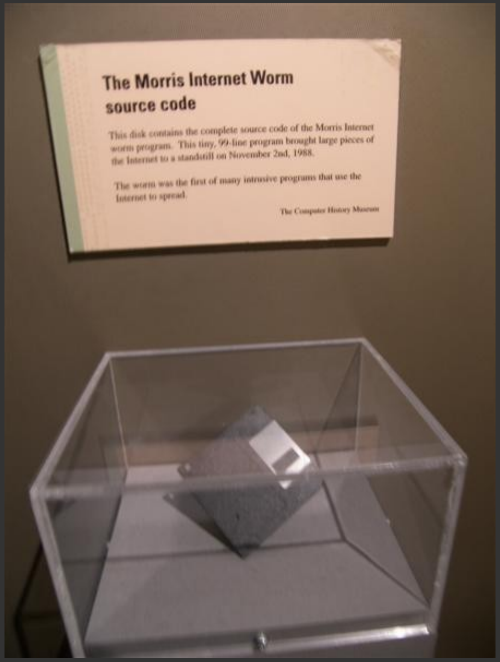
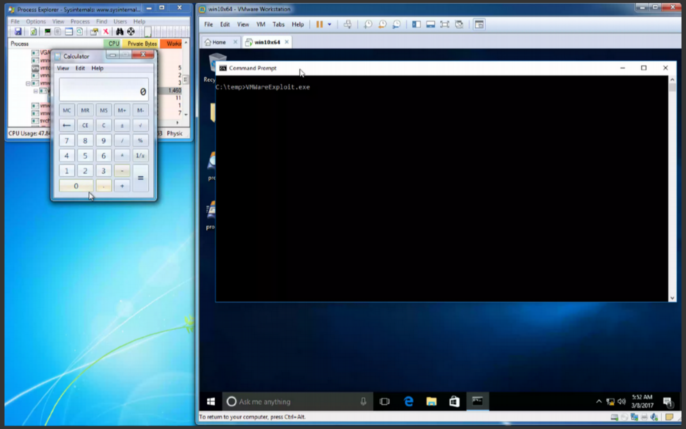
Security Testing
Hard to Reach Code

Mathias Payer <mathias.payer@epfl.ch>

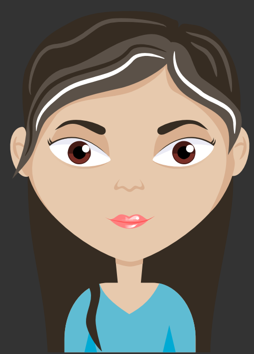
https://hexhive.github.io

Vulnerabilities everywhere?



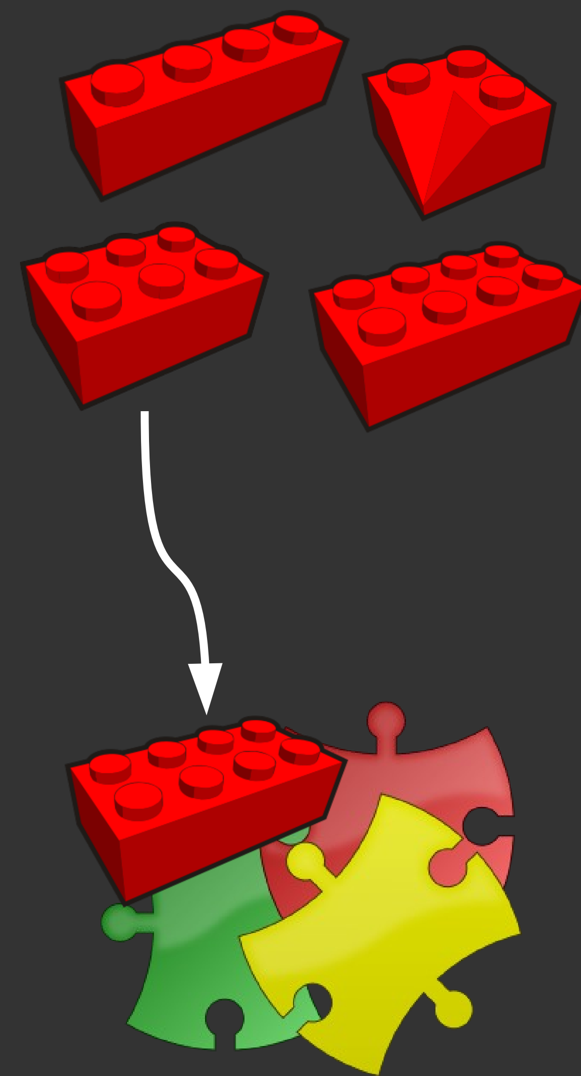


Challenge: broken abstractions



```
                 C/C++
void log(int a) {
  printf("A: %d", a);
}

void vuln(char *str) {
  char *buf[4];
  void (*fun)(int) = &log;
  strcpy(buf, str);
  ...
  fun(15);
}
```

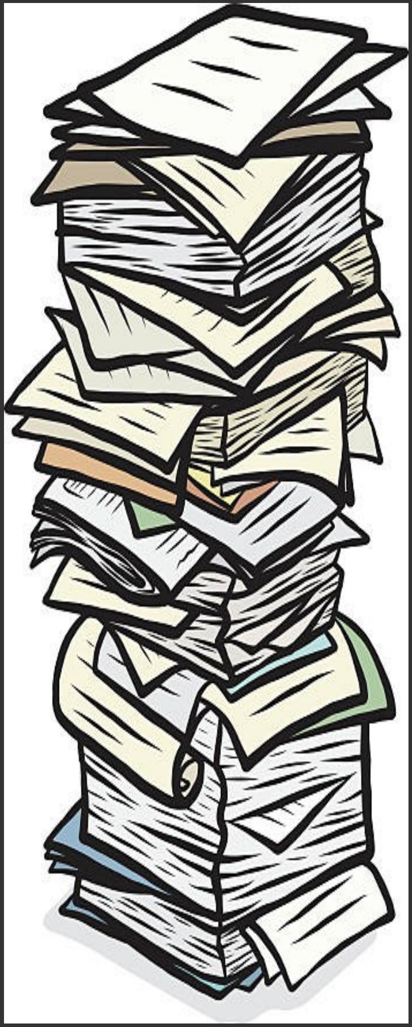
```
                 ASM
log: ...
fun: .quad log
vuln:
  movq log(%rip), 16(%rsp)
  ...
  call strcpy
  ...
  call  16(%rsp)
```



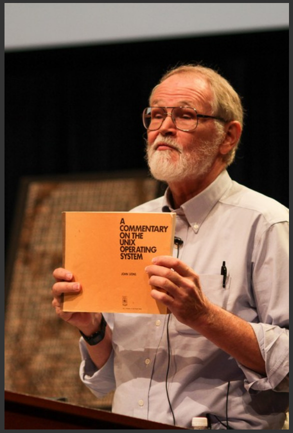
Challenge: software complexity

Google Chrome:	**76 MLoC**
Gnome:	**9 MLoC**
Xorg:	**1 MLoC**
glibc:	**2 MLoC**
Linux kernel:	**17 MLoC**

Chrome and OS
~100 mLoC,
27 lines/page,
0.1mm/page ≈ 370m



Margaret Hamilton with code for Apollo Guidance Computer (NASA, ‘69)



Brian Kernighan holding Lion’s commentary on BSD 6 (Bell Labs, ‘77)

Defense: Testing *OR* Mitigating?

Software Testing

Mitigations

```
vuln("AAA");
```

```
vuln("ABC");
```

```
vuln("AAAABBBB");
```

```
strcpy_chk(buf, 4, str);
```

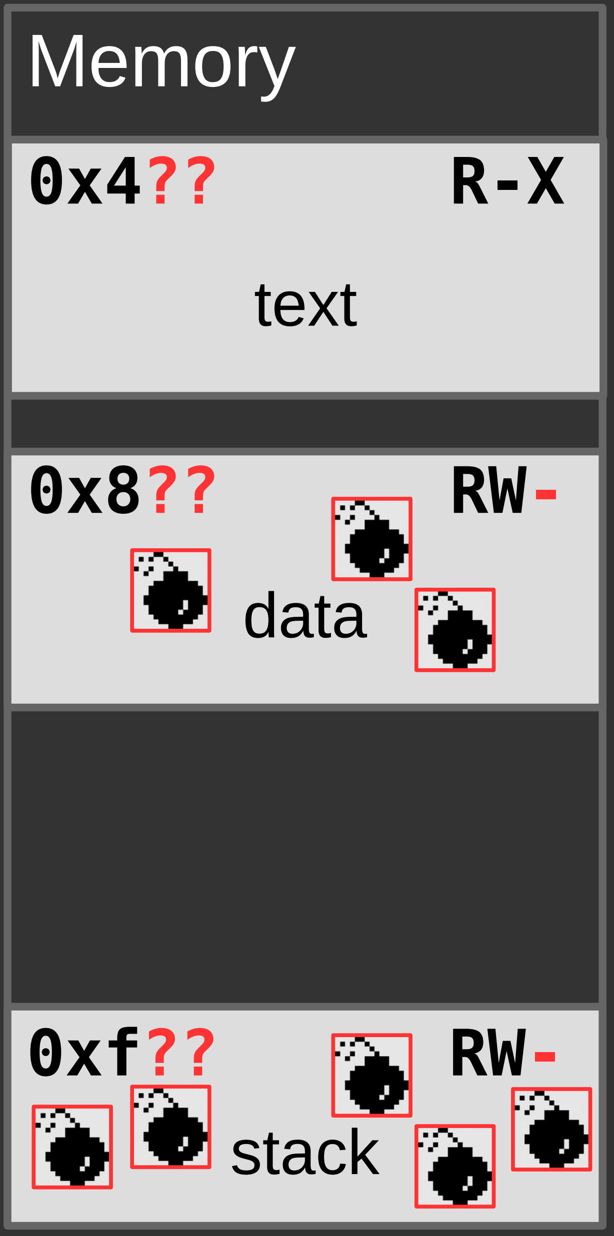
C/C++

```
void log(int a) {
  printf("A: %d", a);
}

void vuln(char *str) {
  char *buf[4];
  void (*fun)(int) = &log;
  strcpy(buf, str);

  fun(15);
}
```

```
CHECK(fun, tgtSet);
```

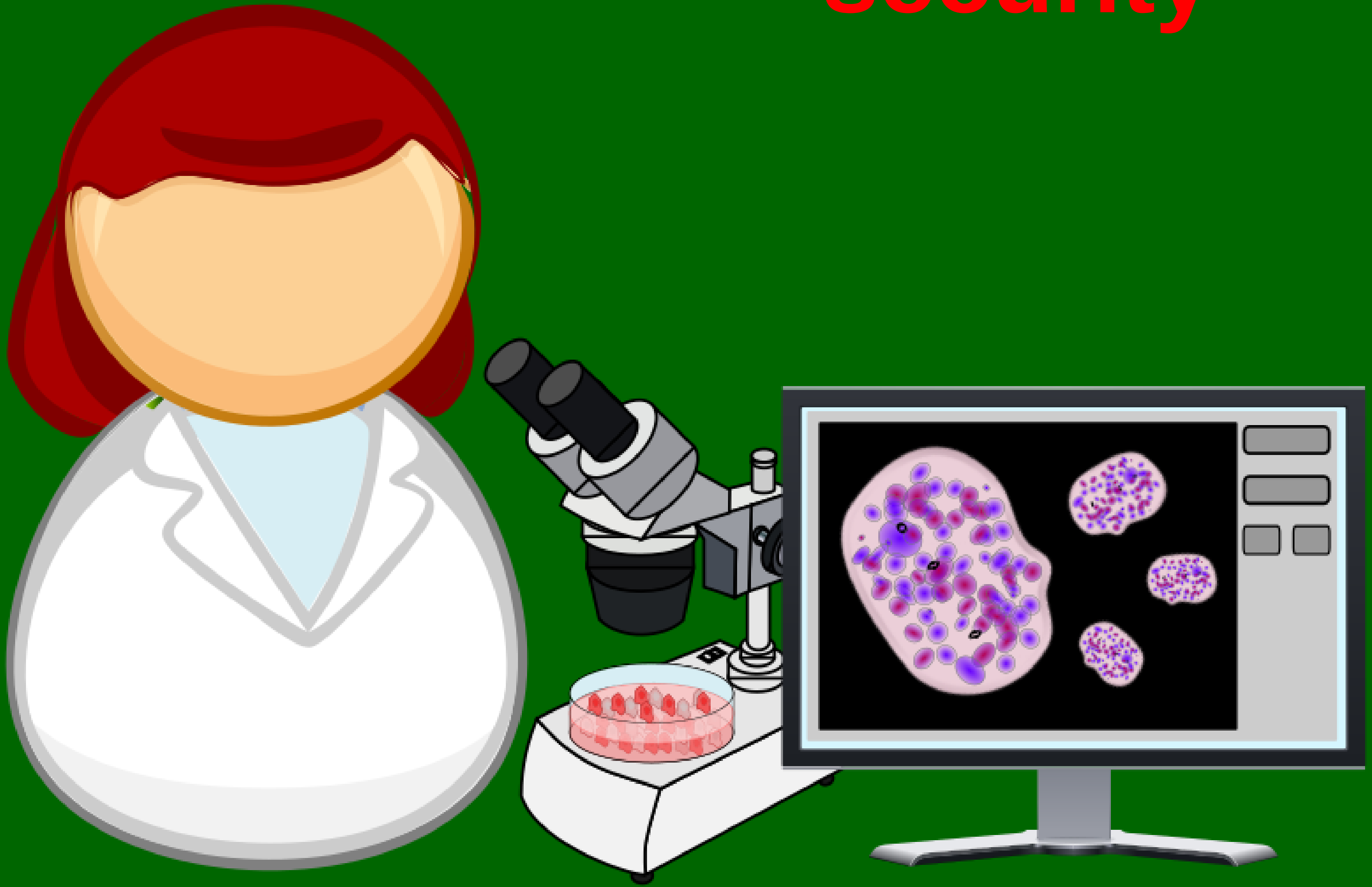
Status of deployed defenses

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack canaries
- Safe exception handlers
- Control-Flow Integrity (CFI): Guard indirect control-flow



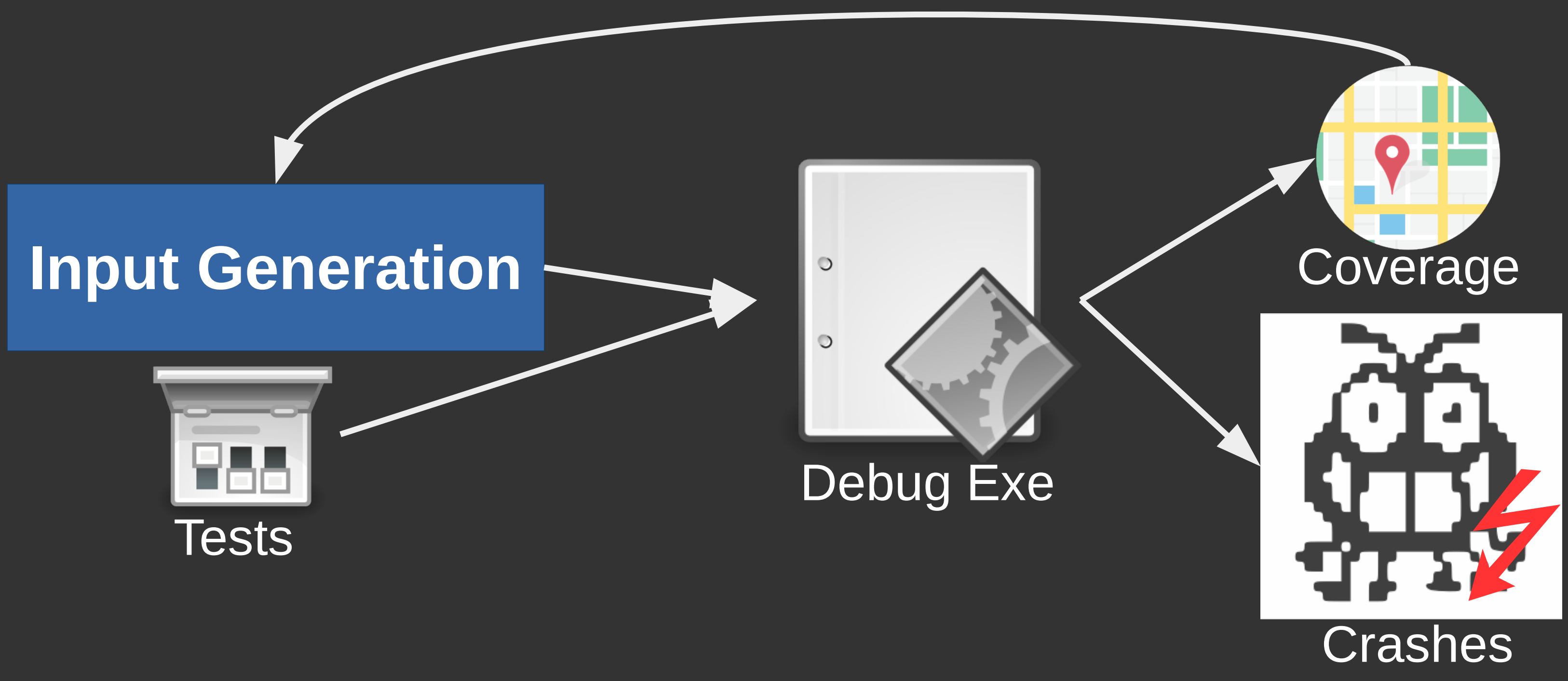
Software testing: discover bugs

security



Fuzz testing

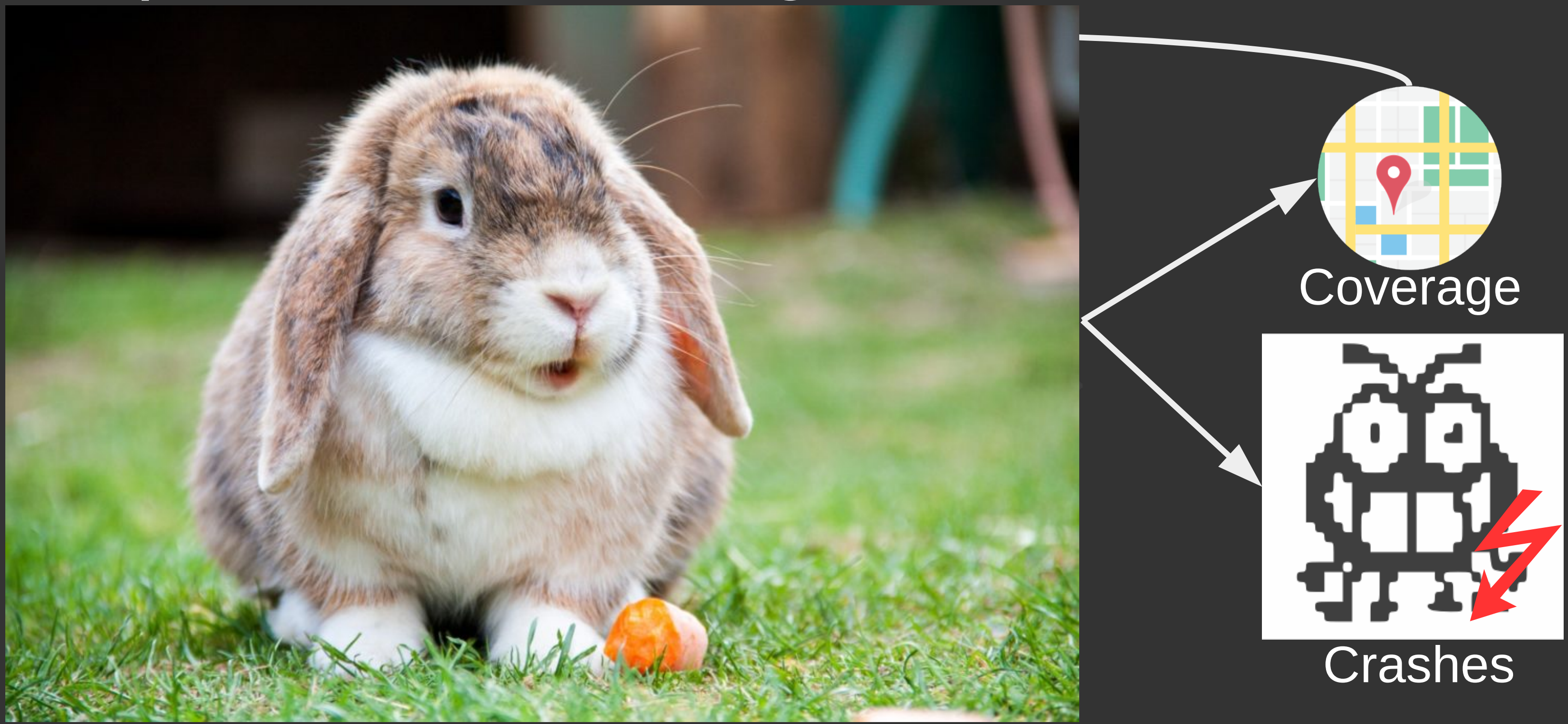
- A random testing technique that mutates input to improve test coverage



- State-of-art fuzzers use coverage as feedback to evolutionarily mutate the input

Fuzz testing

- A random testing technique that mutates input to improve test coverage



- State-of-art fuzzers use coverage as feedback to evolutionarily mutate the input

Fuzzing as bug finding approach

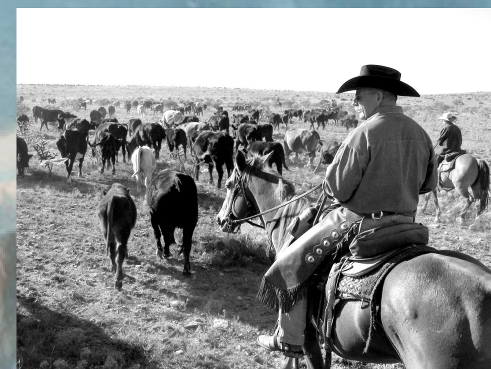
- Fuzzing finds bugs effectively (CVEs)
  - Proactive defense, part of testing
  - Preparing offense, part of exploit development

Academic fuzzing research

Fuzzing frontiers

Fuzzing frontiers

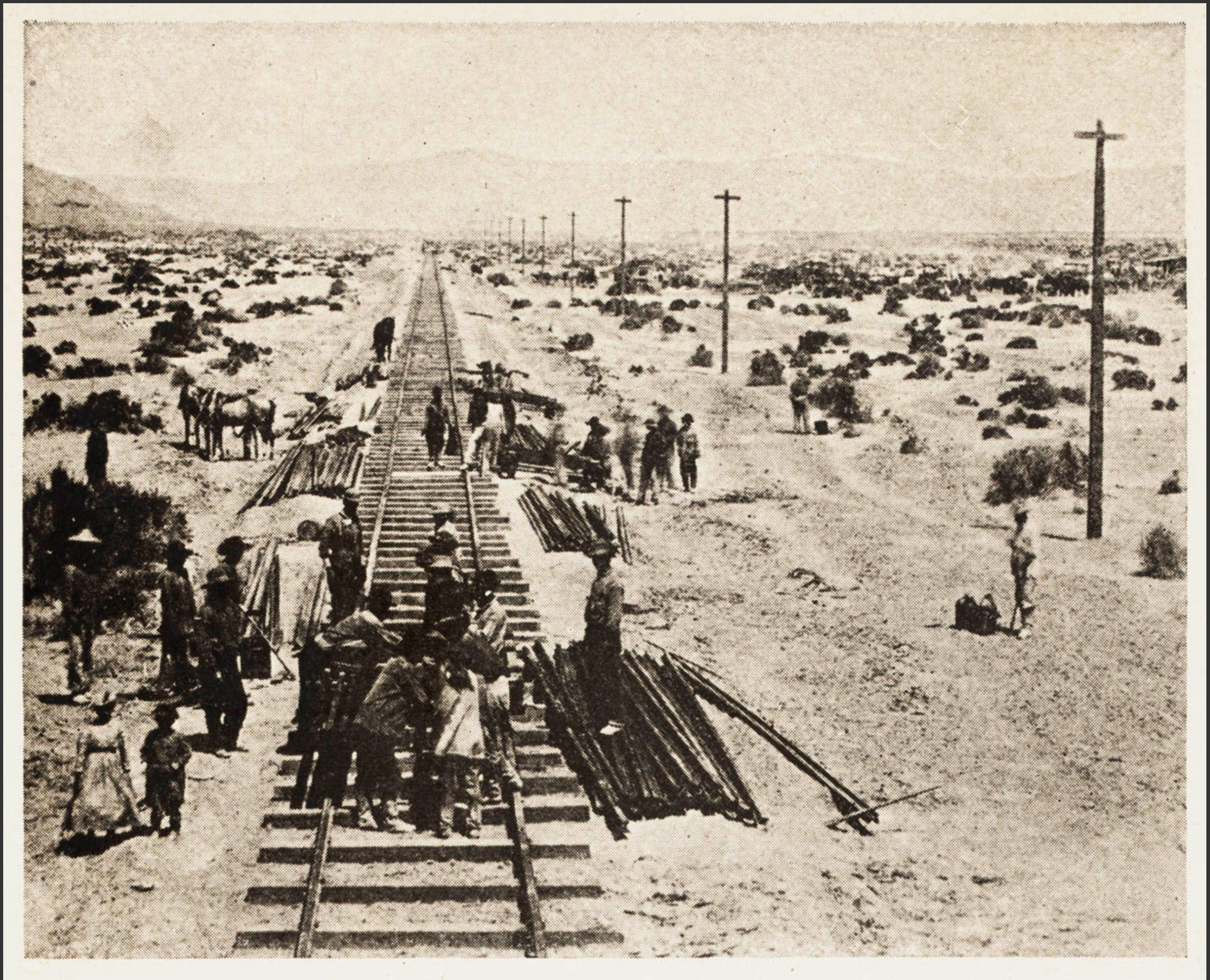
Mine existing code



Cross unknown borders

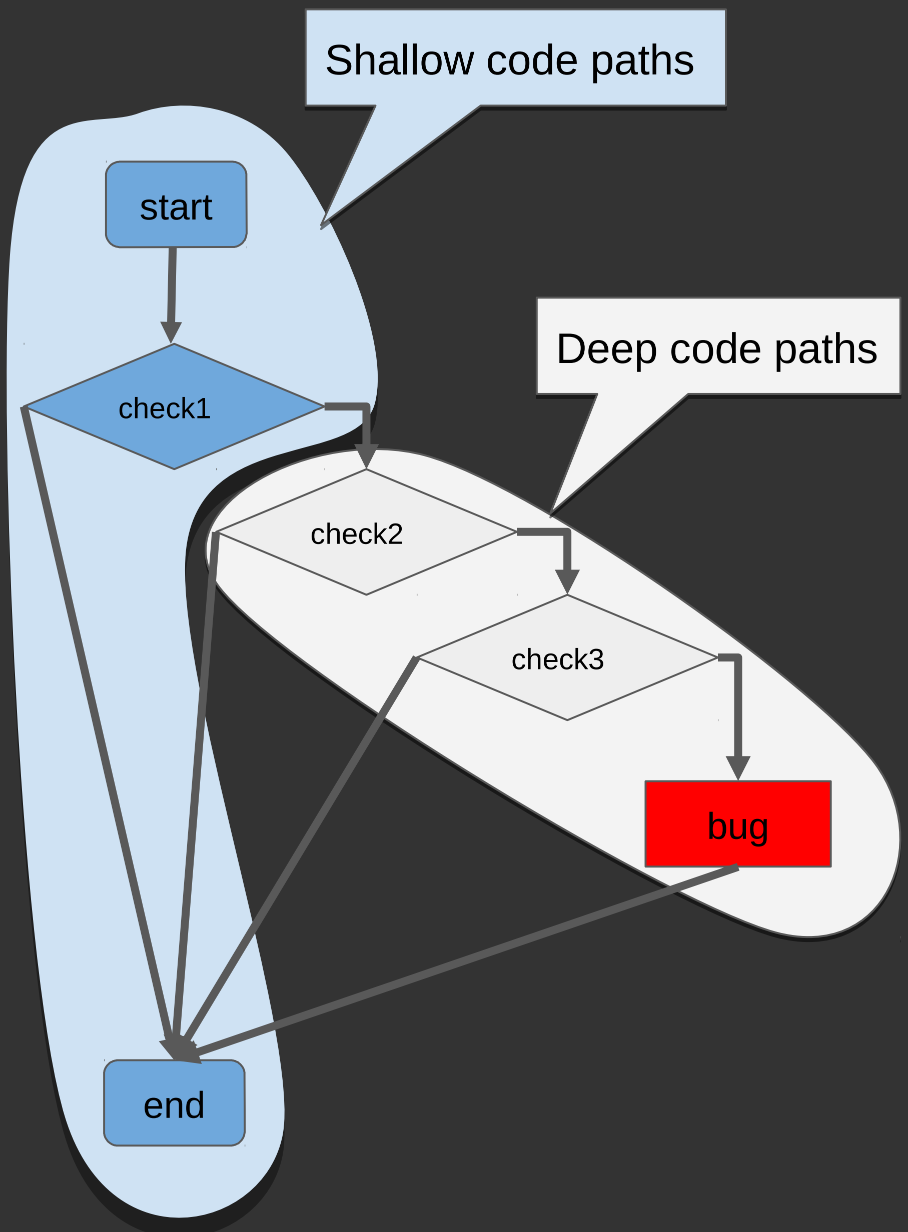
Explore new paths

Exploring hidden program paths



Challenges for Fuzzers

- Challenges
  - Shallow coverage
  - Hard to find “deep” bugs
- Root cause
  - Fuzzer-generated inputs cannot bypass complex sanity checks in the target program



Limitations of existing approaches

- Existing approaches focus on ***input generation***
  - AFL improvements (seed corpus generation)
  - Driller (selective concolic execution)
  - VUzzer (taint analysis, data-/control-flow analysis)
  - QSYM, Angora (symbolic/concolic analysis)
- Limitations: high overhead, not scalable
- Unable to bypass “hard” checks
  - Checksum values
  - Crypto-hash values

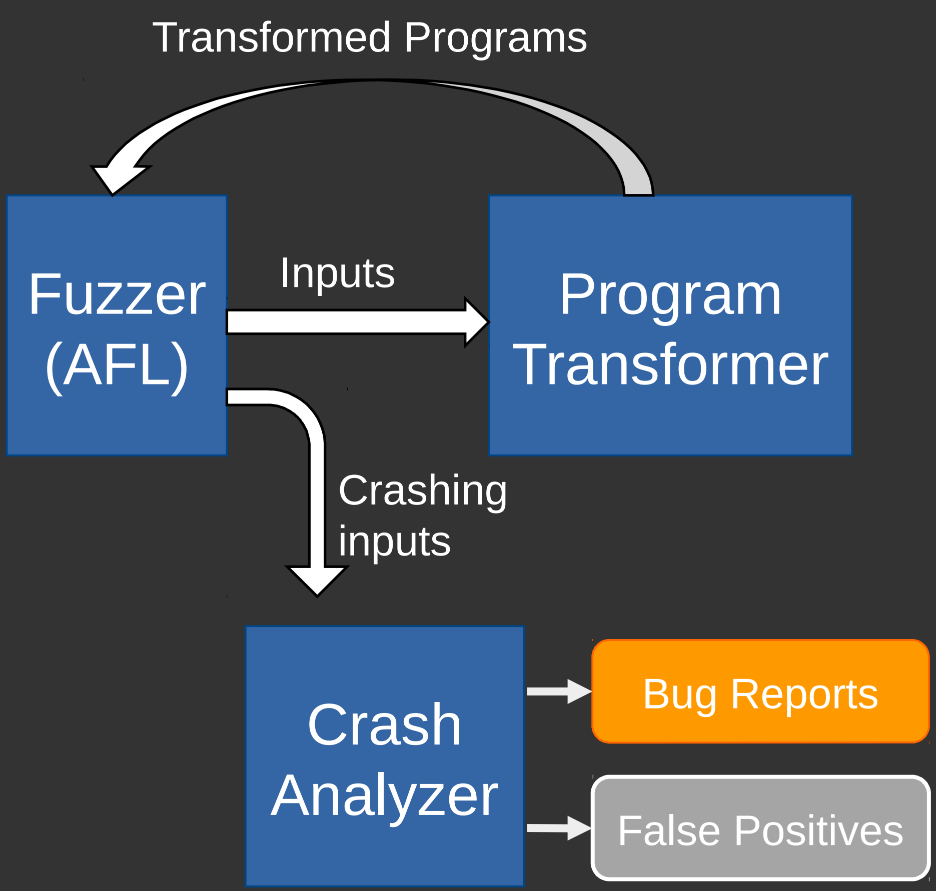
Non-Critical Checks (NCC)

- Some checks are not intended to prevent bugs
  - Checks on magic values, checksum, or hashes
- Removing NCCs
  - Won't incur erroneous bugs, simplifies fuzzing

```
void main() {
  int fd  = open(...);
  char *hdr = read_header(fd);
  if (strncmp(hdr, “ELF", 3) == 0) {
    // main program logic
    // ...
  } else {
    error();
  }
}
```

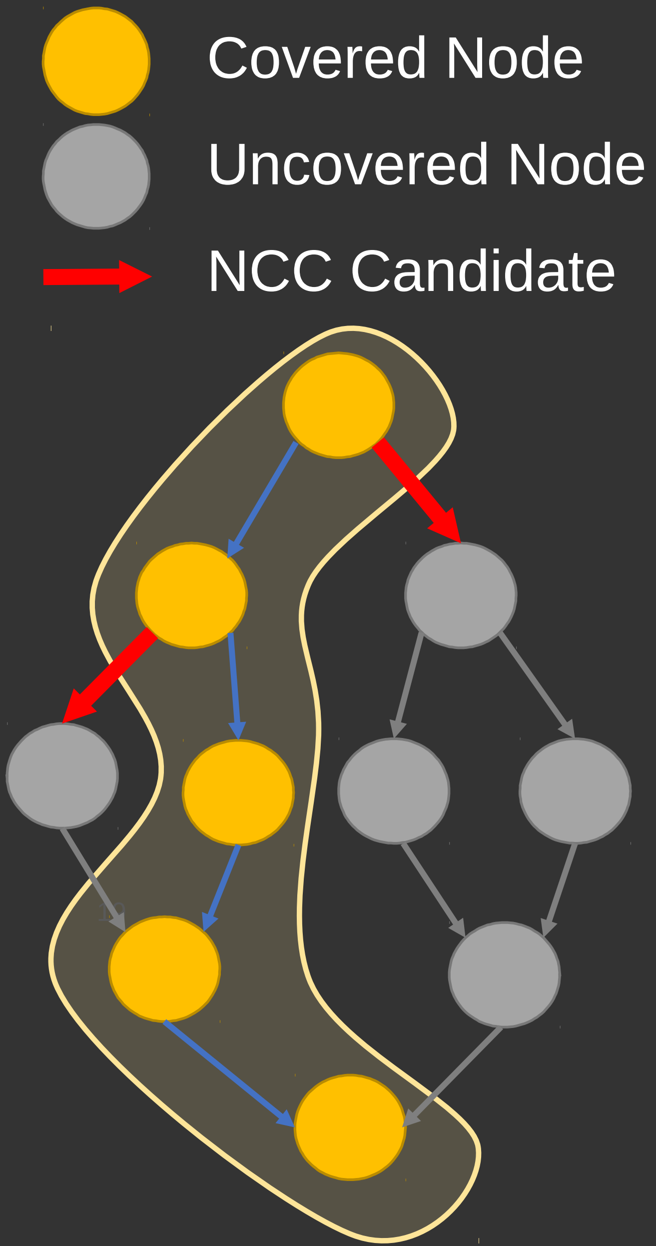
Fuzzing by Program Transformation

- Fuzzer fuzzes
- When stuck
  - Detect NCC candidates
  - Remove NCCs
  - Repeat
- Verify crashes in original program



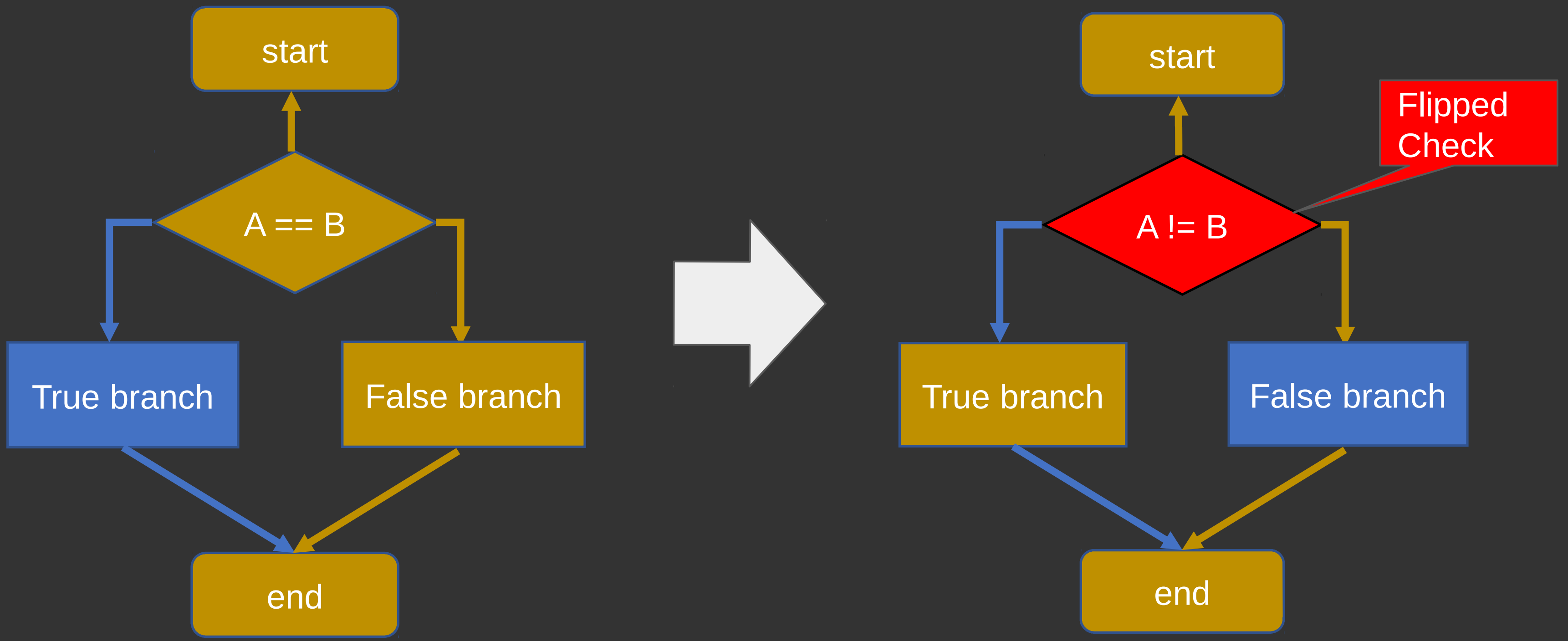
Detecting NCCs: imprecision

- Approximate NCCs as edges connecting covered and uncovered nodes in CFG
  - Over approximate, may contain false positives
  - Lightweight and simple to implement

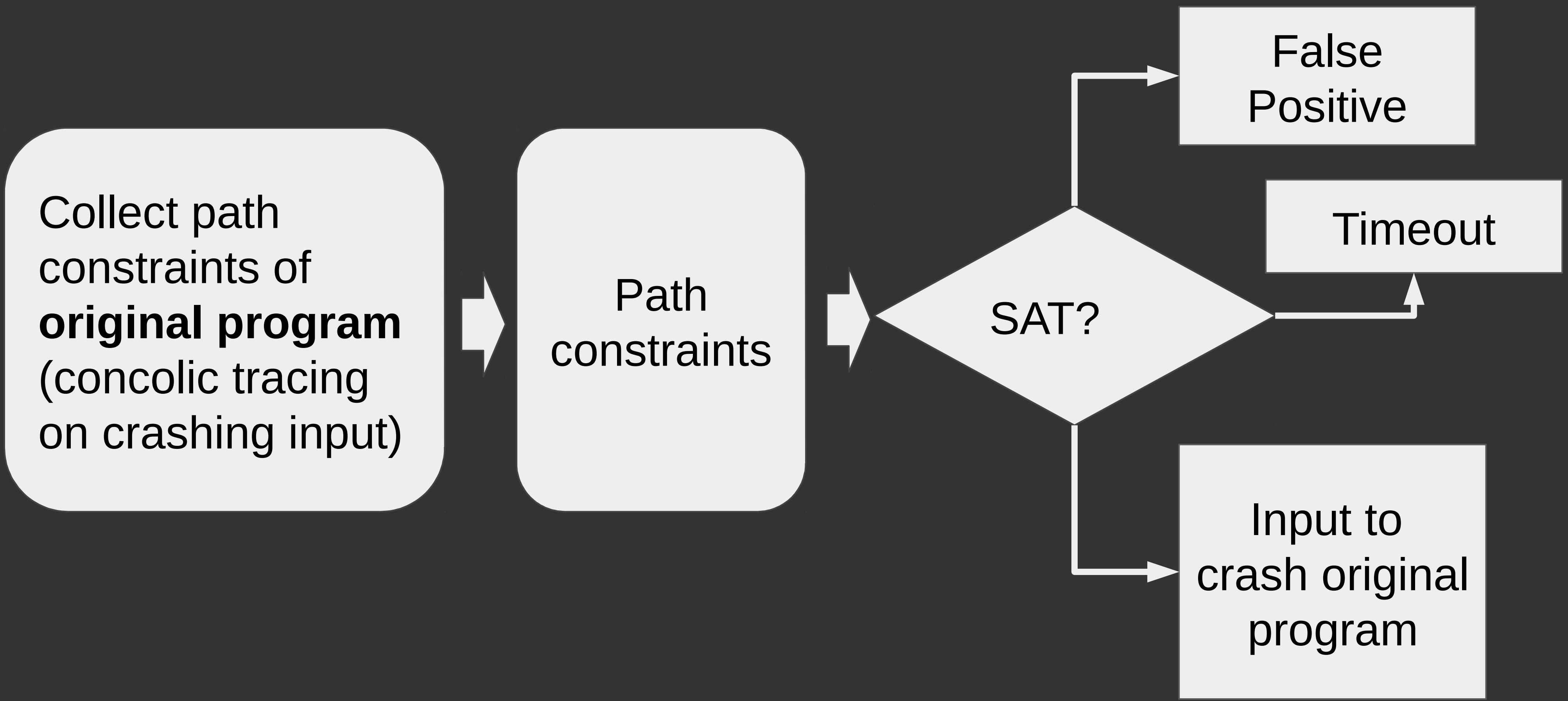


Program transformation

- Our approach: negate JCCs
  - Easy to implement: static binary patching
  - Zero runtime overhead in resulting target program
  - CFG/trace/path constraints remains the same



Crash analysis: false positives?



NCC example

Collected path constraints

{ x > 0, y == 0xdeadbeef }

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y == 0xdeadbeef)
      bug();
  }
}
```

Original Program

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y != 0xdeadbeef)
      bug();
  }
}
```

Transformed Program

NCC example

Collected path constraints

{ x > 0, y != 0xdeadbeef }

SAT

True BUG

flip

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y == 0xdeadbeef)
      bug();
  }
}
```

Original Program

```
int main (){
  int x = read_input();
  int y = read_input();
  if (x > 0) {
    if (y != 0xdeadbeef)
      bug();
  }
}
```

Flipped check

Transformed Program

NCC example 2

Collected path constraints

{ i > 0, i <= 0 }

```
int main (){
  int i = read_input();
  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i <= 0) {
    bug();
  }
  //...
}
```

Original Program

```
int main (){
  int i = read_input();
  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i > 0) {
    bug();
  }
  //...
}
```

Transformed Program

NCC example 2

Collected path constraints

{ i > 0, i > 0 }

UNSAT

False BUG

flip

```
int main (){
  int i = read_input();
  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i <= 0) {
    bug();
  }
  //...
}
```

Original Program

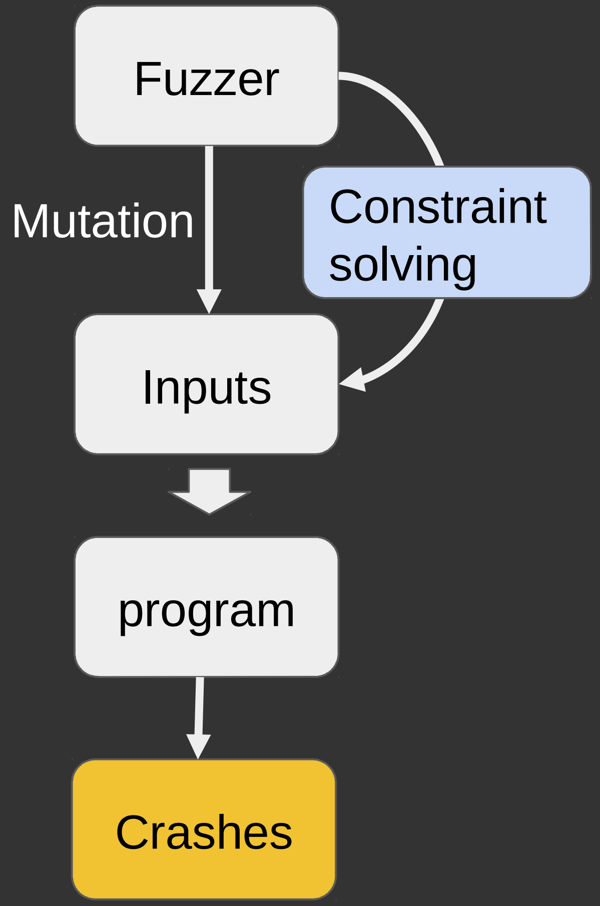
```
int main (){
  int i = read_input();
  if (i > 0) {
    func(i);
  }
}

void func(int i) {
  if (i > 0) {
    bug();
  }
  //...
}
```

Transformed Program

Flipped check

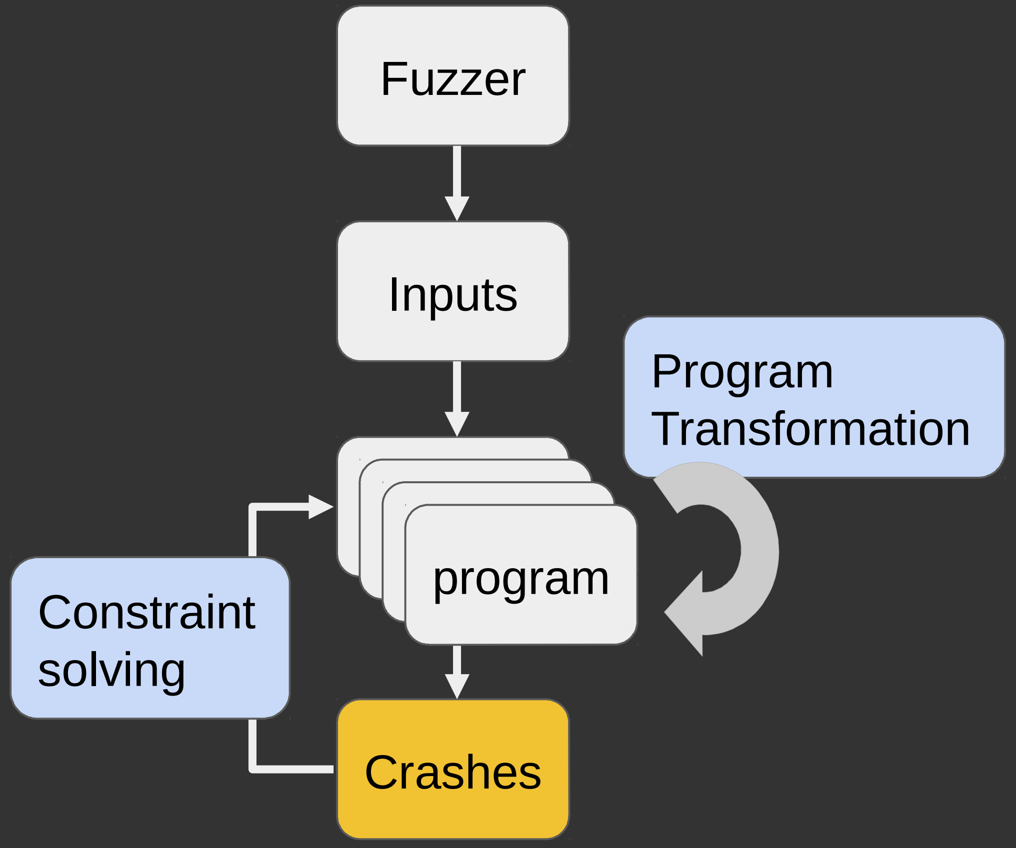
Comparison to Driller

- Fuzzing explores code paths
- Concolic execution explores new code paths when “stuck”
- Limitations
  - Constraints solving is slow
  - Unable to bypass “hard” checks



T-Fuzzing

- Constraint solving and fuzzing are decoupled
- Constraint solving only for crashes
- T-Fuzz detects bug for “hard” checks, but cannot verify it



Limitations

- NCC selection: transformation explosion
- False bugs: fault before bug
- Crash analyzer: constraint solving is hard
  - Length of trace
  - Number of constraints
  - Non-termination

Case study: CROMU_00030 (CGC)

```
void main() {
  int step = 0;
  Packet packet;
  while (1) {
    memset(packet, 0, sizeof(packet));
    if (step >= 9) {
      char name[5];
      int len = read(stdin, name, 128);
      printf("Well done, %s\n", name);
      return SUCCESS;
    }
    read(stdin, &packet, sizeof(packet));
    if(strcmp((char *)&packet, "1212") == 0)
      return FAIL;
    if (compute_checksum(&packet) != packet.checksum)
      return FAIL;
    if (handle_packet(&packet) != 0)
      return FAIL;
    step ++;
  }
}
```

Total time to find the bug: ~4h

Stack Buffer overflow bug

C1: check on magic

C2: checksum

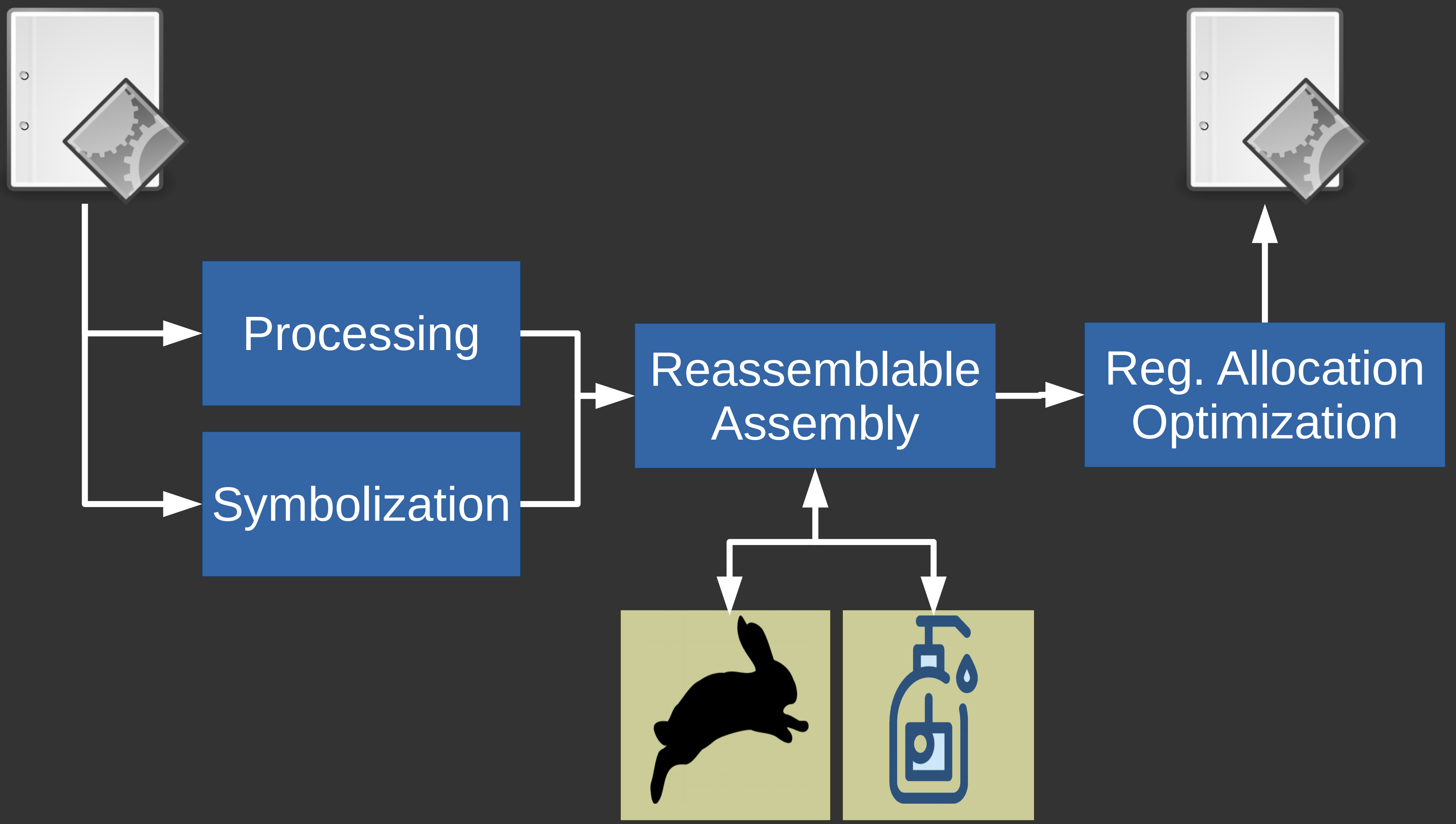
C3: authenticate user info

T-Fuzz summary

- Core idea: mutate both program and input
- T-Fuzz outperforms state-of-art fuzzers
  - Improvement over Driller/AFL by 45%/58%
  - Bugs: 1 in LAVA-M and 3 in real-world programs
- T-Fuzz future work
  - LLVM-based program transformation
  - Crash analyzer: optimize constraint solving
  - NCC detection through static analysis

Security-testing binary-only code

RetroWrite: static binary rewriting

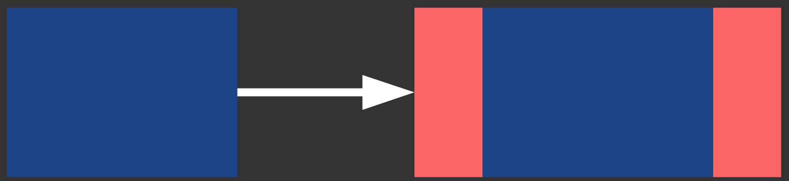


afl-retrowrite

- Instrument basic blocks to update coverage map
- To show interoperability, we reuse afl-gcc
  - afl-gcc / afl-clang instruments assembly files
  - Our symbolized assembly files follow the format of compiler-generated ASM files
  - Enables reuse of existing transformations!

Binary-only ASan (retrowrite-asan)

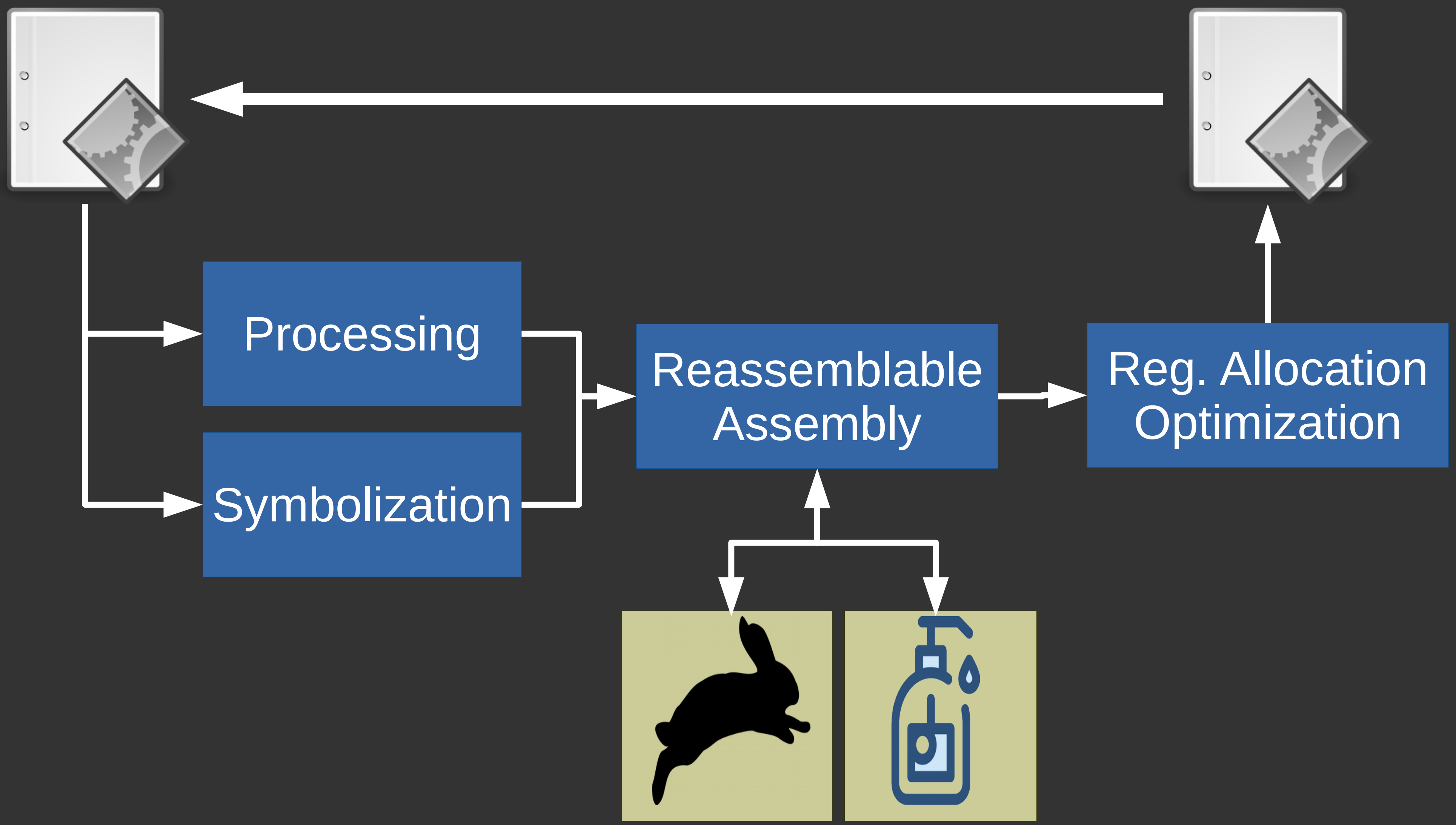
- RetroWrite API to identify instrumentation sites
- Two kinds of instrumentation:
  - Allocation Instrumentation



  - Memory Check Instrumentation

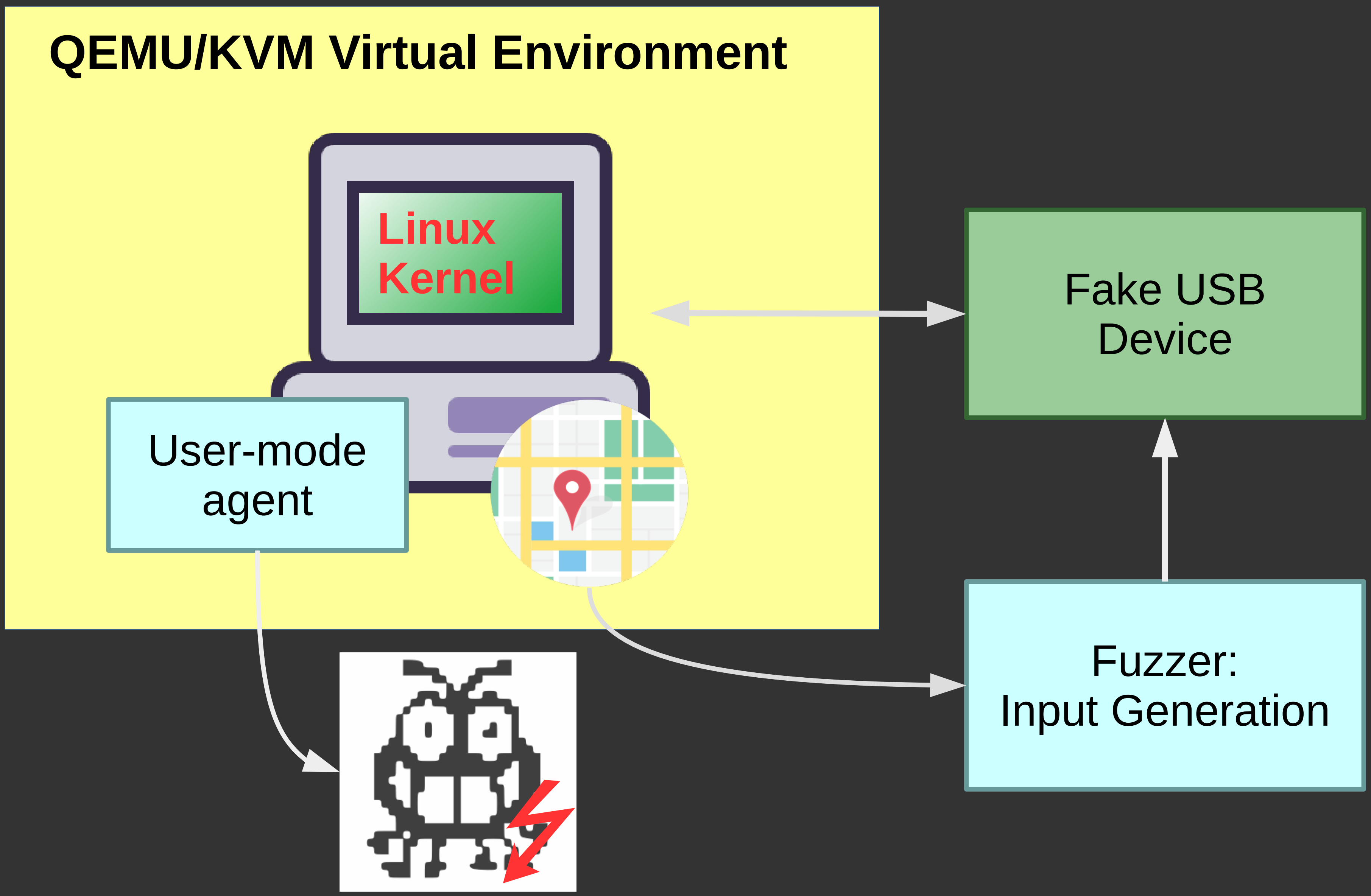
```
If 0x100 is poisoned:
  terminate();
var = access(0x100);
```

RetroWrite: static binary rewriting



Two-ended peripheral testing

USBFuzz: explore peripheral space



USBFuzz Evaluation

- ~60 new bugs discovered in recent kernels
- 36 memory bugs (UaF / BoF)
- 8 bugs fixed (with CVEs)
- Bug reporting in progress

Type	Bug Info	#
Memory Bugs (36)	double-free	2
	NULL pointer dereference	8
	general protection	6
	slab-out-of-bounds access	6
	user-after-free access	16
Unexpected state reached (17)	INFO	6
	WARNING	9
	BUG	2

Security testing hard-to-reach code

- Fuzzing is an effective way to automatically test programs for security violations (crashes)
  - Key idea: optimize for throughput
  - Coverage guides mutation

- T-Fuzz: mutate code ***<u>and</u>*** input
- RetroWrite: ***<u>efficient</u>*** static rewriting
- USBFuzz: enable fuzzing of ***<u>peripherals</u>***

https://github.com/HexHive