EasyChair Preprint

№ 5095

Differential Analysis of X86-64 Instruction Decoders

William Woodruff, Niki Carroll and Sebastiaan Peters

March 3, 2021

Differential analysis of x86-64 instruction decoders

William Woodruff[§]
Trail of Bits
New York, New York
william@trailofbits.com

Niki Carroll[†]
George Mason University
Fairfax, Virginia
ncarrol5@gmu.edu

Sebastiaan Peters[†]
Eindhoven University of Technology
Eindhoven, The Netherlands
s.peters2@student.tue.nl

***Abstract*—Differential fuzzing replaces traditional fuzzer oracles like crashes, hangs, unsound memory accesses with a *difference* oracle, where an implementation of a specification is said to be potentially erroneous if its behavior differs from another implementation's on the same input. Differential fuzzing has been applied successfully to cryptography software and complex application format parsers like PDF and ELF.**

This paper describes the application of differential fuzzing to x86-64 instruction decoders for bug discovery. It introduces MISHEGOS, a novel differential fuzzer that discovers decoding discrepancies between instruction decoders. We describe MISHEGOS's architecture and approach to error discovery, as well as the security implications of decoding errors and discrepancies. We also describe a novel fuzzing strategy for instruction decoders on variable-length architectures based on an over-approximated model of machine instructions.

MISHEGOS produces hundreds of millions of decoder tests per hour on modest hardware. We have used MISHEGOS to discover hundreds of errors in popular x86-64 instruction decoders without relying on a hardware decoder for ground truth. MISHEGOS includes an extensible framework for analyzing the results of a fuzzing campaign, allowing users to discover errors in a single decoder or a variety of discrepancies between multiple decoders. We provide access to MISHEGOS's source code under a permissive license.

***Index Terms*—differential testing, software testing, automatic test generation**

I. Introduction

Instruction *decoding* is the most basic phase in *disassembly* and forms the bedrock of every decompilation and binary analysis workflow. Subsequent steps in these workflows rely on the fidelity of the instruction decoder to perform accurate function identification, code and data disambiguation, and control flow graph recovery. Instruction decoders are also pervasive in modern computing environments, with roles ranging from just-in-time compilers (JITs) in web browsers [1] and operating system kernels to the reverse engineer's workbench [2].

The role of an instruction decoder is singular and hermetic: it must decode *individual* machine code instructions into an abstract representation suitable for analysis or pretty-printing as assembly. Instruction decoders perform the software equivalent of *hardware decoders*, which perform the initial phase of *program interpretation* within physical CPUs. The latter are the sole source of *ground truth* about a machine code instruction's true semantics on a particular machine. Consequently, software decoder errors can result in discrepancies between sets of instructions accepted by the decoder and by real hardware. The instruction decoder may *over-accept* inputs that are not valid instructions, *under-accept* inputs that are valid but with unusual or uncommon encodings, or *mis-accept* inputs as valid but with incorrect semantics. Two separate instruction decoders may, furthermore, incorrectly disagree *or* agree on the validity *or* invalidity of a potential instruction.

Errors and discrepancies in and between individual instruction decoders (or instruction decoders and a particular hardware decoder) have historically been treated as minor problems, best left for manual resolution by the reverse engineer at their workbench. We argue that recent developments in programming practices challenge this treatment, and instead, encourage the treatment of decoding bugs as potential sources of vulnerabilities:

- Antivirus (AV) tools use binary analysis to detect malicious program behavior [3] and deny execution to malware, and are thus dependent on the accuracy of the underlying instruction decoder. Errors in the instruction decoder might result in false negatives, either through "fail-safe" behavior when the AV detects instructions that it cannot decode correctly, or through undetected misdecoding that obfuscates the presence of a malicious payload [4], [5].
- Software sandbox frameworks [6] need accurate instruction decoding to determine the safety and soundness of client code in the sandboxed environment. Differences between the sandbox's instruction decoder and the ground truth of the hardware decoder can result in sandbox escapes.
- JIT compilation is an established technique for improving the performance of interpreted and domain-specific languages (DSLs) in both kernel [7], [8] and user space [9]. JITs depend on decoder-supplied semantics to emit safe and correct machine code for untrusted inputs. Errors in these semantics might cause emission of exploitable machine code.
- Both static and dynamic binary translators, such as Rosetta [10] and McSema [11], rely on the correctness of the instruction decoder when translating machine code to a target architecture or intermediate representation. We speculate that errors in the translator's instruction decoder may allow an attacker to convert benign machine code into exploitable machine code on the target architecture.

[§]Proceedings author.
[†]Research contributor.

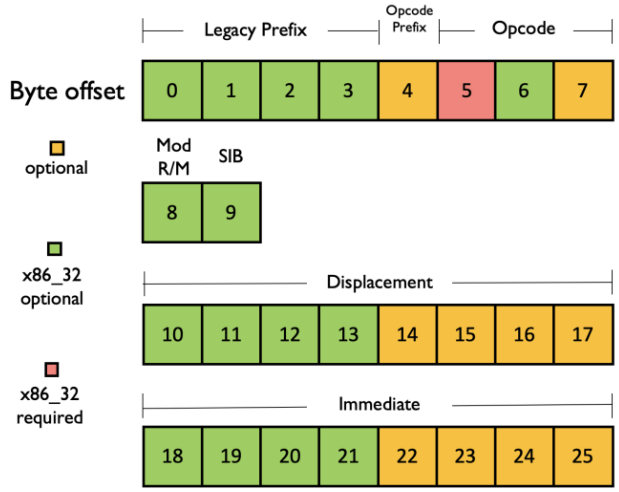

Fig. 1. An overapproximation of a "core" x86-64 instruction.

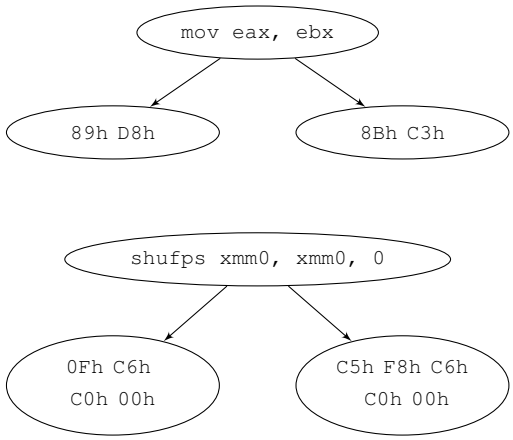

Fig. 2. Instruction semantics and their alternative encodings.

The challenges of accurate instruction decoding vary by architecture, and can be divided into two tasks: correctly *accepting* all hardware-accepted machine instructions and correctly *denying* all hardware-denied instructions. These two tasks are occasionally in conflict: a instruction decoder might exchange simplicity for *over-accepting* invalid instructions, or *under-accepting* valid instructions that a compiler is unlikely to emit.

Among popular instruction sets, the x86-64 instruction format is notoriously complex (Fig. 1) and difficult to decode correctly. Its challenges include:

- Variadic instruction length: a valid x86-64 instruction can range from 1 to 15 bytes. A decoder must *always* calculate the correct instruction length; failure to do so results in both an incorrect decoding for the current instruction *and* incorrect decodings for any subsequent instructions in the stream due to misalignment.
- Multiple internal instruction formats: x86-64 decoders must handle "core," x87, VEX/EVEX, and other formats for different components of the instruction set architecture (ISA). These internal formats increase the complexity of the decoder and thus the likelihood of misinterpretation.
- Host-relative validity: the validity and semantics of a potential instruction are predicated on the host CPU's vendor (Intel, AMD, Cyrix, VIA), feature availability (*e.g.*, AVX512 and CET), operating mode (real, protected, long), and virtualized context (AMD-V, VT-x, or not virtualized).
- Alternative encodings: the "core" x86-64 instructions include duplicate encodings for many register-to-register operations [12], and SIMD instructions can be encoded in either a "legacy" format or with VEX [13]. A decoder must accept all duplicate forms and must not assume that only one form occurs in a particular instruction stream. Fig. 2 shows some duplicate encodings for identical instruction semantics.

These complexities make x86-64 instruction decoders uniquely susceptible to implementation errors. Subsequent sections will discuss automated discovery of decoder errors, prior research efforts, and offer a rationale for the novel approach that we take in MISHEGOS.

II. Background and prior work

A. A taxonomy of decoding errors

We establish the following taxonomy for the purposes of describing errors in instruction decoders. Errors in instruction decoders fall into three categories: *over-supporting*, *under-supporting*, and *mis-supporting*. Examples of all three are given in Table I.

1) Over-supporting: An over-supporting error occurs in an instruction decoder when an *invalid* instruction candidate is *accepted*, resulting in a "successful" but incorrect decoding in a case where a hardware decoder would fail.

2) Under-supporting: An under-supporting error occurs in an instruction decoder when a *valid* instruction candidate is *rejected*, resulting in a unsuccessful decoding in a case where a hardware decoder would succeed.

3) Mis-supporting: A mis-supporting error occurs in an instruction decoder when a *valid* instruction candidate is *accepted* but incorrectly interpreted, resulting in a "successful" but incorrect decoding where a hardware decoder would succeed correctly.

B. Automatic testing of instruction decoders

Efforts to automate the testing of instruction encoders and decoders have historically fallen into two general categories: *exhaustive* search and *fuzz* strategies.

TABLE I
EXAMPLES OF DECODING ERRORS

Type of error	Decoder	Candidate	Expected	Actual
Over-support	DynamoRIO	`662e4d0f37`	INVALID	`getsec`
Under-support	Capstone	`f30f38de18`	`aesenc256kl xmm3, zmmword ptr [rax]`	INVALID
Mis-support	`libopcodes`	`66650f85df03e6d2`	`jnz 0xfffffffffd2e603e7`	`gs jne 0x3e5`

1) Exhaustive search: The exhaustive approach to instruction decoding is the simplest approach, and is based on the observation that most instruction formats are *finite languages* over $\Sigma = \{0, 1, \ldots, 255\}$ of three possible input spaces:

$$L = \Sigma^{N} \tag{1}$$

$$L = \Sigma^{\{N_1, N_2, \ldots, N_n\}} \tag{2}$$

$$L = \Sigma^{\{1 \ldots N\}} \tag{3}$$

where N is either the absolute (for fixed ISAs) or maximum (for variadic ISAs) length of a valid instruction. (2) represents a special case where the ISA is variadic, but only for a specific subset of values in $\{1, \ldots, N\}$. A concrete example of this is ARM with Thumb, where $N = 4$ and instructions can be 2 (Thumb) or 4 (ARM) bytes long but never 1 or 3.

Algorithm 1 shows an exhaustive linear search of L, using the hardware decoder as a source of ground truth. Paleari et al. [14] and Domas [15] describe suitable implementations of DECODE and DIFFER with a hardware decoder as ground truth for Algorithm 1.

Algorithm 1 Exhaustive search with hardware truth.

```
1:  procedure EXHAUSTIVE-SEARCH(L)
Require: L is the set of all possible instructions
2:      H ← HARDWARE-DECODER()
3:      S ← SOFTWARE-DECODER()
4:      for l ∈ L do    ▷ Parallel factor M for M host cores
5:          lh ← H.DECODE(l)
6:          ls ← S.DECODE(l)
7:          if DIFFER(lh, ls) then
8:              RECORD(lh, ls) ▷ Save discrepancy for triage
9:          end if
10:     end for
11: end procedure
```

Exhaustive search is feasible for small $|L|$ on modest hardware, is parallelizable across M host cores, and has been applied successfully to the verification of the ARM encoder-decoder used by LLVM [16], albeit with a vendor supplied "golden" decoder for ground truth rather than a hardware decoder.

Table II shows N and $|L|$ for common ISAs, as well as a time estimate (E) for a complete search using an optimistic target of 1 million tests/second. At E, it would take us approximately $2.54 * 10^{24}$ years to analyze the entire input space of a *single* x86-64 instruction decoder. As such, we currently consider exhaustive search infeasible for x86-64 instruction decoders. Another approach is necessary.

TABLE II
N, $|L|$, AND E FOR COMMON ISAS

ISA	Variadic?	N	$\|L\|$	E (days)
ARMv7 (no Thumb)	No	4	$4.29 * 10^{9}$	2.98
ARMv7 (Thumb)	Yes[a]	4	$4.29 * 10^{9}$	2.98
z/Architecture	Yes[b]	6	$2.81 * 10^{14}$	$1.95 * 10^{5}$
x86-64	Yes	15	$1.33 * 10^{36}$	$9.27 * 10^{26}$
IA-64	No	16[c]	$3.40 * 10^{38}$	$2.36 * 10^{29}$

[a] $\sum^{\{2,4\}}$
[b] $\sum^{\{2,4,6\}}$
[c] IA-64 bundles multiple instructions for decoding.

2) Fuzz strategies: Fuzzing [17] is a well-established technique for testing programs with input spaces that are either difficult to model or impossible to test exhaustively. A fuzzer (or *fuzz tester*) feeds generated inputs into a target program, relying on an *oracle* [18] to determine whether a particular input has caused an erroneous program state. Traditional oracles include abnormal program terminations ("crashes"), hangs, or unexpected memory accesses; triggering the oracle causes the fuzzer to mark the input as the source of a bug and retain it alongside other program state for subsequent minimization, deduplication, and analysis (Fig. 3).

Instruction decoders are typically devoid of traditional fuzzing oracles due to their constrained inputs and outputs. Consequently, instruction decoder fuzzers are typically differential fuzzers [19]. Differential fuzzers utilize a *difference oracle*: the output of the decoder is compared to a *reference* output; if they differ, then the input is considered the source of a potential bug and retained. The *reference* can be a static collection of known good results, a hardware decoder, or an entirely different instruction decoder. Adding multiple sources of reference diminishes the likelihood that all decoders incorrectly accept or reject an input, which in turn decreases the likelihood of missing erroneous inputs (Fig. 4).

Instruction decoder fuzzers fall into one of two categories: "random" or "structured". Each has strengths and weaknesses: a "random" fuzzer ignores the documented instruction format and is therefore well suited to discovery of undocumented or mis-documented instructions [15], but is unlikely to discover decoding bugs that arise from the internal complexities of the instruction format. A "structured" fuzzer uses the instruction format to guide input generation and is therefore more likely to discover discrepancies between the specification and the de-

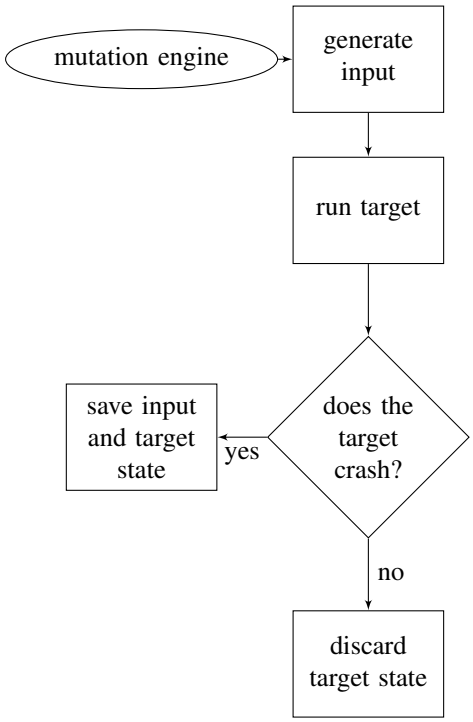

Fig. 3. A traditional fuzz lifecycle with a crash oracle.

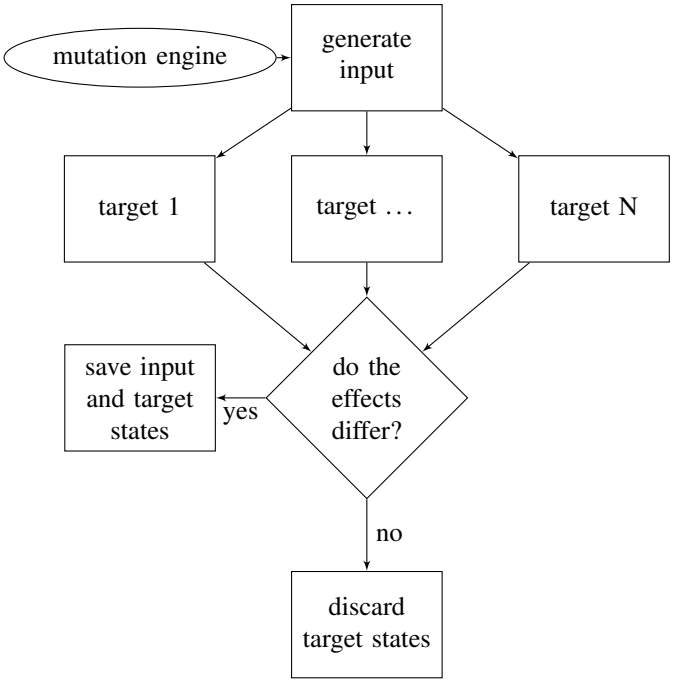

Fig. 4. A differential fuzz lifecycle with a difference oracle.

coder, but is unlikely to discover places where the instruction decoder deviates significantly from the specification.

Two special sub-techniques of "structured" instruction decoder fuzzing have been the subject of prior research. The first case is *CPU-assisted fuzzing* [14], which uses a hardware decoder to generate instructions that are known to decode correctly.

The second case is *inferred structural fuzzing* [20], wherein initially valid inputs are used to seed the fuzzer. The inputs are then mutated bit-by-bit and decoded, allowing the fuzzer to label individual bits as *field*, *reserved*, *unused*, or *structural*. These labelings are further refined to compensate for dependencies between bits, allowing for valid instruction inference without a hardware decoder.

Critically, both of these "structured" techniques focus on discovering *valid* instructions that instruction decoders report as *invalid*: they do not attempt to discover *invalid* instructions that decoders report as *valid*. We judge the latter case to be of equal research interest, particularly in light of the proliferation of binary translation and emulation techniques that are fundamentally dependent on the fidelity of their underlying decoders.

C. Differential fuzzing with multiple instruction decoders

Prior research efforts to fuzz instruction decoders have primarily focused on a singular difference oracle: whether the targeted instruction decoder differs from a single source of *ground truth*. Under a singular difference oracle, the source of ground truth is either a "golden" instruction decoder presumed to be reliable [16], or a hardware decoder.

We judge the use of a "golden" decoder to be problematic for x86-64: Intel's XED, the closest thing to a reference decoder, has a history of implementation errors resulting in erroneous decodings. More generally, we regard difference against a "golden" decoder to be less interesting (from the attacker's perspective) than difference against multiple decoders of unknown quality embedded throughout the software lifecycle.

Prior work on fuzzing multiple x86-64 instruction decoders with *multiple* difference oracles is limited. An important prior multiple-oracle approach is described in Paleari, *et al.* [14], where it is termed *N-version disassembly*. Despite their use of multiple decoders, Paleari, *et al.*, include a hardware decoder (the N^{th} in "N-version") as the ultimate source of ground truth in their approach. Consequently, the *N-version* technique struggles to distinguish *over-supported* instructions in individual decoders from instructions that are not supported by a *particular* hardware decoder.

Another important approach is described in Jay & Miller and implemented in FLEECE [20]. In FLEECE, multiple decoders are fed inputs generated by an *inferred structure fuzzer*; each decoder's result (in the form of pretty-printed assembly) is subsequently *reassembled* and checked for discrepancies. In practice, variations in pretty-printing between different instruction decoders means that this approach requires FLEECE to perform significant normalization of each decoder's output

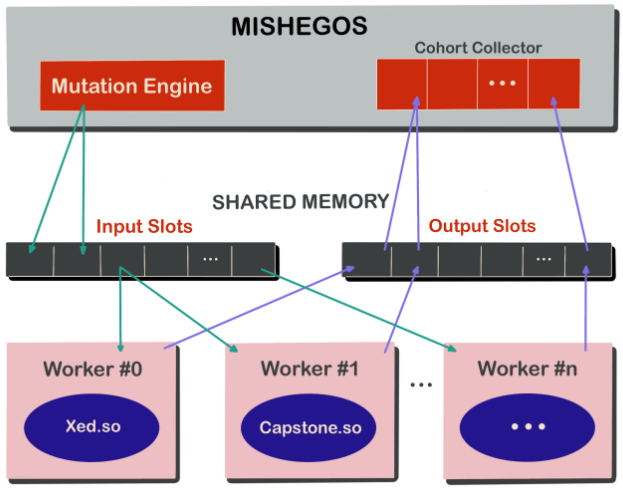

Fig. 5. A bird's eye view of MISHEGOS's architecture.

before attempting reassembly, introducing overhead to the process of fuzzing additional decoders. FLEECE also relies on the GNU Assembler [21] for reassembly, which uses one of its decoder targets (`libopcodes`) internally.

III. MISHEGOS

We describe and justify MISHEGOS's architecture, input mutation strategies, and analysis framework below. Fig. 5 depicts the high level architecture.

A. Overview

MISHEGOS is a differential fuzzer for x86-64 instruction decoders. Instead of relying on a "golden" decoder or hardware decoder for ground truth within its difference oracle, MISHEGOS runs multiple *unreliable* decoders in parallel on an instruction candidate and uses their consensus or disagreement to mark the input as likely erroneous.

As of writing, nine x86-64 instruction decoders written in a variety of languages can be fuzzed by MISHEGOS:

- GNU `libopcodes` (C) [21]
- Capstone (C++) [22]
- DynamoRIO (C and C++) [2]
- fadec (C) [23]
- Udis86 (C) [24]
- Intel XED (C) [25]
- Zydis (C) [26]
- bddisasm (C) [27]
- Iced (Rust) [28]

MISHEGOS compensates for differences in performance between each instruction decoder by implementing an asynchronous input and output *slot* system: no decoder running under MISHEGOS ever waits for its peers to finish decoding the same instruction candidate. Each decoder's results are emitted asynchronously and are grouped into *cohorts* as all outputs for an input become available, with cohorts representing the most basic unit within MISHEGOS's subsequent analyses. The details of the slot and cohort mechanisms are hidden from each decoder by the *worker abstraction*.

We discuss each component in detail below.

B. Mutation strategies

MISHEGOS addresses the problems identified above in the "random" and "structured" strategies by implementing a novel mutation strategy that we term the *sliding strategy*.

The sliding strategy is built around the observation in Fig 1: no valid x86-64 instruction ever exceeds 15 bytes, but the the x86-64 machine code structure allows us to construct an *overapproximation* of a valid instruction, up to 26 bytes in length. We term this overapproximated instruction the *maximal instruction candidate*.

To generate the maximal instruction candidate, we combine the "random" and "structured" strategies: the "structured" strategy is used to produce plausible values for the prefixes (both legacy and REX) and core opcode semantics of the maximal candidate, while the "random" strategy is used to produce random values for the ModR/M, SIB, displacement, and immediate fields of the candidate. We split the fields between the internal strategies by observing their internal structure and search spaces: the prefixes and opcodes are highly structured, while ModR/M and SIB are both single-byte lookup tables and the immediate and displacement are constant literals subject to minimal interpretation by the instruction decoder. Of note is our decision to not always produce a maximal candidate of *exactly* 26 bytes: our strategy may produce shorter candidates as part of selecting shorter individual fields, but the resulting candidate will *always* be an overapproximation. Algorithm 2 shows the process for generating the maximal candidate; the result is a *maximal candidate*. The STRUCTURED and RANDOM procedures for each field are elided for brevity and can be found in MISHEGOS's `mutator.c`.

Algorithm 2 Maximal instruction candidate generation.

```
1:  procedure MAXIMAL-CANDIDATE
2:      C ← EMPTY-CANDIDATE()
3:      C.PREFIXES ← STRUCTURED-PREFIXES()
4:      C.REX ← STRUCTURED-REX-PREFIX()
5:      C.OPCODE ← STRUCTURED-OPCODE()
6:      C.MODRM ← RANDOM-MODRM()
7:      C.SIB ← RANDOM-SIB()
8:      C.IMMEDIATE ← RANDOM-IMMEDIATE()
9:      C.DISPLACEMENT ← RANDOM-DISPLACEMENT()
10:     return C
11: end procedure
```

Once a maximal instruction candidate has been generated, we perform our sliding strategy to convert it into a series of inputs suitable for feeding into each of MISHEGOS's decoders. Each input derived from the maximal candidate is termed a *sliding candidate*.

To generate our sliding candidates, we start at the first byte (index 0) of our maximal candidate and extract a slice of

N bytes (*i.e.*, 15 bytes on x86-64) for our sliding candidate, where N is our absolute or maximum ISA instruction length. We then iterate moving rightwards, extracting further "windows" until we've exhausted our maximal candidate's ability to produce sliding candidates of length N. Algorithm 3 shows the process of generating sliding candidates from a maximal candidate, and Fig. 6 visualizes the results of windowing.

Algorithm 3 Sliding instruction candidate generation.

procedure SLIDING-CANDIDATES(C, N)
Require: C is a candidate from MAXIMAL-CANDIDATE()
Require: N is the ISA's maximum instruction length
    $W \leftarrow \emptyset$
    OFFSET $\leftarrow 0$
    while OFFSET $+ N < C$.LENGTH() **do**
        $I \leftarrow C$.SLICE(OFFSET, N)
        $W \leftarrow W \cup \{I\}$
        OFFSET $\leftarrow$ OFFSET $+ 1$
    end while
    return W
end procedure

This strategy of overapproximation and *sliding* produces sequences of fuzzing inputs with properties from both the "random" and "structured" strategies: the first sliding candidates are generated according to the rough expected structure of x86-64 instructions, while later sliding candidates become increasingly random. Mishegos also implements the "random" and "structured" strategies for the purposes of contrast; we analyze all three in the results below.

C. Worker architecture and ABI

MISHEGOS supports a broad range of instruction decoders, written in a variety of languages and with a variety of resource management patterns.

To ease the integration of diverse instruction decoder implementations, MISHEGOS provides the *worker abstraction*. Each *worker* runs in its own process and handles input and output slot management for the underlying decoder, allowing each decoder to focus solely on the task of decoding.

Workers load their underlying decoders as shared objects, communicating with them through a minimal C ABI (Listing 1). Within the ABI, only `worker_name` and `try_decode` are required—workers may choose not to implement `worker_ctor` and `worker_dtor` if they do not require special initialization before attempting decoding.

The C ABI ensures compatibility with virtually all languages and runtime environments and reduces interactions between each decoder and MISHEGOS to just four touchpoints.

The Capstone worker (`capstone.c`) illustrates the simplicity of the decoder integration into MISHEGOS (Listing 2).

D. Automated analyses

A MISHEGOS fuzzing campaign produces volumes of *cohorts*. Each cohort has K outputs for K decoders enabled

Listing 1
THE MISHEGOS WORKER ABI

```
/* a unique identifier for the worker,
 * like "capstone"
 */
char *worker_name;

/* an optional constructor */
void worker_ctor();

/* try to decode one instruction,
 * storing the results in result.
 */
void try_decode(decode_result *result,
                uint8_t *raw_insn,
                uint8_t length);

/* an optional destructor */
void worker_dtor();
```

Listing 2
THE CAPSTONE WORKER

```
#include <capstone/capstone.h>

#include "../worker.h"

static csh cs_hnd;

char *worker_name = "capstone";

void worker_ctor() {
  if (cs_open(CS_ARCH_X86, CS_MODE_64, &cs_hnd)
      != CS_ERR_OK) {
    errx(1, "cs_open");
  }
}

void worker_dtor() {
  cs_close(&cs_hnd);
}

void try_decode(decode_result *result,
    uint8_t *raw_insn, uint8_t length) {
  cs_insn *insn;
  size_t count = cs_disasm(cs_hnd,
                           raw_insn,
                           length, 0, 1,
                           &insn);
  if (count > 0) {
    result->status = S_SUCCESS;
    result->len =
        snprintf(result->result,
                 MISHEGOS_DEC_MAXLEN,
                 "%s %s\n",
                 insn[0].mnemonic,
                 insn[0].op_str);
    result->ndecoded = insn[0].size;

    cs_free(insn, count);
  } else {
    result->status = S_FAILURE;
  }
}
```

```
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
f0 f3 67 64 46 0f 3a 2d 39 2f f3 65 26 f2 5a c9 bd ea fd 7f 4c c4 0f df c7 fa
```

Fig. 6. Results of sliding candidate generation. The colors, in order, represent the legacy prefixes (blue), REX prefix (red), opcode bytes (green), ModR/M (orange), SIB (brown), displacement (magenta), and immediate fields (purple); the bubbles represent each sliding candidate generated from the overarching maximal candidate.

Listing 3
AN ANALYSIS SPECIFICATION

```
size-discrepancies:
  - dedupe
  - filter-all-failure
  - filter-ndecoded-same
  - filter-incomparable
  - find-size-discrepancies
  - minimize-input
  - normalize
```

during the campaign. Every cohort is associated with its instruction candidate.

The design of the MISHEGOS analysis framework mirrors the design of the LLVM pass planner: *analyses* are described as a sequence of *passes*, with individual passes being capable of expressing their dependency on other passes. The analysis framework loads a user-specified analysis, verifies that its internal passes form a directed acyclic graph, and runs the passes in a pipeline to completion. Listing 3 shows a specification for a single analysis, composed of multiple internal passes.

Individual passes within an analysis have two fundamental actions available to them: they can either *filter* cohorts from the results of a fuzzing campaign, or they can *modify* cohorts for the purposes of normalization or augmentation with additional metadata. We supply examples of each in Listing 4 and Listing 5.

We compensate for the lack of a hardware or "golden" instruction decoder by supplying analyses that target specific decoders, using majority consensus among their peers as our oracle for probable decoding errors. We also supply "generic" analyses that target all decoders enabled during a campaign. All analyses, whether generic or targeted, discover discrepancies consistent with our taxonomy in §II-A:

1) Over-supporting and under-supporting: We define a discrepancy between decoders to be the likely result of *over-supporting* in a particular instruction decoder if a majority of its peers agree that the instruction is *invalid*. Similarly, a discrepancy between decoders is said to be the likely result of *under-supporting* in a particular instruction decoder if a majority of its peers agree that the instruction is *valid*.

Listing 4
A FILTERING ANALYSIS PASS

```
warn "[+] pass: filter-any-failure"

count = 0
$stdin.each_line do |line|
  result = JSON.parse line, symbolize_names: true

  if result[:outputs].any? { |o| o[:status][:name]
      == "failure" }
    count += 1
    next
  end

  $stdout.puts result.to_json
end

warn "[+] pass: filter-any-failure done: #{count}
    filtered"
```

Listing 5
A MODIFYING ANALYSIS PASS

```
warn "[+] pass: minimize-input"

$stdin.each_line do |line|
  result = JSON.parse line, symbolize_names: true

  max_ndecoded = result[:outputs].map { |o| o[:
      ndecoded] }.max

  result[:input] = \
    result[:input][0, max_ndecoded * 2]

  $stdout.puts result.to_json
end

warn "[+] pass: minimize-input done"
```

MISHEGOS supplies targeted analyses both over- and under-supporting errors in specific decoders; `xed-underaccept`, for example, discovers instructions that XED claims are invalid but are marked as valid by a consensus of other "high-quality" decoders, while `xed-overaccept` finds the inverse.

2) Mis-supporting: We define a discrepancy between decoders to be a likely result of *mis-supporting* in a particular instruction decoder when a majority of its peers agree that the instruction's length or semantics differ *consistently* (*i.e.*, all peers agree that it should be a different result, and they agree on the different result). MISHEGOS supplies two generic analyses for detecting mis-supported instructions: `size-discrepancies` and `status-discrepancies`, as well as several targeted analysis for individual decoders.

IV. EVALUATION

To evaluate MISHEGOS, we conducted a fuzzing campaign with 7 of the 9 supported decoders: `libopcodes`, Capstone, DynamoRIO, XED, Zydis, bddisasm, and Iced. The decoders were selected based on their popularity in debugging tools (`libopcodes`), binary analysis (Capstone, DynamoRIO), hardware vendor support (XED), and reputation for quality and reliability (Zydis, bddisasm, Iced). The remaining two decoders were excluded because of their lack of maintenance (Udis86) and relative obscurity (fadec). Table III shows version information for each worker's decoder.

TABLE III
DECODER VERSIONS FOR EVALUATION

Worker	Version
`libopcodes`	Release 2.30
Capstone	Commit `8984920`
DynamoRIO	Commit `568d7a6`
XED	Commit `93c0b83`
Zydis	Commit `ba9431c`
bddisasm	Commit `a0b3eee`
Iced	Commit `0a3f062`

Our fuzzing campaign was conducted over a period of 4 hours on 8 dedicated physical cores from an Intel Xeon Gold 6140, producing 130,574,092 cohorts (39,715,068 after deduplication). Each cohort contained 7 results (one for each worker's decoder), meaning that MISHEGOS processed 228 million decoder outputs per hour of fuzzing.

To evaluate the analysis framework, we selected four individual analyses to run on the results of the fuzzing campaign: `status-discrepancies`, `size-discrepancies`, `xed-overaccept`, and `xed-underaccept`. These analyses were selected for their volumes at both ends of the spectrum (high for `status-discrepancies` and `size-discrepancies`; low for `xed-overaccept` and `xed-underaccept`). The raw cohort counts from each analysis are shown in Table IV.

TABLE IV
EVALUATION ANALYSES

Analysis	Results (# cohorts)
`status-discrepancies`	15,359,832
`size-discrepancies`	3,423,953
`xed-overaccept`	58,808
`xed-underaccept`	100

1) Status discrepancies: The `status-discrepancies` analysis discovered roughly 15 million cohorts where at least one decoder disagreed with its peers on the validity of an instruction candidate. Examples of status discrepancies discovered by this analysis are shown in Table V.

To determine the total number of status discrepancies, we ran a count over each cohort: each decoder in the cohort's status was tested against a majority consensus of its peers, with a consensus failure increasing the decoder's discrepancy count. The result was 17,503,244 unique discrepancies over the original 15,359,832 cohorts, or 1.14 results per cohort.

Of the status discrepancies observed, the majority (57%) occurred between `libopcodes` and the majority consensus. Capstone and DynamoRIO were responsible for significant minorities (22% and 19%, respectively) of the discrepancies, and the remaining 2% of discrepancies were observed across bddisasm (137,316), XED (48,933), Zydis (12,365), and Iced (48,942).

2) Size discrepancies: The `size-discrepancies` analysis discovered roughly 3.4 million cohorts where at least one decoder disagreed with its peers on the size of an instruction candidate. Examples of size discrepancies discovered by this analysis are shown in Table VI.

To determine the total number of size discrepancies, we ran the same count in §IV-1 but with each result's decoded length for the consensus oracle rather than status. The result was 6,108,755 discrepancies over the original 3,423,953 cohorts, or 1.78 discrepancies per cohort.

Of the size discrepancies observed, the majority (54%) again occurred between `libopcodes` and the majority consensus. A significant minority (41%) occurred between DynamoRIO and the consensus, and the remaining 5% of discrepancies were observed across Capstone (328,527), bddisasm (3,461), XED (1), and Iced (1). No size discrepancies were discovered for Zydis. Upon manual analysis, the bddisasm, XED, and Iced size discrepancies were all determined to be false positives caused by over-supporting of nonexistent instructions in `libopcodes`.

3) Over- and under-support in XED: Our last two analyses focused on Intel's XED, typically considered the closest thing to a publicly available reference decoder for x86-64. Although `xed-overaccept` discovered nearly 60,000 discrepancies between XED and a consensus of other "high-quality" decoders (bddisasm, Zydis, and Iced), none of them were errors in XED: all belonged to multi-byte NOP ranges than only XED and Iced were able to consistently decode.

`xed-underaccept`, by contrast, discovered 100 discrepancies that, through manual review, were reduced to 3 mishandled instructions: `XSTORE` (VIA), `MONTMUL` (VIA), and `SPFLT` (Intel L10M/KNC). XED's maintainers have acknowledged all three discrepancies.

V. CONCLUSIONS

MISHEGOS's novel *sliding* strategy and analysis framework successfully discover a large number and variety of errors in individual instruction decoders, as well as potentially exploitable discrepancies between decoders during relatively

TABLE V
STATUS DISCREPANCIES

Decoder	Input	Consensus	Result
libopcodes	36f2f02ecb	INVALID	ss repnz lock cs retf
libopcodes	666565f20fbdb633c22064	bsr si, gs:[rsi+0x6420C233]	INVALID
Capstone	f0f22e770f	INVALID	bnd ja 0x14
Capstone	f264f30f0d02	prefetch byte ptr fs:[rdx]	INVALID
DynamoRIO	656665f30f382446d0	INVALID	pmovsxwq xmm0, oword ptr [gs:rsi-0x30]
DynamoRIO	f2490f38f84754	enqcmd rax, zmmword ptr [r15+0x54]	INVALID
XED	f267470fa7c0	INVALID[a]	repnz xstore
XED	f3440fa6c0	rep montmul	INVALID
Zydis	642e2662d168b05c38	INVALID	vsubps zmm7, zmm18, fs:[r8] {float16} {cdab}
Zydis	f366400f38dc6035	aesenc128kl xmm4, ptr [rax+0x35]	INVALID
bddisasm	673626660f01cc	INVALID[b]	tdcall
bddisasm	66f336650fae6516	ptwrite dword ptr gs:[rbp+0x16]	INVALID
Iced	f326430f01c2	INVALID	vmlaunch[c]
Iced	3ef36466450fa6e8	rep ccs_hash	INVALID

[a,c] Split consensus.
[b] Erroneous consensus.

brief fuzzing campaigns. We believe that these properties make MISHEGOS a strong foundation for future research into instruction decoder fuzzing.

MISHEGOS is available on GitHub under a permissive (Apache-2.0) license: https://github.com/trailofbits/mishegos.

A. Future work and research

We have identified areas of particular interest, both for MISHEGOS and instruction decoder fuzzing more generally.

1) Mutation refinements: MISHEGOS's *sliding* strategy is effective at producing large volumes of decoding errors and discrepancies, but also produces large volumes of *nearly identical* results that vary only in operands and immediates. These nearly identical results are difficult to automatically deduplicate, meaning that human analysis still forms an essential part of the MISHEGOS workflow.

We posit that the quality of the *sliding* strategy's inputs could be further improved by a partial adaptation of Jay & Miller's *inferred structural fuzzing* technique [20].

2) Automated regression detection: MISHEGOS currently evaluates *distinct* instruction decoders for discrepancies, but could just as easily evaluate *different versions* of the same decoder. We posit that work in this direction, combined with a guided novel mutation strategy, could enable MISHEGOS to serve in an assurance capacity for individual instruction decoders.

3) Generation of schizophrenic binaries: Previous research [15] has shown that errors in instruction decoders can be used to craft real binaries that malicious behavior on real hardware but benign behavior when emulated. These binaries are a form of so-called *format schizophrenia* [29], and are more challenging to mitigate than traditional emulator detection in a malicious program.

We posit the feasibility of semi-automatic generation of schizophrenic binaries, with MISHEGOS as the underlying "gadget" discovery component. We further posit the discovery of schizophrenic gadgets that have a *multiplicity* of behaviors across multiple instruction decoders.

ACKNOWLEDGMENTS

The authors would like to thank Trent Brunson and Evan Sultanik (both Trail of Bits) for their guidance and review.

REFERENCES

[1] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, "The devil is in the constants: Bypassing defenses in browser JIT engines." in *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[2] D. L. Bruening and S. Amarasinghe, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, USA, 2004.

[3] M. Novitchi, "Anti-malware emulation systems and methods," U.S. Patent 8 151 352B1, 2006.

[4] C. Jämthagen, P. Lantz, and M. Hell, "A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries," in *2013 Workshop on Anti-malware Testing Research*. IEEE, 2013, pp. 1–9.

[5] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

TABLE VI
SIZE DISCREPANCIES

Decoder	Input	Consensus (length)	Result (length)
libopcodes	26f3660f8ff801f910	jnle 0x10f90201 (9)	es repz jg 0x00000000000001ff (7)
libopcodes	474ac3	ret (3)	ret.RXB (1)
Capstone	66e80f38ee9c	call 0xffffffff9cee3815 (6)	call 0x3813 (4)
Capstone	6665653e4fc29322ca95	ret 0x2293 (8)	ret 0x2293 (10)
DynamoRIO	0fb9accfe498d5b8	ud1 ebp, [rdi+rcx*8-0x472A671C] (8)	ud2b (2)
DynamoRIO	f3f26664460f78fb5126	insertq xmm15, xmm3, 0x51, 0x26 (10)	vmread ebx, r15d (8)

[6] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93.

[7] J. Corbet. (2011) A JIT for packet filters. [Online]. Available: https://lwn.net/Articles/437981/

[8] S. Scannell. (2020) Fuzzing for eBPF JIT bugs in the Linux kernel. [Online]. Available: https://scannell.me/fuzzing-for-ebpf-jit-bugs-in-the-linux-kernel/

[9] The v8 Authors, "v8," https://github.com/v8/v8, 2021.

[10] Apple Inc., "About the Rosetta Translation Environment," 2021. [Online]. Available: https://developer.apple.com/documentation/apple_silicon/about_the_rosetta_translation_environment

[11] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instructions to LLVM," in *Proceedings of the ReCon Conference*, 2014.

[12] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in *Information and Communications Security*, J. Lopez, S. Qing, and E. Okamoto, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 187–199.

[13] A. Fog, "Optimizing subroutines in assembly language," 2020. [Online]. Available: https://www.agner.org/optimize/optimizing_assembly.pdf

[14] R. Paleari, L. Martignoni, G. Fresi Roglia, and D. Bruschi, "N-Version disassembly: Differential testing of x86 disassemblers," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 265–274.

[15] C. Domas, "Breaking the x86 ISA," *Black Hat*, 2017.

[16] R. Barton, "Guaranteeing the Correctness of MC for ARM," 2012. [Online]. Available: https://llvm.org/devmtg/2012-04-12/Slides/Richard_Barton.pdf

[28] 0xd4d, "Iced," 2021. [Online]. Available: https://github.com/0xd4d/iced

[17] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[18] R. McNally, K. Yiu, D. Grove, and D. Gerhardy, "Fuzzing: the state of the art," DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), Tech. Rep., 2012.

[19] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[20] N. Jay and B. P. Miller, "Structured random differential testing of instruction decoders," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 84–94.

[21] The GNU Project, "GNU Binutils," 2021. [Online]. Available: https://www.gnu.org/software/binutils/

[22] N. A. Quynh. (2021) Capstone. [Online]. Available: http://www.capstone-engine.org/

[23] A. Engelke, "Fadec—Fast Decoder for x86-32 and x86-64 and Encoder for x86-64," 2021. [Online]. Available: https://github.com/aengelke/fadec

[24] V. Thampi, "Udis86," 2021. [Online]. Available: https://github.com/vmt/udis86

[25] Intel Corporation. (2021) The X86 Encoder Decoder. [Online]. Available: https://intelxed.github.io/

[26] Zyantific, Inc., "Zydis," 2021. [Online]. Available: https://github.com/zyantific/zydis

[27] Bitdefender, Inc., "bddisasm," 2021. [Online]. Available: https://github.com/bitdefender/bddisasm

[29] A. Albertini, "Abusing file formats; or, Corkami, the novella," *The International Journal of Proof of Concept or GTFO*, no. 0x07, March 2015.