Structured Random Differential Testing of Instruction Decoders

Nathan Jay
Computer Sciences Department
University of Wisconsin
Madison, WI 53706 USA
nholcomb@wisc.edu

Barton P. Miller
Computer Sciences Department
University of Wisconsin
Madison, WI 53706 USA
bart@cs.wisc.edu

***Abstract*—Decoding binary executable files is a critical facility for software analysis, including debugging, performance monitoring, malware detection, cyber forensics, and sandboxing, among other techniques. As a foundational capability, binary decoding must be consistently correct for the techniques that rely on it to be viable. Unfortunately, modern instruction sets are huge and the encodings are complex, so as a result, modern binary decoders are buggy. In this paper, we present a testing methodology that automatically infers structural information for an instruction set and uses the inferred structure to efficiently generate structured-random test cases independent of the instruction set being tested. Our testing methodology includes automatic output verification using differential analysis and reassembly to generate error reports. This testing methodology requires little instruction-set-specific knowledge, allowing rapid testing of decoders for new architectures and extensions to existing ones. We have implemented our testing procedure in a tool name Fleece and used it to test multiple binary decoders (Intel XED, libopcodes, LLVM, Dyninst and Capstone) on multiple architectures (x86, ARM and PowerPC). Our testing efficiently covered thousands of instruction format variations for each instruction set and uncovered decoding bugs in every decoder we tested.**

I. Introduction

Instruction decoding is the process taking the binary representation of a machine instruction and decomposing it into its basic fields. As part of this decomposition, a decoder needs to identify the opcode, the operands and their types, qualifiers to the opcode and operands, and any address calculations that might be used for the operands. While instruction decoding is essentially a syntactic operation, it provides information necessary to support control- and data-flow of the code, and to understand instruction semantics.

Instruction decoding is the first step for any tool that operates on binary code, so represents a critical feature needed for these tools. Tools that operate on binary code include disassemblers [2], [4], [12], [14], [21], reverse engineering tools [5], [9], [16], binary rewriters [4], [18], dynamic instrumentors [4], [13], [22], performance profiling tools [1], [4], and security analysis tools [7], [20].

Accurate instruction decoding of an instruction is essential to the correct operation of the tools that are built on top of the decoder. Incorrect decoding can cause these tools to be useless or even dangerous. For example, in the presence of decoding errors, control flow can be misleading, performance bottlenecks can be obscured, and sandboxing untrusted code can be faulty. Consider the output of Intel's decoder for the x86 architecture, XED [12] (version 6.26.0) for the bytes `CA 48 0C`. These bytes should correspond to instruction `lretl 0xc48` (a function return instruction), but XED produces `lcallq 0xc48` (a function call instruction). These outputs have different implications for control flow, and this error will result in an incorrect control flow graph. Thorough testing of instruction decoders is essential to understanding binary code.

Accuracy of an instruction decoder is difficult to achieve due to the enormous complexity of modern instruction sets. Although manuals detailing early x86 processors are as short as 200 pages, modern manuals for x86 are thousands of pages long [11]. Even manuals for RISC ISAs like PowerPC and ARM exceed a thousand pages [3], [10]. These pages detail hundreds of opcodes, many addressing modes, and a myriad of restrictions on instruction fields or combinations of fields. Just determining which bits contribute to the opcode of an instruction is challenging because many bits not explicitly included in the opcode of an instruction can still change it, or even invalidate the instruction. Many PowerPC and ARM instructions have reserved fields whose bits are specific to the opcode. Changing these fields invalidates the instruction or alters the opcode. The operation of x86 instructions depend not only on the 1-3 byte opcode, but also on prefixes and even fields within prefixes, as with EVEX instructions. Our automated testing framework is able to detect which subsets of the bits in an instruction encode the operation of an instruction; we found that hundreds of different sets encode the operation of PowerPC and ARM instructions, and thousands of sets of bits encode the operation of x86 instructions.

The complexity of modern instruction sets has resulted in numerous errors in the decoding tools for x86, PowerPC and ARM, highlighting the need for comprehensive testing. Unfortunately, comprehensive testing of instruction decoders is challenging for two reasons. First, test inputs must be selected from the input space that maximize the number of errors found while minimizing the number of inputs that must be tested. ARMv8 and PowerPC both use 4-byte, fixed-length instructions, so their decoders must correctly decode more than 4 billion inputs. Decoders for x86 need to decode variable-length instructions up to 15 bytes long, resulting in

2^{120} possible inputs. The number of possible inputs makes a brute force approach to comprehensive testing impractical and requires careful consideration of test instructions. Second, the assembly language output of each decoder needs to be verified for every input. Given the size of the input space, this verification requires an automated method with ground-truth decoding for all input bytes.

Existing work has recognized the complexity of modern ISAs and challenges in testing instruction decoders, seeking to quantify issues with instruction decoder accuracy through differential testing. In 2010, Paleari et al [17] compared the outputs of eight x86 decoders when used to decode a mixture of random and known-valid inputs. Paleari's work introduced random differential testing for instruction decoders; however, it was specific to x86 and leveraged detailed knowledge of the ISA to generate known-valid test cases. Additionally, this work required extensive normalization functions to compare the output of different decoders. While the work by Paleari used mostly random inputs, results from other testing domains have shown that information about input structure can be used to generate inputs that improve code coverage [8], [23].

In this paper, we introduce a framework that provides several benefits over existing tests:

- *Inferred Input Structure:* We automatically infer encoding information using only knowledge of the register sets but not a model of the ISA, significantly reducing the expert knowledge required to test decoders.
- *Error Verification:* Our framework uses reassembly to verify that differences in decoder output correspond to errors in decoding. This process vastly reduces the effort required to analyze output and identify underlying issues in the decoders we test.
- *ISA Agnostic Testing:* Our framework makes few assumptions about the structure of instructions, allowing us to test decoders for x86, PowerPC and ARM (and others in the future) within a single framework. Other testing methods make assumptions that restrict them to a single ISA, or even a specific version of an ISA.

We applied our testing methodology and tools to a wide variety of decoders, including Capstone [2], Dyninst [4], libopcodes [21], LLVM [14], and XED [12], on a variety of architectures including x86 (32 and 64 bit), PowerPC (32 and 64 bit), and ARM (64 bit). Our experience has shown these tools (including those from the processor manufacturer and those from our own project) to be disturbingly buggy. Across all decoders, testing encountered dozens of errors that varied significantly including segmentation faults, incorrect opcodes, incorrect number of operands, invalid operand sizes, invalid use of restricted registers and missing opcodes. We reported all of the decoding errors we identified to the developers of these tools and many have been confirmed and fixed.

In Section 2, we provide background on existing instruction decoder testing. Section 3 describes the complexity of modern ISAs, Section 4 presents the testing procedure, Section 5 presents an evaluation of Fleece, our tool that implements our structured random testing procedure, testing several popular instruction decoders. We conclude in Section 6.

II. Related Work

Several existing testing techniques provide a foundation for our own work, including structured random input generation for fuzz testing [8], [19] and differential testing of instruction decoders [17].

While fuzz testing has been effective at finding bugs using unstructured, pseudorandom input [15], using structured input has been used to provide greater coverage in a short period of time for certain tools [8].

Model-based input generation has been applied to x86 instructions because instructions contain multiple distinct components that can be used in combination, including prefixes, opcodes, and modifier bits. This case was explored by Seidel [19], who models instructions as graphs, with each node representing a byte of the instruction. In their technique, instructions start as an empty string of bytes and new bytes are appended as the graph is traversed with edges leading from the current byte of the instruction to each valid following byte, according to a configurable probability. This approach can generate valid instructions with a specified distribution of instruction set features like prefixes and modifiers. However, this approach requires extensive knowledge of the instruction set, and is only applicable to ISAs where instruction set features correspond to contiguous bits (or bytes) that can be concatenated to create a valid instruction. In many RISC architectures like Arm and PowerPC, instructions are all 4-byte words whose individual bytes do not specify independent parts of the instruction, and whose opcodes and operands are not all contiguous.

Paleari et al introduced CPU execution as a way to generate valid inputs for instruction decoders [17]. Their approach, again specific to x86, takes advantage of the structure of instructions to test all possible 1-3 byte opcodes. The authors executed one instruction for each possible value of the first three bytes to obtain a list of valid 1-3 byte opcodes. Then, using a list of valid prefixes, they created test instructions by prepending prefixes to each of the valid opcodes. This method produced valid instructions with a variety of opcodes to provide better coverage of the tested instruction set than a purely random approach. Unfortunately, this approach is specific to x86, and it assumes that the list of valid prefixes is straightforward for a programmer to enumerate. New EVEX instructions in x86 violate this assumption with complex 3-byte prefixes that contain information about registers and instruction operation. This method requires the tool to explicitly define features of the instruction set (in the form of valid prefixes) and relies on 1-3 byte opcodes whose validity is independent of any prefixes.

Unlike these two methods of that require knowledge of the instruction set to create test cases, grammar-based input generation uses symbolic execution and a pre-defined grammar to construct valid inputs [8]. This approach increases code coverage when compared to purely random approach, but also

requires knowledge of the structure of inputs (in the form of a grammar) to generate test cases.

In addition to detecting failures, our work also seeks to verify the test results. Unfortunately, the specification of instruction decoding requires lengthy manuals with thousands of pages [3], [10], [11]. Verifying even a single output by hand requires lookups in several tables detailing the meaning of different bits, and sometimes referencing multiple manuals, a process that can introduce errors into the testing process. Differential testing provides a way to detect likely errors by comparing the output of several instruction decoders and only producing error reports where the outputs differ. This technique was used by Paleari et al to quantify the number of errors in existing instruction decoders [17]. In their work, the authors use eight x86 decoders in conjunction with execution on a CPU to identify cases where the decoders differ, signifying a probable error.

Ideally, this differential testing would yield no false positives: every difference in the output of decoders would correspond to an error in at least one of the decoders. Unfortunately, instruction decoders do not produce a single, uniform output. Even when configured to produce the same syntax, outputs from different instruction decoders can vary significantly. Paleari et al used nearly 100 functions to normalize the outputs of all decoders into one standard representation. Still, they found that some instructions marked as errors actually corresponded to trivial formatting differences between instruction decoders, rather than an error. Our approach to normalization uses an assembler to detect equivalent decodings, significantly reducing the number of normalization functions.

The DERIVE system [6] introduced the use of assemblers combined with a list of opcodes, registers and special instruction format strings as a way to derive the encoding of instructions. DERIVE's instruction format strings specified which field in assembly language is the opcode, which fields are registers, and which fields are immediates. The DERIVE system then created assembly language instructions with permutations of opcodes, fields and immediates, assembling each instruction. Using a series of constraint solvers, the system determined which bits of the instruction correspond to which fields, as well as appropriate decoding procedure. Unfortunately, modern ISAs violate several of the assumptions of this work, including: multiple binary instructions decode to the same assembly instruction, immediates are not always represented in assembly as they appear in the binary, instruction fields are not always independent, and not all immediate operands are contiguous. Conversely, our approach modifies the binary encoding of instructions and uses decoders to derive similar structural information about instructions without any outside information other than the maximum length of instructions.

III. Instruction Set Complexity

As introduced in Section 1, modern instruction sets are complex, and the challenges of implementing correct instruction decoders are a direct reflection of this complexity. In this section, we try to quantify instruction set complexity.

Complex opcode encoding: Opcodes can depend on many parts of an instruction, and even determining which bits encode the opcode is difficult. While an ISA often has a common set of bits used to encode the opcode, each ISA has many encoding variations that differ from their common opcode encoding. Every PowerPC instruction uses at least the first 6 bits to encode the opcode; many x86 instructions have common encodings based on the prefixes present; and, ARM instructions often use bits 1-6 for the opcode. Despite common opcode-encoding bits in each ISA, a scattering of extra opcode bits, reserved bits and fields that affect the opcode cause the total number of opcode-encoding subsets to explode. In total, we identified 279 different subsets of bits that encode ARMv8 opcodes, 124 subsets that encode PowerPC opcodes and 2900 subsets for x86 opcodes. Each ISA has characteristics that contribute to this complexity. For example, both ARM and PowerPC contain opcodes that are valid only when certain values are supplied for operands, adding those operand bits to the subset that affects the opcode, and the use of prefixes in x86 can shift and modify which bits encode the opcode.

Multi-purpose operand values: Even once the bits encoding an operand are identified, the operand may have several meanings based on other parts of the instruction. For example, x86 instructions accessing a register encoded by the value "001" might be accessing one of 10 different registers, and which register is accessed is determined by prefixes, operand size, and a myriad of other modifier bits scattered throughout the instruction. ARMv8 instructions also use multi-purpose operand encodings where the meaning of an operand encoding depends on the opcode.

Undefined and illegal instructions: Determining whether or not an instruction is valid and defined is difficult even once the opcode is known because many factors can cause an instruction to be undefined or illegal. These factors include a combination of prefixes, addressing modes, interacting operand values and even unallocated bits. For example, x86 instructions are illegal if they contain a lock prefix but do not contain a memory operand. Both ARM and PowerPC have instructions that may be invalid because of a combination of separate fields, like SIMD register loads whose range of destination registers cannot contain the source register. In these cases, only a synthesis of information from different fields of the instruction can determine whether or not it will be valid and defined.

Instruction Set Churn: Since its creation, x86 has received at least 13 extensions (depending on how one counts them) and now contains hundreds of opcodes, most of which were not present in the initial instruction set [11]. Likewise, both ARM and PowerPC have undergone significant revisions since their creation [3], [10]. Each revision or extensions forces programmers to update decoders. Additionally, extensions can produce new interactions between opcodes and operands that might affect existing instructions. The result is noticeable in differential testing: the GNU decoder incompletely deprecated

3DNow!, and Dyninst maintained old versions of PowerPC decodings from POWER 2 that conflict with a more recent vector extension.

These instruction set complexities have resulted in numerous errors in the decoding tools for x86, PowerPC and ARM, highlighting the need for comprehensive testing.

IV. Testing Procedure

The complexity of instruction sets makes thorough testing of instruction decoders difficult. It is challenging to generate input that includes all interesting test cases, and to verify the results of the decoding. Given a set of instruction decoders and an assembler, we generate a variety of test instructions that provide good instruction set coverage and verify the output of each decoder through differential testing and reassembly. When the output of decoders differ, and the reassembly process indicates a likely error, we produce labelled output indicating which decoder appears to be at fault. These tasks are broken into two main components: input generation (Section IV-A) and output verification (Section IV-B), shown in Figure 1.

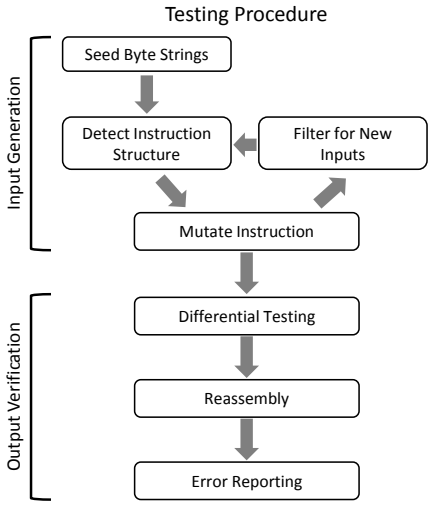

Fig. 1. Overview of testing procedure.

A. Input Generation

Our approach to input generation is based on the observation that, even though billions of byte strings decode to valid instructions in an instruction set, most are similar. If two instructions differ by only a single immediate or register operand, testing both instructions is probably unnecessary. Consider the x86 instruction at the top of Figure 2. While this instruction is 16 bits long, the last 8 bits contain an immediate operand, so changing these bits will produce an instruction that differs only by that immediate. The same goes for bits 6-8, which encode only a register operand. On the other hand, bits 1-5 encode the opcode. If we change any of these bits, we will produce a significantly different instruction, as shown in the bottom five rows of Figure 2. Our goal is to identify the bits that can be changed to produce new and interesting test cases so we can vary them to generate input with a variety of opcodes, prefixes and addressing modes.

```
1011010011011111: movb 0xdf, %ah
0011010011011111: xorb 0xdf, %al
1111010011011111: hlt
1001010011011111: xchgl %eax, %esp
1010010011011111: movsbb (%rsi), (%rdi)
1011110011011111: movb 0x65d5f5, (%esp)
```

Fig. 2. Changes made to any one of the first 5 bits of the top `movb` instruction produce an instruction that is different in a non-trivial way.

To identify bits that encode parts of the instruction that we want to vary is to simply flip bits and determine the new decoding. We observe that, in most of the cases that we consider interesting (new opcodes, prefixes or addressing modes), two or more operands are changed with a single bit flip. We use this observation as the starting point for inferring instruction structure and generating test cases.

The top of Figure 1 shows an overview of the input generation procedure. Given several sequences of random seed bytes, each of which is likely to contain a valid instruction (about 75% for x86 [17]), we detect structural information about these instructions (Section IV-A1). Using this structural information, we mutate the original instruction to create new test cases that efficiently cover the ISA being tested (Section IV-A2). This process continues iteratively by detecting the structure of each new test case and mutating these instructions to produce new inputs (Section IV-A3).

1) Inferring Structure: We start our testing process by generating several random byte sequences, each as long as the maximum instruction length for the ISA being tested (four bytes for ARMv8 and PowerPC, 15 bytes for x86). For each sequence of bytes that forms a valid instruction, we detect the structure of the instruction by repeatedly altering the sequence of bytes and decoding the result with each decoder. During this process, we give each bit a label that will be used during mutation to decide which bits of the instruction should be changed to produce new inputs. Our labelling process has three steps: detecting instruction length, assigning preliminary labels to each bit, and refining the label of certain bits.

First, we determine the length of each instruction. Given a sequence of N bytes, we use a decoder to decode only the first byte. If the result of decoding the first byte is identical to the result of decoding all N bytes, then the instruction is one byte long. We attempt decoding up to a maximum of N times. The fewest bytes that produces the exact same output as decoding all N bytes is considered to be the instruction length. The step only has practical significance for variable length architectures like x86, and it allows our process to function independently of the explicitly returned instruction length provided by decoders.

Next, we flip each bit of the instruction and assign a label based on the decoding of the resulting instruction. For now, we assume that flipping each bit alone is sufficient to determine which parts of the instruction each bit encodes. This

is equivalent to assuming that all bits independently encode fields, i.e., flipping one bit will not affect the result of flipping another. This is not always true, so we relax this assumption later to refine our inferred structures. Bits are assigned one of four labels: *field*, *reserved*, *unused* or *structural*.

Field bit: A bit is a field bit if it encodes only a single field of the decoded instruction. These bits are marked with a number indicating which field they encode. Often, these bits encode only a single immediate or register operand. In cases where the opcode of the instruction can be changed without modifying any other fields, a field bit can also be a part of the opcode. During mutation, field bits will be grouped into the fields that they changed, and each field will be given random and special values intended to generate special instruction forms that depend on the operands.

Reserved bit: A bit is a reserved bit if modifying it causes the instruction to change from a valid instruction to an invalid instruction. Because we focus on testing valid inputs, we do not modify reserved bits during mutation.

Unused bit: A bit is an unused bit if modifying the bit has no effect on the decoded instruction. As they have no effect on the result of decoding, unused bits are not modified during mutation.

Structural bit: A bit is structural if modifying it changes more than one field of the decoded instruction, the number of fields in the instruction, or the preliminary labels of other bits. These bits frequently encode the opcode, addressing mode or operand size. Because we want to test these characteristics of instructions, we modify structural bits in multiple ways to generate new inputs.

The preliminary label given to each bit is determined by comparing the decoding of the the input instruction with that of the input instruction with the bit flipped. Figure 3 gives an example of generating the sequence of bit-flipped instructions from the original `movb` with the resulting labels given to each bit.

We originally assumed that all bits were independent, so flipping each bit alone was sufficient to identify what those bits encoded. This is not always true though; for example, a bit may encode a field-modifying x86 prefix that significantly alters the instruction when changed in combination with the field bits. We refine our preliminary labels by relaxing the assumption that all bits operate independently. Now, we flip two bits at a time to determine if any bits encode structural information conditioned on the values of other bits. We perform this two-bit test only for field bits and unused bits because structural bits are already known to encode structural information and changing reserved bits results in errors. We test two bits flipped at a time by modifying each field one bit at a time and recomputing the preliminary map (which flips every bit once, giving us combinations of two bits flipped at a time). If the new map is different from the original, then the flipped bit encodes some structural information, so we refine the preliminary label from *field* or *unused* to *structural*. To illustrate this issue, we give a more complex example `adc` instruction that contains a REX prefix (the first 8 bits) in

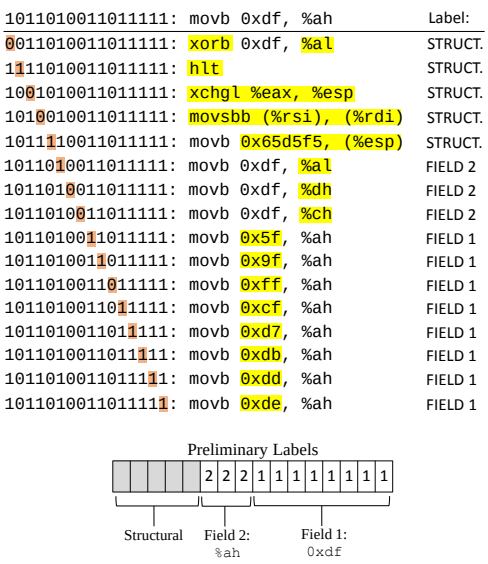

Fig. 3. Example of flipping each bit in the example movb instruction and assigning labels. Grayed-out spaces in the final instruction are structural or reserved bits. Others are given a number corresponding to the field that they encode. Note that changes made to any one of the first 5 bits of the top instruction produce an instruction that is different in a non-trivial way, while changes to the remaining bits produce an instruction that differs only by an immediate or register operand.

Figure 4.

Several bits in the REX prefix are given a preliminary *field* label because they encode only a single field of the assembly language. However, these bits do not match the concept of a field that we intended because they also encode instruction structure. Bit 3 was given the label *field 2* because it appeared to encode field 2, but changing this bit alters the REX prefix to a `data16` prefix. Both of these prefixes modify *field 2*, but they have different structures, so we revise the preliminary label of this bit to be *structural*. We try only combinations of two bits flipped at a time, so we cannot identify structure bits depend on two or more other bits. We chose this value empirically because using three bits provided only marginal benefits at a high cost.

Not all fields are encoded such that any value can be supplied while affecting only that field. Some have special values that cannot be used with certain opcodes, and other fields determine which alias of an instruction should be produced. In these cases, changing a bit that encodes a single operand may change multiple fields or the validity of the instruction, so the labelling process may not give these bits *field* labels. We do not consider this a significant limitation because these field bits are often given the *structural* label, which is used more rigorously to generate new inputs.

While the process of flipping each bit and re-decoding an instruction allows us to label every bit, it requires the instruction to be decoded many times to infer its structure. The final labelling process is an $O(N^2)$ process with respect to the

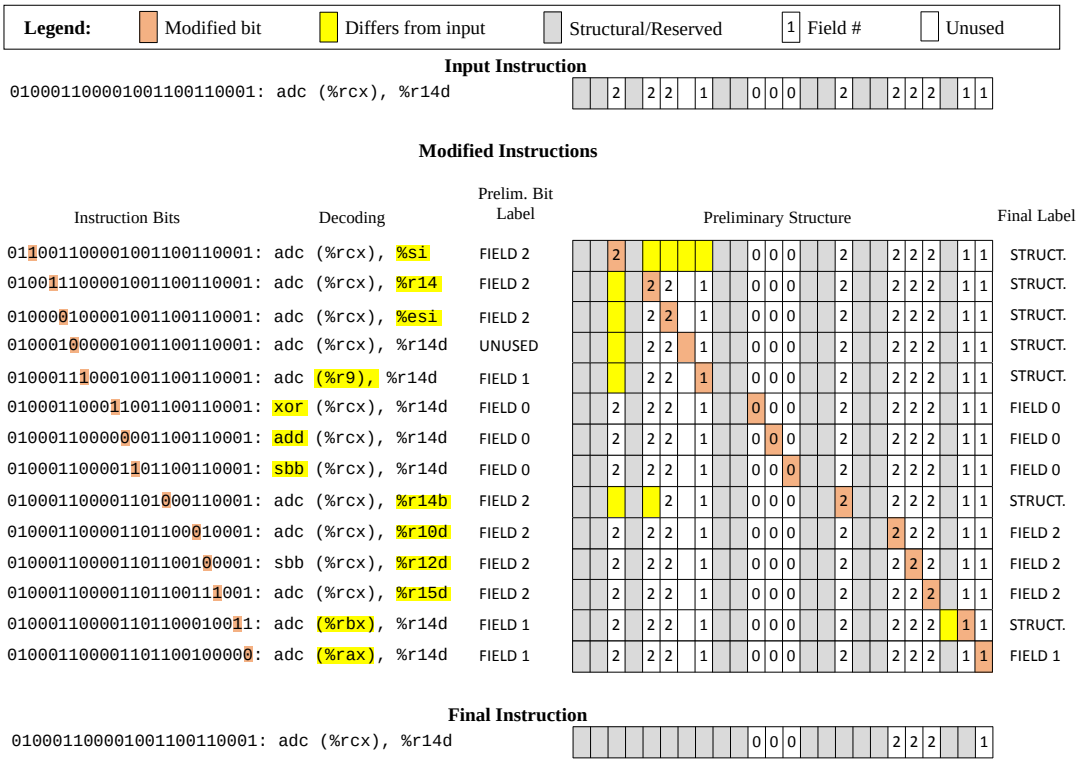

Fig. 4. An example of refining the preliminary map for an adc instruction. The first byte of this instruction is an x86 REX prefix, so the bits of this prefix encode more than just operand 2, they also encode structural information. When each field bit is flipped, the preliminary labels are recomputed as shown on the right side of the figure. Each bit whose change results in different (highlighted) preliminary labels is given the structural label in the final labels.

number of bits because it repeatedly determines all preliminary labels to identify structural bits like the one highlighted in Figure 4. For a 16 bit instruction, this could mean that hundreds of decodings are used to infer structure. To reduce the total number of decodings required, we introduce an optimization based on an observation used in DERIVE [6]: many immediates are contiguous and printed in assembly exactly as they appear in the bytes of the instruction. For example, `movb $0xdf, %ah` has two bytes: `b4 df`. The second byte encodes the immediate, and its value is used directly in the assembly language representation of the instruction. Once we detect the first bit of an immediate field through decoding, we then directly compare the value decoded with the adjoining bits. If they match, we infer that all these bits encode an immediate, without the need for additional decoding. Immediate bytes that are labelled this way do not need to be checked for structural properties because they are used in exactly one field. This optimization reduces the number of times the sample `movb` instruction must be decoded to infer structure from 192 (nearly the number of bits squared) to 36 decodings.

2) Mutating Instructions: Once each bit of the input instruction has been labelled, we use these labels in a mutation process that generates new input instructions through four different types of modifications of the original instruction:

- *Pairwise Structural Bit Flips:* Each pair of structural bits are flipped while all other bits are held constant. Because structural bits often correspond to opcode, addressing mode or operand size, changing a combination of these bits should create inputs with new variations of those characteristics.
- *Single Structural Bit Flips:* Each structural bit is flipped individually while the other bits are held constant (this process is equivalent to simply passing the single structural bit-flipped instructions from Section IV-A1 to the filtering phase).
- *Field Randomization:* We choose a single random value for each operand field. Special operand values (like all zeros, all ones, or the same register used for source and destination) often signify a distinct operation of the instruction and may have a distinct decoding. By choosing a random value for each field, we try to create new inputs that do not contain special operand values.
- *Setting Special Operand Values:* We also set each operand to all zeros and all ones because these values are commonly signify a special operation of an instruction. For example, register operands with the value ”11111” in ARMv8 can (but not always) signify the stack pointer. By generating inputs with these common special values, we try to test variations of instructions with special operations based on their operands.

These mutations create new instructions that are likely to differ in multiple fields and number of fields, while testing each field randomly and for special values. While we are not limited to two structural bit flips at once, we observed a significantly reduced rate of new, valid instructions when changing three or more structural bits. When mutations are

performed on x86 instructions, which can vary in length, the instructions are at the beginning of a byte sequence with 15 bytes, so new instructions created by mutation may be longer than the input instruction, up to the x86 maximum size of 15 bytes.

3) Filtering For New Inputs: While mutating the input instruction creates new inputs to test, it can also create instructions that are redundant with other inputs. For example, the first output from mutating `movb 0xdf, %ah` is `je 0xe1`. When the `je` instruction is mutated, it will recreate the original `movb` instruction, which would again be tested and mutated. To ensure that we do not loop through the same instructions, we create a *format string* for each instruction we test. This string is a modified version of the assembly language representation, where the immediate operands are replaced with a generic symbol and the register operands are replaced with a generic name indicating to which register set they belong. Instructions are only tested and used to generate new input if no other instruction with the same format string has already been tested. This process requires information about the register sets of an ISA, but this is a small subset of the knowledge required by other methods. Figure 5 shows the format string for the `movb` instruction previously discussed.

```
Original:    movb 0xdf,%ah
Format Str: movb IMM, %reg1
```

Fig. 5. The format string of a movb instruction.

We place one final restriction on new test inputs: they cannot have more than two optional bytes. Bytes are considered optional if they can be removed from the instruction while removing a subset of fields in the instruction, but without altering any remaining fields of the instruction. In practice, the only bytes with this characteristic are the x86 legacy prefixes and unused REX prefixes. Consider an x86 instruction (the `movb` from previous examples) with three optional bytes, each of which is a prefix:

```
(data16 addr32 data16) movb 0xdf, %ah
```

This instruction can have up to 13 prefix bytes, each of which can have 11 different values (repetition of some prefixes is allowed). Using only permutations of the prefixes, we could create more than 30 trillion versions of this instruction, making exhaustive testing prohibitive. To limit the number of instructions that differ only by permutations of optional bytes, we filter out inputs with three or more optional bytes. Complex x86 prefixes that affect the opcode or operands like the REX, VEX and EVEX prefixes are never counted toward optional bytes because removing them alters other fields of the instruction.

While it may seem reasonable to save the generated inputs in the form of a model that can be used in future testing, the resulting model will depend on the versions of the tested decoders. If a decoder is updated to support new instructions or remove incorrect decodings, the old model may not correctly capture these changes. Because of this limitation, we choose to re-run input generation each time we test decoders with this framework. For most ISAs tested, this requires only a few minutes. For 64-bit x86, this means testing overnight (about 8 hours). We feel that this is an acceptable cost for a thorough test. Of course, users can re-test specific instructions themselves by simply running the decoders on a binary with those instructions.

B. Output Verification

Our approach to output verification uses differential testing combined with reassembly to avoid the error-prone process of decoding instructions by hand and referencing the ISA manuals. When verifying the output of decoders, we use the assembly language representation because it is a common form of output available from all the instruction decoders that we tested (and because the assembly language decoding can be verified using an assembler). Our approach first uses each decoder to decode an instruction (Section IV-B1), and then attempts to reassemble the resulting assembly language representations of the instruction to compare representations (Section IV-B2). We produce an error report when it appears that one or more decoders produced incorrect assembly language for the input bytes (Section IV-B3).

1) Differential Decoding: We begin output verification with differential decoding: each instruction is decoded with all decoders, and the result is normalized and compared. Our rationale for decoding each instruction with all decoders is that if all decoders produce the same output, it is likely to be the correct output, so no further testing of this input is needed. Our approach differs from Paleari et. al [17], who labelled decodings agreed on by 75% of decoders as correct. We require that *all* decodings match to be considered correct, as we have found cases where only one decoder produces a correct decoding.

Ideally, there would be one assembly language representation of each instruction, so a simple string comparison could identify errors. Unfortunately, the output of decoders can differ in many ways that should not be considered errors, where some of the differences are trivial, like different formatting for register names, and some are complex, like aliases of an opcode with a different number of operands. We address these differences by applying a set of normalization rules. When the assembly language representations from different decoders are almost identical, except for spacing or minor formatting, normalization can produce the same output from each decoder, obviating the need for reassembly. In cases where there are significant differences in decoder output, normalization is used to ensure that the assembly language produced by each decoder conforms to the input expected by the assembler. (Decoders frequently produce assembly language that cannot be reassembled.) Figure 6 shows a case where a single ARM instruction produced different decodings with different decoders.

In this case, there are a few trivial differences, like spacing between operand and register name capitalization, but the outputs of the decoders differ in other, more significant ways including opcode, operands and even validity. While

Decoder	Output	Normalized	Reassembled
libopcodes	`mov v0.d[0], v7.d[1]`	`mov v0.d[0], v7.d[1]`	`0x630844e0`
Capstone	`invalid`	`invalid`	`N/A`
Dyninst	`ins Q0, Q7`	`ins q0, q7`	`Error`
LLVM	`ins v0.d[0], v7.d[1]`	`ins v0.d[0], v7.d[1]`	`0x630844e0`

Fig. 6. A single ARM instruction decoded by each decoder.

differences in decoding often indicate an error in at least on decoder, this may not always be the case. Reassembly of the instruction provides a tool to make this determination.

2) Reassembly: Consider the output of LLVM and libopcodes in Figure 6. While the opcodes appear to differ, the `mov` opcode is actually an alias of `ins`. Explicitly programming all such aliases as normalization rules would require substantial expert knowledge of the ISA being tested, and may introduce errors in the testing process. By using an assembler that accepts different assembly language representations of an instruction, we can determine whether two different assembly language strings encode the same instruction.

If the output of a decoder assembles to the input bytes, we believe that the decoding is correct. For all other cases, we record whether the assembler produced an error and what bytes were produced by assembling the instruction. We use this information during error reporting to determine which decoder outputs appear to be incorrect.

3) Error Reporting: Once we have discovered that the output of instruction decoders differs, and the difference cannot be resolved through reassembly, we suspect that the difference indicates an error in one or more of the decoders. We define three ways that our testing procedure can identify errors in decoders:

Output of decoder results in an error when reassembled: These errors often signify that the input instruction is invalid or that an instruction is not supported by the assembler. These errors are always reported.

Decoder reports "invalid": If a decoder reports an input to be invalid, and the other decoders report the same input to be valid, we check the reassembly of the valid decodings. If any valid decodings can be reassembled without error, then we report the invalid decoding as an error.

Output of decoder does not reassemble to the same input bytes: This often indicates that the decoder has produced a valid but incorrect decoding for the input bytes. However, not all reassembly differences are the result of errors. Some differences occur when the assembler produces an equivalent instruction encoded by different bytes. We detect this case by comparing the reassembled bytes of all valid decodings. If all valid decodings reassemble to the same bytes, then the decodings are equivalent and not reported as errors.

For each decoder with an error, we produce a report that contains the input bytes, the output of every decoder, and any error messages produced by the assembler. We organize these reports into files based on the reassembly error. For example, all x86 decodings that produced the error "expecting lockable instruction after lock prefix" are placed in the same file. This grouping simplifies the task of condensing many similar reports. Ideally, we would like to automatically condense these reports for the developers of each decoder. Unfortunately, the process of determining which errors are the result of the same underlying issue in a decoder is challenging and requires a reference manual and significant expert knowledge of the ISA.

V. Evaluation

We implemented our testing procedure in a tool named Fleece and used it to evaluate popular instruction decoders for three architectures: x86-64, ARMv8.0 and PowerPC version 3.0. We tested libopcodes (2.26), LLVM (3.9), Dyninst (9.2) and Capstone (3.0.4) for all three architectures, and Intel XED (6.26) for only x86-64. We also compared the efficiency of Fleece with random input generation for each of the architectures, and demonstrated that Fleece produces instructions with a substantially greater variety of formats.

A. Decoder Evaluation

For each ISA, we configured our tool to use the GNU assembler (version 2.26) and began our testing procedure with ten seed byte sequences (enough to frequently have at least one valid instruction in our input). We allowed the testing procedure to run until it exhausted the list of inputs that it could discover using our mutation process. Table II provides data summarizing the testing of each ISA.

Arch	Inputs	Differences	Time (mins:secs)
x86	482,711	480,034	508:00
ARM	6,051	4,337	3:09
PowerPC	3,629	3,067	1:02

TABLE II
Testing results from applying Fleece to several decoders. x86 testing generated about 100x as many inputs as ARM and PowerPC testing.

To our initial surprise, the vast majority of inputs resulted in differences. However, because we independently vary fields of an instruction, an error in the decoding of a single type of operand or addressing mode will result in many differences. As a concrete example, few x86 instructions are valid with a `lock` prefix, but our testing procedure tests many different opcodes and addressing modes with a `lock` prefix, resulting in more than 120,000 test cases with lock prefixes, most of which result in differences because only libopcodes decoded them as valid. Similar issues with x86 prefixes exist for multiple decoders, resulting in almost every test case differing among the five decoders.

We handle this large number of differences in two ways. First, as mentioned in Section IV, output is placed into files based on any assembler errors produced, so all 128,331 test

TABLE I
EXAMPLES OF ERRORS FOUND FOR EACH DECODER AND EACH ARCHITECTURE.

	Decoder	Input	Output	Correct
x86-64	XED & LLVM	`67 00 05 00 00 00 00`	`addb %al, (%rip)`	`addb %al, (%eip)`
	libopcodes	`c4 02 51 90 51 19 51 19`	`vpgatherdd %xmm5, 0x19(%r9), %xmm10`	`vpgatherdd %xmm5, 0x19(%r8, %xmm8, 1), %xmm10`
	Dyninst	`de 6c 50 6e`	`fisubr 0x6e(%rsp)`	`fisubr 0x6e(%rax, %rdx, 2)`
	Capstone	`66 3e 97`	`xchgl %di, %eax`	`xchg %ax, %di`
ARMv8	libopcodes	`e8 13 5a 2a`	`mov w8, w26`	`orr w8, wzr, w26, lsr #4`
	LLVM	`f8 e3 4f 08`	`invalid`	`ldaxrb w24, [sp]`
	Dyninst	`a9 1c 20 6e`	`eor q9, q5, q0`	`eor v9.16b, v5.16b, v0.16b`
	Capstone	`6a 2d 1e 6e`	`invalid`	`mov v10.h[7], v11.h[2]`
PowerPC	libopcodes:	`7c 00 12 6e`	`lhzux r0, 0, r2`	`invalid`
	LLVM	`41 80 80 00`	`bt 0, .+32768`	`bt 0, .-32768`
	Dyninst	`43 77 dc 23`	`bdzla- 0xffffdc20`	`bcla+ 26, 4*cr5+so, 0xffffdc20`
	Capstone	`43 77 dc 23`	`bdzla+ 0xffffdc20`	`bcla+ 26, 4*cr5+so, 0xffffdc20`

cases with lock instructions that were decoded incorrectly by libopcodes are in the same file whose name includes "expecting_lockable_instruction". Second, we use shell scripts that search for a certain feature, like an instruction pointer dereference (`(%eip)` or `(%rip)` in x86), and we compare the correct outputs to any incorrect outputs. If they differ by which instruction pointer was used, then we know this decoding error contributed to the report. If the decodings are different, but neither produced an assembler error, then we refer to the ISA manual to determine which decoder is correct. We check several such decodings before using a shell script to automatically categorize the rest. For example, LLVM and Capstone correctly decoded all instruction pointer dereferences (their outputs reassemble to the input bytes without error), but Dyninst, libopcodes and XED did not. To identify all errors of this type, we use a single shell script that selects all reports involving instruction pointer dereferences and outputs those where a decoder deviates from the known-correct decoders.

While many of the differences that we report are the result of an error in at least one decoder, other differences are the result of different ISA support. For example, libopcodes version 2.26 supports ARMv8.1, so it decoded 727 instructions that are only valid in ARMv8.1, while the other decoders viewed these as invalid inputs. In our x86 testing, there were also 129 differences in x86 output due to imperfect normalization. Most (79) of these instructions are variants of `mov` for which the correct representation is not always clear because the assembler error is generic ("unsupported instruction 'mov'") and the assembler will not accept instruction suffixes in some cases while requiring suffixes in other cases of the same opcode. The difference in syntax requirements of the assembler could not be resolved even when considering opcode and addressing mode together. Normalization rules intended to address this issue reached too far and altered correct output, so they were removed. In all other outputs, there was at least one incorrect decoder or difference in support for an assembly feature.

Table III shows the number of real errors reported for each decoder and architecture.

The most common type of error discovered was invalid instructions being decoded as valid. Decoders also had errors decoding control flow targets and memory access locations.

x86-64 Summary: We found the most errors in x86 decoders, the majority of which (44 of the 54 issues) involved instructions with at least one prefix. This finding agrees with the work of Paleari et. al, who also observed that x86 instructions with prefixes were frequently decoded differently by a variety of decoders [17]. While decoders agreed on many general-purpose instructions, instructions from recent extensions were often decoded incorrectly by at least one decoder, particularly when prepended with a legacy prefix. Table I gives an example error reported for each of the decoders. In the example given for XED and LLVM, the first byte, `67` specifies that a 32-bit address should be used to determine the memory operand. Both XED and LLVM ignore this byte when the register specified is the instruction pointer (%rip). The examples given for both libopcodes and Dyninst show cases where an address is computed using the wrong addressing mode and registers. Given the bytes `66 3e 97`, Capstone produces an incorrect `xchg` instruction whose operands are different sizes (`%di` is a 16-bit register while `%eax` is 32 bits) because it incorrectly applies the `data16` prefix (`0x66`) to only one of the two operands. Although operand order is swapped by Capstone, this has no effect for `xchg` instructions.

Arch	XED	libopcodes	LLVM	Dyninst	Capstone
x86	3	13	8	18	12
ARM	n/a	2	1	7	3
PowerPC	n/a	7	4	6	7

TABLE III
NUMBER OF ERRORS REPORTED FOR EACH TOOL. X86 DECODERS HAD FAR MORE ERRORS ON AVERAGE THAN OTHER DECODERS, WHILE ARM DECODERS TENDED TO HAVE THE FEWEST ERRORS.

Decoders can also terminate unexpectedly due to invalid inputs. LLVM will terminate with an "unreachable" message when used to decode an EVEX instruction with opcode byte `0xc2`, and libopcodes will terminate unexpectedly for some EVEX instructions with opcode byte `0x02`.

ARM Summary: ARMv8 decoders had the fewest errors, but each decoder had at least one. LLVM was the most reliable decoder, which is not surprising, because it receives commits from ARM employees and is recommended as the authoritative ARM decoder. The most common error in ARM decoding is the example provided for LLVM in Table I, a load instruction used for transaction-based computation. LLVM decodes this instruction as invalid because unused operands have values other than the typical compiler-generated values (all 1s) for

this instruction. In the error shown for libopcodes, the decoder produced an incorrect alias. While the correct `orr` instruction includes a shift of 4 bits, the `mov` instruction given by libopcodes incorrectly omits this shift. The Dyninst error is a decoding in which registers of the wrong type (general purpose instead of vector) are produced by the decoder. The incorrect Capstone output shows a case where the decoder fails to recognize a valid instruction (this instruction was supposedly supported by the decoder). Compared to the errors identified in x86-64 decoders, the errors we discovered in ARM decoders are more related to aliasing and correctly identifying valid instructions and less related to operands and addressing modes, likely because ARM instructions have distinct forms of each opcode that describe the types of operands used, while x86-64 instructions have a variety of different operands and operand-modifying prefixes that can be used for any given opcode.

PowerPC Summary: For each decoder, PowerPC had fewer errors than x86-64, and for all but Dyninst, they had more errors than ARM. The majority of these errors are cases where an invalid instruction is decoded as valid. Table I shows an example of a difference found for each decoder. libopcodes decodes an invalid `lhzux` instruction as valid. This atomic update `lhzux` instruction cannot use `r0` as the first operand, but the bits for this instruction encode `r0`, so it is invalid. The example error given for LLVM shows a case where it incorrectly decodes the offset of a branch instruction. Both Capstone and Dyninst also provide incorrect decodings of branch instructions for PowerPC. In the case shown in the table, Dyninst and Capstone both give a version of the branch mnemonic with fewer operands, but the alias is incorrect. The correct version of the instruction branches based on the overflow bit (`so`) of condition register 5 (`cr5`), as indicated by the second operand. The incorrect version produced by Dyninst and Capstone implicitly branches based on the less-than bit of `cr0`.

The decoding errors we discovered have implications for a variety of higher-level analysis techniques that rely on accurate decoding. All decoding errors will make debugging more difficult, while many of the errors we found have implications for analysis tools too. Decodings with incorrect addressing information will result in errors when tracing memory accesses for data-flow. Incorrect decodings of branch instructions, including conditions and destinations can result in incorrect control-flow extraction and incomplete or incorrect code discovery as a result. Decoders that fail to decode instructions that do not contain typical compiler-generated options risk failure when given hand-written executables, allowing malicious programmers to obfuscate code and defeat higher-level tools.

TABLE IV
COMPARISON OF TESTING RESULTS USING RANDOM INPUTS AND FLEECE-GENERATED INPUTS. RANDOM TESTING WAS RUN FOR AS LONG AS FLEECE FOR X86, AND 10 MINUTES FOR ARM AND PPC. FOR EACH ARCHITECTURE, FLEECE DISCOVERS MORE UNIQUE DIFFERENCES IN LESS TIME.

	Fleece			Random		
Arch.	Inputs	Unique Diffs.	Time (mins:secs)	Inputs	Unique Diffs.	Time (mins:secs)
x86	482,711	480,034	508:42	2,679,254	9,494	508:00
ARM	6,051	4,337	3:09	1,706,422	600	10:00
PPC	3,629	3,067	1:02	64,360	1,100	10:00

B. Fleece Evaluation

Input generation is a critical part of our testing procedure, but there are few good benchmarks against which we can evaluate our input generation. While Paleari et al. [17] used input generation that was more sophisticated than a purely random approach, they chose most of their input bytes at random. Additionally, their testing for x86 occurred before the introduction of complex EVEX prefixes, so we compare Fleece to a random approach given the same, or greater period of testing time. To evaluate the usefulness of Fleece, we compare the number of unique differences found (meaning no duplicate prefix-opcode-operand type instructions).

Table IV gives an overview of the inputs generated and differences discovered by testing random input compared to testing Fleece-generated input.

Random testing requires very little time to generate inputs, so the primary bottleneck for this approach was reassembly. For ARM random testing, most decodings were exact string matches, which do not require reassembly, allowing random inputs to be tested more quickly. Despite this speed, ARM testing still tested instructions with about 1/10th of the format string variety discovered by Fleece. While the random approach discovered a greater total number of differences, most of these differences are redundant, covering instructions with similar opcodes, addressing modes and register sizes.

Fleece discovered unique differences much more quickly because it varied instruction features using inferred structure, only testing instructions with new structure. For example, a single x86-64 instruction with a `data16` and EVEX prefix will be produced at a rate of about one in 65,000 at random (because it must contain `67` and `62` as prefixes), yet one in five of the Fleece-generated input contained these prefixes because they produce instructions with unique opcode-addressing mode combinations. For both ARM and PowerPC, the difference between Fleece and the random input generation is less dramatic, yet it persists because instructions with 15 bits devoted to the opcode or reserved are still fairly rare in random inputs (1 in 32768), but are frequently generated by Fleece.

For all architectures, Fleece was able to use inferred instruction structure to generate inputs with a variety of features much more efficiently than a random approach.

VI. CONCLUSION

Binary decoders play a critical role in analyzing executable files, a task fundamental to numerous applications in program performance, security analysis and debugging. Decoders provide instruction-set-specific knowledge that allows higher levels of analysis to work with abstractions about control- and data-flow. Unfortunately, the ISA-specific knowledge contained in decoders can be inaccurate, leading to inaccurate

conclusions from higher level analyses. We designed and implemented a method of input generation, differential testing, and reassembly that can be used with very little ISA-specific knowledge to discover decoding errors. We demonstrated the effectiveness of our method by testing a variety of instruction decoders for a variety of architectures and reporting errors for each. In addition, we showed that our testing procedure can use inferred structural information to substantially improve the variety of instruction formats tested during random testing.

VII. Acknowledgements

We are grateful to the efforts of Xiaozhu Meng, Bill Williams and Ben Welton in the preparation of this paper, and the technical support of John Detter and the rest of the Paradyn Project team.

This work is supported in part by Department of Energy grant DE-AC05-00OR22725, National Science Foundation Cyber Infrastructure grants ACI-1547272 and ACI-1449918, Lawrence Livermore National Lab grant B617863, a grant from Cray Inc., and a grant from Intel Corp.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, Apr. 2010.

[2] Anh Quynh, Nguyen. Captstone: The Ultimate Disassembly Framework, http://www.capstone-engine.org/.

[3] ARM. *ARM Architecture Reference Manual*.

[4] A. R. Bernat and B. P. Miller. Anywhere, Any Time Binary Instrumentation. In *ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Szeged, Hungary, Sept. 2011.

[5] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. Technical Report UCB/EECS-2009-133, EECS Department, University of California, Berkeley, Oct 2009.

[6] D. R. Engler and W. C. Hsieh. DERIVE: A Tool that Automatically Reverse-Engineers Instruction Encodings. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, Boston, Massachusets, USA, Jan. 2000.

[7] W. Fang, B. P. Miller, and J. A. Kupsch. Automated Tracing and Visualization of Software Security Structure and Properties. In *9th Internation Symposium on Visualization for Cyber Security (VizSec)*, Seattle, Washintgon, USA, Oct. 2012.

[8] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 2008.

[9] Hex-Rays. IDA, https://www.hex-rays.com/products/ida/.

[10] IBM. *Power ISA*.

[11] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*.

[12] Intel. Intel X86 Encoder Decoder Software Library, https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library.

[13] Intel. Pin - A Dynamic Binary Instrumentation Tool, https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO '04 Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Palo Alto, California, USA, Mar. 2004.

[15] B. P. Miller, L. Fredricksen, and B. So. An Empirical Study of the Reliability of UNIX Systems. *Communications of the ACM*, 33(12), Dec. 1990.

[16] Oleh Yuschuk. OllyDbg, http://ollydbg.de/.

[17] R. Paleari, L. Martigononi, G. Roglia, and D. Bruschi. N-Version Disassembly: Differential Testing of X86 Disassemblers. In *19th international symposium on Software testing and analysis (ISSTA)*, Trento, Italy, July 2010.

[18] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Seattle, Washington, Aug. 1997.

[19] P.-M. Seidel. Directed Test Case Generation for x86 Instruction Decoding. In *15th International Microprocessor Test and Verification Workshop (MTV)*, Austin, Texas, USA, Dec. 2014.

[20] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. PoosanKam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *4th International Conference on Information Systems Security*, Hyderabad, India, Dec. 2008.

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

[22] The Valgrind Developers. Valgrind, https://valgrind.org.

[23] D. Yang, Y. Zhang, and Q. Liu. BlendFuzz: A Model-Based Framework for Fuzz Testing Programs with Grammatical Inputs. In *Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Liverpool, UK, June 2012.