Cross-Architecture Lifter Synthesis

Rijnard van Tonder and Claire Le Goues

Carnegie Mellon University,
`{rvantonder,clegoues}@cs.cmu.edu`

Abstract. Code translation is a staple component of program analysis. A lifter is a code translation unit that translates low-level code to a higher-level intermediate representation (IR). Lifters thus enable a host of static and dynamic analyses for such low-level code. However, writing a lifter is a tedious manual process which must be repeated for every architecture an analysis aims to support. We introduce cross-architecture lifter synthesis, a novel approach that automatically synthesizes lifters for previously unsupported architectures. Our insight is that lifter synthesis can be bootstrapped with existing IR sketches that exploit the shared semantic properties of heterogeneous architecture instruction sets. We show that our approach automates a significant amount of translation effort for a previously unsupported instruction set, and that it enables discovery of new bugs on new architecture targets through reuse of an existing IR-based analysis.

1 Introduction

Intermediate representations (IRs) are a staple component of compilers [20, 23] and program analyses [5, 8, 11, 18]. Code translation can generate programs in an IR from high level source languages (e.g., compilers) or low level machine code (e.g., decompilers). A *lifter* is a code translation unit that emits a higher level, architecture-agnostic intermediate representation of architecture-specific lower-level code. Lifters are central to low-level code analysis because they enable reuse of architecture-agnostic analyses at the IR level (e.g., taint analysis, constraint generation) [14, 22], and provide essential high level abstractions for program analysis (CFG and function recovery) [9].

However, writing the translation layer for an IR is onerous, requiring manual translation of architecture-specific instructions (e.g., for x86, ARM, MIPS) to the target IR while preserving the native semantics. Modeling the semantics of a new instruction set requires an engineer to consult instruction manuals numbering up to 1,000s of pages per architecture [14, 15]. Recent work raises the importance of automating the lifting process [14]. In our own past work, we identify the potential to reuse existing analyses in the IR for new architectures,[1] but are faced with the undesirable prospect of writing new lifters from scratch.

We propose a novel synthesis technique to automate the lifting translation process, with a goal of producing an IR program usable for further program

[1] e.g., https://opam.ocaml.org/packages/bap-warn-unused/bap-warn-unused.1.3.0

analysis (e.g., to find bugs). At a high level, our technique uses inductive synthesis over finite input-output pairs of native instructions to infer semantically equivalent instructions in the IR. We verify the correctness of synthesized instructions by executing the IR (under associated operational semantics) and comparing computational events with that of native execution. Our approach learns *sketches* (templates) from existing IR instructions, that then drive synthesis. Two key insights enable our synthesis approach. First, software exhibits a "natural" property: code structure is repetitive and predictable [16]. Instruction architectures are inherently heterogeneous, but they share similar semantic operations (e.g., move instructions, arithmetic operations). Our approach mines sketches from existing IR programs which preserve this underlying shared semantics. Moreover, because instructions are not distributed uniformly (e.g., move instructions are more common) [6, 16], our approach (1) extends across heterogeneous architectures and (2) achieves high translation coverage. Second, we parameterize synthesis by exploiting statement structure to produce an efficient search.

Prior work only partially addresses the challenge of automatic lifting. Hasabnis et al. [13, 14] observe the forward translation of compiler IR (GCC's RTL) to assembly code and produce an inverse mapping from assembly back to the original RTL IR. However, this approach requires a forward translation from the IR to assembly for *each* architecture. This approach is impossible if no such translation exists (typical for low-level IRs, which lift directly from assembly [8, 22]). Related synthesis approaches automate discovery of symbolic instruction encodings from input-output pairs [10, 15]. By contrast, we address the unique challenge of cross-translating the semantics of instructions to another target language (IR) that supports additional program analysis abstractions (e.g., taint analysis, control flow recovery, function recovery). Recent work in program synthesis has proposed the notion of exploiting existing code for scaling synthesis [6]. To the best of our knowledge, we are the first to demonstrate these ideas toward practical, real-world application by enabling automatic lifter synthesis. Our contributions are as follows:

- **Automatic Lifter Synthesis.** We introduce a technique for automatically synthesizing language translation components that lifts low-level code to an IR. We demonstrate that lifter synthesis enables cross-language translation, allowing analysis reuse on previously unsupported architectures.
- **Learning Synthesis Templates.** We show that mining sketches is effective for translating across heterogeneous instruction architectures. Mining sketches (a) preserves shared semantic properties across architectures and (b) scales synthesis by efficiently constraining the candidate sketch search space.
- **Experimental Evaluation.** We validate our approach by synthesizing a lifter for MIPS, a previously unsupported architecture in the Binary Analysis Platform.[2]. On average, the synthesized lifter successfully translates 84.4% of instructions to IR, across 28 binaries. Our technique complements additional strategies for lifting the remainder of unlifted instructions (e.g., manually, or with more aggressive synthesis exploration). The synthesized lifter allows

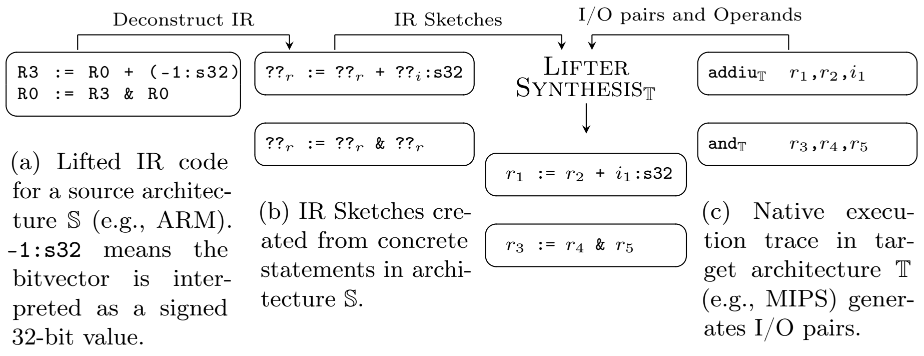
[2] Available at https://github.com/BinaryAnalysisPlatform/bap

a previous IR-based analysis to discover 29 new bugs in binaries for the previously-unsupported architecture.

- **Implementation.** We release our tool and results at https://github.com/squaresLab/SynthLift.

2 Overview and Problem Definition

We formulate IR translation as a syntax-guided synthesis problem [4]. We bootstrap the approach by obtaining an initial set of programs in the IR translated by some existing lifter targeting some other architecture/instruction set (e.g., x86 or ARM).[3] We mine these IR programs to turn concrete program fragments into *sketches* for use in synthesis. Given an unsupported architecture (for which we do not yet have IR translation rules), we (A) collect input-output pairs observed during native execution, and then (B) apply inductive inference over those sketches to discover IR program fragments that satisfy those pairs. We use the oracle-guided inductive synthesis [4] principle to invalidate candidate program fragments using ground-truth input-output pairs.



```
R3  := R0 + (-1:s32)
R0  := R3 & R0
```

$??_r$:= $??_r$ + $??_i$:s32

$??_r$:= $??_r$ & $??_r$

r_1 := r_2 + i_1:s32

r_3 := r_4 & r_5

addiu$_{\mathbb{T}}$ r_1,r_2,i_1

and$_{\mathbb{T}}$ r_3,r_4,r_5

(a) Lifted IR code for a source architecture $\mathbb{S}$ (e.g., ARM). `-1:s32` means the bitvector is interpreted as a signed 32-bit value.

(b) IR Sketches created from concrete statements in architecture $\mathbb{S}$.

(c) Native execution trace in target architecture $\mathbb{T}$ (e.g., MIPS) generates I/O pairs.

Fig. 1: Synthesizing IR from Sketches and I/O pairs.

Overview. Our goal is to use existing IR terms translated from instructions in a source architecture $\mathbb{S}$ (like ARM) to synthesize satisfying IR translation rules for instructions of a new target architecture $\mathbb{T}$ (like MIPS). The first step of lifter synthesis deconstructs concrete IR terms (Figure 1b) from previously lifted code in source architecture $\mathbb{S}$ (Figure 1a). Program sketches are syntactic templates that define the search space for synthesis. A sketch is a partial implementation of a program with missing expressions called *holes* [7]; we denote holes by `??` in Fig. 1b. There are two types of holes in our IR sketches: variables, denoted by $??_r$, and immediate bit vector values, denoted by $??_i$ (respectively corresponding to registers and immediate values in the machine architecture).

The second step of synthesis (Fig 1c) collects concrete input-output pairs, instruction operands, and instruction opcodes from the target architecture $\mathbb{T}$ that

[3] Note that this is not a limiting assumption on generalizing the technique: an existing, functional IR implies at least one existing translation layer implementation, as is the case with, e.g., REIL [8] LLVM [19], VEX IR [21].

we want to lift. In the example, the target architecture is MIPS [4]. We generate traces of input-output pairs by dynamically executing one or more native MIPS instructions. We use the LLVM disassembler to obtain *static* information of instructions: their opcodes, syntactic register names, and immediate values.[5] Static values denoting operands are converted to symbolical IR variables, which we denote in the example by r_x and i_x respectively (x is fresh).

The LifterSynthesis$_{\mathbb{T}}$ procedure then enumerates candidate IR sketches and fills operand holes with the target $\mathbb{T}$'s register and immediate values operand, respectively. The procedure seeks an IR instruction and operand assignment that satisfies all dynamic I/O observations for the native instruction in $\mathbb{T}$ when executed according to the IR's operational semantics. When successful, synthesis produces a *lifter rule* that translates native instructions to the IR for the target $\mathbb{T}$.

Translation Substitution. The synthesis procedure in Fig. 1 identifies IR statements whose *semantics* (specified in Section 3) match the input-output pairs of native execution translation rules. For example, an `addiu`$_{\mathbb{T}}$ operand with opcodes $\langle r_1, r_2, i_1\rangle$ map to an IR statement r_1 `:=` r_2 `+` i_1`:s32` (note that syntactic register and immediate values are both converted into proper typed values when translated into the IR). Translation binds concrete values to IR operand variables r_x, i_x positionally (Figure 2).

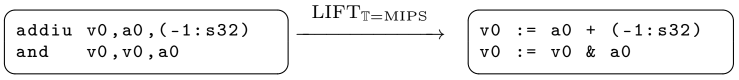

```
addiu  v0,a0,(-1:s32)
and    v0,v0,a0
```
$\xrightarrow{\text{LIFT}_{\mathbb{T}=\text{MIPS}}}$
```
v0 := a0 + (-1:s32)
v0 := v0 & a0
```

Fig. 2: Lifting to a target $\mathbb{T}$ (MIPS)

In general, we do not know the correct order for applying operands obtained from disassembly; we consider permutation of operands during synthesis in Section 4.

Restricting the Synthesis Search Space. The syntactic structure of instructions from native execution allows us to prune the search space of sketches. Fig. 1c gives an intuition: the IR sketch `??`$_r$ `:=` `??`$_r$ `&` `??`$_r$ won't be considered for the `addiu`$_{\mathbb{T}}$ $\langle r_1, r_2, i_1\rangle$ MIPS instruction because the IR does not use an immediate value. In practice, we find that pruning reduces the set of valid candidate sketches to 83% per native instruction, on average.

Problem Scope. Our approach synthesizes instructions including arithmetic operations, bitwise operations, and conditional jumps. We do not consider the details of CPU-specific memory models and modes (e.g., concurrency, memory segments, or privileged instructions). While important, these aspects do not directly support the goal of modeling the essential dataflow properties of instruction semantics in the IR. Extension of the IR to additional architecture-specific memory or permission models is possible, but we leave this consideration for future work. For simplicity, we've demonstrated a one-to-one translation of native instruction to IR instruction, wehereas IRs are typically designed to represent a

[4] https://www.mips.com/products/architectures/mips32/

[5] We use LLVM for convenience–dynamic binary instrumentation techniques can similarly provide instruction operands and opcodes.

single native instruction in one or more IR instructions. We discuss one-to-many translation in Section 4.

3 Synthesis Model

We perform oracle-guided synthesis of IR translation using dynamic execution traces of native instructions for a target architecture $\mathbb{T}$. For simplicity of introducing the model, we consider only one iteration of verifying instruction correctness. In one iteration, our goal is to check whether a sequence of *events* produced during a single step execution of a native instruction is syntactically equal to the sequence of events produced by a executing a *translation* of the native instruction to the IR. Our model assumes a sequential running process, i.e., executing a native instruction is uninterruptible, and memory cannot be modified by concurrent processes. Further, we assume instruction output is invariant under the same inputs. Our assumptions are consistent with the goal of tracking dataflow properties of instruction semantics (e.g., taint analysis, constraint generation), as well as those underlying previous work [10, 15]. In this section we introduce the program model and operational semantics for comparing IR and native execution. We use the BAP IR [1, 3], which performs competitively relative to other IRs [17]. However, the approach generalizes under the synthesis model and assumptions presented in this section.

3.1 Comparing Executions

Program Model. The execution context of a program is modeled by state σ. Both native and IR instructions operate on a state σ that comprises a memory μ and variable bindings $\mathit{\Delta}$. Memory μ is modeled by a partial function from addresses to values $nat \rightarrow int$. Variable bindings $\mathit{\Delta}$ is modeled by a partial function from variable names to values $var \rightarrow int$.

Events. A sequence of events reify the effect of executing an instruction. Events generated during native execution serve as the ground truth oracle for synthesis. We denote events on registers by a 4-tuple $\langle action, \texttt{REG}, reg, value \rangle$ (including flags). An *action* may be either a read operation `R` or a write operation `W`. We use a syntactic value `REG` to disambiguate events on registers from those on memory. A register *reg* may be any syntactic term corresponding to a register for a given architecture (e.g., `EAX` for the x86 architecture). The *value* is a bitvector with a word size for a given architecture. We denote events on memory by a 4-tuple $\langle action, \texttt{MEM}, addr, value \rangle$. Actions on memory are the same as for registers. A read action on memory reads a bitvector *value* from a nonnegative address *addr*. A write action on memory writes a bitvector *value* to a nonnegative address *addr*. All events are syntactic elements; we say $e_1 = e_2$ if an event e_1 is syntactically equal to e_2.

Comparing Events. For every instruction executed in the trace of the native Architecture $\mathbb{T}$, a single native instruction $\mathcal{I}_{\mathbb{T}}$ in state $\sigma_{\mathbb{T}}$ produces a sequence of events $\mathcal{E}_{\mathbb{T}}$. We denote the execution step by $\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}}, \varnothing \rangle \overset{\mathbb{T}}{\rightsquigarrow} \langle \sigma'_{\mathbb{T}}, -, \mathcal{E}_{\mathbb{T}} \rangle$. For convenience, we define a function $\texttt{step}_{\mathbb{T}}$ that returns the sequence of events after executing the instructions: $\mathcal{E}_{\mathbb{T}} = \texttt{step}_{\mathbb{T}}(\sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}})$.

Next, consider execution for IR in an architecture-agnostic language *IR*. Our goal is to generate a sequence of events $\mathcal{E}_{IR}$ which is equivalent to $\mathcal{E}_{\mathbb{T}}$ by executing a logical instruction (comprising one or more IR statements) denoted by $\mathcal{I}_{IR}$. We denote an execution step in the IR by $\langle \sigma_{IR}, \mathcal{I}_{IR}, \varnothing \rangle \overset{IR_*}{\rightsquigarrow} \langle \sigma'_{IR}, -, \mathcal{E}_{IR} \rangle$ and define a convenience function $\texttt{step}_{IR}$ that returns the sequence of events after execution $\mathcal{E}_{IR} = \texttt{step}_{IR}(\sigma_{IR}, \mathcal{I}_{IR})$.

Executing an IR instruction requires an initial state σ_{IR} that simulates the native architecture state $\sigma_{\mathbb{T}}$. We introduce a function α_{IR} that resolves register and memory values from the trace, and maps these values to the initial IR execution state, i.e., $\sigma_{IR} = \alpha_{IR}(\sigma_{\mathbb{T}})$.

We now define an equivalence relation of execution $\sim$ as equal event sequences generated from in-tandem single step execution of source and target languages. Let $\texttt{lift}_{IR}$ be the function that translates a native instruction to target IR instructions, where $\mathcal{I}_{IR} = \texttt{lift}_{IR}(\mathcal{I}_{\mathbb{T}})$. Then synthesis requires:

$$\texttt{step}_{\mathbb{T}}(\sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}}) \sim \texttt{step}_{IR}(\alpha_{IR}(\sigma_{\mathbb{T}}), \texttt{lift}_{IR}(\mathcal{I}_{\mathbb{T}}))$$

which simplifies to checking $\mathcal{E}_{\mathbb{T}} \sim \mathcal{E}_{IR}$. If $\mathcal{E}_{\mathbb{T}} \sim \mathcal{E}_{IR}$ holds, a synthesis iteration is complete and $\mathcal{I}_{\mathbb{T}}$ lifts to $\mathcal{I}_{IR}$. We perform multiple such iterations to refine the accuracy of translation, invalidating IR statements that do not satisfy all input-output equivalence constraints. We defer details of the approach and algorithm to Section 4.

3.2 Operational Semantics

Native Execution. The semantics of native execution is treated as a black box, allowing us to observe input-output pairs of an instruction execution. We use dynamic instrumentation to record sequences of events during an execution trace. We support tracing with popular instrumentation frameworks QEMU [6]and Pin. [7] Dynamic events on registers, flags, and memory are recorded in the trace and processed to produce $\mathcal{E}_{\mathbb{T}}$, accepted as ground truth. For purposes of synthesis, we synchronize byte ordering (endianness) for $\mathbb{T}$ and IR at the dynamic instrumentation level, if needed.

IR Execution. We use an analysis-based IR to execute synthesized statements according to an operational semantics.[8] The BAP interpreter performs architecture-agnostic execution of IR statements. Fig. 3 is a simplified version of the IR grammar. Our work extends the operational semantics and interpreter to generate events during IR execution. For brevity, we elide the rules. The essential changes entail event recording for each rule: variable assignments on registers

[6] https://www.qemu.org/

[7] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[8] Note that IRs may lack a specified operational semantics. Our work emphasizes the importance of using a formally specified IR to enable translation synthesis.

and reads produce `Write` and `R` events, respectively. The same follows for memory accesses; the sequence rule appends events for two instructions, and so on. The full IR grammar and operational semantics is available online [1]. As a concrete example, executing the IR statement $R2 := R0$ results in the event sequence $\mathcal{E}_{IR} = [(\texttt{R,REG,R0,0x1}), (\texttt{W,REG,R2,0x1})]$ (where $R0$ initially stores the value 0x1). The production of ground truth $\mathcal{E}_{\mathbb{T}}$ by native execution and $\mathcal{E}_{IR}$ is compared during synthesis iterations to discover IR statements that satisfy the observed input-output pairs. Note that since we perform a synthesis iteration for one native instruction at a time, execution of the IR code is synchronized with native execution. Our operational semantics therefore does not continue execution by advancing a program counter: instead, it iterates through the sequence of IR statements and executes them until the sequence is empty.[9]

4 Synthesis Approach

We now explain how our synthesis approach generates translation rules that lift native instructions to a sequence of IR statements as a function of the following inputs: (A) A unique identifier for the instruction (i.e., opcode); (B) the set of instruction operands (as purely syntactic values, i.e., register names and immediate values); (C) a set of input-output pairs on register and memory; and (D) a set of candidate sketches in the IR.

4.1 Sketches from Term Deconstruction

Our first key insight is that concrete IR terms (generated from existing lifters) preserve semantic properties to correctly synthesize translation rules for new architectures. Our technique deconstructs concrete IR terms to automatically generate the set of sketch candidates for synthesis. The second key insight is that the syntactic values of native instruction operands (register names and immediate values) reduce the set of possible sketch candidates, making synthesis efficient.

We deconstruct concrete IR terms to generate sketches. Formally, a sketch is a *partial function* $\lambda h.S$, where S accepts a vector of arguments h, or holes, that generate a concrete term S. The arity of S depends on the number of leaf nodes in the AST of the IR term. The input domain consists of two kinds of terms: free variables (e.g., corresponding to registers), and immediate values (i.e., constants). Note that these two kinds of terms correspond to the leaf nodes *var* and *imm* in the IR grammar, respectively (see Fig. 3).

As an example, suppose we encounter the concrete IR statement $\texttt{R1} := \texttt{R0} + 5$. We recursively visit each term in the statement and generate holes for terminal nodes, thereby deconstructing the statement to yield a sketch $[\![S]\!]$ as follows:

$$[\![\lambda h.S]\!] \overset{def}{=} \_{var} := \_{var} + \_{imm}$$

[9] Note that PC-relative instructions, such as jumps, still need access to a program counter variable to enable synthesis. For this, an internal PC is kept in the execution environment.

Three holes are created: the first two operands refer to variables, and the third operand refers to an immediate bitvector value. Given a vector of operands $\overline{o} = \langle \mathtt{R5}, \mathtt{R6}, 2 \rangle$, we can perform a substitution in S:

$$[\![(\lambda h.S)\ \overline{o}]\!] = \mathtt{R5} := \mathtt{R6} + 2$$

To apply a vector of operators, it must match the arity in S over the number of variables $|vs|$ and immediate values $|is|$. In our example, S has arity 3, partitioned as an *arity pair* $\langle 2, 1 \rangle$ since $|vs|= 2$ and $|is|= 1$. We use Algorithm 1 to generate a set of candidate sketches from a program in the IR by visiting each statement in the program. The function ToSketch in line 4 takes a concrete term $\mathcal{I}_{IR}$ and turns it into a sketch containing holes (all concrete values in leaf nodes are converted to holes). Line 5 obtains the operands of $\mathcal{I}_{IR}$ and partitions them to obtain the arity pair $\langle |vs|, |is| \rangle$. The result of Algorithm 1 produces a partial function Lookup mapping unique arity pairs to a set of candidate sketches.

4.2 Synthesis

We perform syntax-guided inductive synthesis over sketches. The program synthesis problem stipulates that the formula $\forall x.\phi(x, [\![P]\!](x))$ be valid for all inputs x for a synthesized program $[\![P]\!]$ [7]. The formula ϕ relates an input and output specification against a synthesized program $[\![P]\!]$. For an oracle-based, syntax-guided synthesis the general formula is

$$\forall x.\phi(x, [\![P]\!](x)) \equiv \forall x.\ oracle(x) = [\![P]\!](x)$$

for some equivalence relation $=$. In Section 3 we defined the equivalence relation for IR and native execution as equivalence of event sequences. In terms of inputs $\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}} \rangle$ from source language $\mathbb{T}$ (as in Section 3) and sketches $[\![S]\!]$, we define the *translation synthesis problem* as finding $[\![S]\!]$ subject to:

$$\begin{aligned} &\forall \langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}} \rangle . \phi(\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}} \rangle, [\![S]\!](\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}} \rangle)) \equiv \\ &\qquad \mathtt{step}_{\mathbb{T}}(\sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}}) \sim \mathtt{step}_{IR}(\alpha_{IR}(\sigma_{\mathbb{T}}), [\![(\lambda h.S)\ \textsc{Ops}(\mathcal{I}_{\mathbb{T}})]\!]) \end{aligned} \tag{1}$$

for each unique $\mathcal{I}_{\mathbb{T}}$. The synthesis algorithm goal is to discover a sketch $[\![S]\!]$ applied to the operands of native instruction $\textsc{Ops}(\mathcal{I}_{\mathbb{T}})$ such that (1) holds.

Algorithm 2 describes the synthesis process. Input consists of the lookup function produced by MineSketches and trace information $\mathcal{T}$ containing a set of triples $\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}} \rangle$ generated from dynamic executions $\mathcal{E}_{\mathbb{T}} = \mathtt{step}_{\mathbb{T}}(\sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}})$. Functions Code and Ops in line 2 of Algorithm 2 extracts a unique identifier *code* associated with $\mathcal{I}_{\mathbb{T}}$, and its operands as a vector $\overline{o}$, respectively. In line 4, instruction operands $\overline{o}$, initial execution state $\sigma_{\mathbb{T}}$, and computed events $\mathcal{E}_{\mathbb{T}}$ are associated with unique instruction codes in the map Ψ. Candidate sketches $\mathcal{S}$ are obtained from a partition of the instruction's operands (lines 5 and 6).

SynthInsn enumerates through all candidate sketches to find a satisfying assignment of operands that satisfy events. As mentioned in Section 2, we cannot assume that the operand order returned by the disassembler guarantees the desired

```
program ::= stmt seq
stmt s ::=
    var := exp
  | jmp exp
  | if (exp) (stmt seq)
    else (stmt seq)

exp e ::=
    exp
  | var                     variable
  | imm                     bitvector value
  | mem[exp_1] := exp_2     memory store
  | mem[exp]                memory load
  | exp_1 binop exp_2       binary operation
  | unop exp                unary operation
  | cast : nat[exp]         casts
```

Fig. 3: Simplified IR Grammar

Algorithm 1: Mine Sketches

```
Input:
    $P_{IR}$ : program in IR.
Output:
    LOOKUP : $\langle |vs|, |is| \rangle \to \mathcal{S}$. LOOKUP returns candidate sketches $\mathcal{S}$ for an arity
    pair (instruction operand sizes)
1  MINESKETCHES($P_{IR}$)
2      for $BasicBlock_{IR} \in$ VISIT($P_{IR}$) do
3          for $Stmt_{IR} \in$
               VISIT($BasicBlock_{IR}$) do
4              $\lambda h.S \leftarrow$ TOSKETCH($\mathcal{I}_{IR}$)
5              $\langle |vs|, |is| \rangle \leftarrow$
                   PARTITION(OPS($\mathcal{I}_{IR}$))
6              LOOKUP $\leftarrow$ UPDATEMAP(
7                  LOOKUP, $\langle |vs|, |is| \rangle, \lambda h.S$
8              )
9      ret LOOKUP
```

Algorithm 2: Synthesis

```
Input:
    LOOKUP, $\mathbb{T}$ : a lookup function produced by MINESKETCHES and trace
    input $\mathbb{T}$.
    $\mathcal{T}$: a dynamic execution trace.
Output:
    LIFT$_{\mathbb{T}}$ : $code \to \mathcal{S}_{IR}$ : a function that returns a set of valid sketches in the
    target IR for the given native instruction code.
1  SYNTHESIZE(LOOKUP, $\mathcal{T}$)
2      for $\langle \sigma_{\mathbb{T}}, \mathcal{I}_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}} \rangle \in \mathcal{T}$ do
3          $code, \overline{o} \leftarrow$ CODE($\mathcal{I}_{\mathbb{T}}$), OPS($\mathcal{I}_{\mathbb{T}}$)
4          $\Psi \leftarrow$
               UPDATEMAP($\Psi$, $code$, $\{\overline{o}, \sigma_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}}\}$)
5          $\langle |vs|, |is| \rangle \leftarrow$ PARTITION($\overline{o}$)
6          $\mathcal{S} \leftarrow$ LOOKUP($\langle |vs|, |is| \rangle$)
7          $\mathcal{R} \leftarrow$ SYNTHINSN(
8              $\sigma_{\mathbb{T}}, \overline{o}, \mathcal{S}, \mathcal{E}_{\mathbb{T}}, \Psi(code)$
9          )
10         LIFT$_{\mathbb{T}}$ $\leftarrow$
               UPDATEMAP(LIFT$_{\mathbb{T}}$, $code$, $\mathcal{R}$)
11     ret LIFT$_{\mathbb{T}}$
```

Algorithm 3: Synthesis Iteration

```
1  SYNTHINSN($\sigma_{\mathbb{T}}, \overline{o}, \mathcal{S}, \mathcal{E}_{\mathbb{T}}, \psi$)
2      $\mathcal{R} \leftarrow \varnothing$
3      for $\lambda h.S \in \mathcal{S}$ do
4          for $\lambda h.S_p \in$ PERM($\lambda h.S$) do
5              $C_{IR} \leftarrow (\lambda h.S_p)\ \overline{o}$
6              $\mathcal{E}_{IR} \leftarrow$
                   step$_{IR}$($C_{IR}, \alpha_{IR}(\sigma_{\mathbb{T}})$)
7              if $\mathcal{E}_{\mathbb{T}} \sim \mathcal{E}_{IR}$ $\wedge$
8                  VERIFY($\psi, \lambda h.S_p$) then
9                  $\mathcal{R} \leftarrow \mathcal{R} \cup \{\lambda h.S_p\}$
10     ret $\mathcal{R}$
11
12 VERIFY($\psi, \lambda h.S$)
13     ret $\bigwedge_{\langle \overline{o}, \sigma_{\mathbb{T}}, \mathcal{E}_{\mathbb{T}} \rangle \in \psi}$
14         ($\mathcal{E}_{\mathbb{T}} \sim$ step$_{IR}$($\alpha(\sigma_{IR}), (\lambda h.S)\ \overline{o}$))
```

Algorithm 4: Lift

```
1  LIFTHELPER(LIFT$_{\mathbb{T}}$, $\mathcal{I}_{\mathbb{T}}$)
2      $\lambda h.S \leftarrow$
           TAKEFIRST(LIFT$_{\mathbb{T}}$(CODE($\mathcal{I}_{\mathbb{T}}$)))
3      $\overline{o} \leftarrow$ OPS($\mathcal{I}_{\mathbb{T}}$)
4      $I_{IR} \leftarrow (\lambda h.S)\ \overline{o}$
5      ret $I_{IR}$
```

semantics. In Algorithm 3, line 4, PERM generates sketches that permute the order of input operands in $\lambda h.S$. We discuss permutation strategies in Section 4.3. Each permuting sketch $\lambda h.S_p$ applies $\overline{o}$ and generates a concrete IR term C_{IR} and executes it to produce $\mathcal{E}_{IR}$. Lines 7 and 8 verify the concrete term satisfies all events for the instruction $\mathcal{I}_{\mathbb{T}}$ observed so far. The check $\mathcal{E}_{\mathbb{T}} \sim \mathcal{E}_{IR}$ short circuits the more expensive VERIFY check (line 12) as an optimization. Each satisfying sketch is added to the result set $\mathcal{R}$. Valid sketches in the result set are updated in the map LIFT$_{\mathbb{T}}$. Algorithm 4 synthesizes `lift`$_{IR}$ (introduced in Section 3) by transforming the LIFT$_{IR}$ map into a lookup function.

In summary, using Algorithms 1–4 we fully derive the desired translation $\mathcal{I}_{IR} = \texttt{lift}_{\mathbb{T}}(\mathcal{I}_{\mathbb{T}})$ from initial inputs P_{IR} and $\mathcal{T}_{\mathbb{T}}$:

$$\textsc{Lift}_{\mathbb{T}} = \textsc{Synthesize}(\textsc{MineSketches}(P_{IR}), \mathcal{T}_{\mathbb{T}})$$
$$\texttt{lift}_{\mathbb{T}} = \lambda\, \mathcal{I}_{\mathbb{T}}.(\textsc{LiftHelper}\ \textsc{Lift}_{IR})$$

4.3 Operand permutations and One-To-Many Translation

The disassembler may return instruction operands in any order to $\textsc{Ops}(\mathcal{I}_{\mathbb{T}})$. We observed that operand order tends to correspond roughly with a left-to-right reading of assembly instruction semantics. For example, an instruction $\texttt{add R0}, 8$ corresponding to a semantic expression $\texttt{R0} = \texttt{R0} + 8$ would disassemble with the operands in the order $\langle \texttt{R0}, \texttt{R0}, 8\rangle$ However, we also observed small discrepancies. For example, memory store instructions in the IR grammar may swap the source and destination operands compared to the disassembled order. Function Perm thus implements a customizable permutation transformation on operands. Though exhaustive enumeration is feasible for small numbers of operands, we have found that only permuting *adjacent* operands proved effective in practice. When we experimented with an exhaustive permutations approach, we observed no increase in successful synthesis. Complexity of trying all adjacency swapping permutations is fast: linear in arity (order $O(|vs|+|is|)$).

Our current implementation synthesizes one-to-many translation by preserving existing one-to-many mappings implemented in current ARM and x86 lifters. This allows synthesis to discover, e.g., conditional branch statements. On the other hand, relying on a rigid mapping may miss sketches such as multiple consecutive assign statements. We leave sketch composition for synthesis to future work.

5 Evaluation

The goal of SynthLift is pragmatic: to synthesize lifter rules for new architectures, alleviating the need to manually translate the majority of instructions. The focus application is to enable existing analyses for unsupported architectures. We target a previously unsupported architecture, MIPS, and show that the synthesized lifter discovers new bugs in commercial off-the-shelf MIPS binaries. Accordingly, we evaluate SynthLift as follows:

– Is SynthLift effective at enabling existing analyses for previously unsupported architectures (Section 5.1).
– What is the speed and accuracy of SynthLift, and what percentage of instructions can SynthLift recover in widely used programs (Section 5.2).
– How well does SynthLift generalize across architectures (Section 5.3)?

5.1 Analysis Reuse

We applied an existing taint-based analysis to find new bugs in COTS binaries for MIPS [2]. The analysis checks for cases where results of C library functions are unused. For example, some C POSIX functions are declared with a "warn unused result" attribute that flags warnings at compile time. Our analysis follows taint flows for function return values to detect such bugs in binaries, where source code is not typically available. The analysis looks for cases where the return value is overwritten without being read. We ran the analysis on 30 binaries in the `sbin` directory of a COTS D-Link router. In total, we discovered 29 bugs in 30 binaries; for brevity, we summarize 8 binaries comprising 17 bugs that span a variety of functions handled by the analysis (Figure 4a). Not shown, we discovered 12 additional bugs across 12 additional binaries for similar functions as in Figure 4a. 11 binaries did not generate bug reports. We manually inspected analysis results using a decompiler to confirm true positives; where possible, we were able to confirm unchecked values for binaries that have source code (such as `ntpclient`). We encountered two false positives. This happened when return values of two consecutive `malloc` calls are inaccurately tracked in our ABI model (note: the inaccuracy is not due to the synthesized instruction semantics). To consider a large real-world example, we also lifted OpenSSL to recover 86% of instructions, and confirmed that the analysis did not find any bugs.

Name	**#**	**Functions**
† iptables	3	`fwrite`
† ntpclient	1	`send`
† pppd	1	`fwrite`
rdnssd	1	`setsockopt`
speedtest	2	`system`, `fgets`
timer	2	`read`, `shutdown`
wakeOnLanProxy	1	`shutdown`
wcnd	8	`system`

(a) # indicates the number of unused return values in COTS MIPS binaries. We show **Function** names for which return values are not used. † indicates that we found evidence of the bug in source code.

%	**ARM-IR Sketches**	**%**	**x86-IR Sketches**
20.9	$\_{}_v$ `:=` $\_{}_v$	11.9	$\_{}_v$ `:=` $\_{}_i$
14.5	$\_{}_v$ `:=` $\_{}_i$	8.6	`jmp` $\_{}_i$
8.9	`jmp` $\_{}_i$	5.1	$\_{}_v$ `:=` `mem`[$\_{}_v + \_{}_i$]
7.0	$\_{}_v$ `:=` `mem`[$\_{}_v + \_{}_i$]	4.4	`mem`[$\_{}_v + \_{}_i$]`:=` $\_{}_v$
5.6	$\_{}_v$ `:=` $\_{}_v = \_{}_i$	4.2	$\_{}_v$ `:=` $\_{}_i = \_{}_v$
5.6	$\_{}_v$ `:=` `hi` : 1[$\_{}_v$]	4.2	$\_{}_v$ `:=` `hi` : 1[$\_{}_v$]
5.5	`mem`[$\_{}_v + \_{}_i$]`:=` $\_{}_v$	4.0	`mem`[$\_{}_v$]`:=` $\_{}_i$
4.6	$\_{}_v$ `:=` $\_{}_v - \_{}_i$	3.9	$\_{}_v$ `:=` $\_{}_v - \_{}_i$

(b) Distribution of single-statement sketches mined from x86 and ARM IR programs. Holes v denote a variable; i, an immediate. We omit bit widths for brevity, though `hi:1` denotes a cast which keeps the high bit of the value (e.g., common for testing IR flags).

Fig. 4

5.2 Synthesizing the MIPS Lifter

To synthesize the MIPS lifter, we used IR sketches generated from 28 ARM Coreutils[10] binaries, and used 5 programs from the Hacker's Delight benchmarks [24] (compiled to MIPS) to generate dynamic input-output pairs. Coreutils is a set of highly popular command-line utilities and representative of typical programs; Hacker's Delight programs perform a variety of bit-manipulation operations that generate input-output pairs for a diverse set of native instructions.

[10] https://www.gnu.org/software/coreutils/coreutils.html

End-to-end synthesis (mining sketches, processing traces, and lifter synthesis) takes 58 seconds. Each native MIPS instruction starts with a set of 29 initial sketches on average (using Alg. 2 Partition and instruction operands). On average, successfully synthesized instructions complete with 2 satisfying sketches (due to commutativity of binary operations). Synthesis converges quickly: Figure 5, left boxplot, shows that synthesis discovers the final set of satisfying sketches after only two input-output pairs for most instructions. The final set of satisfying sketches verify over thousands of input-output events for typical instructions (Figure 5). We observe that the distribution of input-output pairs by Hacker's Delight binaries mirror the intuition that common instructions like "load word" (`LW`) represent a disproportionately large part of the programs.

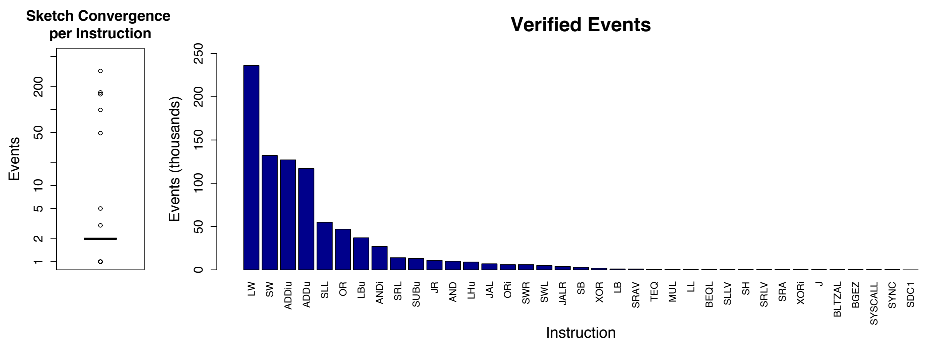


Fig. 5: MIPS Lifter Synthesis. On the left, the number of iterations until synthesis converges on the final set of satisfying sketches over all events. On the right, the number of verified input-output pair events for successfully synthesized instructions.

We ran the synthesized lifter on the 28 MIPS Coreutils binaries in the Debian distribution. To count lifter coverage, we take the percentage of individual native MIPS instructions that fire a translation rule in the lifter. On average, the lifter recovers **84.8%** of instructions. Thus, 15.2% of instructions in the binary could not be lifted (in practice, we substitute NOP instructions in the IR). Synthesis fails when a suitable sketch cannot satisfy the semantics of an instruction. One such instruction is `LUi`, or load upper immediate. To lift this instruction, we ideally want an IR candidate such as $\_{var} := \_{imm} << \_{imm_2}$, where imm_2 is 16. However, such a candidate is never mined from the IR sourced from ARM. In Section 5.4 we suggest further improvements to our technique to address such cases.

5.3 Generalizing Across Architectures

BAP currently supports lifting for both the ARM and x86 architectures. To validate our ability to synthesize across architectures, we targeted MIPS by mining sketches lifted from the suite of x86 Coreutils programs. The x86-sourced MIPS lifter synthesized 6 less instructions than the ARM version due to missing a rotation sketch.[11] Interestingly, the x86-sourced lifter recovered the same 84.8% instructions when lifting the MIPS Coreutils test set. Figure 4b suggests why we gain the same utility when synthesizing under different architectures: the eight most frequent sketches for ARM and x86 are very similar, and account for the majority of IR instructions.

5.4 Discussion

Mining versus Manually Specifying. Our approach demonstrates the applicability and feasibility of mining sketches to enable a cross-architecture translation. Figure 4b also suggests that manually specifying a small set of sketches is competitive to mining sketches. However, we observe that manual specification poses additional challenges compared to mining: (a) it is difficult to anticipate exactly the set of sketches to specify; current approaches usually involve a human-in-the-loop who must iteratively estimate or consult specification manuals [10]; (b) the set of effective sketches changed based on how the IR is designed (i.e., different IRs will require different sketch templates); (c) manual specification does not naturally consider similarities of multiple heterogeneous architectures; our summary in Figure 4b is a first result to show that sketches *do* translate for IRs. Our approach sees manual specification as complementary: mining is an effective approach for revealing initial common sketches (and how the IR-specific design structure relates to sketches), and can automatically discern similarity in e.g., architectures at the IR level. A human-in-the-loop can use this information to make synthesis more effective.

Partial Instruction Set Recovery. We showed in Section 5.2 that the synthesized lifter recovers a high percentage (roughly 85%) of instructions in typical binaries. On the other hand, the lifter has a lower rate of coverage for the entire MIPS instruction set, approximately 33 instructions of 45.[12] While a greater percentage of the instruction set is desirable, our goal is to (a) assess whether translation can be synthesized "out-of-box" without specifically considering the target architecture and (b) validate how well existing analyses can operate with a partial lifter synthesized for a new architecture. Our evaluation reveals ample opportunity for improving instruction set coverage (e.g., manually specifying missing sketches) and existing work has shown nondeterminstic approaches, like stochastic search [15] to be effective. At present, our goal is to demonstrate synthesis effectiveness using a tractable method, i.e., using only the set of finite sketches mined from existing rules. We leave the appeal of combining complementary approaches to future work.

6 Related Work

Bornholt et al. [6] propose mining sketches for structure to scale program synthesis—our work demonstrates the ability to fill this gap by mining IR

[11] The missing rotation operator is however found in subexpressions of IR statements, but we fail to generate the desired statement $_{var}$:=$_{var} << _{imm}$.

[12] Using Fig. 5, (excluding instructions `TEQ`, `SYSCALL`, `SYNC`, and `SDC1` which are modeled differently in the trace than actual MIPS semantics), and compared to a simplified MIPS ISA(`goo.gl/YUEdiy`).

sketches to scale IR translation over heterogeneous architecture instruction sets. Our work relates generally to syntax-guided synthesis over sketches [4, 7]. Related work in inductive synthesis use I/O pairs to recover x86 semantics as SMT encodings [10, 15]. Our approach similarly uses I/O pairs to infer semantics, but targets IR translation for multiple architectures and mines sketches automatically in lieu of manual specification. Hasabnis et al. leverage forward source-to-compiler-IR translation [14] and symbolic execution of compilers [13] to lift low level instructions to the compiler IR. These approaches rely on the existence of a forward translation routine (i.e., compiler) for each architecture, which then reverse the mapping to generate assembly-to-IR rules. In contrast, our approach generalizes to cross-architecture translation using a bootstrapped set of initial candidate sketches and input-output pairs only—no existing translation is required for *each* architecture target. Applications in static binary translation manually translate dynamically executed QEMU instructions to static LLVM IR [9] for multiple architectures; we believe our technique has the ability to automate the translation process. Work on verifying correctness of low level IRs are complementary to the lifter synthesis problem; related techniques can assert correctness of semantics with respect to observed I/O-pairs [10, 12] or symbolic equivalence checking [17].

7 Conclusion

We have presented cross-architecture lifter synthesis, a new way to automatically synthesize IR translation rules for new architectures by leveraging existing IR programs. We demonstrated that our approach is effective at recovering a lifter for a new architecture, and provides sufficient instruction coverage to enable analysis reuse and discovery of new bugs. Synthesis could discover more rules by generating candidates over the IR grammar (e.g., using stochastic search [4, 15]), or by manually supplying a small number of plausible sketches (rather than manual, per-instruction translation). We further believe our work has further application for discovering semantic relations between different languages: lifter synthesis reveals similar semantic properties across heterogeneous architectures and can distinguish differences when cross-translation synthesis fails. Lifter synthesis opens up new methods for language translation, e.g., by complementing manual processes, and is amenable to automation assistance where sketches can be manually specified (e.g., [10]). Finally, we believe the approach has broad application to IRs generally, including automatic discovery and synthesis of common semantics for IR-to-IR translation.

Acknowledgments

This work is partially supported under NSF grant number CCF-1563797. All statements are those of the authors, and do not necessarily reflect the views of the funding agency. The authors would like to thank the BAP Team for their continued open source development and support for the BAP project.

Bibliography

[1] BAP IR Operational Semantics. https://github.com/BinaryAnalysisPlatform/bil/releases/download/v0.1/bil.pdf (2018), online; accessed 23 April 2018

[2] BAP Warn Unused Analysis. https://opam.ocaml.org/packages/bap-warn-unused/bap-warn-unused.1.3.0/ (2018), online; accessed 23 April 2018

[3] Binary Analysis Platform. https://github.com/BinaryAnalysisPlatform/bap (2018), online; accessed 23 April 2018

[4] Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design. pp. 1–8 (2013)

[5] Balakrishnan, G., Reps, T.: Analyzing Memory Accesses in x86 Executables. Compiler Construction pp. 2732–2733 (2004)

[6] Bornholt, J., Torlak, E.: Scaling program synthesis by exploiting existing code. Machine Learning for Programming Languages (2015)

[7] Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: POPL '16. pp. 775–788 (2016)

[8] Dullien, T., Porst, S.: REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest '09

[9] Federico, A.D., Payer, M., Agosta, G.: rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. In: CC '17. pp. 131–141 (2017)

[10] Godefroid, P., Taly, A.: Automated synthesis of symbolic instruction encodings from I/O samples. In: PLDI '12. pp. 441–452 (2012)

[11] Gotovchits, I., van Tonder, R., Brumley, D.: Saluki: Finding Taint-style Vulnerabilities with Static Property Checking. In: BAR '18 (2018)

[12] Hasabnis, N., Qiao, R., Sekar, R.: Checking correctness of code generator architecture specifications. In: CGO '15. pp. 167–178 (2015)

[13] Hasabnis, N., Sekar, R.: Extracting instruction semantics via symbolic execution of code generators. In: FSE '16. pp. 301–313 (2016)

[14] Hasabnis, N., Sekar, R.: Lifting assembly to intermediate representation: A novel approach leveraging compilers. In: ASPLOS '16. pp. 311–324 (2016)

[15] Heule, S., Schkufza, E., Sharma, R., Aiken, A.: Stratified synthesis: automatically learning the x86-64 instruction set. In: PLDI '16. pp. 237–250 (2016)

[16] Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.T.: On the naturalness of software. Communications of the ACM 59(5), 122–131 (2016)

[17] Kim, S., Faerevaag, M., Junk, M., Jung, S., Oh, D., Lee, J., Cha, S.K.: Testing intermediate representations for binary analysis. In: ASE '17 (2017)

[18] Kinder, J., Veith, H.: In: Precise Static Analysis of Untrusted Driver Binaries. pp. 43–50. FMCAD '10 (2010)

[19] Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO '04. pp. 75–88 (2004)

[20] Le, V., Sun, C., Su, Z.: Randomized stress-testing of link-time optimizers. pp. 327–337. ISSTA '15 (2015)

[21] Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: USENIX Security Symposium '09

[22] Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Security and Privacy. pp. 317–331 (2010)

[23] Sun, C., Le, V., Zhang, Q., Su, Z.: Toward understanding compiler bugs in GCC and LLVM. In: ISSTA '16. pp. 294–305 (2016)

[24] Warren, H.S.: Hacker's delight. Pearson Education (2013)