Week 10: Lecture B

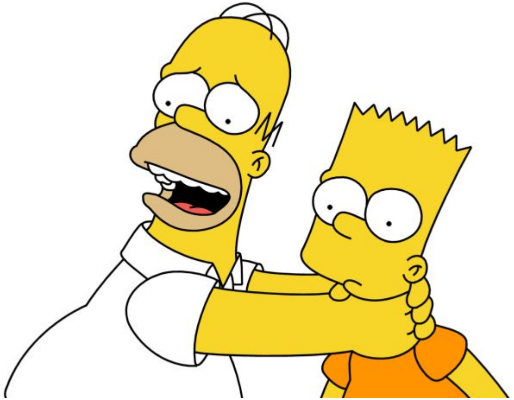
Hybrid Fuzzing II

Wednesday, March 20, 2024

How are projects going?

Problems?

Successes?



Questions?

Hybrid Fuzzing Recap

What is hybrid fuzzing?

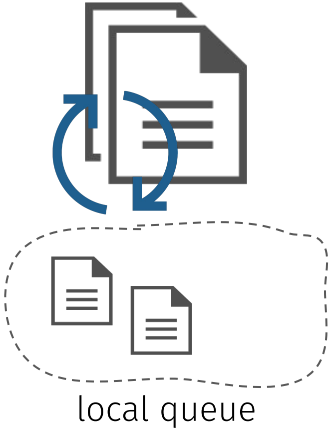
- **Combining crude fuzzing with smarter fuzzing**
  - E.g., **random** + **concolic execution** (Driller, QSYM, Savior)
  - E.g., **random** + **taint tracking** (VUzzer, RedQueen, Angora)

- **Goal is to balance strengths of both techniques**
  - Use generic fuzzing for **most test cases**
    - Use **speed** to brute-force easy branches
  - Deploy more elegant approach **selectively**
    - Focus its **precision** on harder branches

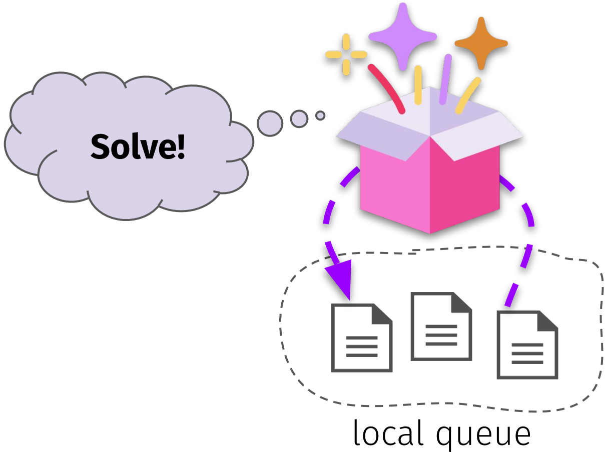
How most hybrid fuzzers work

- Leverage AFL-style **parallel fuzzing** mode with conventional fuzzer as parent

Conventional (e.g., AFL)

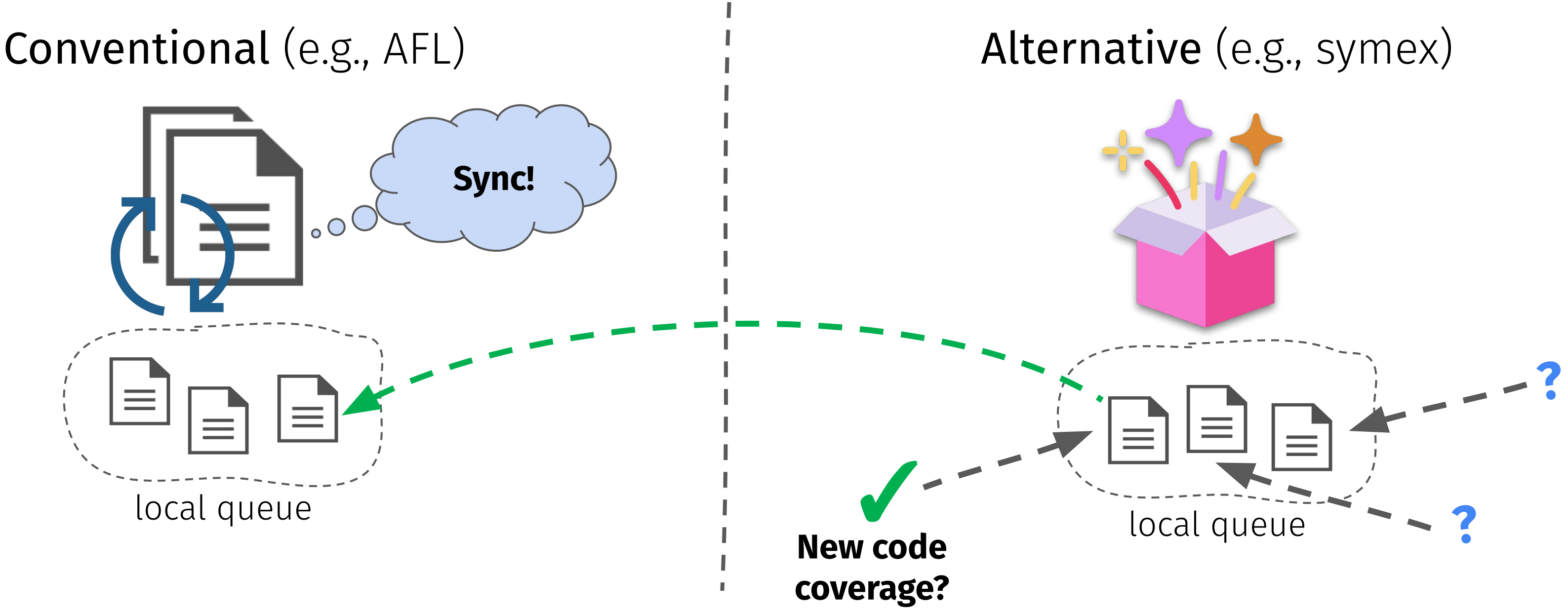


Alternative (e.g., symex)



How most hybrid fuzzers work

- Leverage AFL-style **parallel fuzzing** mode with conventional fuzzer as parent



What could go wrong?

- **Ineffective seed scheduling**
  - There are fundamental differences in **speed**
    - AFL can solve basic branch conditionals fast
    - Fancier approaches generally are much slower
  - Heavyweight approaches are best applied to a **subset** of paths
    - Invoking on all paths will lead to **path explosion**
    - E.g., by the time it’s solved, fuzzer is already way past

Questions?

Adventures in Hybrid Fuzzing:
Driller

Fuzzing

```
def f (x) {
 if x > 10 {
   if x < 100:
     print "You win!"
   else:
     print "You lose!"
 }else:
   print "You lose!"
```

1	⇒	"You lose!"
593	⇒	"You lose!"
183	⇒	"You lose!"
4	⇒	"You lose!"
498	⇒	"You lose!"
48	**⇒**	**"You win!"**

Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

Where fuzzing falls short

```
def f (x) {
 if x > 10 {
    if x^2 == 152399025:
       print "You win!"
    else:
       print "You lose!"
 }else:
    print "You lose!"
```

1 ⇒ "You lose!"

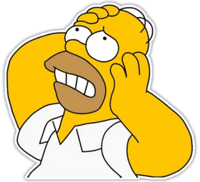
593 ⇒ "You lose!"

183 ⇒ "You lose!"

4 ⇒ "You lose!"

… …

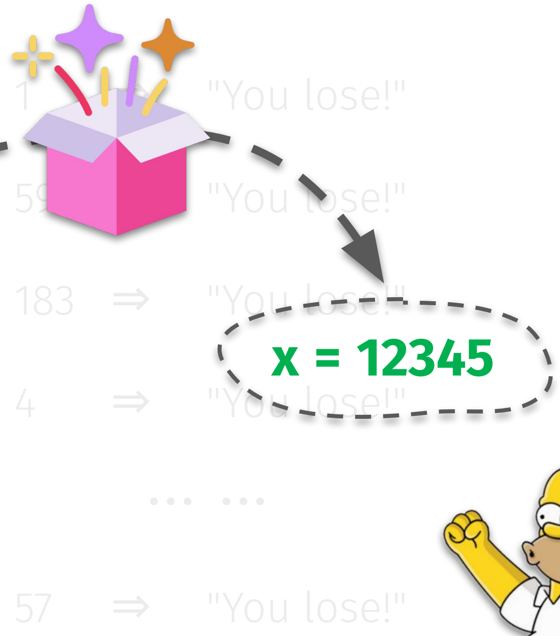
57 ⇒ "You lose!"



Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

Symbolic Execution to the rescue!

```
def f (x) {
 if x > 10 {
   if x^2 == 152399025:
     print "You win!"
   else:
     print "You lose!"
 }else:
   print "You lose!"
```



Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

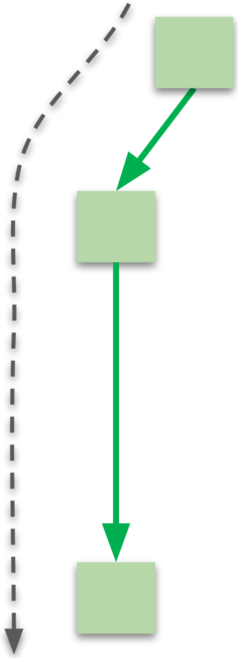
Driller

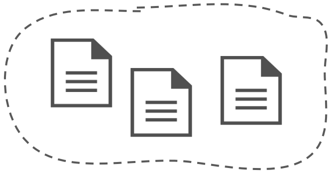
- **Idea:** invoke concolic execution via **demand launch**
  - **Heuristic 1:** a pre-determined # of mutations based on test case length
  - **Heuristic 2:** after a pre-determined time interval without new coverage

- Concolic executor based on **angr**
  - Binary-level instrumentation and analysis framework
  - Heavily maintained and used in many research projects
  - Translates, analyzes binary in **intermediate form** (VEXIR)

Driller in action

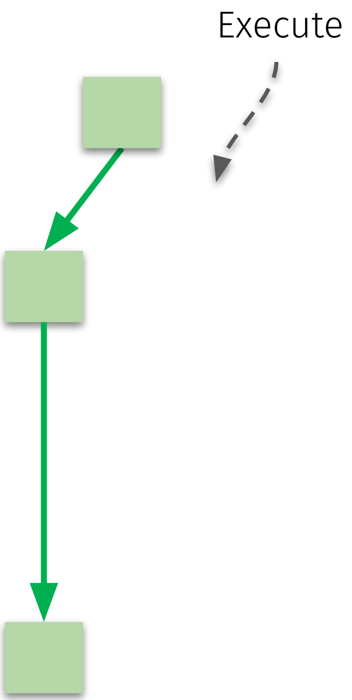
AFL-found
coverage

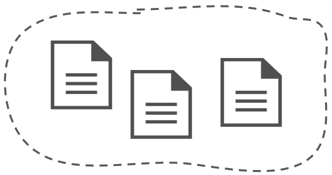




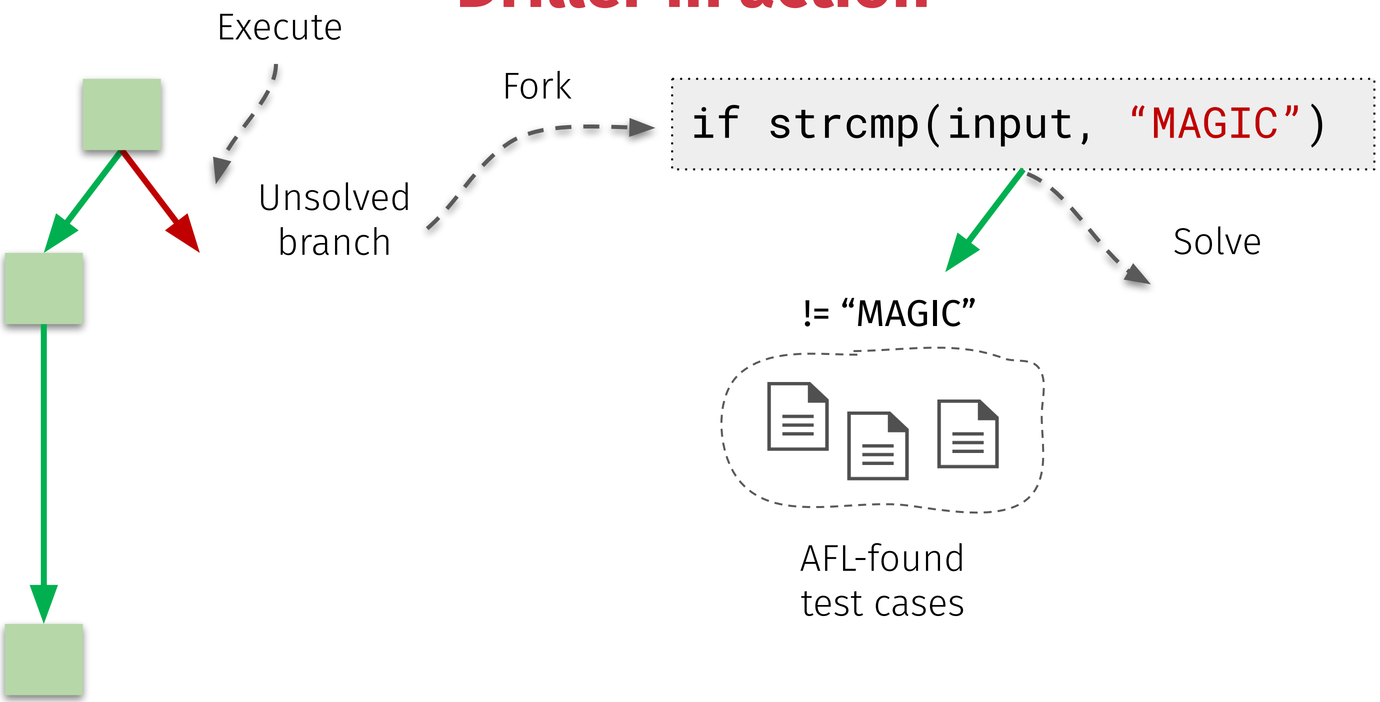
AFL-found
test cases

Driller in action

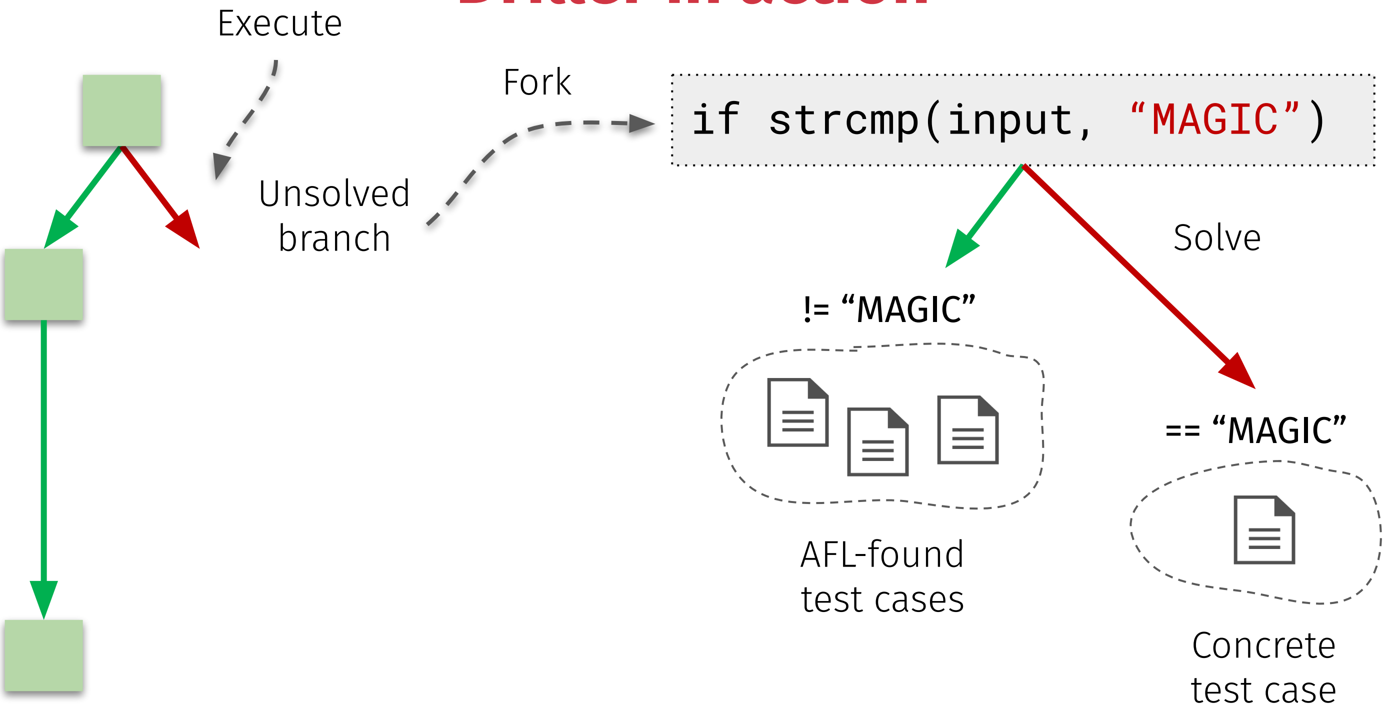




Driller in action

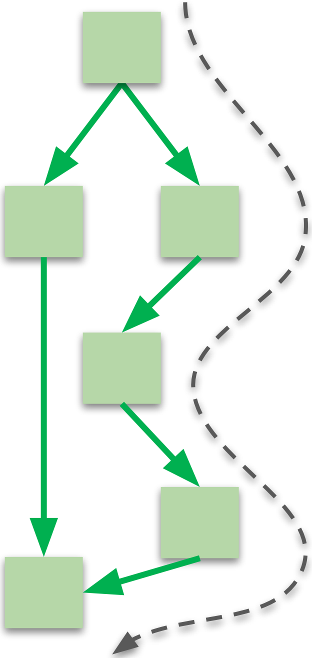


Driller in action

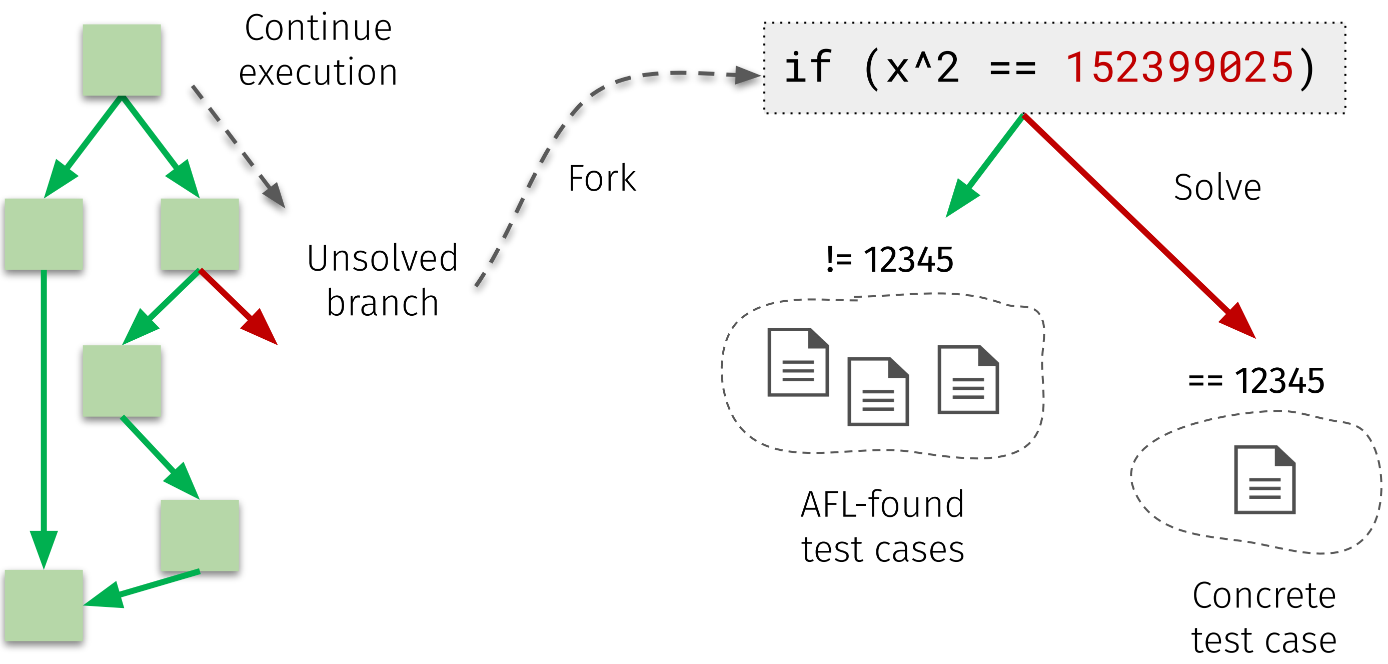


Driller in action

AFL-found coverage

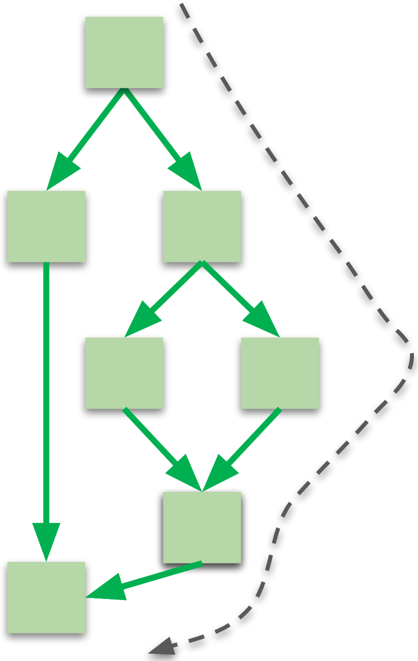


Driller in action



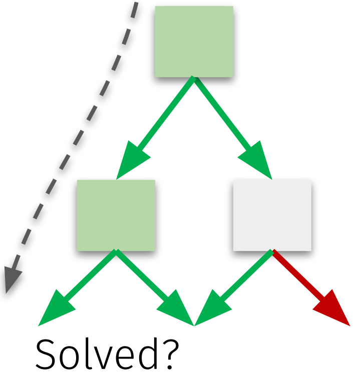
Driller in action

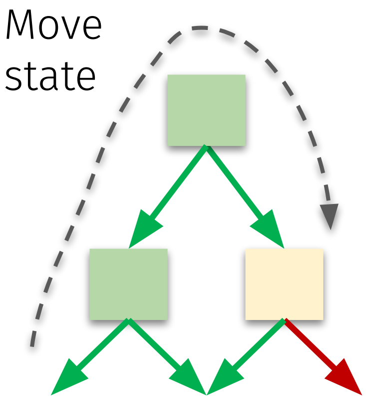
AFL-found coverage



When to turn solving elsewhere?

- When the path is already **fully solved**
  - Track all branches and which have been solved
  - A fundamental piece of info that is tracked



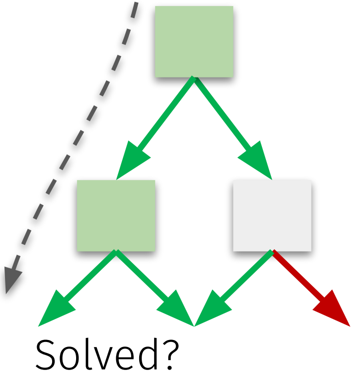


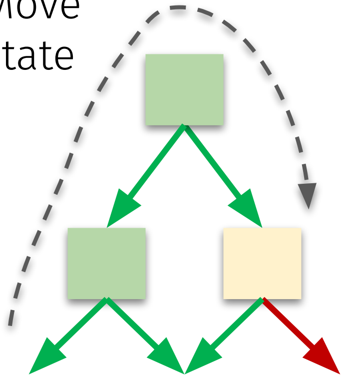
When to turn solving elsewhere?

- When the path is already **fully solved**
  - Track all branches and which have been solved
  - A fundamental piece of info that is tracked
- When symbolic executor **cannot solve**
  - Biggest culprit: **hashes**

```
if MD5(input) == "......."
```

A very large search space!





Questions?

Adventures in Hybrid Fuzzing:

QSYM

Problem: relying on an IR is costly

Executor	chksum	md5sum	sha1sum	md5sum(mosml)
Native	0.008	0.014	0.014	0.001
KLEE	26.243	32.212	73.675	0.285
angr	-	-	-	462.418

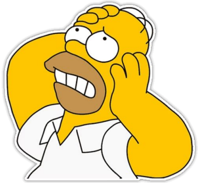
Table 1: The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used `chksum`, `md5sum`, and `sha1sum` in coreutils to test KLEE, and `md5sum (mosml)` [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.

Source: https://taesoo.kim/pubs/2018/yun:qsym.pdf

Problem: relying on an IR is costly

Executor	chksum	md5sum	sha1sum	md5sum(mosml)
Native	0.008	0.014	0.014	0.001
KLEE	26.243	32.212	73.675	0.285
angr	-	-	-	462.418

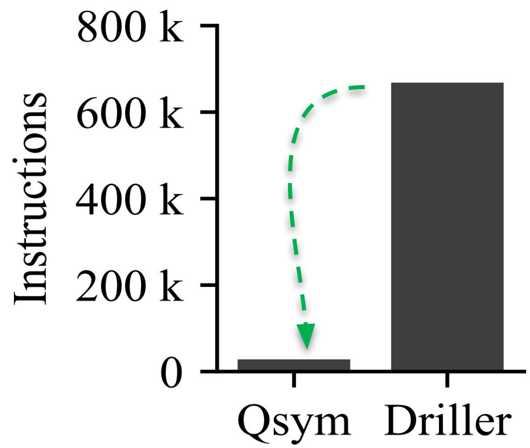
Table 1: The emulation overhead of KLEE and angr compared to native execution, which are underlying symbolic executors of S2E and Driller, respectively. We used `chksum`, `md5sum`, and `sha1sum` in coreutils to test KLEE, and `md5sum (mosml)` [12] to test angr because angr does not support the `fadvise` syscall, which is used in the coreutils applications.



Source: https://taesoo.kim/pubs/2018/yun:qsym.pdf

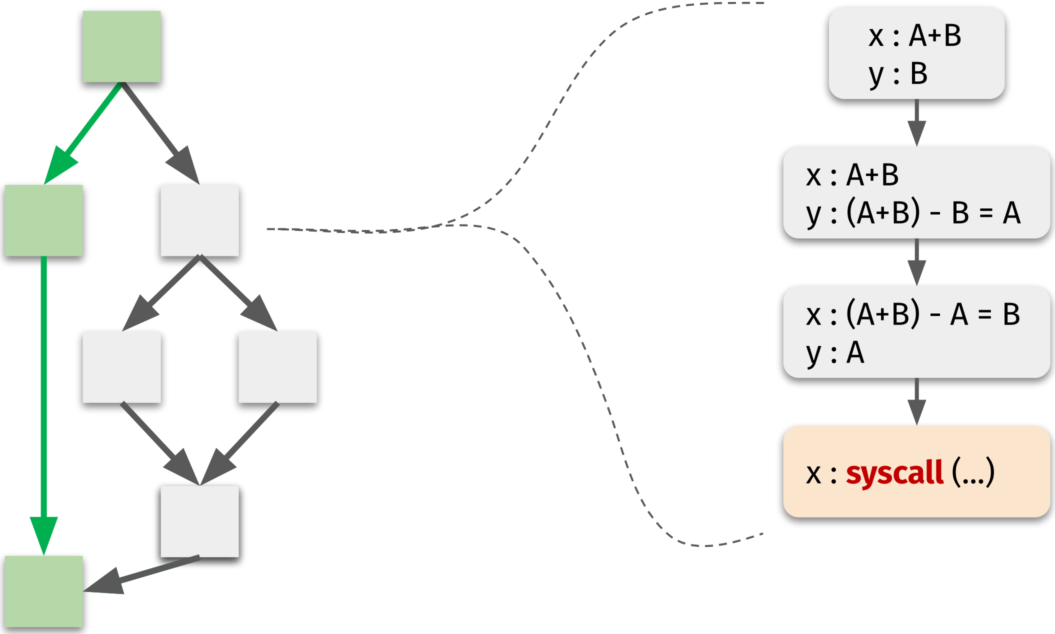
QSYM: operate on *native* instructions

- **Omit lifting to intermediate representation**
  - Use Intel PIN dynamic binary instrumentation
- **Trade-offs:**
  - A much higher **implementation complexity**
  - Significant **decrease in symbolic instructions**
    - **4X fewer** than Driller

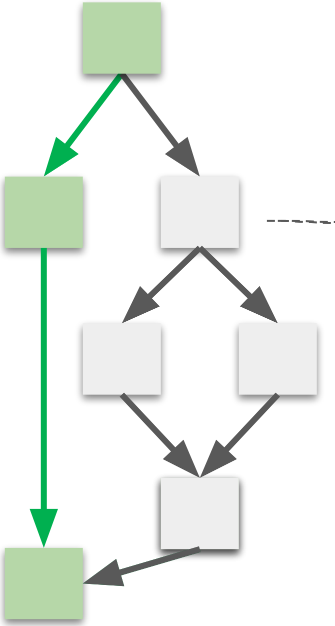


Source: https://taesoo.kim/pubs/2018/yun:qsym.pdf

Problem: incomplete environment modeling



Problem: incomplete environment modeling



x : A+B
y : B

x : A+B
y : (A+B) - B = A

x : (A+B) - A = B
y : A

x : **syscall** (...)

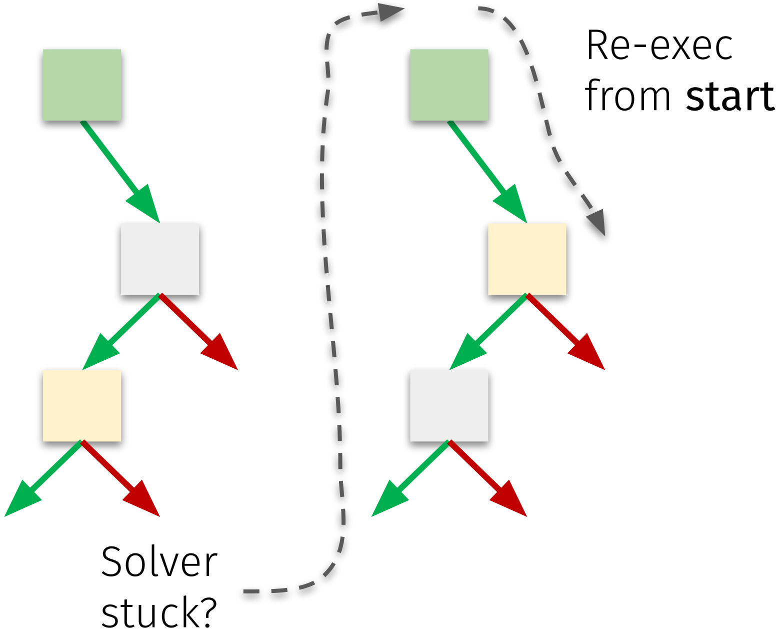
Non-trivial to model symbolically

Expensive to emulate and fork

QSYM: leave the environment as-is

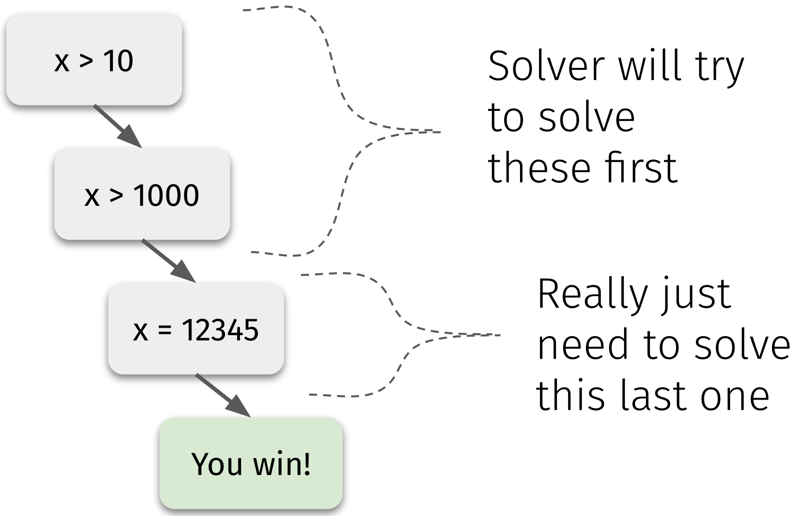
- **Omit translating the environment**
  - Use **concrete execution** to model it
    - Model only relevant system calls
    - E.g., standard input, reads, etc.
  - **What about kernel state forking?**
    - Avoid—just **re-execute** from the start

- **Trade-offs:**
  - Re-execution adds **more overhead**
    - Cannot “go back in time” like Driller



Problem: overconstrained paths

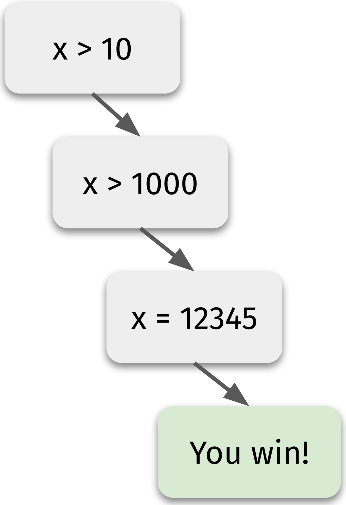
```
def f (x) {
 if x > 10 {
   if (x > 1000){
     if x^2 == 152399025:
       print "You win!"
     else:
       print "You lose!
   }else:
     print "You lose!"
 }else:
   print "You lose!"
```



Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

Problem: overconstrained paths

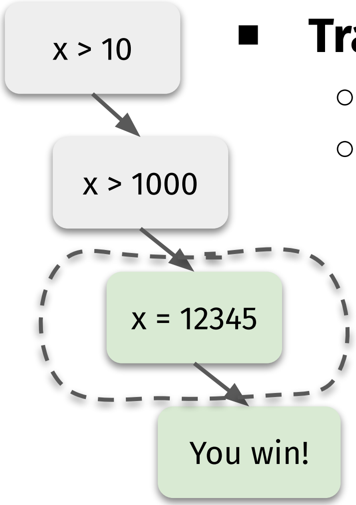
```
def f (x) {
 if x > 10 {
   if (x > 1000){
     if x^2 == 152399025:
       print "You win!"
     else:
       print "You lose!
   }else:
     print "You lose!"
 }else:
   print "You lose!"
```



Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

QSYM: optimistically solve *last* constraint

```
def f (x) {
 if x > 10 {
   if (x > 1000){
     if x^2 == 152399025:
       print "You win!"
     else:
       print "You lose!
   }else:
     print "You lose!"
 }else:
   print "You lose!"
```



- **Trade-offs:**
  - Does not always work
  - Can just let the fuzzer quickly rule these out

Source: https://www.ndss-symposium.org/wp-content/uploads/2017/09/07_3-ndss2016-slides.pdf

Questions?

Adventures in Hybrid Fuzzing:

RedQueen

Problem: symbolic and concolic execution is slow

```
if( u64(input) == hash(input[8..len]) )
    if( u64(input+8) == hash(input[16..len]) )
        if( input[16] == 'R' && input[17] == 'Q')
            print "You win!"
```

Source: https://www.ndss-symposium.org/wp-content/uploads/ndss2019_04A-2_Aschermann_slides.pdf

Problem: symbolic and concolic execution is slow

```
1.  if( u64(input) == hash(input[8..len]) )
2.      if( u64(input+8) == hash(input[16..len]) )
3.          if( input[16] == 'R' && input[17] == 'Q')
4.              print "You win!"
```

Source: https://www.ndss-symposium.org/wp-content/uploads/ndss2019_04A-2_Aschermann_slides.pdf

RedQueen's solution: input-to-state tracking

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with **compared-to value** (lifted directly from the operand)

source `if (x[0:3] == "ABCD")` `CMP (eax, 0x44434241)` binary

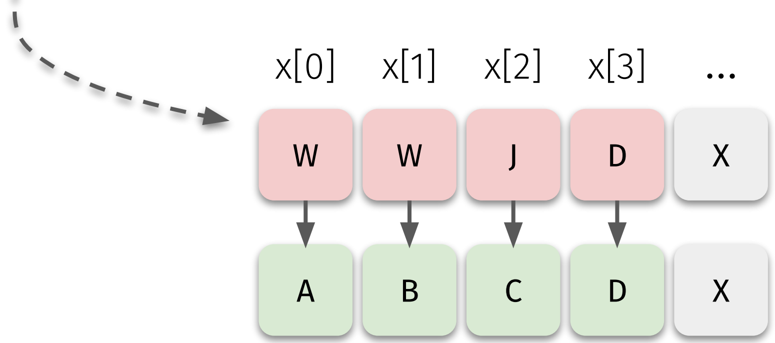
x[0]	x[1]	x[2]	x[3]	...
W	W	J	D	X

RedQueen's solution: input-to-state tracking

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with **compared-to value** (lifted directly from the operand)

source `if (x[0:3] == "ABCD")`

`CMP (eax, 0x44434241)` binary



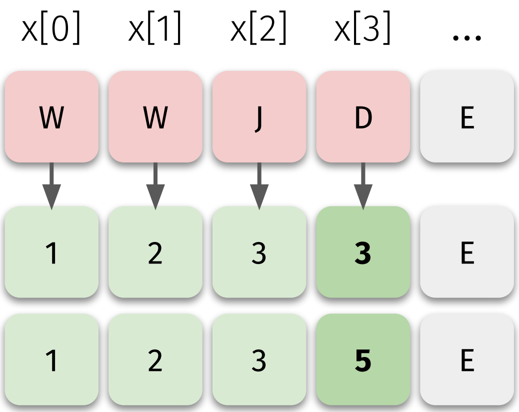
Supporting other comparisons

- **Idea:** hook comparison instructions and identify their input bytes
  - Replace with **compared-to value** (lifted directly from the operand)

source
```
if (x[0:3] < 1234)
```

Replace(x[0:3], 1234 **- 1**)

Replace(x[0:3], 1234 **+ 1**)



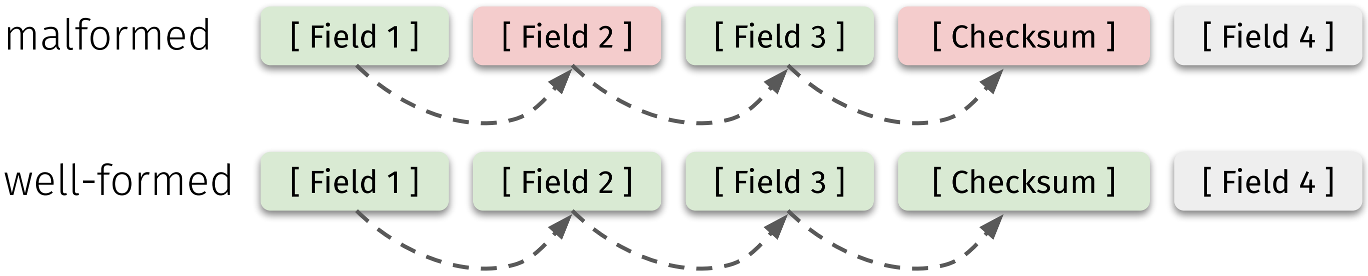
What about checksums?

- **Finding these at the binary-level is difficult**
  - **Assumption:** can identify input bytes that affect the checksum hash
  - **Colorize the input:** inject random bytes and see if they influence the outcome

```
if( u64(input) == hash(input[8..len]) )
```

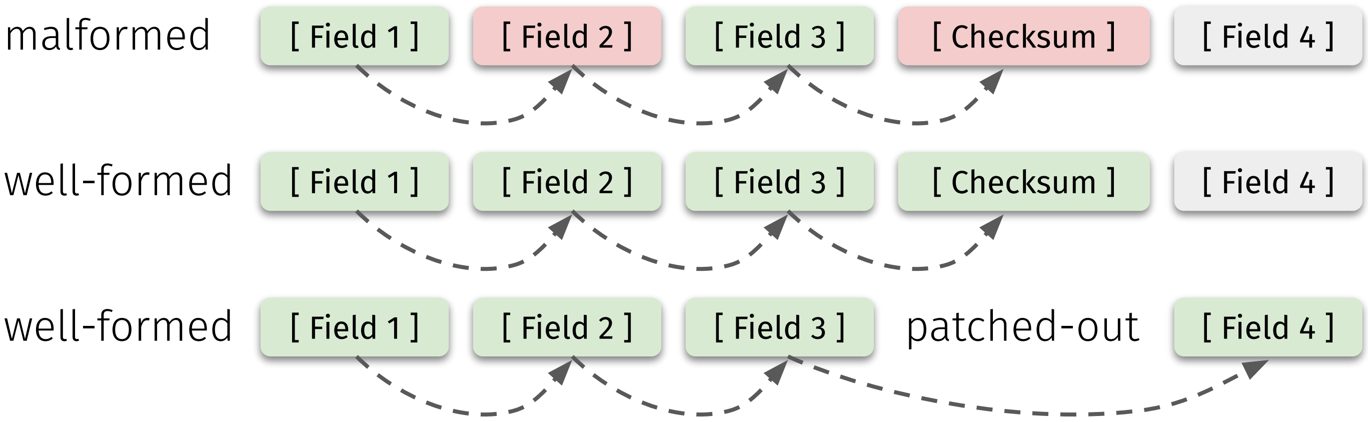
What about checksums?

- Then, **patch-out the checksum** with an always-true operation
  - **Assumption:** checksum is only passed if the input is well-formed



What about checksums?

- Then, **patch-out the checksum** with an always-true operation
  - **Assumption:** checksum is only passed if the input is well-formed
    - Thus, skipping over checksum **won't matter if well-formed**
    - New input found afterwards? Great—restore the checksum



Questions?