Cross-Architecture Lifter Synthesis

Rijnard van Tonder and Claire Le Goues
Cross-Architecture Lifter Synthesis

Rijnard van Tonder and Claire Le Goues
What is a Lifter?
What is a Lifter?

- x86
- ARM
- ...

Lower level code
What is a Lifter?

- x86
- ARM
- ...

Lower level code

Architecture-specific
What is a Lifter?

- x86
- ARM
- ...

Intermediate Representation
What is a Lifter?

- x86
- ARM
- ...

Intermediate Representation

Higher level code
What is a Lifter?

- x86
- ARM
- ...

Intermediate Representation

Higher level code

Architecture-agnostic
Why Lifters?
Why Lifters?

• Binary analysis
Why Lifters?

• Binary analysis

• Simpler semantic abstraction
  – E.g., symbolic execution
Why Lifters?

• Binary analysis

• Simpler semantic abstraction
  – E.g., symbolic execution

• Reuse analysis components
  – Control flow graph construction
  – Single Static Assignment (SSA) conversion
Compilers and Decompilers

- Compilers lose information
  - Translation is not a bijection
Compilers and Decompilers

• Compilers lose information
  – Translation is not a bijection

• Decompilers recover more features
  – Often architecture-specific (e.g., ABI)
The Problem: Translating Multiple Architectures

- x86
- ARM
- ...

Intermediate Representation

Manual
The Problem: Translating Multiple Architectures

- x86
- ARM
- ...

Intermediate Representation

Manual

1000s of manual pages
Our Goal: Automate across Architectures

- x86
- ARM
- ...

Intermediate Representation
Our Goal: Automate across Architectures

- x86
- ARM
- ...

Intermediate Representation

Our Goal: Automate across Architectures

- x86
- ARM
- ...

Intermediate Representation

Cross-Architecture reuse
Our Goal: Automate across Architectures

x86

ARM

...  

Intermediate Representation

Bug-finding Analyses “for free”
Our Goal: Automate across Architectures

Dataflow framework
VC Generation
Bug-finding Analyses "for free"
Taint analysis
What We Do
What We Do

• IR translation as a Syntax-Guided Synthesis problem
What We Do

• IR translation as a Syntax-Guided Synthesis problem

\[ \forall x . \varphi(x, P(x)) \]
What We Do

• IR translation as a Syntax-Guided Synthesis problem

\[ \forall x . \varphi(x, P(x)) \equiv \forall x . \text{oracle}(x) = P(x) \]
What We Do

• IR translation as a Syntax-Guided Synthesis problem

\[ \forall x. \varphi(x, P(x)) \equiv \forall x. \text{oracle}(x) = P(x) \]
What We Do

• IR translation as a Syntax-Guided Synthesis problem

$$\forall x . \varphi(x, P(x)) \equiv \forall x . \text{oracle}(x) = P(x)$$

Dynamic Input/Output Pairs
What We Do

• IR translation as a Syntax-Guided Synthesis problem

\[ \forall x . \varphi(x, P(x)) \equiv \forall x . \text{oracle}(x) = P(x) \]
What We Do

- IR translation as a Syntax-Guided Synthesis problem

\[ \forall x . \varphi(x, P(x)) \equiv \forall x . \text{oracle}(x) = P(x) \]

Key Insight:
Learn IR Sketches from existing Lifter Productions
Learning Sketch Templates: Example
Learning Sketch Templates: Example

ARM
add R3, R0
Learning Sketch Templates: Example

ARM
add R3, R0

SOURCE
Native Instruction
Learning Sketch Templates: Example

ARM
add R3, R0

SOURCE
Native Instruction

Op codes and register names
Learning Sketch Templates: Example

ARM
add R3, R0

SOURCE
Native Instruction

Manual translation rules

LIFT
Learning Sketch Templates: Example

ARM
add R3, R0

SOURCE
Native Instruction

LIFT

Manual translation rules

Intel x86: 1500-page manual
ARM: 1000-page manual
Learning Sketch Templates: Example

ARM
add R3, R0

SOURCE
Native Instruction

LIFT

TARGET
IR Instruction

R3 := R3 + R0
Learning Sketch Templates: Example

**ARM**

add R3, R0

**SOURCE**

Native Instruction

**LIFT**

**TARGET**

R3 := R3 + R0

**IR with operational semantics**
Learning Sketch Templates: Example

ARM
add R3, R0

IR SKETCH
_ := _ + _

LIFT

R3 := R3 + R0
Learning Sketch Templates: Example

ARM
add R3, R0

LIFT

IR SKETCH
_ := _ + _

R3 := R3 + R0

Create sketches from concrete expressions
Learning Sketch Templates: Example

- Learn templates from lifter output for a **supported** architecture to synthesize one for an **unsupported** architecture
Learning Sketch Templates: Example

ARM
add R3, R0

LIFT

IR
R3 := R3 + R0

IR SKETCH
_ := _ + _

Create sketches from concrete expressions
Learning Sketch Templates: Example

ARM
add R3, R0

MIPS
add $1, $2

IR
R3 := R3 + R0

IR SKETCH
_ := _ + _

LIFT
Learning Sketch Templates: Example

ARM
add R3, R0

MIPS
add $1, $2

Unsupported Architecture

IR SKETCH
_ := _ + _

LIFT

IR
R3 := R3 + R0
Learning Sketch Templates: Example

**ARM**
add R3, R0

**MIPS**
add $1, $2

Unsupported Architecture

**IR**
R3 := R3 + R0

**IR SKETCH**
_ := _ + _

**LIFT**

**IR**
$1 := $1 + $2
Learning Sketch Templates: Example

MIPS
add $1, $2

Unsupported Architecture

IR SKETCH
_ := _ + _

IR
$1 := $1 + $2
Learning Sketch Templates: Example

MIPS
add $1, $2

Unsupported Architecture

IR SKETCH
_=,_+__

SYNTHESIS

IR
$1 := $1 + $2
Learning Sketch Templates: Example

MIPS
add $1, $2

Unsupported Architecture

IR
$1 := $1 + $2

I/O from executing add $1, $2

IR SKETCH

_ := _ + _
Learning Sketch Templates: Example

MIPS
add $1, $2

Unsupported Architecture

IR
$1 := $1 + $2

I/O from executing add $1, $2

SYNTHESIS

IR SKETCH
_ := _ + _
Learning Sketch Templates: Example

ARM
add R3, R0

MIPS
add $1, $2

Unsupported Architecture

IR
R3 := R3 + R0

IR
$1 := $1 + $2

IR SKETCH
_ := _ + _

SYNTHESIS

LIFT

I/O from executing add $1, $2
Learning Sketch Templates: Example

- Learn templates from lifter output for a supported architecture to synthesize one for an unsupported architecture
Learning Sketch Templates: Example

• Learn templates from lifter output for a **supported** architecture to synthesize one for an **unsupported** architecture

• Code is “natural” ➔ shared semantic properties
Learning Sketch Templates: Example

• Learn templates from lifter output for a supported architecture to synthesize one for an unsupported architecture

• Code is “natural” shared semantic properties

• Sketch templates exploit structural qualities to guide synthesis
Synthesis Approach
Synthesis Approach

Native Behavior

- I/O pairs from native execution
Synthesis Approach

Native Behavior

• I/O pairs from native execution

• QEMU and PIN traces
Synthesis Approach

Native Behavior

• I/O pairs from native execution

• QEMU and PIN traces

IR Target

• Binary Analysis Platform (BAP) IR
Synthesis Approach

Native Behavior

• I/O pairs from native execution

• QEMU and PIN traces

IR Target

• Binary Analysis Platform (BAP) IR

• IR Interpreter
Synthesis Approach

Native Behavior

• I/O pairs from native execution

• QEMU and PIN traces

\[ \langle \sigma_T, I_T, \emptyset \rangle \xrightarrow{T} \langle \sigma'_T, -, E_T \rangle \]

IR Target

• Binary Analysis Platform (BAP) IR

• IR Interpreter

\[ \langle \sigma_{IR}, I_{IR}, \emptyset \rangle \xrightarrow{IR^*} \langle \sigma'_{IR}, -, E_{IR} \rangle \]
Synthesis Approach

$$\forall \sigma_T, I_T, \emptyset \in T \quad \mapsto \quad \forall \sigma'_T, \sim, E_T$$

$$\forall \sigma_{IR}, I_{IR}, \emptyset \in IR \quad \mapsto \quad \forall \sigma'_{IR}, \sim, E_{IR}$$
Synthesis Approach

\[ \langle \sigma_T, I_T, \emptyset \rangle \xrightarrow{T} \langle \sigma'_T, -, \mathcal{E}_T \rangle \]

\[ \langle \sigma_{IR}, I_{IR}, \emptyset \rangle \xrightarrow{IR^*} \langle \sigma'_{IR}, -, \mathcal{E}_{IR} \rangle \]
Synthesis Approach

\[
\langle \sigma_T, \mathcal{I}_T, \emptyset \rangle \xrightarrow{T} \langle \sigma'_T, -, \mathcal{E}_T \rangle
\]

\[
\langle \sigma_{IR}, \mathcal{I}_{IR}, \emptyset \rangle \xrightarrow{IR^*} \langle \sigma'_{IR}, -, \mathcal{E}_{IR} \rangle
\]
Synthesis Approach

\[ \langle \sigma_T, I_T, \emptyset \rangle^T \sim \langle \sigma'_T, \neg, E_T \rangle \]
\[ \langle \sigma_{IR}, I_{IR}, \emptyset \rangle^{IR*} \sim \langle \sigma'_{IR}, \neg, E_{IR} \rangle \]
Synthesis Approach

\[ R2 := R0 \]
\[ [(R,\text{REG},R0,0x1),(W,\text{REG},R2,0x1)] \]
Inferring Semantics from I/O pairs

SOURCE

Native Instruction

\text{add\ R3,\ R0}
Inferring Semantics from I/O pairs

**SOURCE**

Native Instruction

```
add R3, R0
```

**Execution**

**Input**

```
R3 = 2
R0 = 2
```

**Output**

```
R3 = 4
R0 = 2
```
Inferring Semantics from I/O pairs

**SOURCE**

Native Instruction

add R3, R0

**Execution**

**Input**

- R3 = 2
- R0 = 2

**Output**

- R3 = 4
- R0 = 2
Synthesizing Translation

**TARGET**

IR Instruction

??

**Execution**

**Input**

R3 = 2
R0 = 2

**Output**

R3 = 4
R0 = 2
Synthesizing Translation

**TARGET**

**IR Instruction**

R3, R0, R3

**Execution**

**Input**

- R3 = 2
- R0 = 2

**Output**

- R3 = 4
- R0 = 2
Syntax-guided Synthesis

TARGET

IR Instruction

R3, R0, R3

_ := _ + _
Syntax-guided Synthesis

TARGET
IR Instruction

Sketch Templates

R3, R0, R3
_ := _ + _

R3 := R0 + R3
R3 := R0 - R3
R3 := R0 * R3
R3 := R0 << R3

...
Syntax-guided Synthesis

TARGET
IR Instruction

Sketch Templates

R3, R0, R3

_ := _ + _

_ := _ - _

_ := _ * _

_ := _ << _

... 

R3 := R0 + R3
R3 := R0 - R3
R3 := R0 * R3
R3 := R0 << R3

Exhaustive Enumeration
Syntax-guided Synthesis

TARGET
IR Instruction

Sketch Templates

Exhaustive Enumeration

Permute adjacent operands

R3, R0, R3

_ := _ + _

R3 := R0 + R3

_ := _ - _

R3 := R0 - R3

_ := _ * _

R3 := R0 * R3

_ := _ << _

R3 := R0 << R3

...
Syntax-guided Synthesis

TARGET

IR Instruction

R3 := R0 + R3

IR Interpreter

Input

R3 = 2
R0 = 2

Output

R3 = 4
R0 = 2
Syntax-guided Synthesis

Native Execution

Input

R3 = 2
R0 = 2

Expected Output

R3 = 4
R0 = 2

IR Interpreter

Input

R3 = 2
R0 = 2

Output

R3 = 4
R0 = 2
Validating Correctness

TARGET

IR Instruction

R3 := R0 * R3

IR Interpreter

Input

R3 = 2
R0 = 2

Expected Output

R3 = 4
R0 = 2
Validating Correctness

TARGET

IR Instruction

R3 := R0 * R3

Oops...

IR Interpreter

Input

R3 = 2
R0 = 2

Expected Output

R3 = 4
R0 = 2
Validating Correctness

TARGET

IR Instruction

R3 := R0 * R3

Oops...
Validating Correctness

**TARGET**
IR Instruction

\[ R3 := R0 \times R3 \]

**Native Execution**

**Input**
- \( R3 = 3 \)
- \( R0 = 2 \)

**Expected Output**
- \( R3 = 5 \)
- \( R0 = 2 \)
Validating Correctness

**TARGET**

IR Instruction

R3 := R0 * R3

R3 = 6

**Native Execution**

Input

R3 = 3
R0 = 2

Expected Output

R3 = 5
R0 = 2
Validating Correctness

TARGET

IR Instruction

\[ \text{R3} := \text{R0} \times \text{R3} \]

Native Execution

Input

\[ \begin{align*}
\text{R3} &= 3 \\
\text{R0} &= 2
\end{align*} \]

Expected Output

\[ \begin{align*}
\text{R3} &= 5 \\
\text{R0} &= 2
\end{align*} \]

R3 = 6

Invalidate sketch
Lifter Synthesis System

SOURCE Programs → LEARN SKETCHES → SKETCHES

TARGET Programs → TARGET DISASSEMBLER → SYNTHESIS LOOP → TARGET LIFTER

TARGET Programs → NATIVE EXECUTION → TRACES
Experimental Setup
Experimental Setup

• Synthesize unsupported lifter target (MIPS)
Experimental Setup

• Synthesize unsupported lifter target (MIPS)

• Collect dynamic traces from arithmetic-heavy benchmark
  – 5 Hacker’s delight programs
Experimental Setup

• Synthesize unsupported lifter target (MIPS)

• Collect dynamic traces from arithmetic-heavy benchmark
  – 5 Hacker’s delight programs

• Mine IR sketches from Coreutils, lifted from ARM and x86
Results: Analysis Reuse
Results: Analysis Reuse

• Taint analysis for warn-unused-result bugs
Results: Analysis Reuse

• Taint analysis for warn-unused-result bugs
  – Forget to check memory is allocated
  – Forget to check privileges are dropped
Results: Analysis Reuse

- Taint analysis for warn-unused-result bugs
  - Forget to check memory is allocated
  - Forget to check privileges are dropped

- Ran on 30 binaries of COTS D-Link router
Results: Analysis Reuse

• Taint analysis for warn-unused-result bugs
  – Forget to check memory is allocated
  – Forget to check privileges are dropped

• Ran on 30 binaries of COTS D-Link router

• 29 bugs, 2 false positives
## Results: Analysis Reuse

<table>
<thead>
<tr>
<th>Name</th>
<th>#</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pppd</td>
<td>1</td>
<td>fwrite</td>
</tr>
<tr>
<td>iptables</td>
<td>3</td>
<td>fwrite</td>
</tr>
<tr>
<td>rdnssd</td>
<td>1</td>
<td>setsockopt</td>
</tr>
<tr>
<td>ntpclient</td>
<td>1</td>
<td>send</td>
</tr>
<tr>
<td>speedtest</td>
<td>2</td>
<td>system, fgets</td>
</tr>
<tr>
<td>timer</td>
<td>2</td>
<td>read, shutdown</td>
</tr>
<tr>
<td>wakeOnLanProxy</td>
<td>1</td>
<td>shutdown</td>
</tr>
<tr>
<td>wcnd</td>
<td>8</td>
<td>system</td>
</tr>
</tbody>
</table>
## Results: Analysis Reuse

<table>
<thead>
<tr>
<th>Name</th>
<th>#</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pppd</td>
<td>1</td>
<td>fwrite</td>
</tr>
<tr>
<td>iptables</td>
<td>3</td>
<td>fwrite</td>
</tr>
<tr>
<td>rdnssd</td>
<td>1</td>
<td>setsockopt</td>
</tr>
<tr>
<td>ntpclient</td>
<td>1</td>
<td>send</td>
</tr>
<tr>
<td>speedtest</td>
<td>2</td>
<td>system, fgets</td>
</tr>
<tr>
<td>timer</td>
<td>2</td>
<td>read, shutdown</td>
</tr>
<tr>
<td>wakeOnLanProxy</td>
<td>1</td>
<td>shutdown</td>
</tr>
<tr>
<td>wcnd</td>
<td>8</td>
<td>system</td>
</tr>
</tbody>
</table>

```c
uint32_t data[12];
struct timeval now;

memset(data, 0, sizeof(data));
data[0] = htonl (    ( LI << 30 ) | ( VN << 27 ) | (    ( STRATUM << 16 ) | ( POLL << 8 );
data[1] = htonl(1<<16); /* Root Delay */
data[2] = htonl(1<<16); /* Root Dispersion */
gettimeofday(&now,NULL);
data[10] = htonl(now.tv_sec + JAN_1970);
data[11] = htonl(NTPFRAC(now.tv_usec));
send(sd,data,48,0);
```
## Results: Analysis Reuse

<table>
<thead>
<tr>
<th>Name</th>
<th>#</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>pppd</td>
<td>1</td>
<td>fwrite</td>
</tr>
<tr>
<td>iptables</td>
<td>3</td>
<td>fwrite</td>
</tr>
<tr>
<td>rdnssd</td>
<td>1</td>
<td>setsockopt</td>
</tr>
<tr>
<td>ntpclient</td>
<td>1</td>
<td>send</td>
</tr>
<tr>
<td>speedtest</td>
<td>2</td>
<td>system, fgets</td>
</tr>
<tr>
<td>timer</td>
<td>2</td>
<td>read, shutdown</td>
</tr>
<tr>
<td>wakeOnLanProxy</td>
<td>1</td>
<td>shutdown</td>
</tr>
<tr>
<td>wcnd</td>
<td>8</td>
<td>system</td>
</tr>
</tbody>
</table>

```c
uint32_t data[12];
struct timeval now;

memset(data, 0, sizeof(data));
data[0] = htonl (  
    ( LI << 30 ) | ( VN << 27 ) | (  
        ( STRATUM << 16) | ( POLL << 8 ) |
    data[1] = htonl(1<<16); /* Root Delay */
data[2] = htonl(1<<16); /* Root Dispersion*/
gmtimeofday(&now,NUL);
data[10] = htonl(now.tv_sec + JAN_1970);  
data[11] = htonl(NTPFRAC(now.tv_usec));

send(sd,data,48,0);
```

No false positives on OpenSSL
Results: Synthesis
Results: Synthesis

• End-to-end synthesis takes 58 seconds
  – mining sketches, processing traces, and lifter synthesis

• 29 sketches per instruction, on average
  – Synthesis converges after 2 input output pairs, on average

• Lifts 84.8% of native instructions.
  – Missed example: “load upper immediate”
Generalizing across Architectures
Generalizing across Architectures

- IR Sketches mined from ARM and x86 are common to both

<table>
<thead>
<tr>
<th>%</th>
<th>ARM-IR</th>
<th>%</th>
<th>X86-IR</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.9</td>
<td>_v := _v</td>
<td>11.9</td>
<td>_v := _i</td>
</tr>
<tr>
<td>14.5</td>
<td>_v := _i</td>
<td>8.6</td>
<td>jmp _i</td>
</tr>
<tr>
<td>8.9</td>
<td>jmp _i</td>
<td>5.1</td>
<td>_v := mem[_v + _i]</td>
</tr>
<tr>
<td>7.0</td>
<td>_v := mem[_v + _i]</td>
<td>4.4</td>
<td>mem[_v + _i] := _v</td>
</tr>
<tr>
<td>5.6</td>
<td>_v := _v = _i</td>
<td>4.2</td>
<td>_v := _v = _i</td>
</tr>
<tr>
<td>5.6</td>
<td>_v := hi : 1[_v]</td>
<td>4.2</td>
<td>_v := hi : 1[_v]</td>
</tr>
<tr>
<td>5.5</td>
<td>mem[_v + _i] := _v</td>
<td>4.0</td>
<td>mem[_v + _i] := _v</td>
</tr>
<tr>
<td>4.6</td>
<td>_v := _v - _i</td>
<td>3.9</td>
<td>_v := _v - _i</td>
</tr>
</tbody>
</table>

- Similar MIPS accuracy from either ARM or x86 sketches
Discussion

• How good “out-of-box”?

• Improve recovery with nondeterministic sketch search and generation

• Expand to more architectures
  – SPARC, PPC, ...

• Complement automatic synthesis with (reduced) manual effort
  – Verify and fix manual translation
Summary

Lifter Synthesis

Verify with Dynamic Traces

\[ R2 := R0 \]

\[ [(R, REG, R0, 0x1), (W, REG, R2, 0x1)] \]

Events from execution

Analysis Reuse for Previously Unsupported Instruction Set

Learn Sketches