Random and Exhaustive Testing of Instruction Parsers

Nathan Jay
Paradyn Project

Scalable Tools Workshop
Granlibakken, California
August 2016

Motivation

Lots of tools parse binaries

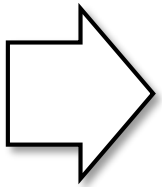
GNU

Motivation

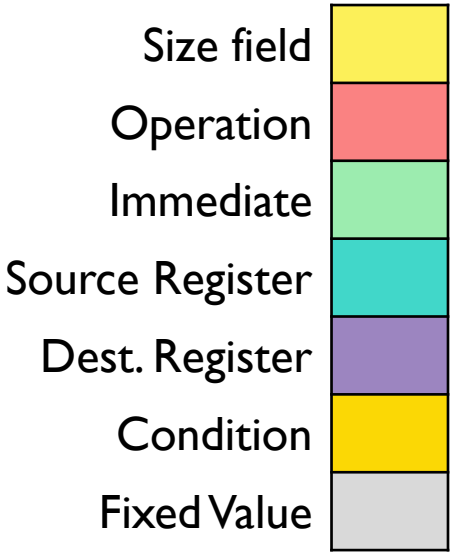
Parsers rely on a disassembly step:

Converting object code into a higher-level language with semantic information

Hex	Assembly
00: 55	push %rbp
01: 48 89 e5	mov %rsp, %rbp
04: 89 7d fc	mov %edi, -0x4(%rbp)
07: 8b 45 fc	mov -x4(%rbp), %eax
0a: 83 c0 0a	add $0xa, %eax
0d: 0f af 45 fc	imul -x04(%rpb), %eax
11: 5d	pop %rbp
12: c3	retq



Motivation



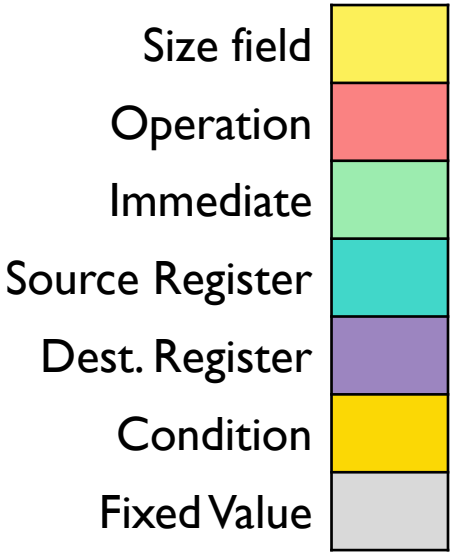
Converting object code to assembly is easy for a single format, like this from ARMv8:

Compare and branch (immediate)

No single format is difficult to decode. Just extract the fields and translate binary to assembly for each field.

Motivation

Unfortunately, the format varies between instructions.



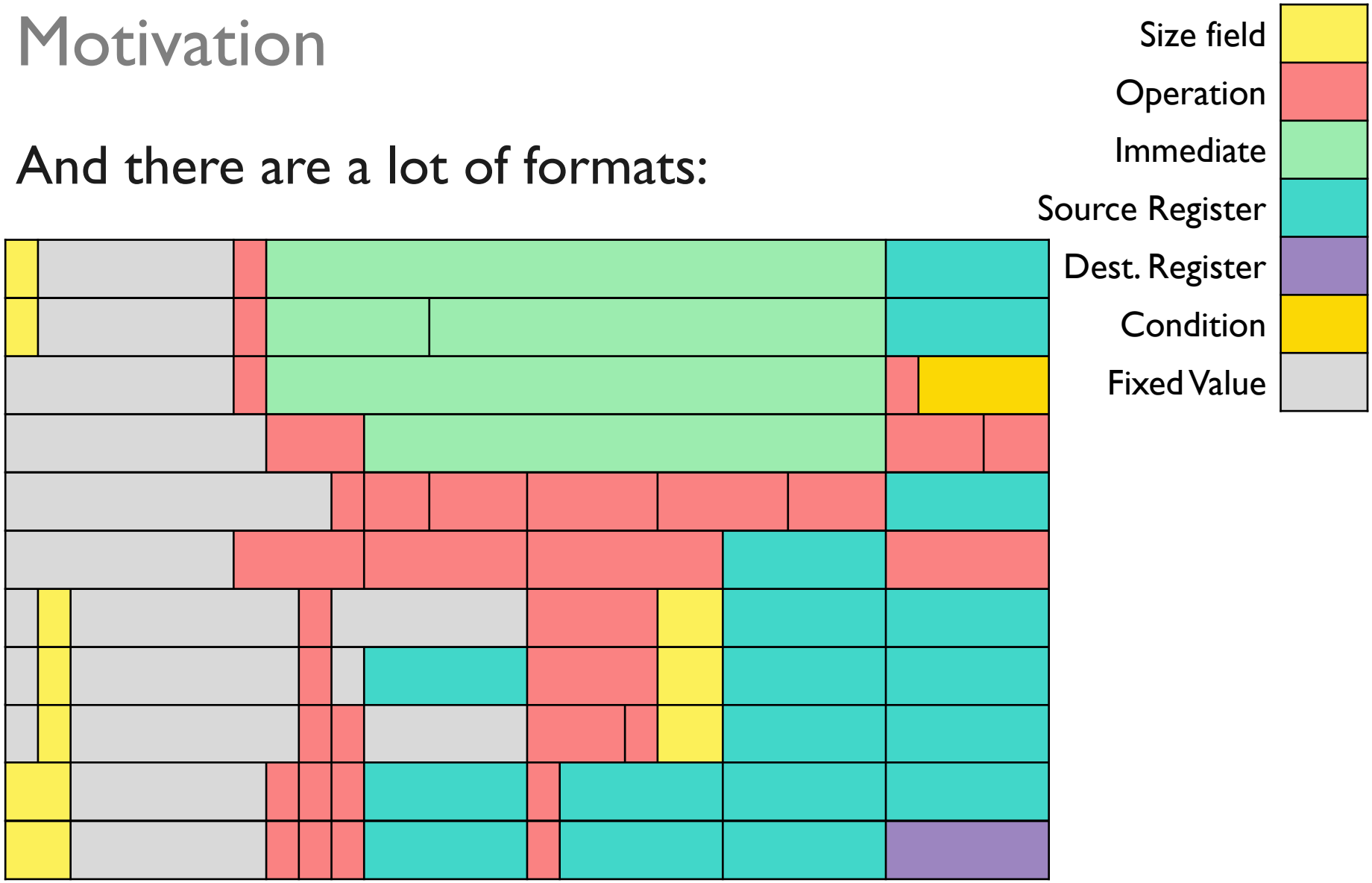
Compare and branch (immediate)

Test and branch (immediate)

Conditional branch (immediate)

Motivation

And there are a lot of formats:



Motivation

These formats only partially cover:

- load/store
- branching

The manual specifies more than 5 times as many different, general formats.

ARM can vary between implementations:

Apple, Samsung, AMD, Nvidia, Broadcom, Applied Micro, Huawei, Cavium…

Motivation

x86 has other challenges with variable length instructions.

This format works for some 1 or 2 byte opcodes:

Prefixes	opcode		mod R/M	SIB	displacement	immediate
Seg, Rep, Lock, 66, 67 REX	0F	XX	*	*	0, 1, 2 or 4 byte value	0, 1, 2 or 4 byte value

There is another format for some 3 byte opcodes:

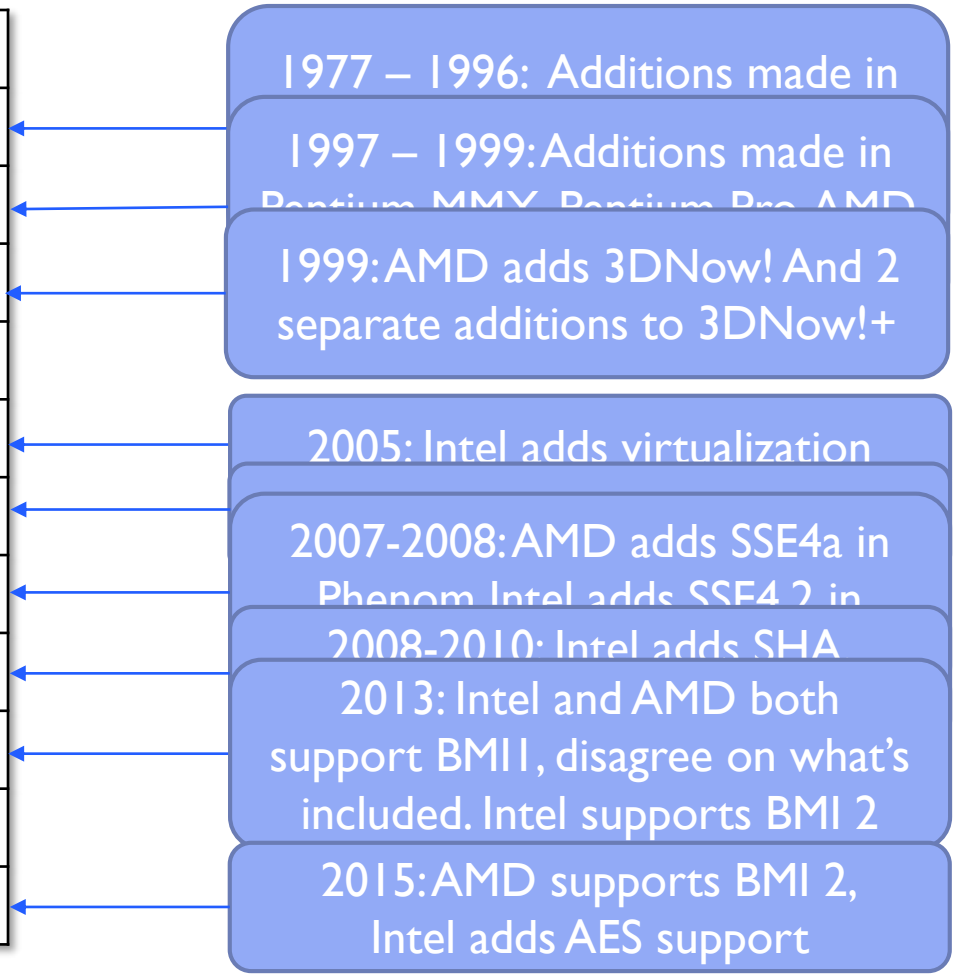
Prefixes	opcode			mod R/M	SIB	displacement	imm		
Seg, Rep, Lock, 66, 67 REX	0F	*	XX	*	*	0, 1, 2 or 4 byte value	byte		

This is less than a 3rd of byte level maps, and there are bit level maps as well.

Motivation

Moreover, instruction sets change over time:

x86 Extensions	
NPX (x87)	1977
MMX	1997
SSE	1999
SSE2	2000
SSE3	2004
SSSE3	2006
SSE4	2007
AVX	2008
AVX2	2011
AVX512	2013
MPX	2013



Goals

- Find disassembler errors
  - Test enormous instruction space quickly
  - Consolidate duplicate reports of an error
- Avoid instruction set specifics
  - Work for multiple instruction sets
  - Don't rely on specific instruction set versions
- Work with any disassembler

Previous Work

Some past efforts:

- Comparison of disassembly and execution results, Ormandy 2008
  - Generate instructions randomly or by brute force
  - Disassemble instructions, execute instructions and compare results
- Generation of known valid or invalid x86 prefixes and opcodes, Seidel 2014
  - Start with empty string of bytes
  - Use look up tables for next valid byte to build instruction, byte-by-byte
  - Arbitrary values can be appended after opcode
- N-version differential disassembly, Paleari et. al 2010

Previous Work – Paleari et. al 2010

Input:

- Randomized bytes (40,000 sequences used)
- CPU-tested instructions (20,000 sequences picked at random)
  - Enumerate all possible 1, 2 and 3 byte sequences
  - Execute each byte sequence with a few operands
  - Prepend a few prefixes to each sequence

Test:

- Compare 8 disassemblers' outputs and execution results
- Remove disassembly output that conflicts with execution in:
  - Instruction length
  - Operand type
- Declare the most common output to be correct

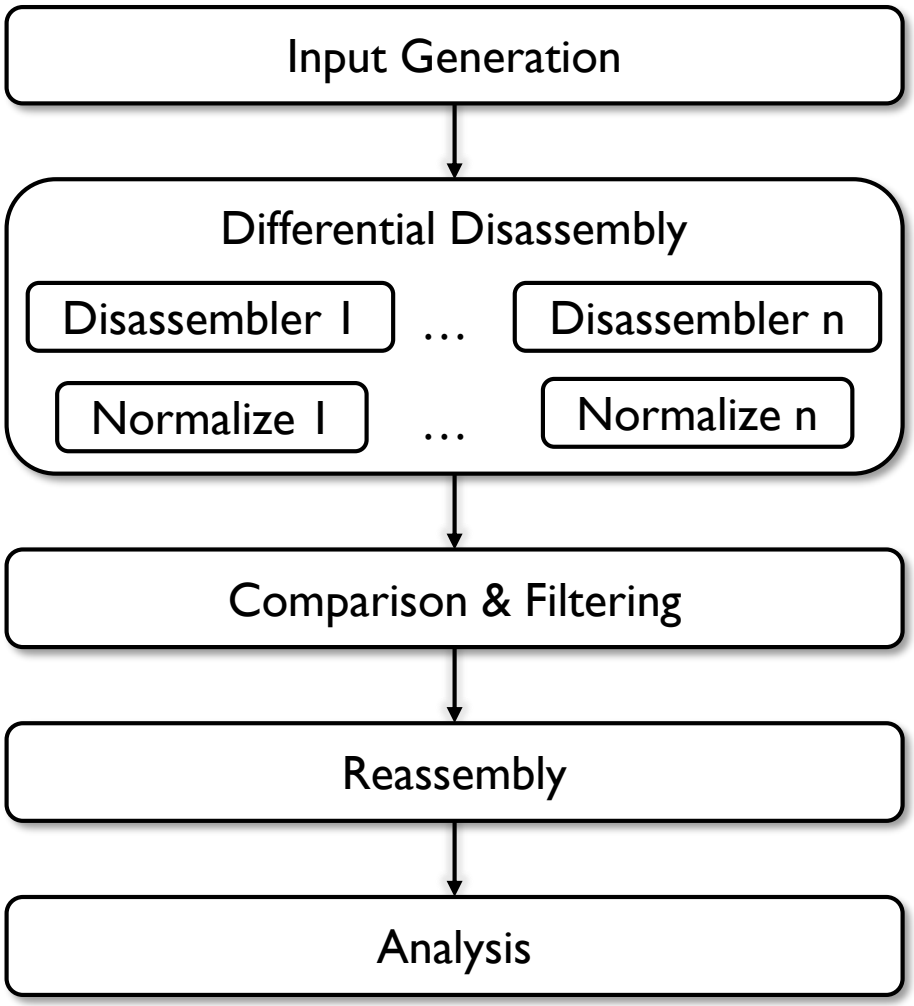
Previous Work - Limitations

- Naïve input generation
  - Randomly choosing instructions inefficiently tests whole space
  - A brute force approach would require 2^{120} instructions
- Required expert knowledge of x86
  - Semantic specification for decoding to compare to execution
  - List of all valid bytes, prefixes, knowledge of operand position
- Relied on details of the ISA
  - Opcode length and position
  - Byte boundaries
- No means to coalesce similar error reports

Approach

- Generate instructions more effectively
  - Avoid repetitions of similar instructions
  - Cover instruction space more thoroughly than purely random within a reasonable timeframe
- Test all functional parts of instructions
- Avoid ISA dependencies and expert knowledge

Workflow



Create object code to disassemble

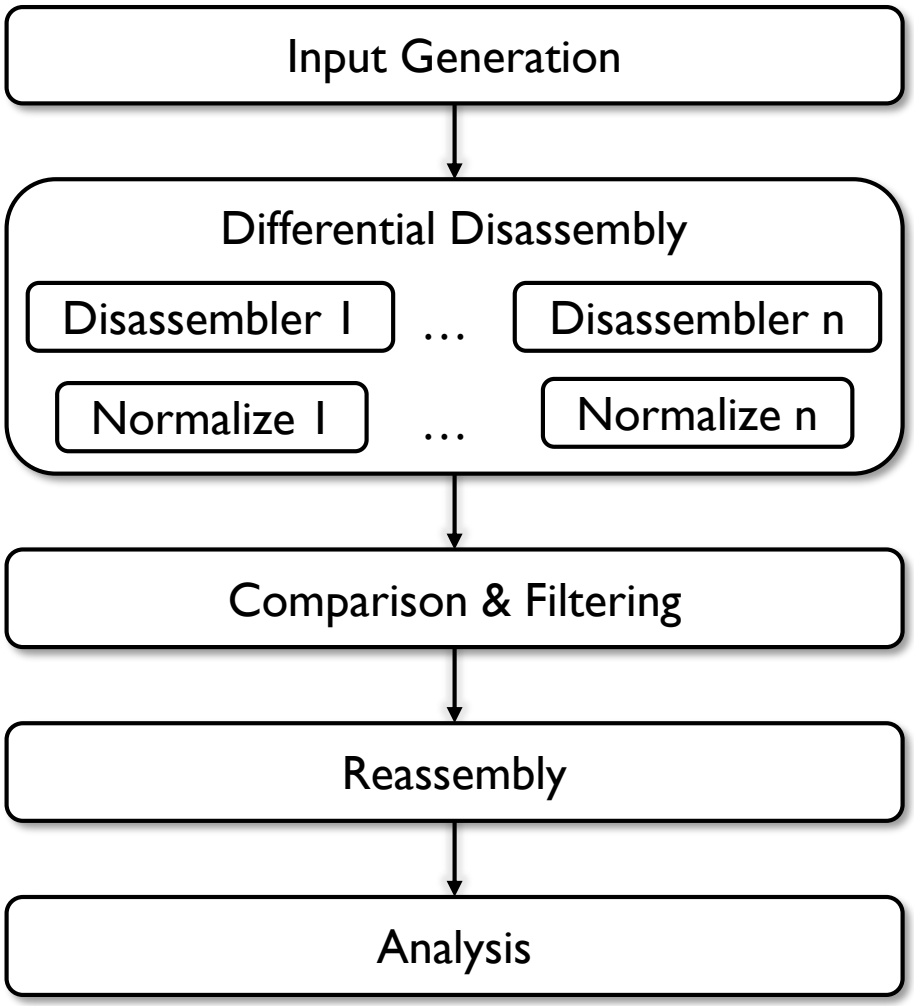
Disassemble object code with each disassembler and normalize results to uniform representation

Compare disassembled code and suppress duplicate differences

Reassemble output, looking for differences with object code

Determine which disassembly is correct

Workflow – Current State



Generalized, works for x86 and ARMv8. PPC64 lacks some register info

Differential disassembly tested on all "In-progress" decoders.

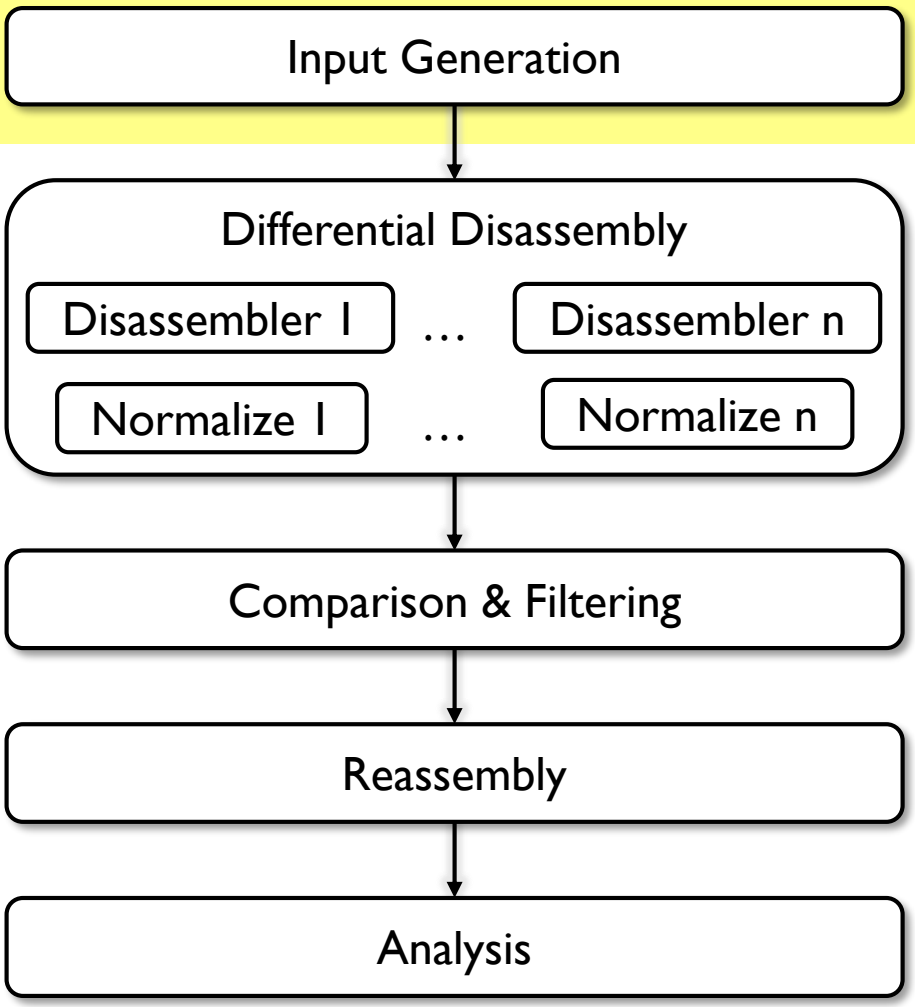
Normalization ongoing in each.

Generalized, works for x86, PPC64 and ARMv8. PPC64 lacks register info.

Primitive support for x86 and ARMv8

Preliminary results on x86 and ARMv8 outputs

Workflow – Current State



Generalized, works for x86 and ARMv8. PPC64 lacks some register info

Differential disassembly tested on all "In-progress" decoders.

Normalization ongoing in each.

Generalized, works for x86, PPC64 and ARMv8. PPC64 lacks register info.

Primitive support for x86 and ARMv8

Preliminary results on x86 and ARMv8 outputs

Input Generation – Observations

- Naïve brute force is too slow
  - x86 instructions are up to 15 bytes long
- There are much less than 2^{120} significantly different instructions
- Many instructions differ only slightly
  - Immediate values do not change meaning or decoding of instructions
  - Registers names (usually) do not change meaning or decoding of instructions

Input Generation – Observations

Disassemblers are likely to decode similar instructions all correctly or all incorrectly.

Binary Code	Decoded Instruction
1011 0100 1101 1111	mov $0xdf, %ah
1011 0100 0101 1111	mov $0x5f, %ah
1011 0110 1101 1111	mov $0xdf, %dh
1011 1100 1101 1111	movsbb (%rsi), (%rdi)

Not all bits flips are equally interesting, so can we find those that are most interesting?

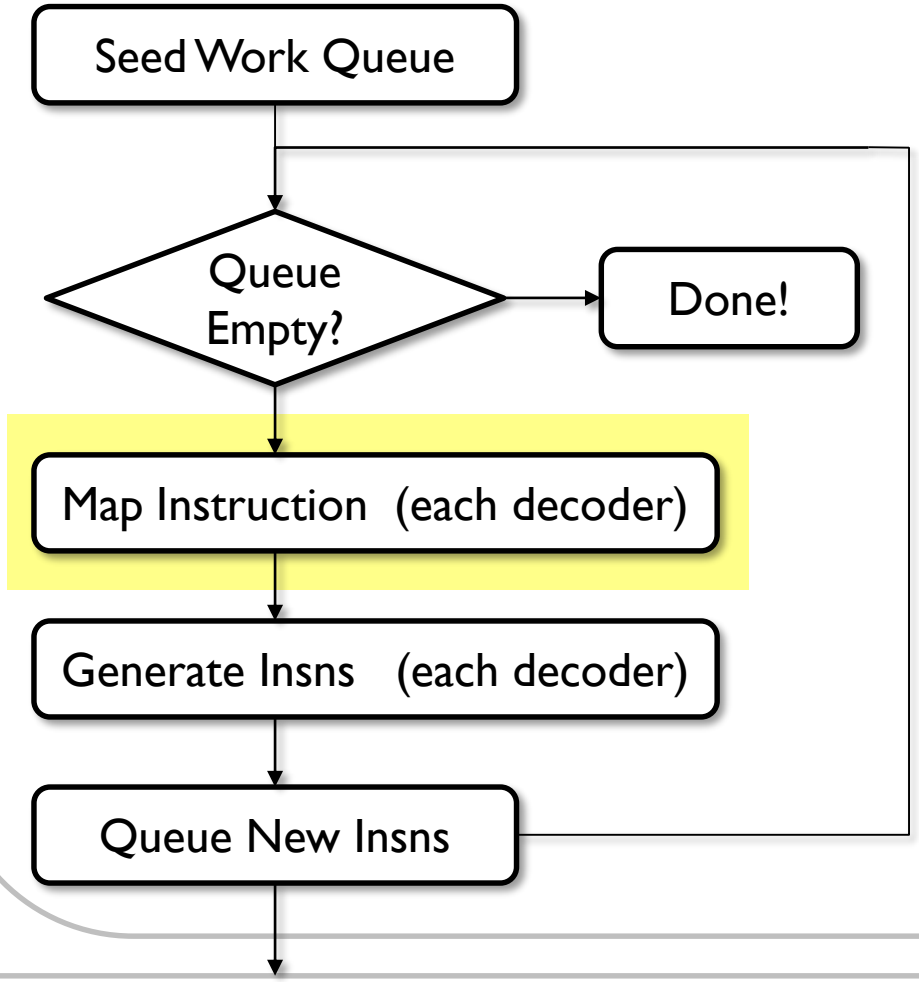
Input Generation – Observations

Goal: Find and ignore bits that encode only register names or immediate values.

```
mov $0xdf, %ah:

1011 0100 1101 1111
```

We can identify 11 of 16 bits that will not be interesting to vary

Input Generation



Add some random byte strings to the queue

Check if there are more instructions to evaluate

Find interesting bits to vary for new instructions

Flip interesting bits to create instructions

Add new instructions to the queue

Differential Disassembly

Producing a Map of Interesting Instruction Bits

```
Map:          *
Base Bits:  1011 0100 1101 1111
New Bits:   0011 0100 1101 1111

Base Insn:  mov    $0xdf,    %ah
New Insn:   xor    $0xdf,    %al
```

Producing a Map of Interesting Instruction Bits

```
Map:          **
Base Bits: 1011 0100 1101 1111
New Bits:  0111 0100 1101 1111

Base Insn: mov   $0xdf,   %ah
New Insn:  hlt
```

Producing a Map of Interesting Instruction Bits

```
Map:          ***
Base Bits:    1011 0100 1101 1111
New Bits:     1001 0100 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     xchg   %eax,     %esp
```

Producing a Map of Interesting Instruction Bits

```
Map:          ****
Base Bits:    1011 0100 1101 1111
New Bits:     1010 0100 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     movsbb (%rsi), (%rdi)
```

Producing a Map of Interesting Instruction Bits

```
Map:          **** *
Base Bits: 1011 0100 1101 1111
New Bits:  1011 1100 1101 1111

Base Insn: mov   $0xdf,    %ah
New Insn:  mov $0x6d5f5…, %esp
```

Producing a Map of Interesting Instruction Bits

```
Map:          **** *2
Base Bits:    1011 0100 1101 1111
New Bits:     1011 0000 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     mov    $0xdf,    %al
```

Producing a Map of Interesting Instruction Bits

```
Map:          **** *22
Base Bits:    1011 0100 1101 1111
New Bits:     1011 0110 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     mov    $0xdf,    %dh
```

Producing a Map of Interesting Instruction Bits

```
Map:          **** *222
Base Bits:    1011 0100 1101 1111
New Bits:     1011 0101 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     mov    $0xdf,    %ch
```

Producing a Map of Interesting Instruction Bits

```
Map:          **** *222 1
Base Bits:    1011 0100 1101 1111
New Bits:     1011 0100 0101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     mov    $0x5f,    %ah
```

The changed value `5f` has the same binary representation as the new bits, `0101 1111`, is a multiple of 8 bits, and occurs on a byte boundary, so we mark the next 8 bits

Producing a Map of Interesting Instruction Bits

```
Map:          **** *222 1111 1111
Base Bits:    1011 0100 1101 1111
New Bits:     1011 0100 1101 1111

Base Insn:    mov    $0xdf,    %ah
New Insn:     mov    $0x5f,    %ah
```

All bits after the decoded instruction length will be marked unused with a ‘U’.

Refining the Map

Sometimes, even a single field change is interesting

Bytes	Instruction	Length
83FE39	cmp $0x39, %esi	24 bits
81FE392D317C	cmp $0x7c312d39, %esi	48 bits

The number of fields changed is an insufficient criterion for detecting interesting bits.

We can re-map the changed instruction to learn structural information and find more interesting changes.

Input Generation – Making the Next Insns

We have a map, so how should we generate new instructions?

We know that only 5 bits produced interesting changes:

```
Insn: mov    $0xdf,    %ah
Map:  **** *222 1111 1111
```

We generate all sequences with every combination of 1 or 2 highlighted bits flipped.

Input Generation – Queueing New Insns

Issue: We do not want to re-evaluate redundant instructions

- The last instruction is only 1 or 2 bit flips away, so we could go right back if we do not record what we have tested

Solution: We record instruction *templates*, which are:

- Generic forms of an instruction based on opcode and operand types
- Identical for trivially different instructions
- Different for interestingly different instructions

Input Generation – Queueing New Insns

To make a template:

- Replace immediates with generic symbols:

```
Base Insn: mov    $0xdf,    %ah
Template:  mov    $0x,      %ah
```

- Replace registers with generic names:

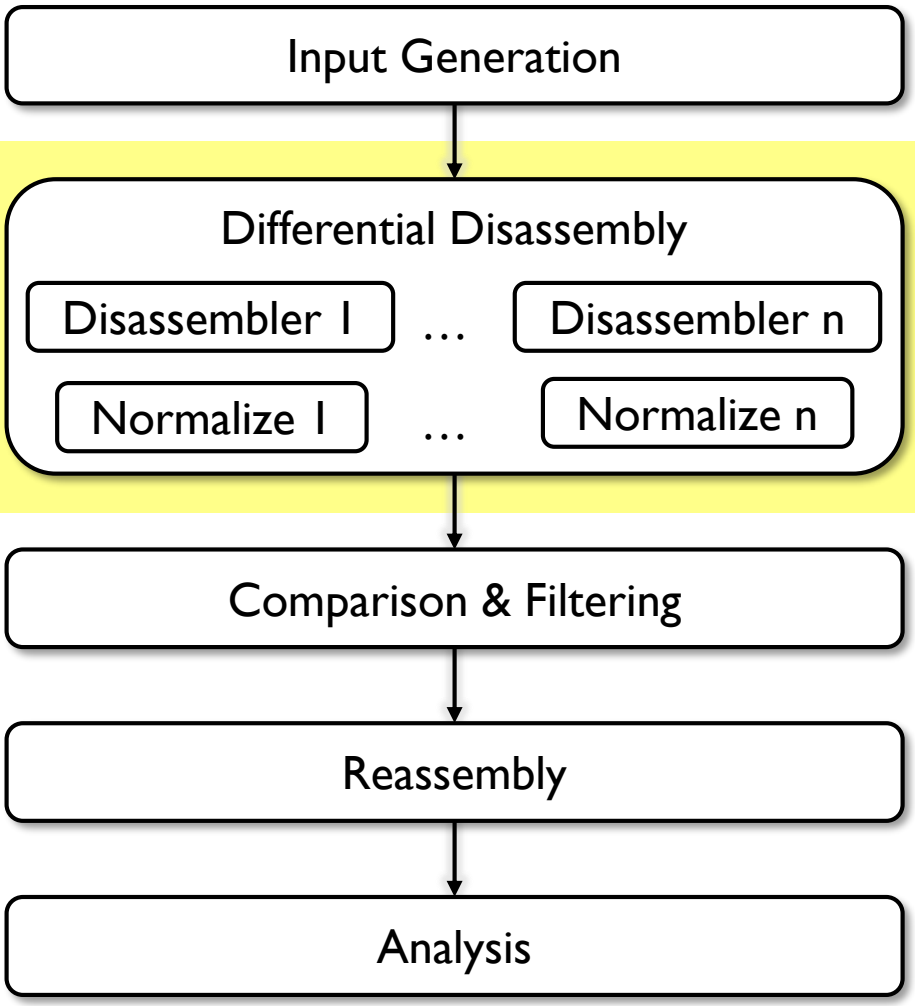
```
Base Insn: mov    $0xdf,    %ah
Template:  mov    $0x,      %gp_8bit
```

Templates coalesce instruction records, but require knowledge of register sets

Input Generation - Summary

- We generate test input using only the given decoders
- We don't rely on a single decoder to be correct
- We reduce input redundancy
- Our process does not heavily rely on a specific ISAs:
  - Opcode/operand placement doesn't matter
  - Byte order doesn't matter
  - Instruction length doesn't matter
- Unfortunately, we rely on register set information for templates.

Workflow



Create object code to disassemble

Disassemble object code with each disassembler and normalize results to uniform representation

Compare disassembled code and suppress duplicate differences

Reassemble output, looking for differences with object code

Determine which disassembly is correct

Differential Decoding

Goal: Compare results of multiple decoders to detect errors.

Caveats:

- Disassemblers can produce slightly different output for semantically identical instructions
- We do not assign correctness at this stage
- We do not rely on any disassembler to be correct

Differential Decoding – Normalization

Challenge: Decoders vary even for equivalent output.

Some differences are trivial:

- Spacing
- Comments
- Immediate base (hex vs. decimal)

We handle those differences first with a few generic normalization steps applied to all decoders.

Differential Decoding – Normalization

Other differences are a bit more complex:

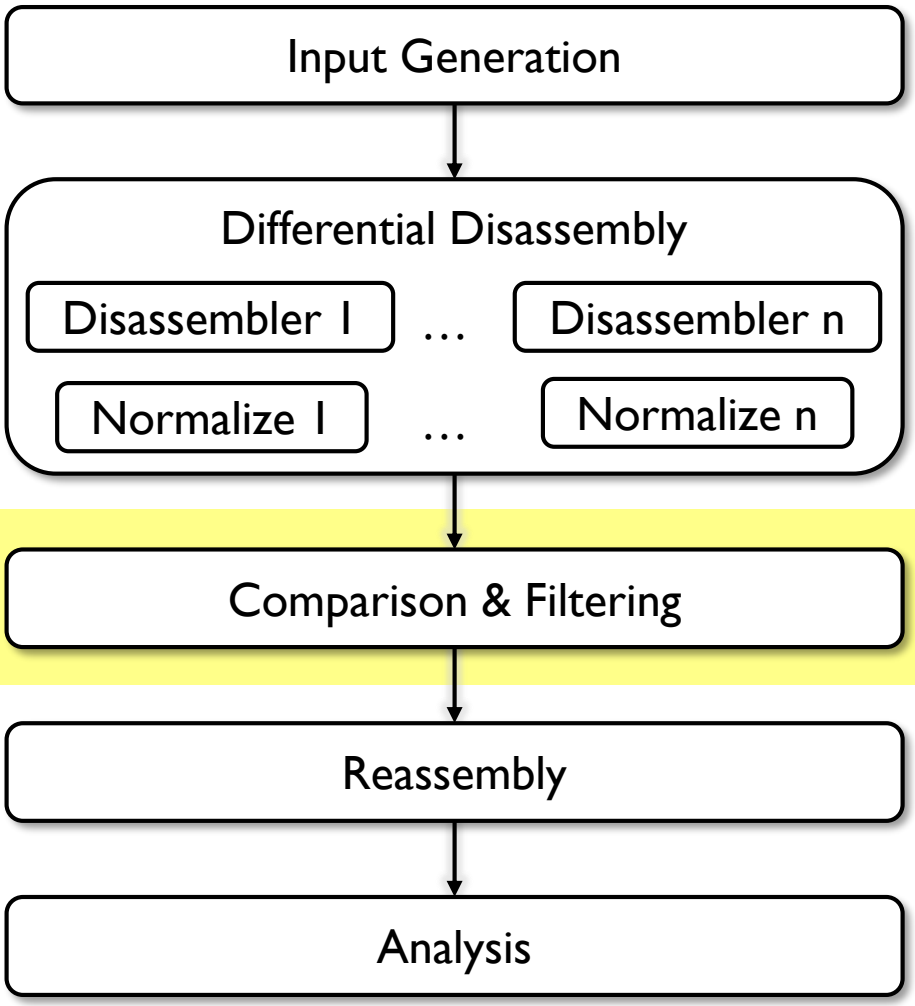
```
LLVM: movn x5, #0x97fc, lsl #16
GNU:  mov x5, #0xffffffff6803ffff

XED: fisttpw  %st0, -0x79c72fc5(%rcx)
GNU: fisttp -0x79c72fc5(%rcx)
```

Differ in:

- Equivalent opcodes that can affect operand encoding
- Operand padding (zero ext. vs. sign ext.)
- Implicit operands

These differences may require decoder-specific normalization.

Workflow



Create object code to disassemble

Disassemble object code with each disassembler and normalize results to uniform representation

Compare disassembled code and suppress duplicate differences

Reassemble output, looking for differences with object code

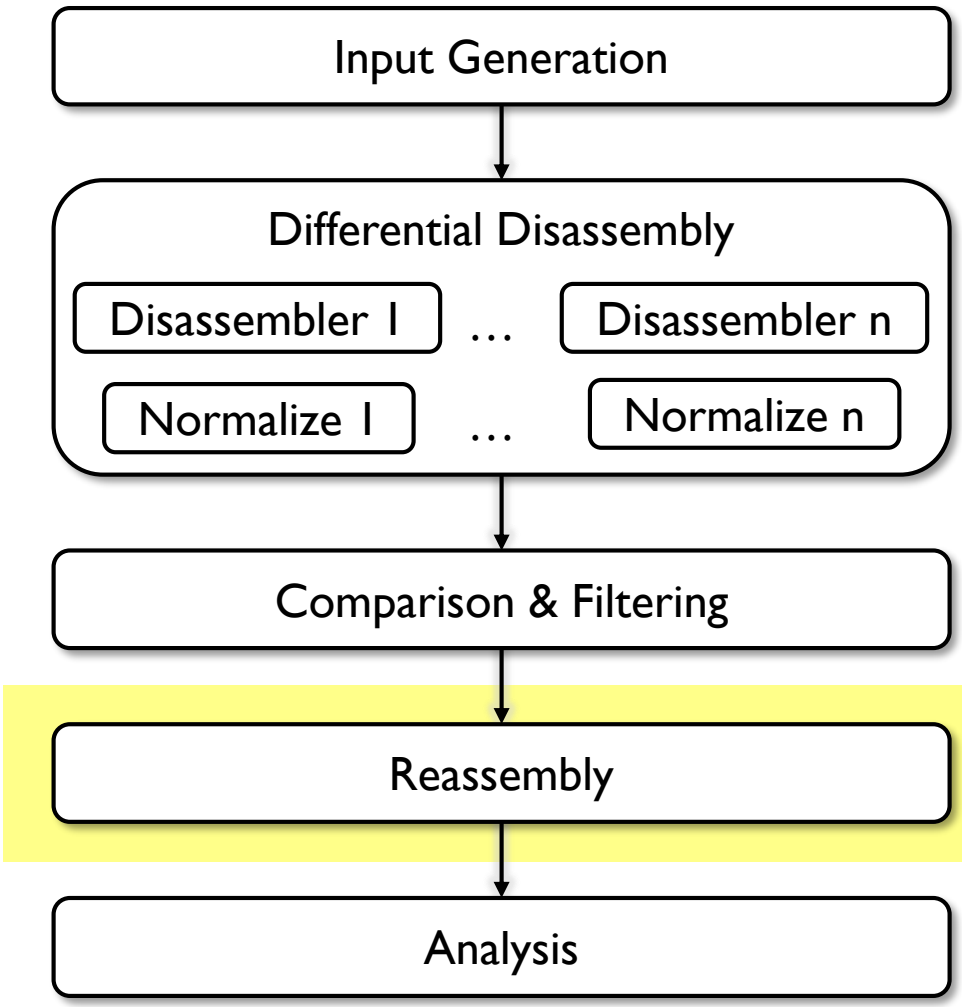
Determine which disassembly is correct

Comparison and Filtering

Comparison and filtering works by:

- Automatically checking aliases
  - Some register names are known aliases, and both are valid, so their difference should not be recorded.
- Producing templates
  - Each decoder output is made into a template
- Examining past templates
  - If the current combination of templates has been seen already, do not issue another report
- Recording this combination of templates

Workflow



Create object code to disassemble

Disassemble object code with each disassembler and normalize results to uniform representation

Compare disassembled code and suppress duplicate differences

Reassemble output, looking for differences with object code

Determine which disassembly is correct

Reassembly

Goal: We want to minimize the expert ISA knowledge needed during previous steps, which includes:

- Equivalent opcodes
- Equivalent register names
- Named constants
- Implicit operands

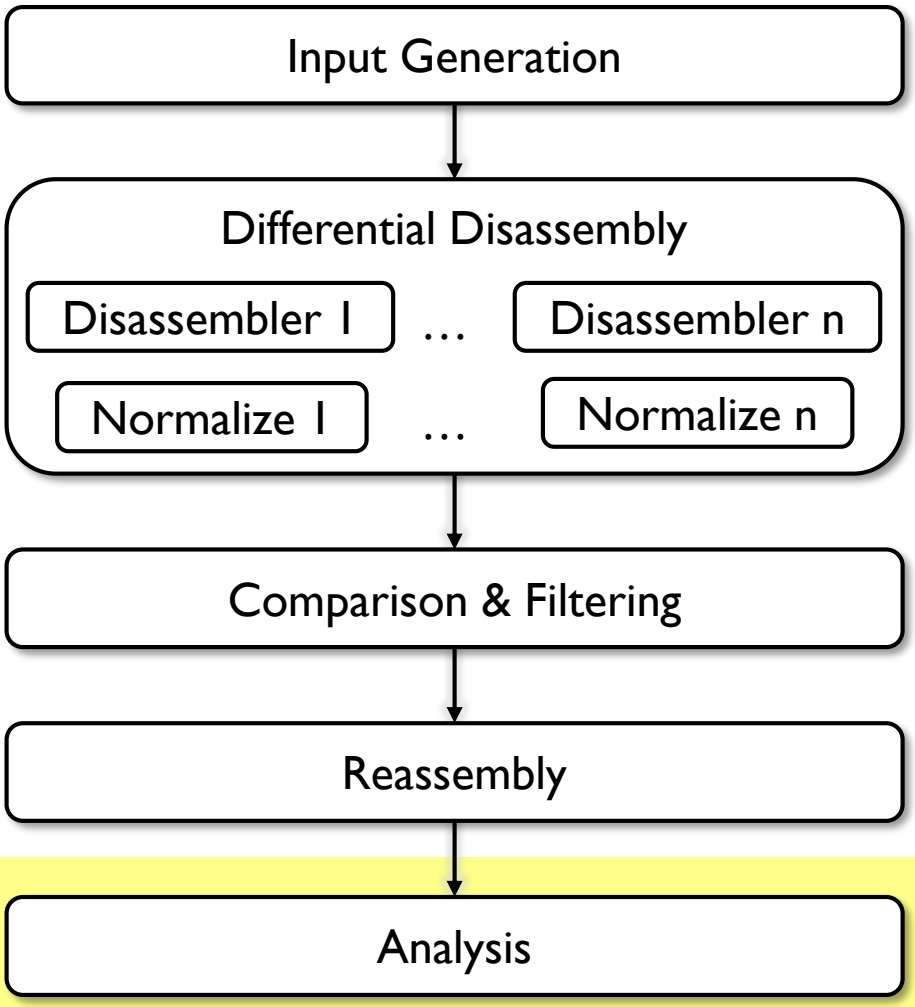
Solution: Learn aliases and implicit operands through reassembly

Reassembly

We can learn these parts by analyzing the output of reassembly.

- If decodings reassemble to the same bytes, they are equivalent and any different fields are likely aliases
- If decodings reassemble differently, they could have:
  - Ignored prefixes
  - Unused bits
  - An error
- If reassembly produces an error, either the decoder or the assembler is wrong

Workflow



Create object code to disassemble

Disassemble object code with each disassembler and normalize results to uniform representation

Compare disassembled code and suppress duplicate differences

Reassemble output, looking for differences with object code

Determine which disassembly is correct

Analysis

Manually examining differences allows us to:

- Verify correctness with ISA manual
- Execute instructions and compare processor state
- Logically group reported differences

Tradeoff:

Requires human involvement and significant time, but verifies correctness as thoroughly as necessary.

Results – x86 (Dyninst, GNU, XED)

Although normalization is incomplete, we have been able to test Dyninst against other decoders and found issues with:

- Invalid instruction handling
  - Asserts halted execution instead of returning an error
- Ignoring REX prefixes when computing operand size
- Decoding illegal instructions with lock prefixes as legal
- Opcodes, including:
  - Failure to translate XCHG to NOP in certain conditions
  - Missing decoding data for certain SHL instructions
- Incorrectly marking valid instructions as invalid involving at least half a dozen opcodes.

Results – ARMv8 (Dyninst, GNU, LLVM)

Testing was done during development of Dyninst ARMv8 support and highlighted:

- Issues recognizing invalid instructions
  - Found multiple asserts and segmentation faults
- Incorrect sign and zero extension
- Offset operand decoding (some are divided by 2 or 4)
- Special operand formatting (implicit adds, inversions)
- Failure to change operands for aliases
- Incorrect opcode aliasing in several opcodes including
  - MOV, SBFIZ, SBFIX, ORR, …

Results – ARMv8 (Dyninst, GNU, LLVM)

GNU Issues

- Incorrectly aliases ORR, changing semantics
- Decodes invalid LD1R, LD2R, LD3R and LD4R instructions as valid, ignoring a reserved bit
- Decodes invalid 16-bit floating point registers, affects nearly 50 opcodes.

LLVM Issues

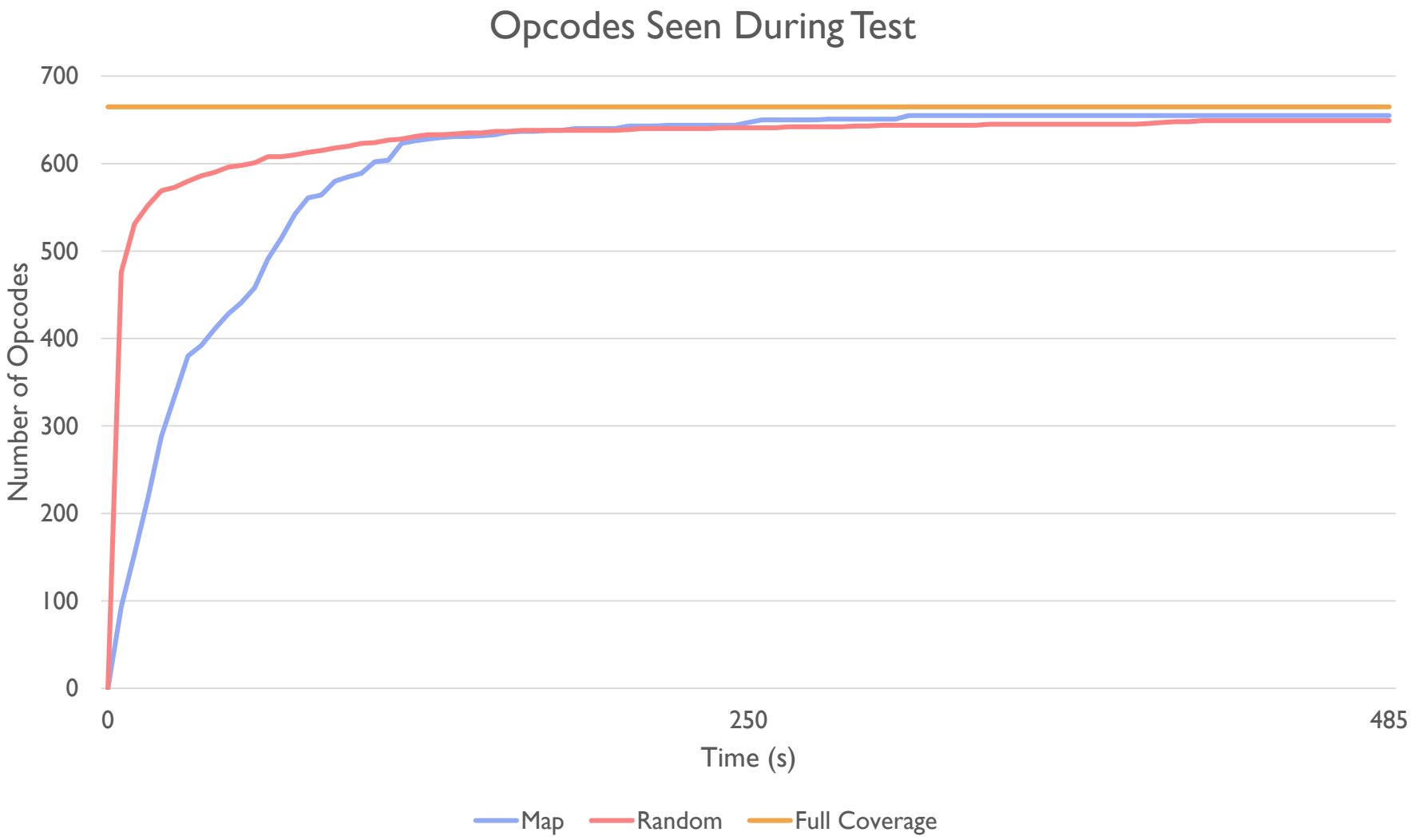
- Aliasing to invalid BFC instruction from semantic equivalent
- Inconsistent enforcement of "Should Be Zero" and "Should Be One" constraints across more than a dozen opcodes

Results – ARMv8 (Dyninst, GNU, LLVM)

Three scenarios were compared to test input generation:

- Random
  - 300 million decoded instructions
  - 50 minutes
- Brute Force
  - 12 billion decoded instructions (4 billion per decoder)
  - Distributed over 32 jobs from total 48 hours elapsed time
- Mapped (the method presented here)
  - 75 million decoded instructions (includes mapping steps)
  - 8 minutes

Results – ARMv8 (Dyninst, GNU, LLVM)



Results – ARMv8 (Dyninst, GNU, LLVM)

Mapped input generation terminated after 8 minutes because the work queue was emptied and no new templates were found

Time	Random	Mapped
8 Minutes	649 opcodes	655 opcodes (done)
50 Minutes	652 opcodes	655 opcodes

A brute force test of every 4-byte binary string revealed 665 opcodes.

Results – ARMv8 (Dyninst, GNU, LLVM)

Missed by Mapped Input

- MOVN, MOVZ
  - Aliased by MOV, these opcodes only appear with a few specific values for a 16-bit imm.
- CASP
  - Has many variants like CASPL, CASPAL, CASPA seen by both
- BLR
  - 27 bits fixed
- DCPS, DRPS , ERET
  - Exactly one 32-bit encoding

Missed by Random Input

- DSB, DMB, ESB, PSB
  - Various synchronization barriers, each with 28 bits fixed (less than 1 in 100 million)
- CLREX
  - Again, 28 bits fixed
- NOP, SEV, SEVL, WFE, WFI, YIELD
  - Exactly one 32-bit encoding

Ongoing Work

Input generation:

- Test special register values (all 0s, all 1s)
- Detect and vary opcode bits

Normalization:

- x86 and PPC have major normalization issues left

Differential Disassembly:

- Consider comparing internal semantic representations

Reassembly:

- Use error messages to help find decoder errors

Include new decoders – each one tests our assumptions

Our framework, Fleece is available at:

https://github.com/dyninst/tools/tree/master/fleece