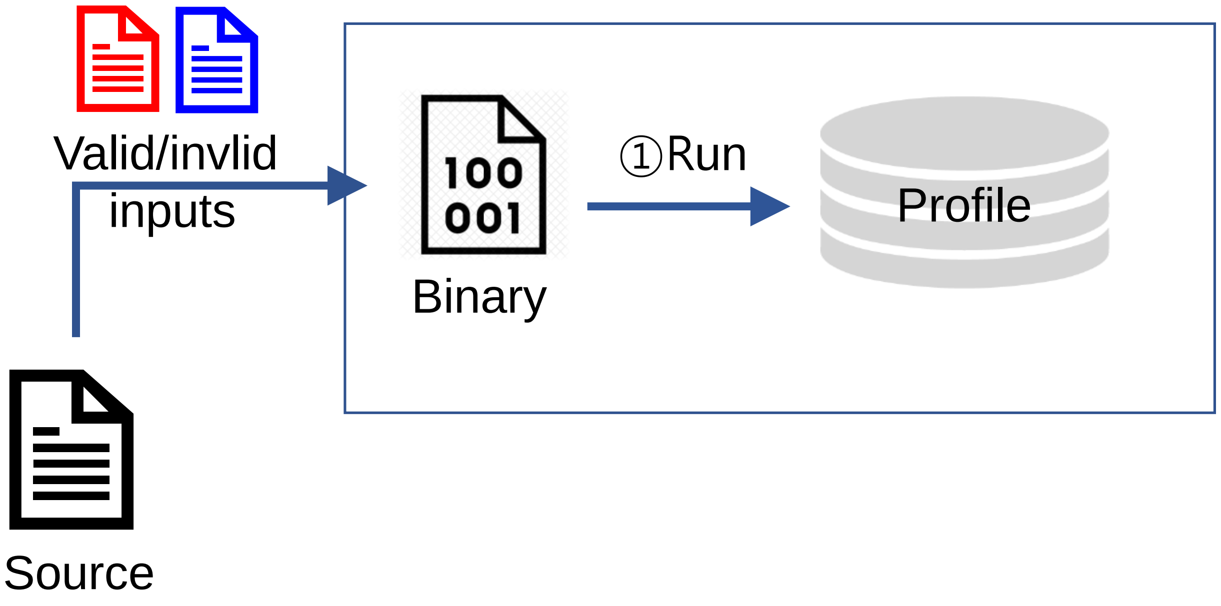
```
if(a == 30)
{ … }
```

```
if(Hash(a) == 0x300df11)
{ … }
```

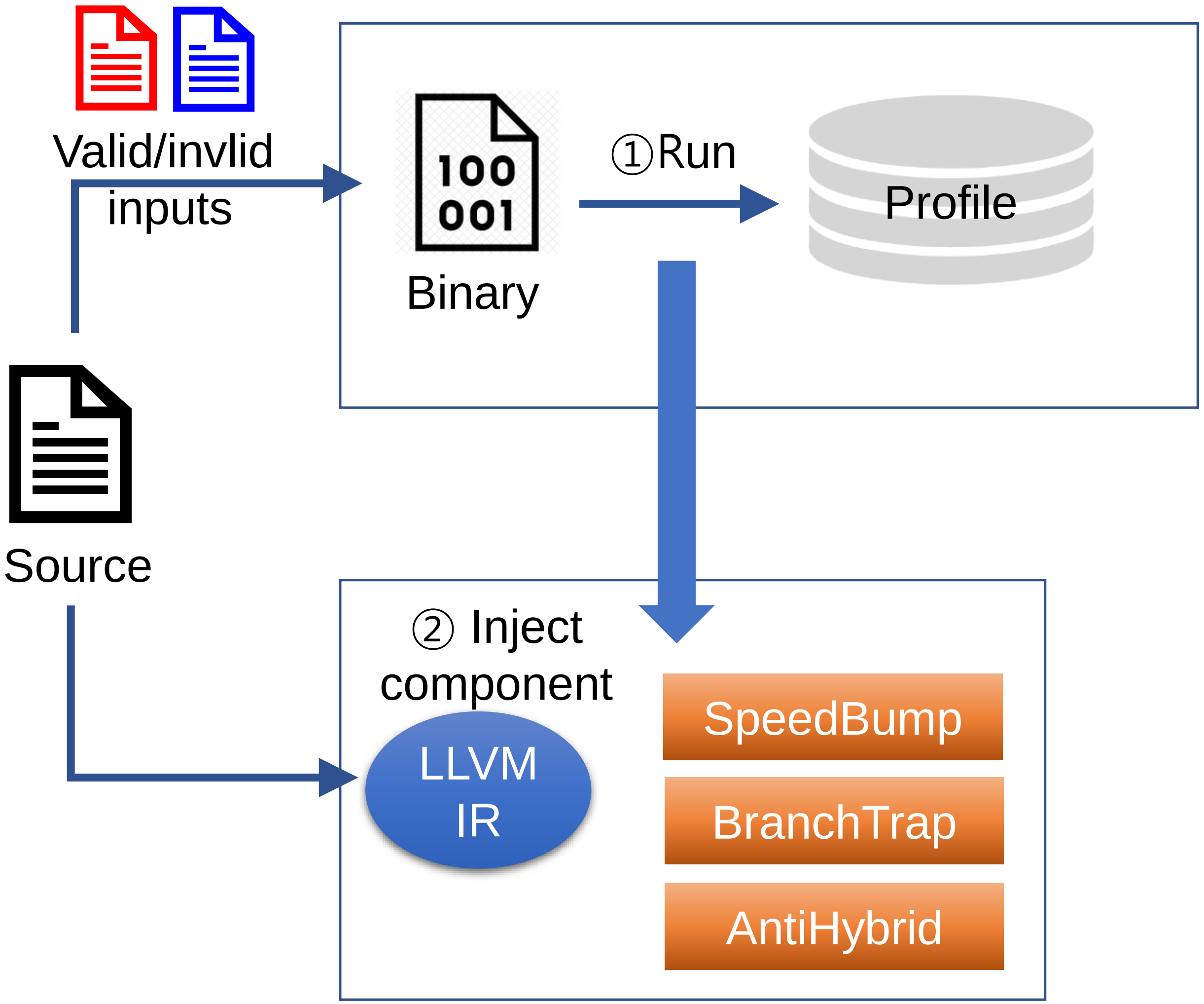


Path Explosion

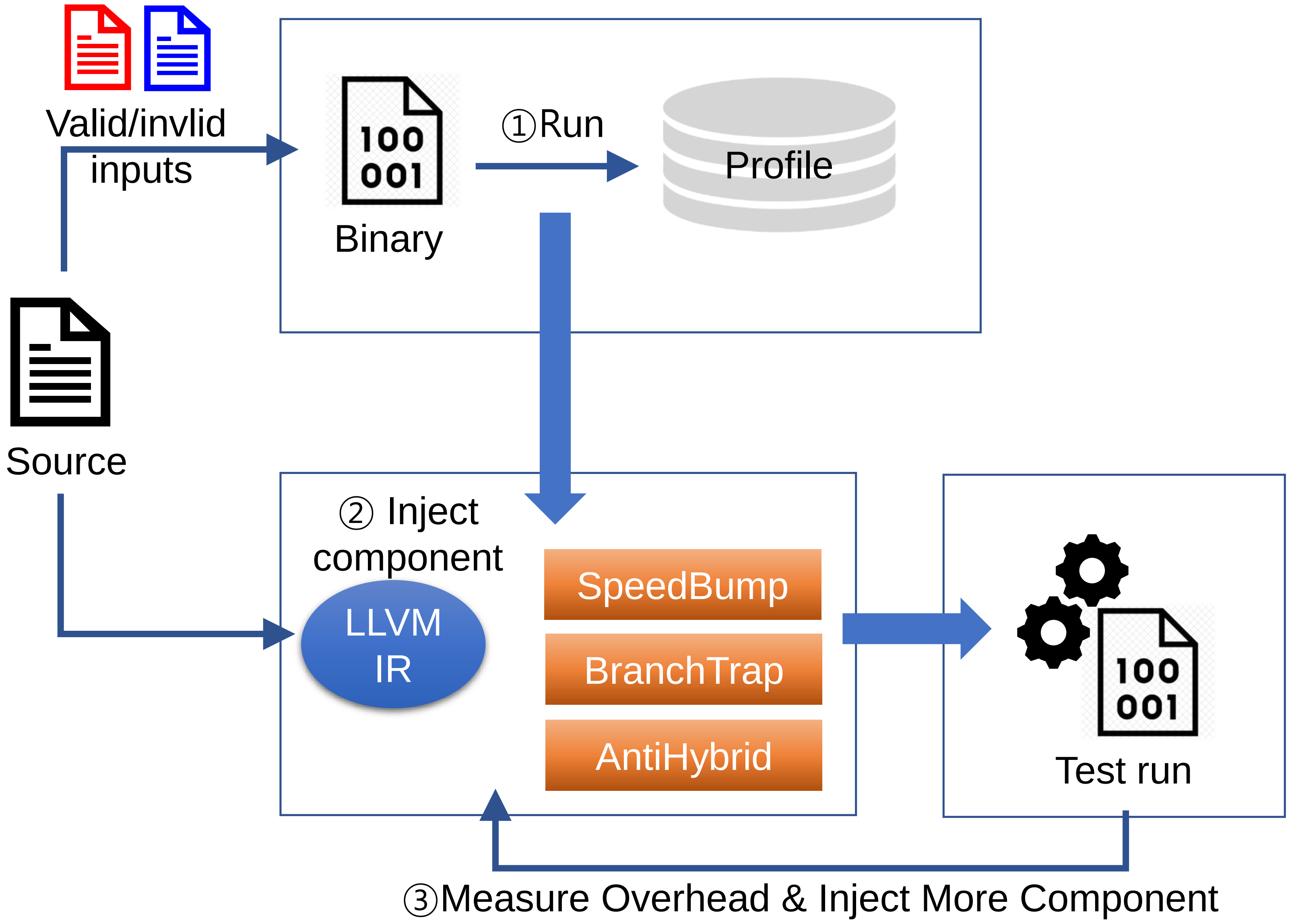
Fuzzification Work-flow



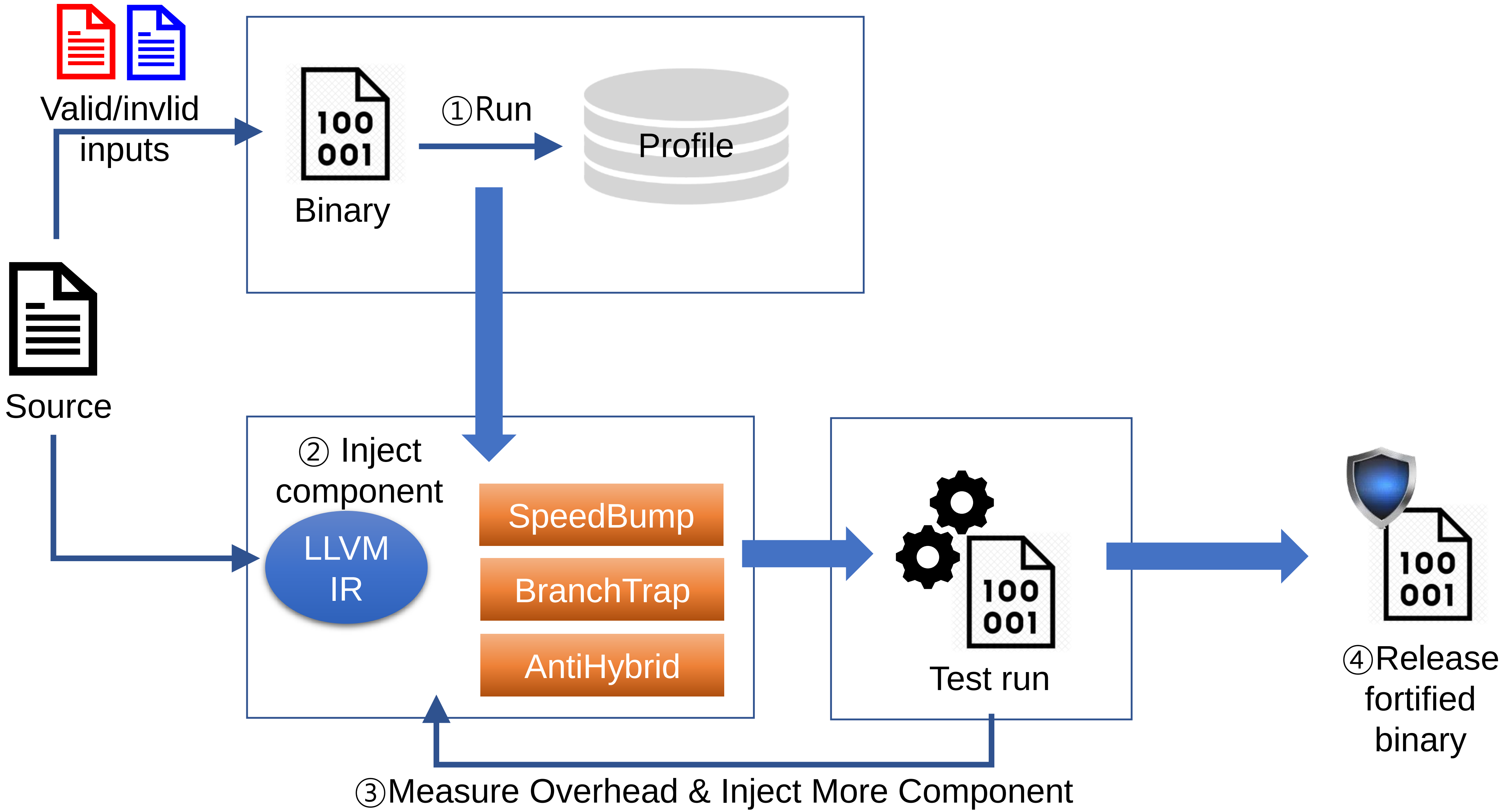
Fuzzification Work-flow



Fuzzification Work-flow



Fuzzification Work-flow

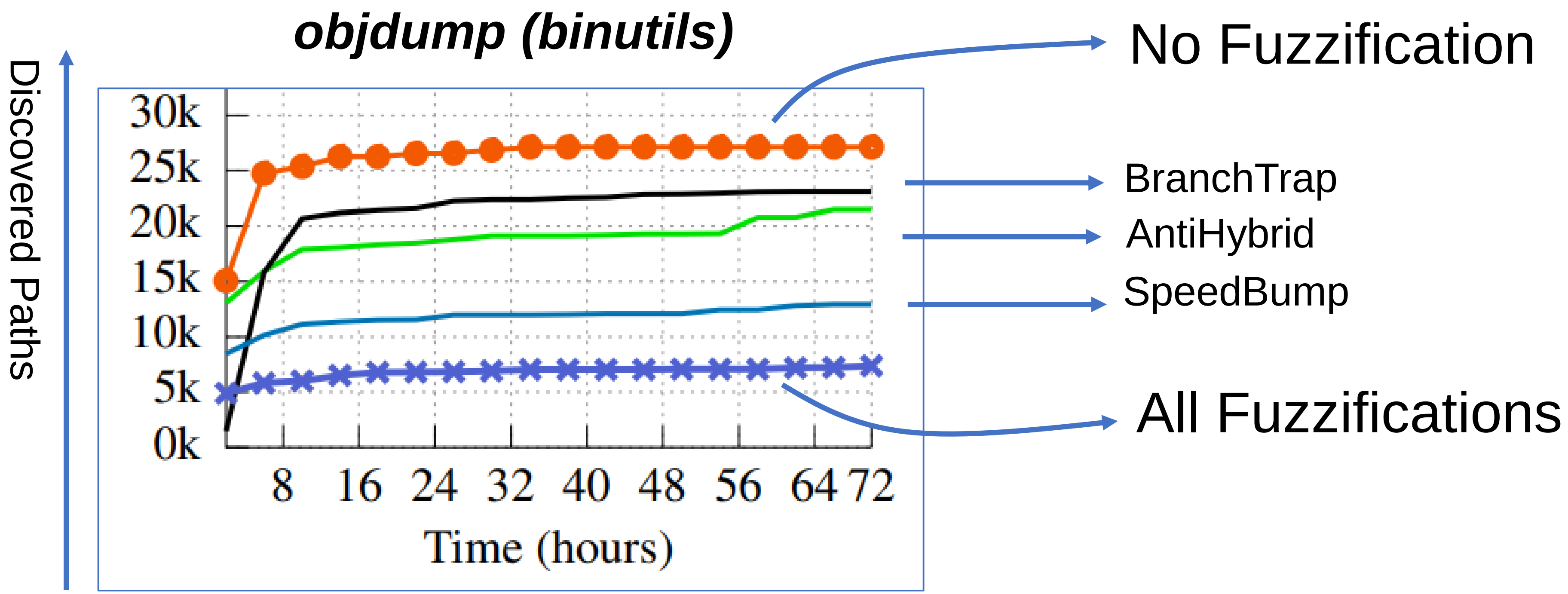


Evaluation Summary

- Implementation
  - 6,599 lines of Python and 758 lines of C++

- Evaluation questions:
  - Effective in "Reducing discovered paths and bugs?"
  - Effective on "Various fuzzers?
  - Impose "Low overhead" to the normal user?

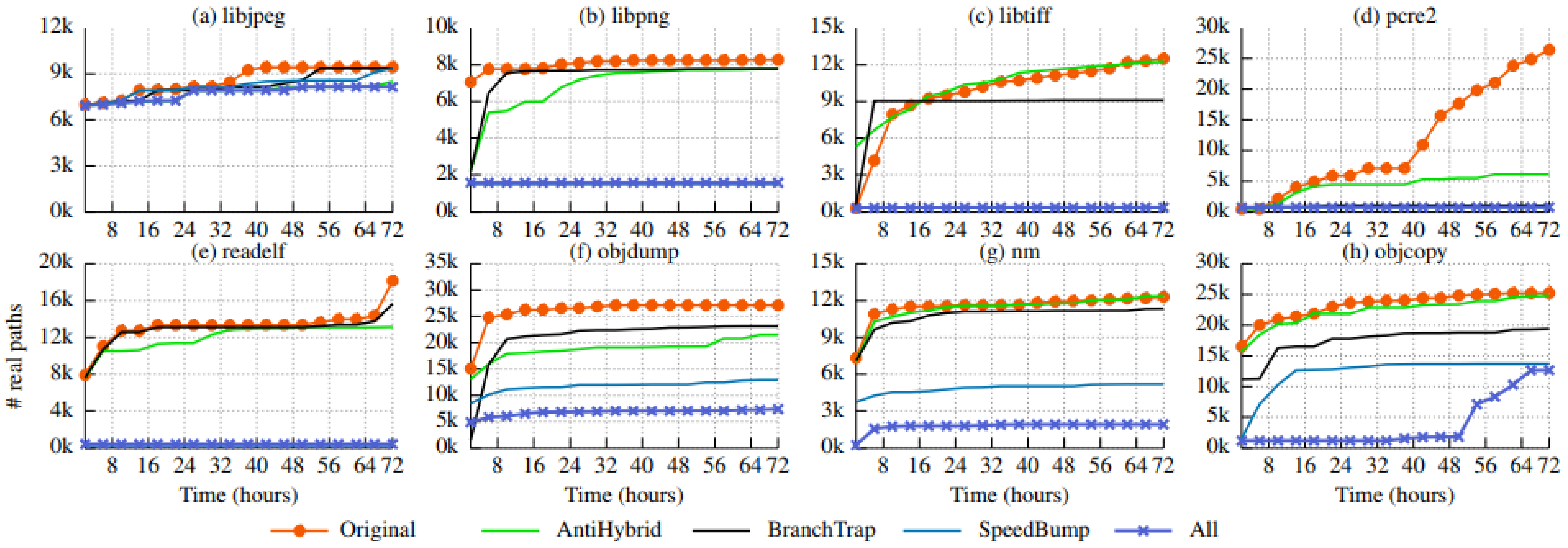
Reduced the Discovered Coverage By 71%



* Fuzzing result on AFL-QEMU

Reduced the Discovered Coverage By 71%

Other binaries



* Fuzzing result on AFL-QEMU

Fuzzification is Effective on Various Fuzzers

Fuzzer	Result
AFL (QEMU)	**74%**
HonggFuzz (PT)	**61%**
QSym (AFL-QEMU)	**80%**
Average	**71%**

Reduced code coverage

Reduced the Discovered Bugs

binutils v2.3.0

Fuzzer	Result
AFL (QEMU)	**88%**
HonggFuzz (PT)	**98%**
QSym (AFL-QEMU)	**94%**
Average	**93%**

LAVA-M dataset

Fuzzer	Result
Vuzzer	**56%**
QSym (AFL-QEMU)	**78%**
Average	**67%**

File size & CPU Overheads

binutils v2.3.0

Overhead	Result
File Size	1.4MB (62.1%)
CPU Overhead	3.7%

Real-world applications (e.g., GUI)

Overhead	Result
File Size	1.3MB (5.4%)
CPU Overhead	0.73%

* Both overheads are configurable

Discussion

- Best-effort protections against adversarial analysis

- Complementary to other defense techniques
  - Not hiding all vulnerabilities
  - But introducing significant cost on attacker

Comparison: Fuzzification vs. AntiFuzz

Component	Fuzzification	AntiFuzz
Delay execution	● (+ cold path)	●
Fake coverage	● (randomized return)	● (fake code)
Saturate coverage	●	○
Prevent crash	○	●
Anti-hybrid	● (+ anti-DTA)	●
Countermeasures	◐	○

Conclusion

Make the fuzzing only effective to the testers

- **SpeedBump**: Inject delays and only affects attackers
- **BranchTrap**: Insert input-sensitive branches
- **AntiHybrid**: Hinder hybrid fuzzing techniques

https://github.com/sslab-gatech/fuzzification