Scalable Validation of Binary Lifters

Anonymous Author(s)

Abstract

Validating the correctness of binary lifters is pivotal to gain trust in binary analysis, especially when used in scenarios where correctness is important, e.g., in security analysis, binary patching, or recompilation to other ISAs. Existing approaches focus on validating the correctness of lifting a single instruction and do not scale to full programs. In this work we show that formal translation validation of single instructions for x86-64 is practical and develop a novel technique that uses validated instructions to scale to program level validation, thereby eliminating the bottleneck of heavy-weight equivalence checkers at the program level. Our work is the first to to do translation validation of single instructions on an architecture as extensive as x86-64, uses the most precise formal semantics available, and has the widest coverage in terms of number of instructions tested for correctness. To scale to whole programs, our *compositional lifter* composes the validated IR sequences to create a reference standard. The semantic equivalence check between the reference and the lifter output is then reduced to a syntactic equivalence check through use of a normalizer — a procedure that reduces IR sequences to a canonical representation. Using our approach, we find 29 new bugs in McSema, a mature open-source lifter from x86-64 to LLVM IR. For whole programs, our approach was able to prove equivalence of lifted code for 2189/2348 functions taken from single-source benchmark test-suite.

Keywords x86-64, Translation Validation, Formal Semantics, LLVM IR, Compiler Optimizations, Graph Matching

1 Introduction

The ability to directly reason about binary machine code is desirable, not only because it allows analyzing binaries even when the source code is not available (e.g., legacy code, closed-source software, or malware), but also because it avoids the need to trust the correctness of compilers [14, 68]. Analyzing binary code is important in certain subfields of software engineering and security tools, including binary instrumentation [17, 18, 45, 48, 55], binary re-targeting [25, 30], software hardening [21, 35, 73, 74], software testing [13, 23, 36], CPU emulation [16, 49], malware detection [19, 24, 31, 43, 65, 72], automated reverse engineering [5, 11, 26, 47, 58, 61, 70], sand-boxing [33, 42, 71], profiling [38, 66], and automatic exploit generation [20].

To reason about binary code, binary analysis frameworks, e.g., [8, 11, 19, 58, 64], first convert raw bytes from the binary into a stream of instructions through *disassembly*. To enable greater retargetability of the frameworks to multiple instruction sets, these tools often use a binary *lifter* to lift, or translate, all supported ISAs to a uniform intermediate representation (IR), and then apply architecture-independent passes on this IR. After lifting, several analysis passes may operate on the IR to: (i) recover higher-level constructs, such as functions, stack frames, variables, and types, (ii) re-target to a different ISA, or (iii) instrument and recompile the binary for various purposes. Many binary analysis frameworks published in academia [19, 32, 34, 65], or as open-source code [7, 8, 11, 58, 64], use such a lifter as a first step in their pipeline.

Developing a lifter, especially for complex modern ISAs, is challenging and error-prone [53], mainly because manually encoding the effects of a vast number of instructions (and their variants) is hard. This is made even harder when the informal specifications provided by the hardware manufacturers run into thousands of pages [1, 9], have mistakes [27], or allow for implementation-dependent undefined behaviors. Once such a lifter is developed, the developers then run into the problem of not having a way to test their implementation completely as there are no formal, machine-readable semantics available for automated testing. Lastly, to make it worse, these lifters need to be updated and rechecked for correctness every time new instructions are added to an ISA.

Despite the correctness challenges in binary lifting, such lifters are sometimes used for tasks where correctness is especially important, e.g., when looking for security vulnerabilities in binary code, binary emulation of one processor ISA on another, or recompiling embedded software to run on new platforms. Beyond these more critical tasks, gaining confidence in decompilers through more effective testing is also generally important for developers of decompilers, especially for complex ISAs.

Making matters worse, there has been very limited work to date on validating correctness of binary decompilers, and that work has focused on translation of single instructions. All of this existing work falls short in at least one of the following criteria: (i) require random test-inputs which leads to incomplete coverage [22, 51, 52], (ii) rely on symbolic analysis and therefore do not scale to full program translation validation [41, 50], or (iii) require modification of the lifting frameworks under test to emit additional information required to prove correctness [39].

The goal of the work in this paper is to develop formal and informal techniques to achieve high confidence in the correctness of a lifter from a complex machine ISA (e.g., X86-64) to a rich IR (e.g., LLVM IR). We present four key contributions.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

1. First, we show that formal translation validation of single machine instructions can be used as a building block for scalable full-program translation validation. The underlying insight is that decompilers are usually designed to perform simple instruction-by-instruction lifting (using a fixed and canonical representation of architectural state at the IR level), followed by standard IR optimization passes to achieve simpler IR code.
2. Second, we develop the first single instruction translation validation framework for x86-64. Our work applies the *most precise formal semantics* for x86-64 known to date, and has the *widest coverage* in terms of the number of instructions tested when compared to earlier work. We experimentally verify that such single instruction validation is capable of finding bugs, and in fact *all bugs we have uncovered were found using said single instruction validation*. In particular, we find discrepancies in the lifting of 29 instructions in McSema [8, 58], a well-tested, mature [6], and open source lifter for x86-64 to LLVM IR, clearly showing the effectiveness of our technique. All of these have been confirmed as bugs in the lifter by the McSema developers.
3. Third, given a lifter, *D*, we show that we can automatically construct a simple, *compositional* decompiler, D', by essentially just concatenating the lifted IR sequences for individual instructions (which are individually proven correct by part 1, above). Although we do not *prove* full formal equivalence of the compositional decompiler to the original, the tool is exceedingly simple, and moreover, *produces code sequences (say, T') that are syntactically very similar* to those (T) produced by the decompiler we want to validate. This serves as a foundation for scaling translation validation to full programs.
4. Fourth, we propose a scalable approach for full-program translation validation that does not require heavyweight symbolic execution or theorem provers. Our key insight is that there exists a semantics-preserving transformation — dubbed a normalizer — of T and T', say $\mathsf{norm}(\mathsf{T})$ and $\mathsf{norm}(\mathsf{T}')$, such that $\mathsf{norm}(\mathsf{T})$ and $\mathsf{norm}(\mathsf{T}')$ are *syntactically* equivalent iff T and T' are *semantically* equivalent. If such a normalizer exists, then we can reduce the problem of program-level semantics checking to a much cheaper program-level syntactic check! In this paper, we construct a normalizer out of a very short sequence of 15 LLVM IR passes, of which 11 are transformation passes. We have found this approach to be effective at proving syntactic equivalence between T and T' for 2189 out of 2348 functions taken from "single-source-benchmark" of LLVM test-suite.

The rest of this paper proceeds as follows. The next section gives a high-level overview of our approach. Section 3 gives some background on building blocks used in our work. Section 4 describes our approach for formal single-instruction Translation Validation. Section 5 describes how we scale to full-program Translation Validation. Section 6 describes our experimental evaluation, including the bugs in McSema uncovered by our work, and the effectiveness of full-program Translation Validation. Section 7 briefly discusses some of the key limitations of this work to date, along with avenues for future work. Section 8 places our work in context of prior work, and Section 9 concludes.

2 Approach Overview

In this section we provide a high-level overview of the two main components of our approach, i.e., single-instruction validation and program-level validation. Before we begin, we first describe the scope of our problem.

Problem Scope. Our techniques are generally applicable to verify binary lifters from any ISA, e.g., x86, ARM, RISC-V, PowerPC, to an intermediate representation, such as LLVM IR [44], VEX IR [55] etc., as long as (a) formal semantics for both the ISA and the target language is available, and (b) the target language can be normalized to a canonical representation through a series of semantic-preserving transformations. Through the rest of this paper, we fix our discussion to lifting x86-64 to LLVM IR using the most mature, open-source lifter McSema [58] and our normalizer to be a subset of LLVM IR optimization passes. Other notable lifters [7, 29] from x86-64 to LLVM IR may be directly supported in our framework through minimal engineering effort. Additionally, we restrict our work to the common case of compiler generated binaries, and we do not consider binaries that are deliberately obfuscated to deter reverse engineering, which is in-line with previous work on translation validation. Lastly, as our aim is to validate the lifted code, we do not focus on finding bugs lower in the pipeline, e.g., in the loading and disassembly of binaries. This is orthogonal to our work and has been shown to be relatively mature for the common case of compiler generated binaries [12].

Our overall approach, is a composition of two techniques as shown in 1 with the goal of validating the translation of an x86-64 program P to a lifted LLVM IR program T using a lifter D.

Single Instruction Validation. The single-instruction validation proceeds in the following steps: (i) each x86-64 instruction I is lifted to an LLVM IR sequence S using the lifter D, (ii) Next, we identify the input/output variable correspondence between I and S, i.e., we determine a mapping of registers/memory in I to IR entities in S, (iii) Using the formal semantics of the IR and x86-64, we perform symbolic execution to generate symbolic summaries, (iv) Lastly, we say I and S are semantically equivalent when each corresponding symbolic summary pair are equivalent. We employ the $\mathbb{Z}3$ [28] solver for the equivalence checks. If the two

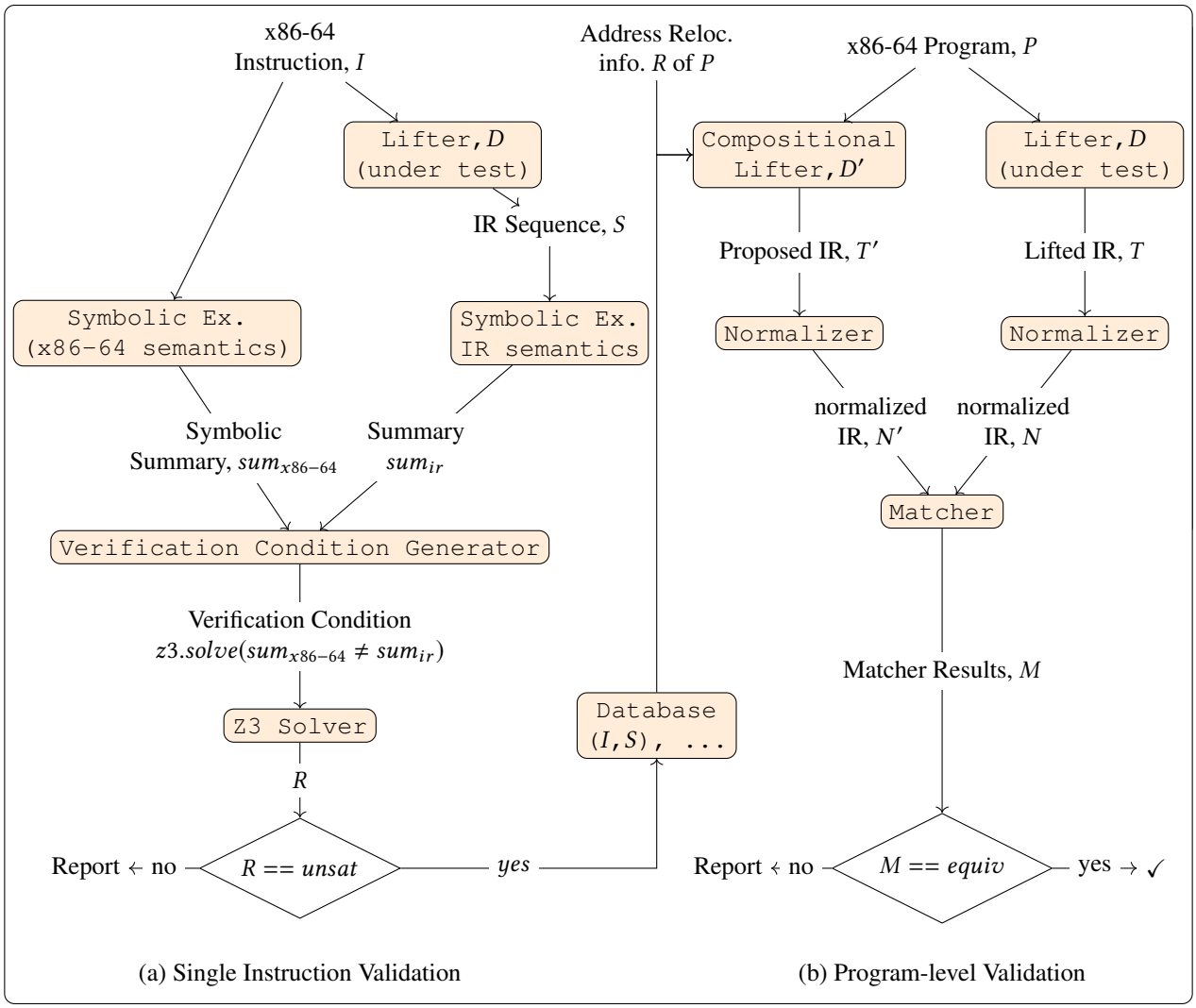

Figure 1. Overview diagram of the translation validation framework

summaries mismatch, meaning we find a bug, they are then reported. Otherwise, we add the pair <I,S> to a validated database.

Program-Level Validation The program-level validation starts with lifting the x86-64 program P using the lifter D to an IR program T. We employ a Compositional Lifter (section 5.1) which composes the validated (or yet to be validated) IR sequences, corresponding to the x86-64 instructions to generate an LLVM IR program, T′. Next, we seek to compare T and T′ for a syntactical match, *one function at a time*. Towards that goal, we use a set of 15 LLVM optimization passes as a normalizer to close the syntactic gap between T & T′ to be compared for syntactic equivalence by a matcher (Section 5.2).

Composing the Techniques As such, the two techniques are independent and their results do not depend on the other, with one minor caveat: the results from program-level validation, either a complete equivalence match, or a potential mismatch, are not sound until the IR instruction sequences used to construct T′ are validated by the single-instruction validation. However, the ordering between the two techniques does not matter, i.e., single-instruction validation may be done ahead of time, when composing instruction during program-level validation, or done in a batch after program-level validation.

3 Preliminaries

In this section, we provide background on various pieces of our work: (i) The binary lifter under test, McSema, (ii) The formal x86-64 semantics, and (iii) The formal LLVM IR semantics.

McSema McSema [58] is the most mature, well tested, open-source lifter to raise binaries from x86-64 instructions

to LLVM bitcode. At a high-level, McSema is split into two parts: (a) frontend, and (b) backend. The frontend is responsible for parsing, loading, and disassembling a binary and exports an interface to the backend to query for the required information, e.g., the defined symbols, sizes of various binary sections, instruction listings etc. The backend then uses this information and Remill [8] library to lift the individual instructions. McSema supports multiple different frontends with IDA Pro being the most robust, and supported option.

Conceptually, the backend implementation of McSema is fairly straightforward: McSema exposes all of architecture state, i.e., the program registers, conditional flags, and program memory, through an LLVM *struct*, aptly named *State*. McSema simply scans through the disassembly of the binary and lifts each instruction one by one, emitting code to read and update the members of *state* as defined by the instruction semantics. In essence, the code lifted by McSema simply *simulates* the binary in LLVM IR.

x86-64 Formal Semantics Our work uses the state-of-the-art x86-64 semantics developed by Dasgupta et al. [27], which presents the most complete and thoroughly tested formal semantics of x86-64 to date, and faithfully formalizes all non-deprecated, sequential user-level instructions of x86-64 Haswell instruction set architecture. This totals to 3155 instruction variants, corresponding to 774 mnemonics. Their semantics are fully executable, and includes a symbolic execution engine automatically generated by $\mathbb{K}$ framework [1].

LLVM IR Formal Semantics We use the LLVM formal semantics as defined in $\mathbb{K}$ [2], which models LLVM types (integers, composite arrays and structs, corresponding pointers), the `getelementptr` instruction (used to compute the address of an element nested within a composite), integer arithmetic & comparison operators, memory operations (`load`, `store`, and `alloca`), control flow instructions for unconditional and conditional branches, as well as function calls and returns. However, the semantics does not support: floating point, vector types, and most LLVM intrinsic functions and therefore we cannot validate the translation validation for such instructions. This is a limitation of the available LLVM semantics, and not a limitation of our work.

4 Single Instruction Validation

The single-instruction validation is responsible for validating the lifting (using McSema) of an x86-64 instruction I to LLVM IR sequence S. This is achieved by (1) Establishing variable correspondence between I and S, (2) Generating symbolic summaries individually for I and S for each output variable, (3) Generating verification conditions meant to establish semantic equivalence between the corresponding pair of summaries, and solving those using an SMT solver ($\mathbb{Z}3$). Next, we describe each one of these steps.

(1) Establishing variable correspondence: "Variable correspondence" between I and S refers to identifying the correspondence between the input/output variables of I[2] and the virtual registers in S. As described in Section 3, McSema uses a *State* structure to model the architecture state, which holds all the simulated architectural registers at different offsets of the structure.[3] Hence, the input and output variables in the context of McSema are particular *struct* fields, identified by constant offsets. As an example, for an instruction **`adcq %rax, %rbx`**, the input variables are `%cf`, `%rax` & `%rbx`, and output variables are `%rbx`, `%cf`, `%pf`, `%sf`, `%zf`, `%of` and `%af`. The following shows how these input/output registers are mapped to the McSema *State* structure.

```
// State structure type (irrelevant fields shown as ...)
%struct.State ↦ type { %struct.ArchState, ...,
    %struct.ArithFlags,..., ..., ..., %struct.GPR, ...}

// Pointers to simulated registers are accessed as below
getelementptr inbounds %struct.State, %struct.State*
    %state, i64 0, i32 m, i32 n, i32 0, i32 0

// Mapping of various simulated registers to
    getelementptr offsets
    rax ↦ m = 6  n = 1;  rbx ↦ m = 6  n = 3
     cf ↦ m = 1  n = 1;   pf ↦ m = 1  n = 3
     af ↦ m = 1  n = 5;   zf ↦ m = 1  n = 7
     sf ↦ m = 1  n = 9;   of ↦ m = 1  n = 13
```

We use the above architectural state representation of McSema to infer the "variable correspondence" between the x86-64 instruction and its corresponding lifted IR sequence[4].

(2) Generating symbolic summaries: The $\mathbb{K}$ framework takes the formal semantics of x86-64 and LLVM IR and generates symbolic execution engines automatically, which we leverage to do symbolic execution of an x86-64 instruction and the corresponding lifted LLVM IR sequence individually. Before symbolic execution, we assign symbolic values to the input variables to obtain a summary over those. For the running example of **`adcq, %rax, %rbx`**, the following shows the symbolic summary for just the output register `%rbx`[5].

[1] Given a syntax and a semantics of a language, $\mathbb{K}$ automatically generates a parser, an interpreter, a symbolic execution engine, as well as formal analysis tools such as model checkers and deductive program verifiers, at no additional effort.

[2] By input/output variables of an instruction we mean implicit and explicit register/memory/flags which are read or written.

[3] Another decompiler, fcd [7], also has the similar approach of modeling. Rev.Ng [29] models the architecture registers as LLVM globals.

[4] Similar inference for fcd [7]. For Rev.Ng [29], "variable correspondence" refers to mapping between the x86-64 registers and the LLVM globals

[5] All the values or addresses, stored in registers, memory or flags, are represented as bit-vectors which are depicted in this paper as V_W and interpreted as a bit-vector of size W and value V.

```
// VX_CF, VX_RAX and VX_RBX are the symbolic values
// assigned to input variables.
extract (
    add (
        (#if eq ( VX_CF1 , 11 ) #then
            add ( concat ( 01 , VX_RAX64 ) , 165 )
        #else
            concat ( 01 , VX_RAX64 )
        #fi)
        , concat ( 01 , VX_RBX64)
        )
    , 1 , 65 )
```

Similar symbolic summaries will be obtained for every simulated register in the lifter IR sequence, which is omitted for brevity.

The x86-64 ISA includes instructions with Repeat String Operation Prefix (e.g. **rep**, **repz** etc.) to repeat a string instruction the number of times specified in the count register or until the indicated condition by the prefix is no longer met. That is, their specification involves a loop which the symbolic execution must handle. We address this by symbolically executing those instruction with symbolic input state and comparing the summaries (using solver checks) of any single i^{th} iteration of the two loops. This suffices to establish equivalence between the two loops, by coinductive reasoning [60] and the fact that such loops are bounded by a constant thus must terminate.

(3) Generating & Solving the verification conditions: First, we convert the summaries written in $\mathbb{K}$ builtin operators to SMTLIB expressions. Given two symbolic summaries $\text{sum}^{rbx}_{x86-64}$ and sum^{rbx}_{ir} for output x86-64 register `%rbx` and corresponding simulated register, we emit a query

$$\texttt{(assert (not (= sum}^{rbx}_{x86-64}\ \text{sum}^{rbx}_{ir}\texttt{)))}$$

Similar queries are generated for all registers, memory and flags (for examples, refer to [10]). Note that we generate queries for all registers/flags, not just the ones clobbered, because the registers and flags not modified by the instruction should have equivalent summaries (which is the unmodified value of the input symbolic value).

The verification condition queries are then dispatched to the $\mathbb{Z}3$ solver. If any two summaries fail to match, we have found a bug in McSema.

Note that, even though we are using solver checks during the first phase, this should not hamper the scalability of our program validation pipeline for the following reasons. First, the instruction-level validation is done for each instruction. Thus its verification condition is much simpler than that of whole program-level validation. Second, the validation result of each instruction can be reused within a program or across different programs, thus the validation cost can be amortized, or, done offline.

5 Program-Level Validation

The goal of program-level validation is to validate the translation of the input x86-64 program P to the Mcsema-lifted LLVM IR program T. Towards that goal, the first step is to construct an alternative program T′ generated using the Compositional Lifter (Section 5.1), which are then compare for syntactic equivalence using (Section 5.2).

5.1 Compositional Lifter

The Compositional Lifter is responsible for generating the proposed LLVM IR T′ by composing the validated McSema-lifted IR sequences of the constituent binary instructions of the x86-64 program P. Importantly, the Compositional Lifter design (Algorithm 1) is simple—and took us about three man-weeks to implement.

P is disassembled (line 2) to identify function boundaries, and to decode instructions. If the disassembled instruction I_{disass} is already in Store, then its corresponding (validated) IR sequence is reused. Otherwise I_{disass} is assembled (line 6) and lifted (using McSema) to generate an LLVM IR sequence that will be validated using Phase 1. The validated IR sequences are then composed (line 17) following program order.

Algorithm 1: Compositional Lifting

Inputs :
P: x86-64 binary program.
Store: Validated pairs (<I, S>) of instruction I and lifted IR sequence S. (possibly empty)
R: Address Relocation information of binary P.
Output : Lifted IR Program T′

```
1  T′ ← φ
2  P_disass ← ObjDump(P)
3  foreach function F_disass in P_disass do
4      foreach instruction I_disass in F_disass do
5          if I_disass not in Store then
6              I ← Assembler(I_disass)
7              S ← McSema(I)
8              Perform Translation Validation of I and S
                (Phase 1)
9              if Validation successful then
10                 Add < I_disass, S > to Store
11             else
12                 Report Bug
13             end
14         else
15             Extract S from Store for I_disass
16         end
17         T′ ← Compose(T′, S, R)
18     end
19 end
20 return T′
```

Single instruction validation of control-flow instructions (line 8) The single-instruction validation strategy described in Section 4 cannot be applied naively to control flow instructions. This is because the instruction fed to McSema (line 7), for single-instruction validation, is obtained from the disassembled instruction I_{disass}, wherein the relative offsets of binary jump/call instructions are specified as labels. Hence, the binary instruction I which we obtain from assembling I_{disass} (line 6), without program context, has an incorrect relative offset, which gets propagated to the lifted IR S.

We get around this problem by symbolically executing I with symbolic values assigned to the current PC and label. That way, we get symbolic summaries agnostic of the actual (incorrect) value of the relative offset. Similarly, we symbolically execute lifted IR sequence by assigning symbolic values to the virtual register holding the simulated relative offset.

The "Compose" step Below we describe the step "Compose" (line 17), responsible for composing the IR sequences together, using a few example binary instructions.

The composed program is initially empty. Upon encountering a function label, we append the following code to it[6].

```
define %struct.Mem* @composedFunc(%struct.State*, i64,
        %struct.Mem* mem)  {}
```

For an instruction `adcq %rax, %rbx`, McSema generates the following IR sequence.

```
define internal %struct.Mem* @ADCImpl(
    %struct.Mem*, %struct.State*, i64*, i64, i64) {
    ; Does adc computation and updates destination RBX
    ; and flags (omitted for brevity)
}

define %struct.Mem* @sub_adcq_rax_rbx(%struct.State*,
        i64, %struct.Mem* ) {
 %RIP = getelementptr ... ; Compute simulated RIP address
 %RAX = getelementptr ... ; Compute simulated RAX address
 %RBX = getelementptr ... ; Compute simulated RBX address
 %VAL_RBX = load i64, i64* %RBX
 %VAL_RAX = load i64, i64* %RAX
 ; RIP update based on instruction size
 %VAL_RIP = load i64, i64* %RIP
 %UPDATED_RIP = add i64 %VAL_RIP, 3
 store i64 %UPDATED_RIP, i64* %RIP
 %retval = call %struct.Mem* @ADCImpl(
        %struct.Mem* %2, %struct.State* %0, i64* %RBX,
        i64 %VAL_RBX, i64 %VAL_RAX)
 ret %struct.Mem* %retval
}
```

The above IR sequence is then appended to the composed program as below.

```
define %struct.Mem* @composedFunc(%struct.State*,
        i64, %struct.Mem* mem)  {
    ; Code: adcq %rax, %rbx
    %loadMem = load %struct.Mem*, %struct.Mem** %mem
    %retval = call %struct.Mem* @routine_adcq_rax_rbx(
        %struct.State* %0, %struct.Mem* %loadMem)
    store %struct.Mem* %call, %struct.Mem** %mem

    ret %struct.Mem* retval
}
; Definitions of called functions omitted for brevity
```

[6] *mem* is pointer to an opaque struct type which together with return type allows ordering of memory operations if required.

A similar composition happens for most instructions, the exceptions being the control-flow data section-accessing instructions, which we elaborate on next.

Composing control-flow instructions As mentioned previously, the "labeled" control-flow assembly instructions, when assembled without program context, generate incorrect offsets which get propagated to the lifted IR. We fix this IR by replacing said incorrect relative offsets with the correct offsets.

For instance, when McSema lifts `jne .L_40087e` in isolation, it generates the following IR sequence:

```
define %struct.Mem* @sub_jne_.L_40087e(%struct.State*,
        i64, %struct.Mem* ) {
  %RIP = getelementptr ... ; Compute simulated RIP
       address
  %RIP_VAL = load i64, i64* %RIP
  ; Compute true target (using incorrect offset)
  %TARGET1 = add i64 %RIP_VAL, <incorrect_val>
  ; Compute fall-through target
  %TARGET2 = add i64 %RIP_VAL, <instr. size>
  %retval = call %struct.Mem* @JNEImpl(..., i64 %TARGET1,
        i64 %TARGET2)
  ret %struct.Mem* %retval
}
```

And the composed program with transformed IR looks like

```
define %struct.Mem* @sub_jne_.L_40087e(%struct.State*,
        i64, %struct.Mem*,
        i64 %true_tgt, i64 %false_tgt) {
  %RIP = getelementptr ... ; Compute RIP address
  %RIP_VAL = load i64, i64* %RIP
  ; Transformed code
  %TARGET1 = add i64 %RIP_VAL,  %true_tgt
  %TARGET2 = add i64 %RIP_VAL,  %false_tgt
  ; Rest same as above
}

define %struct.Mem* @composedFunc( %struct.State*,
        i64, %struct.Mem* mem)  {
  ; ... previously composed code ...
  ; Code: jne .L_40087e  RIP: 400855     Bytes: 6
  %loadMem = load %struct.Mem*, %struct.Mem** %mem
  %retval = call %struct.Mem* @routine_jne_.L_40087e(
        %struct.State* %0, %struct.Mem* %loadMem,
        i64 41, i64 6)
  store %struct.Mem* %retval, %struct.Mem** %mem
  ret %struct.Mem* %retval
}
```

Composing data-section access instructions Instructions accessing the data section, like `movq 0x602040, %rdi` with the first operand being an address, cannot be lifted correctly in isolation (without program context) because McSema does not have sufficient information to to distinguish constant from address. Single-instruction validation can only validate the fact whether the constant (which could potentially be an address) is correctly moved to the destination register. However, the problem is the program-level validation cannot use that lifting because the resulting composed IR T', with a constant moved to `%rdi`, will be different from the one lifted by McSema T, with a global address moved to `%rdi`. Upon normalization, two such IRs will be optimized differently by LLVM, leading to two syntactically divergent normalized forms, even when the initial programs were equal.

To aid in testing, we compile binaries with options to retain auxiliary information. To disambiguate between cases where a constant is a reference into the data section (e.g., an `int*`) v/s a scalar (e.g., an `int`), we use relocation information, denoted by R in algorithm 1. We allow McSema to (incorrectly) lift such instructions in isolation and then we course-correct the lifted IR by consulting the binary's relocation information to determine if an immediate operand should be considered as an address or constant — every immediate operand that is a reference has a corresponding entry in the relocation table. Missing this entry would automatically mean that the immediate is a constant.

For example, the incorrect IR generated by McSema when lifting **`movq 0x602040, %rdi`** in isolation is:

```
define %struct.Mem* @sub_movq_0x602040___rdi(
        %struct.State*, i64, %struct.Mem* ) {
    ...
    %retval = call %struct.Mem* @MOVImpl(
        %struct.Mem* %2, %struct.State* %0,
        ; data-section addr 0x602040
        ; lifted as a constant
        %i64* %RDI, i64 6299712)

    ret %struct.Mem* %retval
}
```

The address relocation information in the binary allows us to identify the address and the following correct lifting:

```
%G_0x602040_type = type <{ [8 x i8] }>
@G_0x602040= global %G_0x602040_type zeroinitializer
define %struct.Mem* @sub_movq_0x602040___rdi(
        %struct.State*, i64, %struct.Mem* ) {
    ...
    %retval = call %struct.Mem* @MOVImpl(
     %struct.Mem* %2, %struct.State* %0,
     %i64* %RDI,
     i64 ptrtoint( %G_0x602040_type* @G_0x602040 to i64))

    ret %struct.Mem* %retval
}
```

We reiterate that Compositional Lifter only uses relocation information to strengthen the generated golden reference, T′, when such information is available, e.g., during test or development time. This allows for a tighter specification, allowing our technique to find bugs at testing that would otherwise be missed. During use of Compositional Lifter in the field to validate the lifting of McSema on an unknown, blackbox binary, we do not require this additional information, at the cost of potentially missing bugs described above. Note that this is a fundamental limitation because x86-64 semantics for an instruction has no notion of types, and therefore T′, which is based on x86-64 semantics, should allow for the ambiguity and cannot enforce stricter type requirements. McSema on the other hand is never given this additional information as it is expected to work in the field where relocation information is rarely available, except in library code.

5.2 Matcher

Algorithm 2 summarizes our overall strategy to check equivalence between the IRs generated by McSema (T) and Compositional Lifter (T′). Due to the nature of the composition, T & T′ are structurally very similar. We leverage this observation to establish semantic equivalence between the two using a graph-isomorphism based algorithm assisted by normalization. The algorithm is realized by a tool we develop called the Matcher. If the Matcher fails to match T & T′, there is a *potential* bug in the lifted program.

Algorithm 2: Matcher Strategy

Inputs : **T:** McSema-lifted IR.
T′: Compositional Lifter lifted IR.
Output : **True** $\implies$ **T** & **T′** semantically equivalent
False $\implies$ **T** & **T′** *may-be* non-equivalent

```
1 foreach corresponding function pair (F,F′) in (T, T′) do
2     if !Matcher(F, F′) then
          // A potential bug in McSema while lifting F
3         return false
4     end
5 end
6 return true
```

The matcher algorithm is based on the following key observations on input IR programs, T & T′, informally stated: (I) Both exhibit identical control-flow and identical sequences of memory allocation and reference behaviors (because McSema does not modify control flow or memory operations during lifting). (II) The single-instruction validation step proves that a memory store (respectively, load) in T writes (resp., reads) the equivalent set of memory locations as does the corresponding operation in T′. This property holds for each dynamic instance of the corresponding instructions.

These two observations motivate an intuitively simple graph isomorphism strategy for proving the equivalence of T and T′. Let us name the normalized versions of the function pair, F & F′, as F_N & F'_N. The Matcher algorithm works on data dependence graphs, G_{F_N} & $G_{F'_N}$, generated from F_N & F'_N. A vertex of the graph represents an LLVM instruction and an edge between two vertices captures SSA def-use or memory dependence relations. Memory dependence edges, extracted from alias analysis results, appear between LLVM load and store instructions.

Soundness of Equivalence via Graph Isomorphism We argue informally that isomorphism of G_{F_N} & $G_{F'_N}$ implies semantic equivalence of the programs T and T′. We consider all of memory used in an execution as a single "SSA variable," which gets renamed at every store operation in the program. A store modifies some (unknown) subset of the locations in memory, and a load reads some (unknown) subset of the bytes. Given two isomorphic graphs G_{F_N} and $G_{F'_N}$, consider a

matching pair of nodes representing a store instruction S in T and the corresponding store S′ in T′. A key to the correctness argument, below, is that single-instruction validation proves that, if the initial state of memory and registers is identical before executing S and S′, then the final state of memory and registers is also identical, i.e., the same bytes have been written into the corresponding memory locations. Similarly, a matching pair of load instructions transfers identical bytes from memory to SSA registers in T and T′. The correctness argument then works as follows:

1. A node N and corresponding node N′ are equivalent because of the equivalence proof constructed by single-instruction validation (Section 4).
2. An SSA edge A → B and corresponding edge A′ → B′ carry identical bytes of data, because A is equivalent to A′ and B is equivalent to B′, by (1), above.
3. A memory edge S → L representing a *true* memory dependence (i.e., a store-to-load dependence) and corresponding edge S′ → L′ carry identical bytes of data, because S and S′ store identical bytes into identical memory locations, and L and L′ read identical bytes from identical memory locations. The argument for *anti* and *output* memory dependences is similar.

Note that the above argument is independent of the precision of any static analysis used to identify memory dependences. A highly imprecise analysis (e.g., one that says every store-load or store-store pair may be aliased) might lead to a failure to prove isomorphism between T and T′, but will not claim isomorphism if the two programs are not equivalent. In practice, we find in our experiments, described in Section 6, that the memory dependence edges from such a highly imprecise analysis do indeed reduce the success rate of the Matcher, but only by a small amount. A more precise analysis may improve the success rate, reducing the number of false negatives.

Checking Graph Isomorphism We build on a subgraph-isomorphism algorithm from Saltz et al. [59], named *dual-simulation*, to check if both G_{F_N} & $G_{F'_N}$ are subgraph-isomorphic to each other. The algorithm, in general, first retrieves initial potential match sets, Φ, for each vertex in one graph based on semantic and/or neighborhood information in the other graph. In our case, the initial potential match set for an instruction I_N in G_{F_N} contains all the instructions in $G_{F'_N}$ which have the same instruction opcode. Also, if I_N has constant operands then its potential matches must share those. Then the algorithm iteratively prunes out elements from the potential match set of each vertex based on its parents/child relations until it reaches a fixed-point. Therefore, nodes A and A′ in G_{F_N} and $G_{F'_N}$ will be marked as isomorphic if they have identical (isomorphic) sets of predecessors and successors. Two edges will be marked as isomorphic if their source and sink nodes are isomorphic.

Comparison with LLVM-MD & Peggy At this point, it is important to differentiate our approach to establish equivalence between two LLVM IR programs, using normalization followed by matching, from some of the existing approaches for validating LLVM IR-to-IR optimization passes, e.g. LLVM-MD [69] and Peggy [67], which, like our approach, move away from simulation proofs, and instead use graph isomorphism techniques to prove equivalence. Both build graphs of expressions for each program, transform the graphs via a series of "expert-provided" rewrite rules, and check for equality. The rewrite-rules mimic various compiler-IR optimizations and hence the technique is precise when the output program is an optimization of the input program and the optimizations are captured by the rewrite rules.

Compared to these approaches, our normalizer is simpler, requires no additional implementation effort, and re-uses off-the-shelf, well-tested compiler passes to reduce the two programs to syntactic equivalence. Nevertheless, the normalizer is still very effective as shown in our evaluations.

6 Evaluation

In this section, we present the experimental evaluation of single-instruction validation and program-level validation. All the experiments are run on an Intel Xeon CPU E5-2640 v6 at 3.00GHz and an AMD EPYC 7571 at 2.7GHz. We aim to address three questions through these experiments:

Q1. Is single-instruction validation by itself useful for finding bugs in a sophisticated decompiler, even though no context information is used during decompilation?
Q2. What fraction of function translations are successfully proven correct by program-level validation, and what is the false alarm rate of the tool?
Q3. Is program-level validation effective at finding additional potential bugs in a complex lifter like McSema, beyond those found by single-instruction validation alone? We studied this question using artificially injected bugs, because all real bugs were caught by single-instruction validation.

Usefulness of single-instruction validation The goal here is to validate the lifting of individual x86-64 instruction to LLVM IR sequences using McSema. Haswell x86-64 ISA supports a total of 3736 instruction variants of which 3155 are formally specified in [27]. McSema supports 1922 instructions all supported by [27]. We had to exclude 573 instruction variants because of limitations of the LLVM IR semantics [2], which does not support vector and floating points types and associated operations, and various intrinsics functions. (However, we do support intrinsics like llvm.ctpop, which is pervasively generated in the lifted IR for updating the `%pf` flag.) This brings us to a total of 1349 viable instruction variants, and we apply Translation Validation to each of them individually.

Out of the 1349 translation validations, 29 cases fail (hence are bugs) and 6 terminate with timeouts. The timeouts all correspond to `paddb`, `psubb`, and `mulq` family of instructions, and are because of complex summaries coming out of the lifted IR. For all the cases that timed out, we manually inspected them to check that the generated code fragments are semantically equivalent.

The 29 failures were all reported as possible bugs [10] to the McSema project, and all 29 have been confirmed as bugs by the McSema developers. The following gives some brief examples of some of the discrepancies we found.

First, `xaddq %rax, %rbx` expects the operations (1) temp ← `%rax` + `%rbx`, (2) `%rax` ← `%rbx`, and (3) `%rbx` ← temp, in that order. McSema performs the same operation differently as (A) old_rbx = `%rbx`, (B) temp ← `%rax` + `%rbx`, (C) `%rbx` ← temp, and (D) `%rax` ← old_rbx. This will fail to work when the operands are the same registers.

Second, for instruction `andnps %xmm2, %xmm1`, the Intel Manual [9] says the implementation should be `%xmm1` ← ~`%xmm1` & `%xmm2`, whereas McSema interchanges the source operands.

Third, for `pmuludq %xmm2, %xmm1` instruction, both the higher and lower double-words of the source operands need to multiply, whereas McSema multiplies just the lower double-words.

Fourth, for `cmpxchgl %ecx, %ebx`, McSema compares the entire 64-bit `%rbx` (instead of just `%ebx`) with the accumulator Concat(0x00000000, `%eax`).

Finally, for `cmpxchgb %ah, %al`, the lower 8-bits of `%rax` should be replaced with the higher 8-bits at the end of the instruction, whereas McSema keeps them unchanged.

Program-level validation: Success rate and false alarms: The goal here is to validate the translation of programs, one function at a time, using the Matcher strategy (Section 5.2). For this purpose, we use programs from LLVM-8.0 "single-source-benchmarks". The benchmark suite consists of a total of 102 programs, of which 11 cannot be lifted by McSema due to missing instruction semantics. The remaining programs contain 3062 functions in total. We excluded 714 functions because the corresponding binary uses floating point instructions which are not supported in the LLVM formal semantics and so could not be validated using single instruction validation. This brings us to a grand total of 2348 usable functions which we compiles (using both gcc/clang) and feed the binaries to Compositional Lifter and McSema for lifting. The length of (inlined) lifted IR functions ranges from 86 – 21729, with an average of 777.

Of the 2348 usable functions, our matcher can correctly and formally prove the correctness of the translations for 2189 using graph-isomorphism, i.e., a success rate of 93%. We manually checked the remaining 159 and found them to be false alarms. Overall, a false alarm rate of about 7% is low enough that we believe our Matcher can be of practical use for validation and testing of a decompiler.

Program-level validation: Effectiveness at finding bugs: In order to evaluate whether our strategy is effective in finding bugs in lifters like McSema, we manually injected bugs in their implementation. The injected bugs covers the following aspects of McSema's lifting: (1) *Instruction lifting*: McSema while lifting uses code templates to generate IR sequences for each instruction. The injected bug forces the tool to choose wrong templates. The injected bug is targeted to affect the translation of 491 unique instruction mnemonics that we collected from the compiled binaries of our evaluation test-suite. (2) *Inferring data-section access constant*: McSema uses information from IDA [37] to know if an immediate operand used in data-section access instructions is a constant or a memory address. The introduced bug forces McSema to take the wrong decision. (3) *Maintaining correct data dependence among instructions*: The injected bug changes the order in which instructions are lifted, causing data dependences between the dependent instructions to be violated (4) *Correctness of hoisting address computation code*: As mentioned earlier, McSema hoists the address computations of simulated registers back to the function entry block, to be reused by later instructions throughout the function. Our injected bug forces the use of incorrect simulated address for general purpose registers and flags.

Each of the above bugs are injected one at a time and in combination and the resulting buggy lifter is tested against the Compositional Lifter on the same evaluation test-suite before. All the injected bugs are correctly detected by the Matcher, showing that program-level validation is useful for detecting bugs, not just proving correctness.

Note that only the first of these bugs would be caught by single-instruction validation: the binary instruction semantics would not match with the LLVM IR sequence semantics in that case. The second case would (in general) produce equivalent semantics between the X86 and the LLVM IR sequence, and so single-instruction validation would not detect the bug. The third and fourth bugs inherently span multiple instructions, and generate wrong code even if the individual instructions are translated correctly, so single-instruction validation would *not* be able to detect them.

7 Discussion

In this section we discuss some limitations of our work and avenues for future work.

Incomplete LLVM Semantics The LLVM IR semantics [2] is currently under development and does not support all LLVM abstractions, e.g., vector and floating point types and their associated operations, and various intrinsics functions at the time of writing the paper. This is a limitation of existing semantics and we believe the verification of lifted instructions

that use such unsupported features will work out-of-the-box when semantics are available.

Formally Verified Normalizer Our current implementation of the normalizer uses a small number (of 15) LLVM passes to improve syntactic matching between the McSema generated T and T' proposed by Compositional Lifter. To prove soundness, these passes need to be formally verified to perform only semantic preserving transformations. An alternative, more promising approach is to develop simple graph transformations on SSA graphs to mimic the transformations of LLVM passes and formally prove the transformations preserve program semantics. We leave this to future work.

Extending to Other Lifters Our current work focuses on McSema, the most mature, open-source, binary to LLVM IR lifter. However, there are a plethora of other lifters that are not formally verified. Extending our work to support these systems is important for two reasons: (i) improving the trust in binary lifters, and (ii) the improvements made to our system would make it more generic enough for future binary lifters to get validation for (nearly) free. We believe that this is mainly engineering effort that involves customization of Compositional Lifter to capture the idiosyncrasies of various lifters.

8 Related Work

Traditionally, translation validation [56] uses compiler instrumentation to help generate a simulation relation to prove the correctness of compiler optimizations [40, 57]. In our initial attempt to solve the problem of translation validation of the lifting of x86-64 program, we tried to borrow insights from such efforts. However, to be effective, we believe our validator should not instrument the lifter mainly because most the available lifters, being in early development phase, are updated and improved at a frantic pace. Without instrumentation, such simulation relations can be inferred by collecting constraints from the input and output programs (as demonstrated in Necula's work [54]). However, in the context of translation validation of binary lifting, such inference is not straight-forward mainly because the two program (x86-64 binary and lifted IR) are structurally very different with potentially different number of basic blocks [7]. Unsurprisingly, a similar challenge poses a hard requirement of branch equivalence in Necula's approach. Consequently, we decided to move away from simulation-based validation approaches.

All the previous efforts, establishing the faithfulness of the binary lifters, can be broadly categorized to be based on (1) Testing, or (2) Formal Methods.

[7] Instructions like `adcq` exhibit different semantic behavior based on input values and hence generate additional basic blocks upon lifting, which are not explicit in the binary program

8.1 Testing based Approaches

This approach is similar to black-box testing in software engineering. Most notable work include Martignoni et al. [50–52] and Chen *et al.* [22].

Martignoni et al. [51, 52] proposes hardware-cosimulation based testing on QEMU [16] and Bochs [46]. Specifically, they compared the state between actual CPU and IA-32 CPU emulator (under test) after executing randomly selected test-inputs on randomly chosen instructions to discover any semantic deviations. Although, a simple and scalable approach, it's effectiveness is limited because many semantics bugs in binary lifters are triggered upon a specific input and exercising all such corner inputs, using randomly generated test-cases, is impractical.

In general, such testing based approaches are unsuitable for validating whole program (or even basic block) translations because even a correctly translated program may not always produce exactly the same output as the original program due to the differences in modeling of the architectural states in the translated program (or basic block) vs the original program. Although, it is possible to engineer out such architecture-state-comparison problems, but still these approaches might not detect some intermediate mistranslated instructions which are course corrected at the end. As an example, suppose a register is assigned values twice in a program and the first assignment is mistranslated but the second assignment statement is translated correctly. Comparing the architecture states at the end of the program (or basic block) may not discover the mis-translation.

Chen *et al.* [22] proposed validating the static binary translator LLBT [63] and the hybrid binary translator [62], retargeting ARM programs to x86 programs. First, an ARM program is translated offline to x86 program (via an intermediate translation to LLVM IR). Next, the translated x86 binary is executed directly on a x86 system while the original ARM binary runs on the QEMU emulator. During run time, both the ARM binary and the translated x86 binary produce a sequence of architecture states, which are compared at the granularity of single instruction after solving the architecture-state-comparison problem, as mentioned above. The validator is evaluated using the ARM code compiled from EEMBC 1.1 benchmark. Like previous approach, the validation of single instruction's translation is based on testing and hence shares the same limitation of not being exhaustive.

Martignoni *et al.* [50] applied symbolic execution on a Hi-Fi ("faithful and more complete in terms of IA-32 ISA") emulator[46]'s implementation of an instruction semantics to generate high-fidelity test-inputs to validate a "buggier and less complete" Lo-Fi emulator [16], by executing the binary instruction twice, once on a real hardware and next on the Lo-Fi emulator, and the output states are matched. However, the work [50] does not aim to validate the translation of x86-64 programs, which is one of out primary contributions.

Note that an approach as above cannot scale naturally to binary function validation; A set of high-coverage test-inputs for all the constituent instructions of a function cannot trivially derive high-coverage test-inputs for the whole function.

8.2 Formal Methods based Approaches

Followings are the effort to establish strong guarantees for binary translations using formal methods.

MeanDiff [41] proposed an N-version IR testing to validate three binary lifters, BAP [19], BINSEC [15], and PyVEX [3] by comparing their translation of a single binary instruction to BIL, DBA, and VEX IRs respectively. The individual IRs are then converted to common IR representations which are then symbolically executed to generate symbolic summaries for comparison using a SAT solver. The above approach shares the same fundamental limitations of any differential testing techniques. For example, if all the binary lifters are in sync on the behavior of a particular instruction, we get more confidence in correct implementation of that instruction's semantics in all of them, but we cannot rule out the possibility of all being incorrect. Also, even if there a disagreement in the behavior of two or more translators, still it might just be a false alarm in case all the candidates are buggy. However, the work [41] is primarily focused on validating single instruction and the problem of handling multiple instruction is left as future work.

Moreover, as candidly mentioned in the paper [41], one of the motivations for relying on differential testing is that there were no formal specification of x86-64 ISA at the time of writing the paper. Whereas we do not have such limitation because of the formal x86-64 ISA specification [27] made public recently. Empowered with that, we can build a symbolic formula that encodes all execution paths of an IR instance lifted from a single machine instruction and then check if the symbolic formula matches the formal specification of the instruction, which is exactly what we did in this work.

The work closest to ours, in term of the goals, is the translation verifier, Reopt-vcg [39], which is developed to cater the verification challenges specific to the translator Reopt [4]. The verifier, which validates the translations at basic-block level, is assisted by various manually written annotations, which are prone to errors. Such annotations could have been generated by instrumenting the lifter. Contrary to that, our approach does not need any such instrumentations, thereby avoided the overhead of maintaining instrumentation patches whenever the lifter design/implementation is updated. Moreover, the validator uses the semantics definitions of a small subset of x86-64, which in turn limits its applicability to small programs. We avoided this limitation by incorporating the most complete and heavily tested x86-64 semantics [27] available to date.

9 Conclusion

In conclusion, we demonstrated that validation of lifters w/o instrumentation or heavy-weight equivalence checking is feasible. The design is based on a simple insight: Formal translation validation of single machine instructions is not only practical, but also can be used as a building block for scalable full-program translation validation. Our experimental evaluation shows that single instruction validation is valuable in finding real bugs in a mature lifter. With single-instruction validation as build-block, we propose a scalable approach for full-program translation validation; One that does not require heavyweight symbolic execution or theorem provers and based on normalization and isomorphic graph matching.

References

[1] 1996. ARM Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html. Last accessed: November 23, 2019.

[2] 2012. Formal semantics of LLVM IR in K. https://github.com/kframework/llvm-semantics. Last accessed: November 23, 2019.

[3] 2013. Python bindings for Valgrind's VEX IR. https://github.com/angr/pyvex. Last accessed: November 23, 2019.

[4] 2014. reopt: A tool for analyzing x86-64 binaries. https://github.com/GaloisInc/reopt. Last accessed: November 23, 2019.

[5] 2018. Angr: A powerful and user-friendly binary analysis platform! http://angr.io/. Last accessed: November 23, 2019.

[6] 2018. Comparison with other machine code to LLVM bitcode lifters. https://github.com/lifting-bits/mcsema#comparison-with-other-machine-code-to-llvm-bitcode-lifters. (2018). Last accessed: November 23, 2019.

[7] 2018. fcd: An optimizing decompiler. https://zneak.github.io/fcd/. Last accessed: November 23, 2019.

[8] 2018. Remill: Library for lifting of x86, amd64, and aarch64 machine code to LLVM bitcode. https://github.com/trailofbits/remill. Last accessed: November 23, 2019.

[9] 2019. Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm. Published on October 12, 2016, updated on September 26, 2019.

[10] 2019. Supplemental Materials Submitted Along with Paper. Link to repository removed for double blind review. Last accessed: November 23, 2019.

[11] Sergi Alvarez. 2018. Radare2. https://rada.re/r/. Last accessed: November 23, 2019.

[12] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[13] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[14] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[15] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. 2011. The BINCOA Framework for Binary Code Analysis. In *Proceedings of the 23rd International*

Conference on Computer Aided Verification (CAV'11). Springer-Verlag, Berlin, Heidelberg, 165–170. http://dl.acm.org/citation.cfm?id=2032305.2032318

[16] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[17] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '03)*. IEEE Computer Society, Washington, DC, USA, 265–275.

[18] Derek L. Bruening. 2004. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0807735.

[19] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[20] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31

[21] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, Washington, DC, USA, 725–741. https://doi.org/10.1109/SP.2015.50

[22] Jiunn-Yeu Chen, Wuu Yang, Bor-Yeh Shen, Yuan-Jia Li, and Wei-Chung Hsu. 2015. Automatic Validation for Binary Translation. *Comput. Lang. Syst. Struct.* 43, C (Oct. 2015), 96–115. https://doi.org/10.1016/j.cl.2015.05.002

[23] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[24] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. 2005. Semantics-Aware Malware Detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (SP '05)*. IEEE Computer Society, Washington, DC, USA, 32–46. https://doi.org/10.1109/SP.2005.20

[25] Cristina Cifuentes and Mike Van Emmerik. 2000. UQBT: Adaptable Binary Translation at Low Cost. *Computer* 33, 3 (March 2000), 60–66. https://doi.org/10.1109/2.825697

[26] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. 2008. Tupni: Automatic Reverse Engineering of Input Formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820

[27] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. 2019. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. https://doi.org/10.1145/3314221.3314601

[28] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[29] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction (CC 2017)*. ACM, New York, NY, USA, 131–141. https://doi.org/10.1145/3033019.3033028

[30] Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, and Alexander Meduna. 2011. Design of aÂăRetargetable Decompiler for aÂăStatic Platform-Independent Malware Analysis. In *Information Security and Assurance*, Tai-hoon Kim, Hojjat Adeli, Rosslin John Robles, and Maricel Balitanas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–86.

[31] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. 2007. Dynamic Spyware Analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, USA, Article 18, 14 pages. http://dl.acm.org/citation.cfm?id=1364385.1364403

[32] Alexey Loginov Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta. 2018. Evolving Exact Decompilation, In Binary Analysis Research (BAR), 2018. *Ndss* February. https://doi.org/10.14722/ndss.2018.23xxx

[33] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. 2006. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 75–88. http://dl.acm.org/citation.cfm?id=1298455.1298463

[34] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. 2011. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11)*. IEEE Computer Society, Washington, DC, USA, 347–356. https://doi.org/10.1109/WCRE.2011.49

[35] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 293–306. http://dl.acm.org/citation.cfm?id=1404014.1404039

[36] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of NDSS (Network and Distributed Systems Security)*. 151–166.

[37] Ilfak Guilfanov. 2008. Decompilers and Beyond. In Black-Hat USA.

[38] Laune C. Harris and Barton P. Miller. 2005. Practical Analysis of Stripped Binary Code. *SIGARCH Comput. Archit. News* 33, 5 (Dec. 2005), 63–68. https://doi.org/10.1145/1127577.1127590

[39] Joe Hendrix, Guannan Wei, and Simon Winwood. 2019. Towards Verified Binary Raising. In *Workshop on Instruction Set Architecture Specification (co-located with ITP 2019)*.

[40] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. 2006. A PVS Based Framework for Validating Compiler Optimizations. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM '06)*. IEEE Computer Society, Washington, DC, USA, 108–117. https://doi.org/10.1109/SEFM.2006.4

[41] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 353–364. http://dl.acm.org/citation.cfm?id=3155562.3155609

[42] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206. http://dl.acm.org/citation.cfm?id=647253.720293

[43] Christopher Kruegel, William Robertson, and Giovanni Vigna. 2004. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*. IEEE Computer Society, Washington, DC, USA, 91–100.

https://doi.org/10.1109/CSAC.2004.19

[44] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[45] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 175–183. https://doi.org/10.1109/ISPASS.2010.5452024

[46] Kevin P. Lawton. 1996. Bochs: A Portable PC Emulator for Unix/X. *Linux J.* 1996, 29es, Article 7 (Sept. 1996). http://dl.acm.org/citation.cfm?id=326350.326357

[47] Zhiqiang Lin and Xiangyu Zhang. 2008. Deriving Input Syntactic Structure from Execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 83–93. https://doi.org/10.1145/1453101.1453114

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[49] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A Full System Simulation Platform. *Computer* 35, 2 (Feb. 2002), 50–58. https://doi.org/10.1109/2.982916

[50] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2150976.2151012

[51] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/1831708.1831730

[52] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA '09)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/1572272.1572303

[53] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[54] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314

[55] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89, 2 (2003), 44 – 66. https://doi.org/10.1016/S1571-0661(04)81042-9 RV '2003, Run-time Verification (Satellite Workshop of CAV '03).

[56] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, Berlin, Heidelberg, 151–166. http://dl.acm.org/citation.cfm?id=646482.691453

[57] Xavier Rival. 2004. Symbolic Transfer Function-based Approaches to Certified Compilation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/964001.964002

[58] Andrew Ruef and Artem Dinaburg. 2014. Static Translation of X86 Instruction Semantics to LLVM with McSema. https://github.com/trailofbits/mcsema

[59] M. Saltz, A. Jain, A. Kothari, A. Fard, J. A. Miller, and L. Ramaswamy. 2014. DualIso: An Algorithm for Subgraph Pattern Matching on Very Large Labeled Graphs. In *2014 IEEE International Congress on Big Data*. 498–505. https://doi.org/10.1109/BigData.Congress.2014.79

[60] Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA.

[61] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. 2013. Native x86 Decompilation Using Semantics-preserving Structural Analysis and Iterative Control-flow Structuring. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 353–368. http://dl.acm.org/citation.cfm?id=2534766.2534797

[62] B. Shen, J. You, W. Yang, and W. Hsu. 2012. An LLVM-based hybrid binary translation system. In *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 229–236. https://doi.org/10.1109/SIES.2012.6356589

[63] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. 2012. LLBT: An LLVM-based Static Binary Translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '12)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2380403.2380419

[64] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).

[65] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[66] Amitabh Srivastava and Alan Eustace. 1994. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/178243.178260

[67] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based Translation Validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742. http://dl.acm.org/citation.cfm?id=2032305.2032364

[68] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. https://doi.org/10.1145/358198.358210

[69] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating Value-graph Translation Validation for LLVM. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[70] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *NDSS*.

[71] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 79–93. https://doi.org/10.1109/SP.2009.25

[72] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 116–127. https://doi.org/10.1145/1315245.1315261

[73] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 559–573. https://doi.org/10.1109/SP.2013.44

[74] Mingwei Zhang and R Sekar. [n. d.]. Control Flow Integrity for COTS Binaries. ([n. d.]).