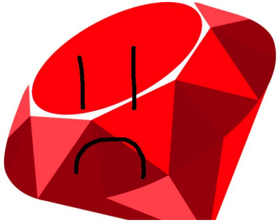
Differential fuzzing, or: how to find bugs when (ground) truth isn’t real

William Woodruff
OSIRIS Lab

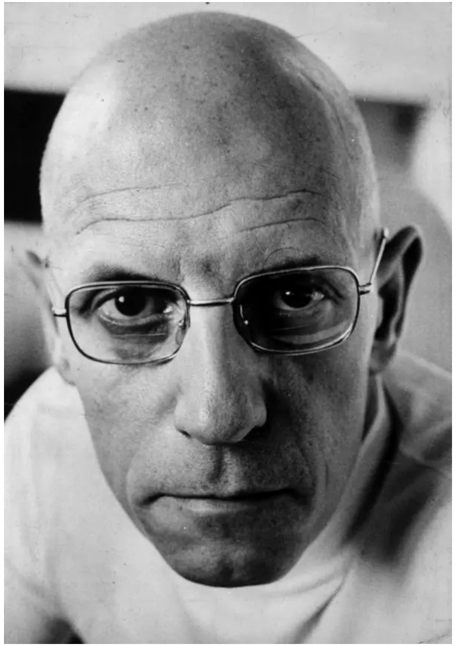
Hi

- **Me**
  - senior security engineer (certified good at computers)
  - R&D: program analysis research (mostly LLVM)
  - Engineering: open-source C, C++, Rust, Ruby, Python
- **Trail of Bits**
  - About 60 people, ~30% in NYC, rest remote
  - Research, engineering, assurance
  - Very good
  - Summer and winter internships



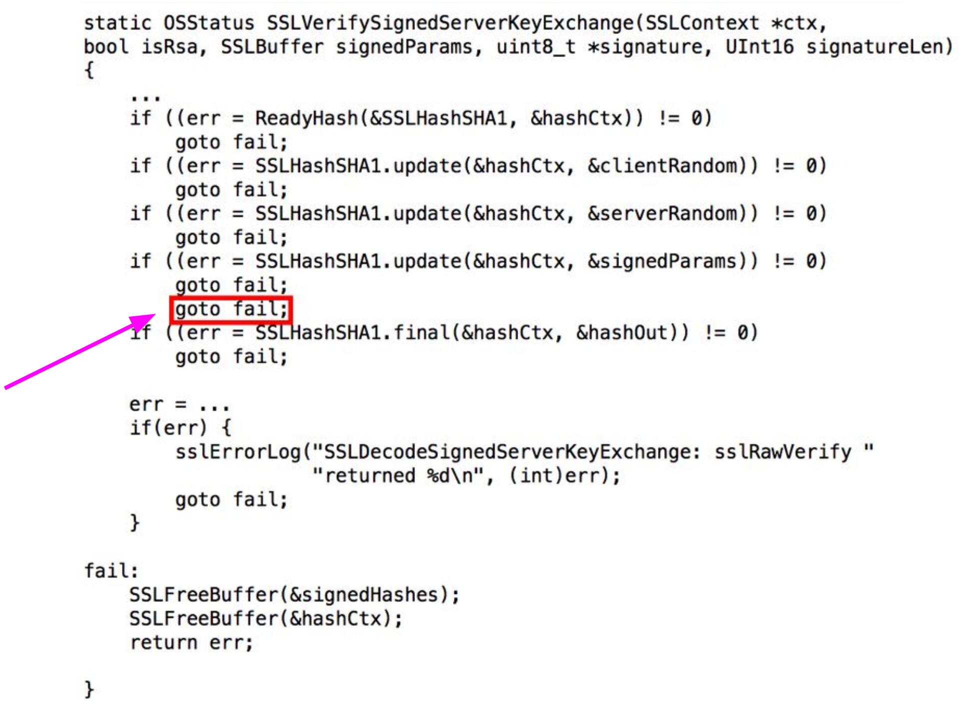
Today's agenda

- Finding bugs
- Finding bugs with fuzzing
- Normal fuzzing can't find some bugs :(
  - We don't even know what bugs are, actually
- Spicy (differential) fuzzing to find those bugs
- Example case: x86_64 decoding
  - Demo!!!



Finding bugs

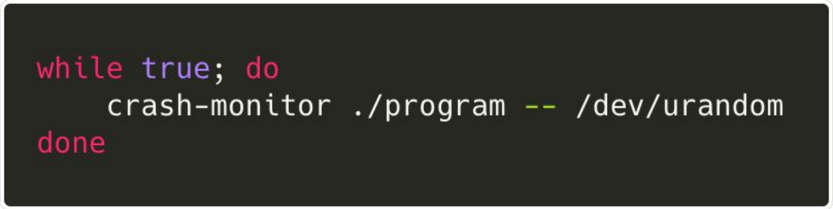
- **Why? We want to...**
  - write reliable, safe code (cred)
  - embarrass our coworkers (more cred)
  - ~~embarrass~~ help our clients (money)
- **How?**
  - Manual code review
    - Expensive (money & time), fallible (goto fail)
  - Static analysis, formal methods, symbolic execution
    - Cheap-ish (compute heavy), sometimes effective, often indefeasible (luv 2 explode state)
  - Fuzzing



```
static OSStatus SSLVerifySignedServerKeyExchange(SSLContext *ctx,
bool isRsa, SSLBuffer signedParams, uint8_t *signature, UInt16 signatureLen)
{
    ...
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;

    err = ...
    if(err) {
        sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                    "returned %d\n", (int)err);
        goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;

}
```

Finding bugs with fuzzing

- **The TL;DR of fuzzing: feed garbage into the program until it crashes**



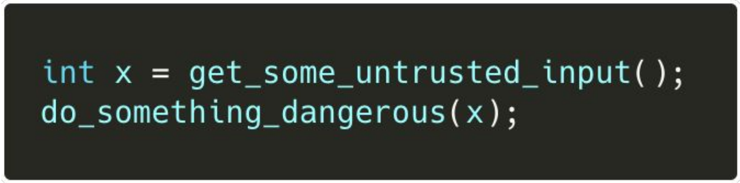
```
while true; do
    crash-monitor ./program -- /dev/urandom
done
```

- **Inexpensive, *shockingly* good at discovering (exploitable) bugs**
- **Problem: random inputs won’t explore the program much**
  - Intuition: Most random inputs don’t resemble HTML, ZIP streams, PNGs...
- **Solution: Use a feedback mechanism to guide inputs**
  - “If it <runs longer/calls more functions/has more coverage>, try similar inputs”

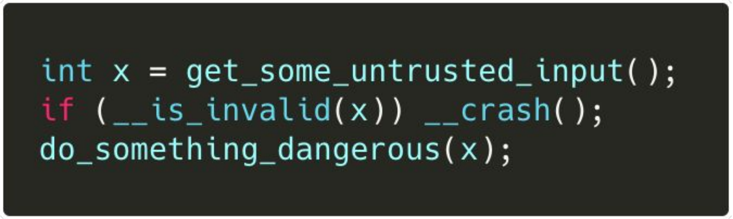
SCIENCE IS A LIAR SOMET
ARISTOTLE
BITCH
GALILEO
2+2=5?
BITCH
LIGHT
NEBVLOSA ORIONIS
NEBVLOSA
EVOLUTION
GOD
GHT
MAC
Coors
LIGHT

Not all bugs are easily fuzzable

- **Not all bugs...**
  - cause easy-to-observe crashes (segfaults, aborts, non-zero exits, &c)
  - are memory corruptions (logic errors, permission errors, DoS, &c)
- **If we have a specification, we can instrument the program to turn non-crashing errors into discoverable crashes**



```
int x = get_some_untrusted_input();
do_something_dangerous(x);
```



```
int x = get_some_untrusted_input();
if (__is_invalid(x)) __crash();
do_something_dangerous(x);
```

What's a bug without a specification?

- **Not everything has a real specification**
- **Lots of things have "specifications" that are basically ignored**
- **The real spec is generally-agreed-upon behavior**
    - "What does Adobe Acrobat do? Make our program do that"
- **Lots of things are written in memory safe languages**
    - == no memory corruption == no crashes on "bugs"
    - + no specification == no easy instrumentation approach :(

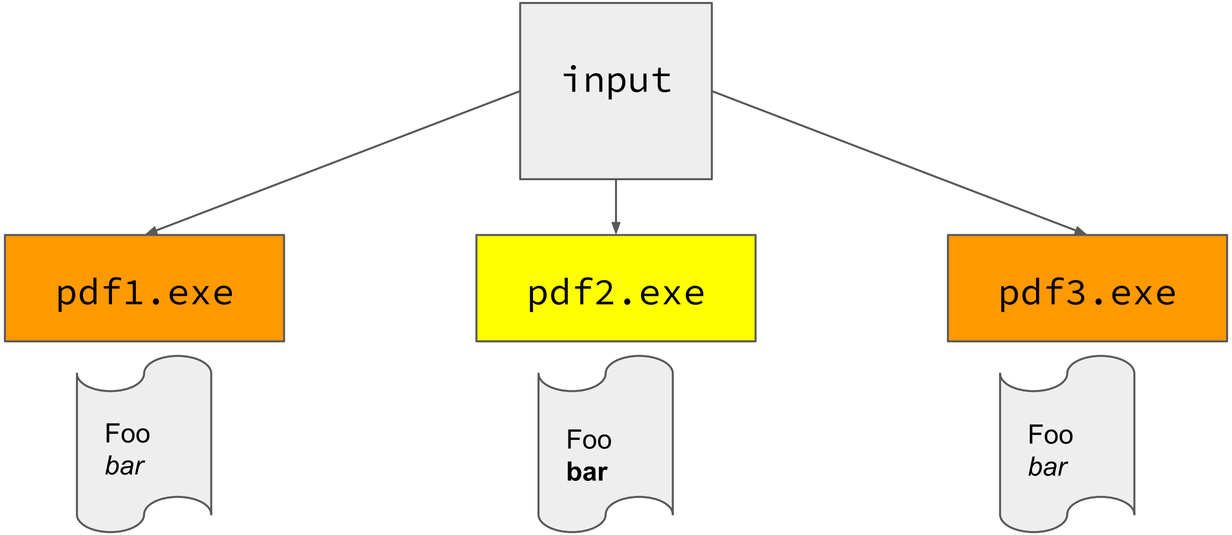
Another perspective: ground truth and oracles

- **Restate the problem: instead of bugs, we want an oracle**
- **Oracle supplies some notion of “ground truth”**
  - a yes or no answer for whether some behavior is correct
- **Different oracles:**
  - C and C++: segfaults, assertions, non-zero exits
  - Memory safe languages: exceptions, assertions, contract violations
- **Still no oracle if a “bug” doesn’t cause any of these!**
  - Back where we started :(

Constructing ground truth from difference

- **Observation: lots of things have multiple implementations**
    - Multiple PDF parsers, ZIP extractors, HTTP header parsers
- **Observation: lots of programs copy ideas and features from competitors**
    - “Acrobat can do $X so our program needs to be able to do $X!”
- **Observation: copying features without a specification means underspecification + lots of variation on unexpected inputs**
- **What if we compared different implementations?**
    - What if we define “bug” == “difference between impls”?

Differential fuzzing: we can’t all be right



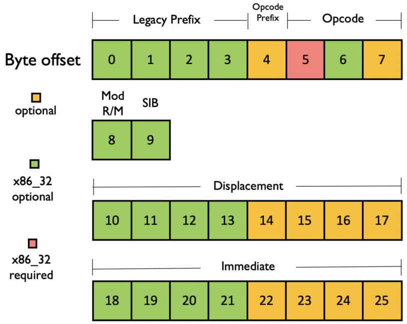
- Three programs, two different results (🟧and 🟨, italic vs. bold)
- Not clear which is “right”, but both *probably* aren’t
- No crashes needed!
- What if this but automated?

Differential fuzzing: applications

- **Any complex, popular format with competing implementations**
  - PDF, Word, media container formats (MKV, MP4)
- **Crypto primitives (hashing, digital signatures)**
  - Prior work: Wycheproof (Google), CDF (Kudelski)
- **Competing hardware implementations of ISAs**
  - x86_64: sandsifter
- **Competing software decoders for ISAs**
  - ARM: MC-Hammer
  - x86_64: mishegos (us!)

Case study: x86_64 decoding

- **Ideal target for differential fuzzing:**
  - Large, messy ISA with thousands of unique instructions
  - Complex encoding format with >50 years of backwards compatibility
  - Variable-length instructions, unlike ARM! Up to 15 bytes!!!
  - Two major vendors with totally independent implementations: Intel, AMD
  - Lots of popular, open source decoders to compare:
    - Capstone (mostly LLVM), zydis, XED (Intel), libopcodes (GNU)
  - High-interest/impact bugs:
    - Mess up debuggers, RE platforms, static analysis tools, ...



Legacy Prefix | Opcode Prefix | Opcode

Byte offset	0	1	2	3	4	5	6	7

🟨 optional

Mod R/M	SIB
8	9

🟩 x86_32 optional

Displacement

10	11	12	13	14	15	16	17

🟥 x86_32 required

Immediate

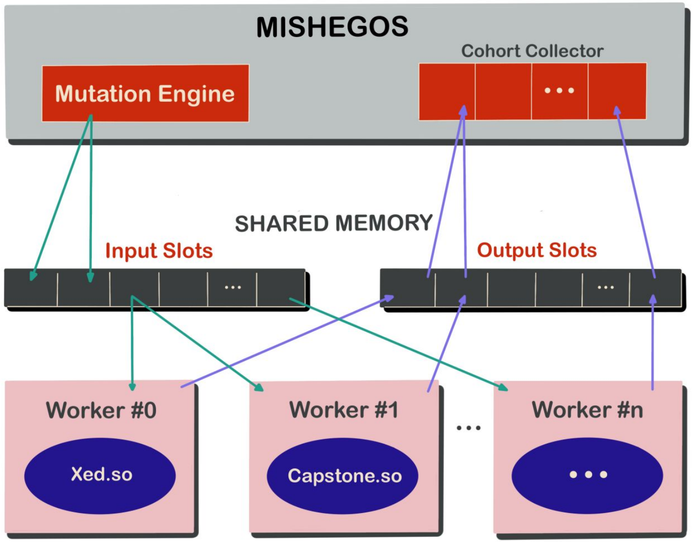
18	19	20	21	22	23	24	25

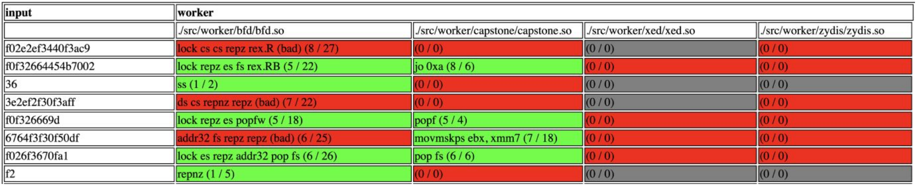
Case study: x86_64 decoding

The basic idea:

- **Spawn a bunch of workers that wrap different decoder impls.**
- **Blast “random” inputs at the workers**
  - Not really random: use x86_64’s structure to inform our choices
    - Legacy prefixes, SIB byte, &c
- **Record what each worker claims each input decodes to**
- **Compare and contrast**
- **?? Bugs ??**

Mishegos: differential fuzzing for x86 decoders

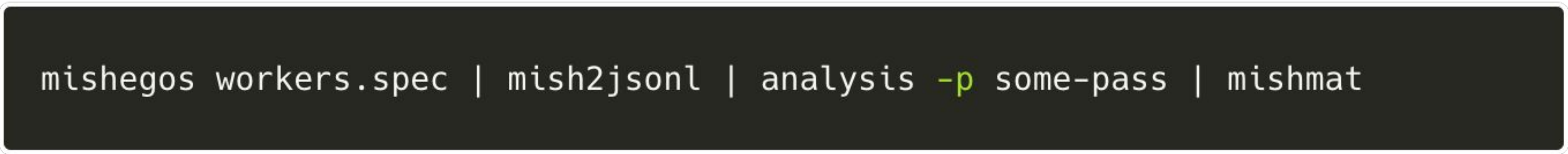




input	worker			
	./src/worker/bfd/bfd.so	./src/worker/capstone/capstone.so	./src/worker/xed/xed.so	./src/worker/zydis/zydis.so
f02e2ef3440f3ac9	lock cs cs repz rex.R (bad) (8 / 27)	(0 / 0)	(0 / 0)	(0 / 0)
f0f32664454b7002	lock repz es fs rex.RB (5 / 22)	jo 0xa (8 / 6)	(0 / 0)	(0 / 0)
36	ss (1 / 2)	(0 / 0)	(0 / 0)	(0 / 0)
3e2ef2f30f3aff	ds cs repnz repz (bad) (7 / 22)	(0 / 0)	(0 / 0)	(0 / 0)
f0f326669d	lock repz es popfw (5 / 18)	popf (5 / 4)	(0 / 0)	(0 / 0)
6764f3f30f50df	addr32 fs repz repz (bad) (6 / 25)	movmskps ebx, xmm7 (7 / 18)	(0 / 0)	(0 / 0)
f026f3670fa1	lock es repz addr32 pop fs (6 / 26)	pop fs (6 / 6)	(0 / 0)	(0 / 0)
f2	repnz (1 / 5)	(0 / 0)	(0 / 0)	(0 / 0)

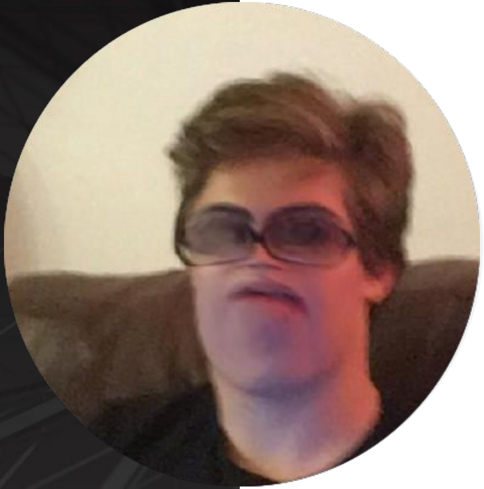
Mishegos: making sense of the noise

- Results: ~tens of millions of results per hour
  - Depends on the number of workers, system load, ...
- Need an automated strategy for filtering the interesting results
- Observation: we want a list (or DAG?) of filters to run, biggest first
- Implemented as “passes” (think LLVM) on transformed (JSON) output
- Boils down to this pipeline:



```
mishegos workers.spec | mish2jsonl | analysis -p some-pass | mishmat
```

Contact Slide



William Woodruff

Senior Security Engineer

william@trailofbits.com

@8x5clPW2 | github.com/woodruffw

TRAIL OF BITS