TRAIL OF BITS

Differential analysis of x86-64 decoders

William Woodruff, Niki Carroll, Sebastiaan Peters

Contact Information

William Woodruff
Trail of Bits

william@trailofbits.com
@8x5clPW2

Niki Carroll
George Mason University

ncarrol5@gmu.edu
@inventednight

Sebastiaan Peters
Eindhoven University of Technology

s.peters2@student.tue.nl

Items

- **Overview of instruction decoding**
  - Why is x86-64 especially challenging?
- **Instruction decoding as an attack surface**
- **Differential analysis of instruction decoders**
- **Mishegos: a differential fuzzer for x86-64 decoders**
- **Results and future work**

Instruction decoding

- **Individual machine code instructions → assembler mnemonics**
- ***The* fundamental building block for correct disassembly**
  - Function, control/data flow, call graph recovery all depend on correct decoding
  - Small mistakes → incorrect control/data flow, misaligned decodings, ...
- **Decoding depends on ISA/vendor-specific parameters:**
  - Fixed length or variable length encoding?
  - Revisions to the ISA? Backwards compatible, incompatible?
  - Open, machine-readable specification?
  - Hidden opcodes/unintended functionality?
  - Vendor-specific behavior?
  - Completeness vs. decoding what the common compilers emit?

Instruction decoding: not just for REs anymore!

- **Previously: primarily reverse engineers and debuggers**
    - Decoder errors here are annoying, but there's a human in the loop
    - Human fixes/stubs the error and continues
- **Now: on the hotpath of user code:**
    - Interpreted languages: JITs, sandboxes
    - Antivirus tools: static disassembly and analysis
    - Static & dynamic binary translation (Apple's Rosetta)
    - Self-modifying code (runtime patching in the Linux kernel)
    - Constrained runtimes/VMs (WASM, eBPF)
    - *No human in the loop to catch errors!*

Can we exploit instruction decoding errors?

- **Errors in instruction decoding have security consequences**
    - Sandboxes, VMs, JITs: run mis-generated code, escape the environment
        - eBPF: violate runtime guards
    - Antivirus: confuse AV checks into skipping/not flagging malicious code
        - *…or trip up code/data disambiguation*
    - SBTs/DBTs: convert a benign program into a misbehaving one
        - Turn benign code into vulnerable code
        - Introduce timing channels or information leaks?
- **Speculation: x86-64's complexity makes it particularly susceptible to decoder implementation errors**

x86-64 decoding is (particularly) hard

- **Numerous features that make x86-64 hard to decode correctly:**
  - ✅ Complex, variable-length instruction format (up to 15 bytes per instruction)
  - ✅ >40 years of semi-compatible ISA revisions
  - ❌ No formal specification, two separate major vendors with references aimed at developers rather than decoder implementers
  - ❌ A long history of undocumented opcodes, including backdoors!
    - 😭 Intel: AAD/AAM variants, SALC, ICEBP, UD1
    - 😢 AMD: 3DNow! variants (Domas 2017)
    - 😿 VIA: ALTINST + C3 family backdoors (Domas 2017)
  - ❌ Variations in vendor behavior (operand size prefix, SYSRET differences)
  - ❌ Under- and undefined semantics
    - 😈 Multiple prefixes, EFLAGS state on BSF/BSR variants

Where are the bugs in x86-64 decoders?

- **Speculation 1: many bugs will exist where discrete regions of the instruction format meet**
  - Unlikely to completely mess up displacement decoding, but *might* mess up displacement decoding with multiple legacy prefixes and an uncommon SIB encoding
- **Speculation 2: many bugs will also exist where x86-64 has undergone historical changes**
  - Subtleties with the REX prefix, validity of instructions in long mode, …
- **Speculation 3: x86-64 instruction decoders are primarily tested against compilers**
  - Decoder authors under-test against instructions/formats that compilers don't use
- **Speculation 4: most bugs in x86-64 instruction decoders won't cause memory unsafety in the decoder itself**
  - Not a good target for traditional crash-driven fuzzing…

Differential analysis of instruction decoders

Instruction decoders are an ideal target for differential fuzzing:

- **Relatively little memory unsafety → simple harnesses**
- **Lots of competing (and open source!) implementations**
- **Simple interfaces (byte string → instruction)**
- **Lots of signals to diff against:**
  - Decoding success/failure, failure kind
  - Decoded length
  - Decoded semantics (instruction ground, operand count, ...)

Differential analysis of instruction decoders

Prior work:

- **Paleari et al. (2010): Compare *N* decoders against a hardware decoder as a source of ground truth**
- **Jay & Miller (2018): Normalize each decoder's assembly output and compare for discrepancies**

Both approaches focus on false negatives; we also want to identify false positives!

Mutation strategies for x86-64

Prior work:

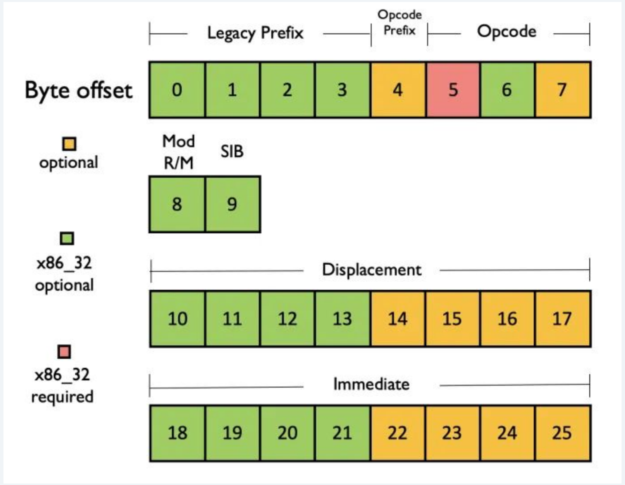
- **Paleari: Generate random input sequences, mix in with sequences from hardware-guided generation**
- **Jay & Miller: Seed the mutator with valid inputs, perform bitflips and sample outputs based on inferred instruction structure**

We want to do better than random generation *without* relying on hardware ground truth (Paleari) or structure inference from a potentially unreliable decoder (Jay & Miller)!

A “sliding” mutation strategy

Observation: the maximum x86-64 instruction length is 15 bytes

… but the maximum size of each instruction field adds up to 26:



We can soundly overapproximate the structure of a potential x86-64 instruction by filling up to 26 bytes!

We can evaluate speculations 1 and 2 by being better-than-random about legacy prefixes, REX, ModR/M, SIB, etc.

A “sliding” mutation strategy

Mutation engine lifecycle

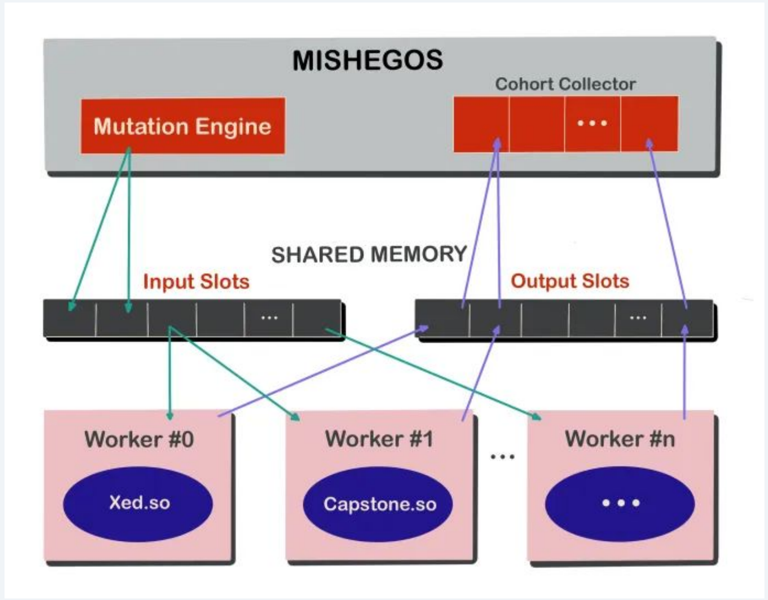
- **Generate a “maximal” overapproximated candidate**
- **Extract individual 15-byte “sliding” candidates from the “maximal” candidate**
- **Once the “maximal” is exhausted, generate a new one**
- **Repeat until fuzzing is halted**

```c
if (insn_cand.off == 0) {
  build_sliding_candidate();
}

if (insn_cand.len <= MISHEGOS_INSN_MAXLEN) {
  memcpy(slot->raw_insn, insn_cand.insn, insn_cand.len);
  slot->len = insn_cand.len;
  insn_cand.off = 0;
} else {
  memcpy(
    slot->raw_insn,
    insn_cand.insn + insn_cand.off,
    MISHEGOS_INSN_MAXLEN);
  slot->len = MISHEGOS_INSN_MAXLEN;
  insn_cand.off =
    (insn_cand.off + 1) % (insn_cand.len - MISHEGOS_INSN_MAXLEN + 1);
}
```

Tests our speculations by feeding a wide range of prefix/structured interactions to the decoders!

Mishegos: our differential fuzzer for x86-64

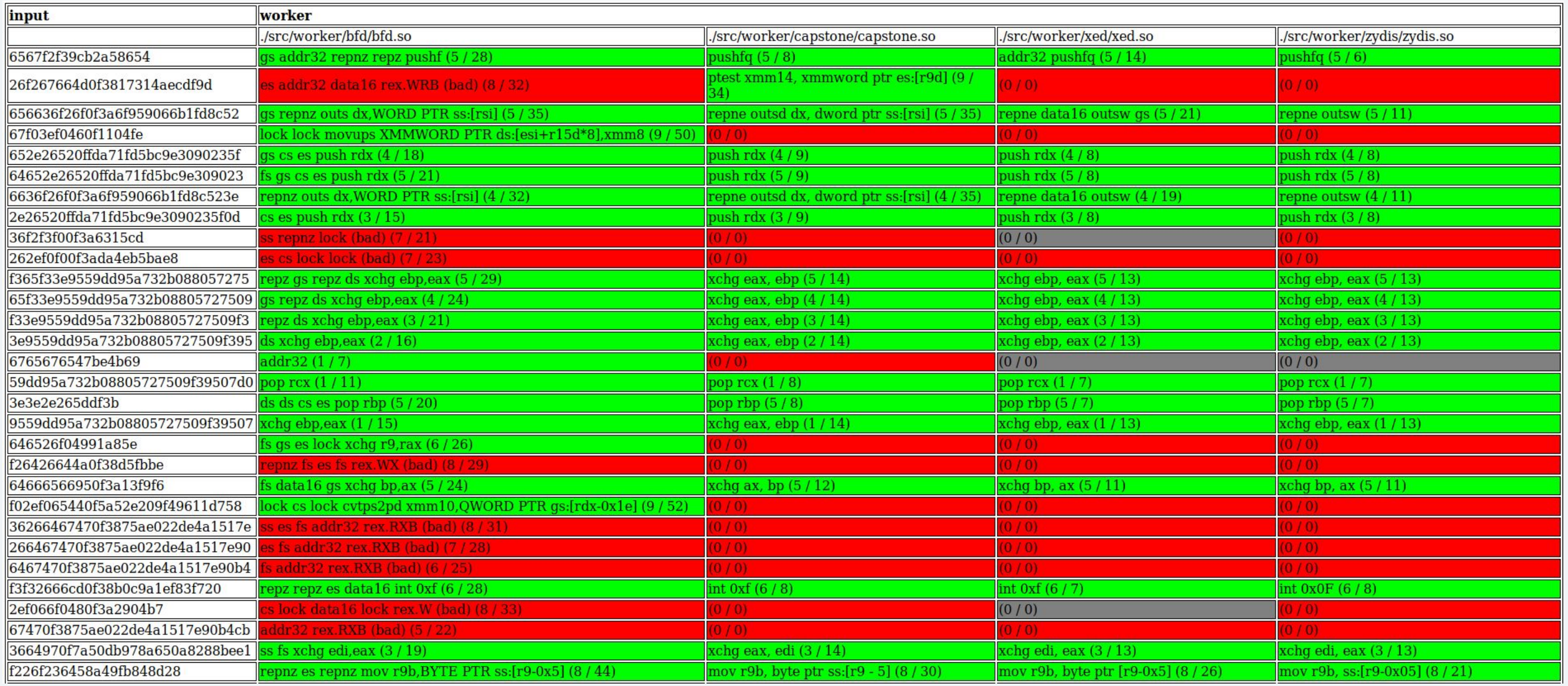


- Each “sliding” candidate is placed in an input slot, to be consumed by each decoder’s worker
- Workers implement a simple ABI to wrap their underlying decoder
- Each decoder’s worker puts its result for an input into an output slot; inputs are pruned once all workers have attempted it
- Outputs are collected into “cohorts” of *N* for *N* decoders for a single input

Mishegos: analysis framework

- **Fuzzing with mishegos produces cohorts of outputs**
  - Each output in a cohort contains the decoder state for the cohort's input:
    - Status (success, failure, kind of failure)
    - # of bytes decoded
    - Disassembly of instruction, length of disassembly
- **Cohorts need to be analyzed for errors and discrepancies**
  - Observation: analyses compose well, perform cheap + effective ones first
  - Remove cohorts that only contain errors, only contain successes, ...
  - Select cohorts where outputs disagree on status, length
  - Treat particular outputs as "high-quality" and use them as ground truth

Mishegos: the bird’s eye view



input	worker			
	./src/worker/bfd/bfd.so	./src/worker/capstone/capstone.so	./src/worker/xed/xed.so	./src/worker/zydis/zydis.so
6567f2f39cb2a58654	gs addr32 repnz repz pushf (5 / 28)	pushfq (5 / 8)	addr32 pushfq (5 / 14)	pushfq (5 / 6)
26f267664d0f3817314aecdf9d	es addr32 data16 rex.WRB (bad) (8 / 32)	ptest xmm14, xmmword ptr es:[r9d] (9 / 34)	(0 / 0)	(0 / 0)
656636f26f0f3a6f959066b1fd8c52	gs repnz outs dx,WORD PTR ss:[rsi] (5 / 35)	repne outsd dx, dword ptr ss:[rsi] (5 / 35)	repne data16 outsw gs (5 / 21)	repne outsw (5 / 11)
67f03ef0460f1104fe	lock lock movups XMMWORD PTR ds:[esi+r15d*8],xmm8 (9 / 50)	(0 / 0)	(0 / 0)	(0 / 0)
652e26520ffda71fd5bc9e3090235f	gs cs es push rdx (4 / 18)	push rdx (4 / 9)	push rdx (4 / 8)	push rdx (4 / 8)
64652e26520ffda71fd5bc9e309023	fs gs cs es push rdx (5 / 21)	push rdx (5 / 9)	push rdx (5 / 8)	push rdx (5 / 8)
6636f26f0f3a6f959066b1fd8c523e	repnz outs dx,WORD PTR ss:[rsi] (4 / 32)	repne outsd dx, dword ptr ss:[rsi] (4 / 35)	repne data16 outsw (4 / 19)	repne outsw (4 / 11)
2e26520ffda71fd5bc9e3090235f0d	cs es push rdx (3 / 15)	push rdx (3 / 9)	push rdx (3 / 8)	push rdx (3 / 8)
36f2f3f00f3a6315cd	ss repnz lock (bad) (7 / 21)	(0 / 0)	(0 / 0)	(0 / 0)
262ef0f00f3ada4eb5bae8	es cs lock lock (bad) (7 / 23)	(0 / 0)	(0 / 0)	(0 / 0)
f365f33e9559dd95a732b088057275	repz gs repz ds xchg ebp,eax (5 / 29)	xchg eax, ebp (5 / 14)	xchg ebp, eax (5 / 13)	xchg ebp, eax (5 / 13)
65f33e9559dd95a732b08805727509	gs repz ds xchg ebp,eax (4 / 24)	xchg eax, ebp (4 / 14)	xchg ebp, eax (4 / 13)	xchg ebp, eax (4 / 13)
f33e9559dd95a732b08805727509f3	repz ds xchg ebp,eax (3 / 21)	xchg eax, ebp (3 / 14)	xchg ebp, eax (3 / 13)	xchg ebp, eax (3 / 13)
3e9559dd95a732b08805727509f395	ds xchg ebp,eax (2 / 16)	xchg eax, ebp (2 / 14)	xchg ebp, eax (2 / 13)	xchg ebp, eax (2 / 13)
6765676547be4b69	addr32 (1 / 7)	(0 / 0)	(0 / 0)	(0 / 0)
59dd95a732b08805727509f39507d0	pop rcx (1 / 11)	pop rcx (1 / 8)	pop rcx (1 / 7)	pop rcx (1 / 7)
3e3e2e265ddf3b	ds ds cs es pop rbp (5 / 20)	pop rbp (5 / 8)	pop rbp (5 / 7)	pop rbp (5 / 7)
9559dd95a732b08805727509f39507	xchg ebp,eax (1 / 15)	xchg eax, ebp (1 / 14)	xchg ebp, eax (1 / 13)	xchg ebp, eax (1 / 13)
646526f04991a85e	fs gs es lock xchg r9,rax (6 / 26)	(0 / 0)	(0 / 0)	(0 / 0)
f26426644a0f38d5fbbe	repnz fs es fs rex.WX (bad) (8 / 29)	(0 / 0)	(0 / 0)	(0 / 0)
64666566950f3a13f9f6	fs data16 gs xchg bp,ax (5 / 24)	xchg ax, bp (5 / 12)	xchg bp, ax (5 / 11)	xchg bp, ax (5 / 11)
f02ef065440f5a52e209f49611d758	lock cs lock cvtps2pd xmm10,QWORD PTR gs:[rdx-0x1e] (9 / 52)	(0 / 0)	(0 / 0)	(0 / 0)
36266467470f3875ae022de4a1517e	ss es fs addr32 rex.RXB (bad) (8 / 31)	(0 / 0)	(0 / 0)	(0 / 0)
266467470f3875ae022de4a1517e90	es fs addr32 rex.RXB (bad) (7 / 28)	(0 / 0)	(0 / 0)	(0 / 0)
6467470f3875ae022de4a1517e90b4	fs addr32 rex.RXB (bad) (6 / 25)	(0 / 0)	(0 / 0)	(0 / 0)
f3f32666cd0f38b0c9a1ef83f720	repz repz es data16 int 0xf (6 / 28)	int 0xf (6 / 8)	int 0xf (6 / 7)	int 0x0F (6 / 8)
2ef066f0480f3a2904b7	cs lock data16 lock rex.W (bad) (8 / 33)	(0 / 0)	(0 / 0)	(0 / 0)
67470f3875ae022de4a1517e90b4cb	addr32 rex.RXB (bad) (5 / 22)	(0 / 0)	(0 / 0)	(0 / 0)
3664970f7a50db978a650a8288bee1	ss fs xchg edi,eax (3 / 19)	xchg eax, edi (3 / 14)	xchg edi, eax (3 / 13)	xchg edi, eax (3 / 13)
f226f236458a49fb848d28	repnz es repnz mov r9b,BYTE PTR ss:[r9-0x5] (8 / 44)	mov r9b, byte ptr ss:[r9 - 5] (8 / 30)	mov r9b, byte ptr [r9-0x5] (8 / 26)	mov r9b, ss:[r9-0x05] (8 / 21)

Mishegos: performance, results, evaluation

- **Fuzzer: 33M cohorts/hour → 228M decoder results/hour**
- **Analyzer:**
  - **68.4M decoder results/hour** after deduplication
  - **130M net cohorts** for a 4 hour campaign:
    - **15M status discrepancies** (one or more decoders disagree on validity)
    - **3.4M size discrepancies** (one or more decoders disagree on decoded length)
    - **59K cases** of XED "overaccepting" instructions
      - *Probably* all others underaccepting e.g. multi-byte NOPs!

Decoders: XED, libopcodes, Zydis, Capstone, DynamoRIO, bddisasm, Iced

Evaluated on 8 physical cores of an Intel Xeon 6140, running 7 decoder workers + the mutation engine and collector process

Mishegos: analysis highlights

Discovered a variety of decoder bugs through discrepancies:

- **Capstone: Incorrect control flow targets (`call`), incorrect length decoding esp. with legacy prefixes**
- **XED: False positives and false negatives for ISA extensions**
- **bddisasm: False positives, particularly around decoding instructions that don't work on AMD64 in long mode**
- **libopcodes (GNU bfd): Remarkably broken! Failed to decode a large number of valid instructions**

Future research

Mishegos constitutes the first step (discovery) in evaluating the security posture of instruction decoders

Important future work:

- **Evaluating the salience of discrepancies (control/data flow, decoder confusion, misaligning the instruction stream)**
- **Automatically emplacing discrepancies (generating "schizophrenic" binaries)**
- **Further refinement of "sliding"; minimizing duplicate work**

Resources

GitHub: https://github.com/trailofbits/mishegos

Blog post: Destroying x86-64 decoders with differential fuzzing

Selected citations:

Paleari et al. 2010: N-version disassembly

Jay & Miller 2018: FLEECE

Domas 2017: Breaking the x86 ISA