IRFuzzer: Improving IR Fuzzing with more Diversified Input
Our experience fuzzing LLVM Backends

Yuyang (Peter) Rong
Stephen Neuendorffer
Hao Chen

AMD, UC Davis
AMD
UC Davis
What is fuzzing

Compilation (Instrumentation)

Executable to be fuzzed

Mutator / Generator

New input

Seed store

Behavior observation

Program feedback

Has new behavior

No new behavior

discard

Initial seed

Has new behavior

Initial seed

No new behavior

What are our expectations when fuzzing LLVM?

- **Completeness**: Sample the entire input space
- **Validity**: Sample only valid inputs
- **Precise feedback** (e.g.: Correctly identifies interesting input)
- **Throughput**, i.e. How fast is this loop?
What are our expectations when fuzzing LLVM?

- **Completeness**: Sample the entire input space
- **Validity**: Sample only valid inputs
- Precise feedback (e.g.: Correctly identifies interesting input)
- Throughput, i.e. How fast is this loop?
What are our expectations when fuzzing LLVM?

- Completeness: Sample the entire input space
- Validity: Sample only valid inputs
- **Precise feedback (e.g.: Correctly identifies interesting input)**
- Throughput, i.e. How fast is this loop?
What are our expectations when fuzzing LLVM?

- Completeness: Sample the entire input space
- Validity: Sample only valid inputs
- Precise feedback (e.g.: Correctly identifies interesting inputs)
- Throughput, i.e. How fast is this loop?

Diagram:

- Executable to be fuzzed
  - Compilation (Instrumentation)
  - Behavior observation
  - Seed store
  - Initial seed
  - Mutator / Generator
  - New input
  - Program feedback
  - Throughput, i.e. How fast is this loop?
  - Has new behavior
  - No new behavior
  - discard
## Overall comparison

<table>
<thead>
<tr>
<th>Tool</th>
<th>Generates</th>
<th>Feedback</th>
<th>Completeness</th>
<th>Validity</th>
<th>Throughput</th>
<th>Overall Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Csmith[1]</td>
<td>C code</td>
<td>None</td>
<td>Low</td>
<td>100%</td>
<td>Low</td>
<td>Poor</td>
</tr>
<tr>
<td>isel-fuzzer[2]</td>
<td>Scalar IR</td>
<td>CFG edge coverage</td>
<td>Low</td>
<td>100%</td>
<td>High, but hard to parallelize</td>
<td>Poor</td>
</tr>
<tr>
<td>AFLplusplus[3]</td>
<td>Byte array</td>
<td>Hashed CFG edge coverage</td>
<td>100%</td>
<td>&lt;0.01%</td>
<td>Highest</td>
<td>Poor</td>
</tr>
<tr>
<td>IRFuzzer (This work)</td>
<td>Scalar IR + more IR features</td>
<td>Hashed CFG edge coverage + MatcherTable Monitoring</td>
<td>High</td>
<td>100%</td>
<td>High</td>
<td>Good</td>
</tr>
</tbody>
</table>

[1]: https://embed.cs.utah.edu/csmith/
[2]: https://www.youtube.com/watch?v=UBbQ_s6hNgg
[3]: https://github.com/AFLplusplus/AFLplusplus
Structured mutator

• Key idea: by mutating IR in known valid ways we can avoid invalid inputs.
  • Faster to generate valid IR with LLVM API

• FuzzMutate
  • Scalar operations, limited CFG generation.

• Our improvements
  • More instructions supported
  • Vector operations
  • Function calls
    • Random function signature
    • Intrinsic call
  • Random CFG
    • Switch, br, and ret instructions
  • Global variables
Structured mutator

- **Key idea**: by mutating IR in known valid ways we can avoid invalid inputs.
  - Faster to generate valid IR with LLVM API

- **FuzzMutate**
  - Scalar operations, limited CFG generation.

- **Our improvements**
  - More instructions supported
  - Vector operations
  - Function calls
    - Random function signature
    - Intrinsic call
  - Random CFG
    - Switch, br, and ret instructions
  - Global variables

A piece of code generated by us.
Many places seems uncommon, but they are legal.
Structured mutator

- Key idea: by mutating IR in known valid ways we can avoid invalid inputs.
  - Faster to generate valid IR with LLVM API

- FuzzMutate
  - Scalar operations, limited CFG generation.

- Our improvements
  - More instructions supported
  - Vector operations
  - Function calls
    - Random function signature
    - Intrinsic call
  - Random CFG
    - Switch, br, and ret instructions
  - Global variables

```assembly
@G = global i16 256
@G.1 = global i32 42
define <1 x i16> @f() {
  BB:
    %RP = alloca <1 x i16>, align 2
    %8 = load <1 x i16>, <1 x i16>* %RP, align 2
    %A = alloca i1, align 1
    %L = load i1, i1* %A, align 1
    switch i1 %L, label %SW_D [
      i1 false, label %SW_C
    ]
  BB1: ; preds = %SW_C, %SW_D
    %A5 = alloca i32, align 4
    %L_C4 = load i32, i32* @G.1, align 4
    %A2 = alloca i1*, align 8
    %L_C = load i16, i16* @G, align 2
    %G = getelementptr i1, i1* %A, i16 %L_C
    %B = mul i32 65536, %L_C4
    store i1* %G, i1** %A2, align 8
    store i32 %B, i32* %A5, align 4
    ret <1 x i16> %8
  SW_D: ; preds = %BB
    br label %BB1
  SW_C: ; preds = %BB
    br label %BB1
}
```

A piece of code generated by us.
Many places seems uncommon, but they are legal.

- Key idea: by mutating IR in known valid ways we can avoid invalid inputs.
  - Faster to generate valid IR with LLVM API
Matcher table monitoring

```cpp
void <Arch>::SelectCode(SDNode *N) {
    static const unsigned char MatcherTable[......] = {
        /*42*/ OPC_CheckOpcode,
        /*43-44*/ TARGET_VAL(ISD::Constant)
        ...;
    }
    // TableGen-ed rules.
    SelectCodeCommon(N, MatcherTable, sizeof(MatcherTable));
}

void SelectCodeCommon(SDNode *N, char *Table) {
    while (true) {
        auto OpCode = Table[Idx++];
        switch (OpCode) {
            case OPC_CheckOpcode: {
                uint16_t Op = Table[Idx++];
                Op |= (unsigned short) Table[Idx++] << 8;
                bool Result = (Op == N->getOpcode());
                break;
            }
            case ......
        }
    }
}
```
Matcher table monitoring

```cpp
global <Arch>::SelectCode(SDNode *N) {
    static const unsigned char MatcherTable[......] = {
        /*42*/ OPC_CheckOpcode,
        /*43-44*/ TARGET_VAL(ISD::Constant)
    };
    // TableGen-ed rules.
    SelectCodeCommon(N, MatcherTable, sizeof(MatcherTable));
}
```

```cpp
void SelectCodeCommon(SDNode *N, char *Table) {
    while (true) {
        auto OpCode = Table[Idx++];
        switch (OpCode) {
            case OPC_CheckOpcode: {
                uint16_t Opc = Table[Idx++];
                Opc |= (unsigned short) Table[Idx++] << 8;
                bool Result = (Opc == N->getOpcode());
                break;
            }
            case ......
        }
    }
}
```

- Edge coverage
  - An input is interesting if a new edge is covered.

```
<table>
<thead>
<tr>
<th>Edge</th>
<th># Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>while(true)</td>
<td>10</td>
</tr>
<tr>
<td>case</td>
<td></td>
</tr>
<tr>
<td>OPC_CheckOpcode</td>
<td>1</td>
</tr>
<tr>
<td>case ......</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

Hashed CFG edge coverage
Matcher table monitoring

```cpp
void <Arch>::SelectCode(SDNode *N){
    static const unsigned char MatcherTable[......] = {
        /*42*/ OPC_CheckOpcode,
        /*43-44*/ TARGET_VAL(ISD::Constant)
    };
    // TableGen-ed rules.
    SelectCodeCommon(N, MatcherTable, sizeof(MatcherTable));
}

void SelectCodeCommon(SDNode *N, char *Table) {
    while (true) {
        auto OpCode = Table[Idx++];
        switch (OpCode){
            case OPC_CheckOpcode: {
                uint16_t Opc = Table[Idx++];
                Opc |= (unsigned short) Table[Idx++] << 8;
                bool Result = (Opc == N->getOpcode());
                break;
            }
        ...
    }
}
```

- Machine instructions don’t correlate with control flow.
- Instruction selection is driven by tables (TableGen).
- Our solution: Track coverage of values from matcher table.
- Edge coverage and table coverage work together.

```
<table>
<thead>
<tr>
<th>Index</th>
<th>isIndexed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>True</td>
</tr>
<tr>
<td>43</td>
<td>True</td>
</tr>
<tr>
<td>44</td>
<td>True</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Matcher table coverage
Matcher table monitoring

```cpp
void <Arch>::SelectCode(SDNode *N) {
    static const unsigned char MatcherTable[......] = {
        ... /*42*/ OPC_CheckOpcode,
        /*43-44*/ TARGET_VAL(ISD::Constant)
    }
    while (true) {
        auto OpCode = Table[Idx++];
        switch (OpCode) {
            case OPC_CheckOpcode: {
                uint16_t Opc = Table[Idx++];
                Opc |= (unsigned short) Table[Idx++] << 8;
                bool Result = (Opc == N->getOpcode());
            }
            case ......
        }
    }
}
```

- Machine instructions don’t correlate with control flow.
- Instruction selection is driven by tables (TableGen).
- Our solution: Track coverage of values from matcher table.
- Edge coverage and table coverage work together.

<table>
<thead>
<tr>
<th>Edge</th>
<th># Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>while(true)</td>
<td>10</td>
</tr>
<tr>
<td>case</td>
<td></td>
</tr>
<tr>
<td>case OPC_CheckOpcode</td>
<td>1</td>
</tr>
<tr>
<td>case ......</td>
<td>0</td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
</tbody>
</table>

Hashed CFG edge coverage

<table>
<thead>
<tr>
<th>Index</th>
<th>isIndexed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>True</td>
</tr>
<tr>
<td>43</td>
<td>True</td>
</tr>
<tr>
<td>44</td>
<td>True</td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
</tbody>
</table>

Matcher table coverage
Findings[4]

- 8 unimplemented features
  - GlobalIsel still has much work to do, even for mature architectures.
- 4 Infinite recursions result in compiler hangs
  - Fix-point algorithms sometime never converge.
- 15 bugs result in compiler crashes
  - Length 1 vector may cause unexpected problems.
  - Invalid or unexpected value in IR.
  - Assertion that can’t be guaranteed.
- 12 bugs fixed

[4]: https://github.com/DataCorrupted/LLVM-fuzzing-trophies
Conclusion

• **Fuzzing helps you find unexpected behaviors**
  • Untested code
    • Issue #57326: A buggy branch is not unit tested for **six** years.
  • Unclear documentation
    • Issue #57452: An index is treated as **SExt** and translates true into -1.
  • Unreliable assumptions
    • Issue #57404: Can’t multiply Boolean.
  • Unimplemented features
    • Cannot select/legalize MIR in GlobalIsel.

• **Specialized fuzzing can discover bugs better than general purpose fuzzing**
  • Parsers - AFL++
  • Frontend - Csmith
  • Middle end - IRFuzzer, opt-fuzzer
  • Backend - IRFuzzer, isel-fuzzer
Contacts – Any questions are welcome

**Yuyang (Peter) Rong**  
AMD, UC Davis  
PeterRong96@gmail.com

**Stephen Neuendorffer**  
AMD  
stephenn@amd.com

**Hao Chen**  
UC Davis  
chen@ucdavis.edu

[QR Code: Peter’s GitHub]