Scalable Validation of Binary Lifters

Sandeep Dasgupta
University of Illinois at Urbana-Champaign
USA
sdasgup3@illinois.edu

Sushant Dinesh
University of Illinois at Urbana-Champaign
USA
sdinesh2@illinois.edu

Deepan Venkatesh
University of Illinois at Urbana-Champaign
USA
deepanv2@illinois.edu

Vikram S. Adve
University of Illinois at Urbana-Champaign
USA
vadve@illinois.edu

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
USA
cwfletch@illinois.edu

Abstract

Validating the correctness of binary lifters is pivotal to gain trust in binary analysis, especially when used in scenarios where correctness is important. Existing approaches focus on validating the correctness of lifting instructions or basic blocks in isolation and do not scale to full programs. In this work, we show that formal translation validation of single instructions for a complex ISA like x86-64 is not only practical, but can be used as a building block for scalable full-program validation. Our work is the first 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 the number of instructions tested for correctness. Next, we develop a novel technique that uses validated instructions to enable program-level validation, without resorting to performance-heavy semantic equivalence checking. Specifically, we compose the validated IR sequences using a tool we develop called *Compositional Lifter* to create a reference standard. The semantic equivalence check between the reference and the lifter output is then reduced to a graph-isomorphism check through the use of semantic preserving transformations. The translation validation of instructions in isolation revealed 29 new bugs in McSema – a mature open-source lifter from x86-64 to LLVM IR. Towards the validation of full programs, our approach was able to prove the translational correctness of 2254/2348 functions taken from LLVM's single-source benchmark test-suite.

PLDI '20, June 15–20, 2020, London, UK
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7613-6/20/06...$15.00
https://doi.org/10.1145/3385412.3385964

CCS Concepts: • **General and reference → Validation**; • **Software and its engineering → Formal language definitions**; **Compilers**.

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

ACM Reference Format:
Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable Validation of Binary Lifters. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3385412.3385964

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 [17, 80]. Analyzing binary code is important in certain subfields of software engineering and security tools, including binary instrumentation [21, 22, 51, 54, 63], binary re-targeting [29, 36], software hardening [25, 38, 84, 85], software testing [16, 28, 39], CPU emulation [19, 55], automated reverse engineering [6, 13, 30, 53, 66, 69, 82], sand-boxing [37, 48, 83], profiling [41, 78], and automatic exploit generation [24].

To reason about binary code, binary analysis frameworks, e.g., [9, 13, 23, 66, 74], 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 instructions 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 [23, 77], or as open-source code [8, 9, 13, 34, 66, 74], use such a lifter as the first step in their pipeline.

Developing a lifter, especially for complex modern ISAs, is challenging and error-prone [59], mainly because manually encoding the effects of a vast number of instructions (and their variants) is hard. This is made even harder because the informal specifications provided by the hardware manufacturers of most of the ISAs run into thousands of pages [2, 11], have mistakes [32], 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 thoroughly, as generally, 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 binary lifters through effective validation techniques is also generally crucial for developers of decompilers, especially for complex ISAs.

Surprisingly, there has been very limited work to date on validating the correctness of binary decompilers, and that work has focused on the translation of single instructions or basic-blocks. 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 [26, 57, 58], (ii) do not scale to full program translation validation [46, 56], or (iii) require modification of the lifting frameworks under test to emit additional information required to prove correctness [42].

Our goal 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). Our approach is inspired by a key observation that most decompilers [4, 6, 8, 13, 20, 23, 27, 34, 35, 47, 49, 66, 76, 77] are 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. We capitalize on this observation by deriving the following insight:

Formal translation validation of single machine instructions can be used as a building block for scalable full-program translation validation.

With that insight, the overview of our approach is as follows: First, given the formal semantics of x86-64 and LLVM IR, we formally validate the translational correctness of individual instructions by asserting equivalence of symbolic summaries of each x86-64 instruction and the corresponding lifted LLVM IR sequence using an SMT solver. If the summaries are equivalent, the lifted LLVM IR sequence is the correct translation of the instruction, else the equivalence check fails, and the solver generates a counter-example that we use to report a bug. Second, for program-level validation, we compose these validated sequences of instructions using a tool we developed, called Compositional Lifter, to form a *reference translation* (T′). Next, we transform T (the IR program generated by the lifter we wish to validate) and T′, one function at a time, using semantics-preserving transformations, to prune any syntactic differences except for the names of virtual registers and the order of non-dependent instructions. Finally, we check if the data dependence graphs corresponding to the transformed function pairs F & F′, of T and T′ resp., are isomorphic in which case the two lifted sequences (T & T′) are deemed as semantically equivalent.

The four key contributions of our work are:

1. 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 *most comprehensive* coverage in terms of the number of instructions tested when compared to earlier work [46]. We experimentally verify that such single instruction validation is capable of finding bugs, and in particular, we find discrepancies in the lifting of 29 instructions in McSema [9, 66], a well-tested, mature [7], and open-source lifter from x86-64 to LLVM IR, clearly showing the effectiveness of our technique. All of these discrepancies have been confirmed as bugs in the lifter by the McSema developers.
2. Given a lifter (*D*), we show that we can construct an alternate lifter, called *Compositional* Lifter (D′), which essentially concatenates the lifted IR sequences for individual instructions (which are proven correct by part 1, above) to provide a reference translation T′. We do not provide a formal guarantee that the above composition T′, of validated lifted IR sequences, is the correct translation of P. However, the tool is exceedingly simple to construct and we added it to our trust base. More importantly, the translation T′ generated by the Compositional Lifter is syntactically very similar to the translation (say T) generated by the lifter we aim to validate. Such code similarity serves as a foundation for scaling translation validation to full programs.
3. 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 canonicalizer— for each pair of functions F and F′ of T and T′, say canonical(F) and canonical(F′), such that, stated informally, we can check if F and F′ are semantically equivalent by checking whether canonical(F) and canonical(F′) have isomorphic data dependence graphs. If such a canonicalizer exists, then we can reduce the problem of

program-level semantics checking to a much cheaper graph-isomorphism check! In this work, we construct an approximation of a canonicalizer, called Transformer, out of a very short sequence of 17 manually selected LLVM passes, of which 11 are transformation passes. The lifted functions F & F′ being syntactically very close to begin with, we found such an approximation to work quite well. For example, the data dependence graphs extracted from the optimized versions when matched for graph isomorphism yield a low false-alarm rate of 7% based on our evaluation setup (Section 6). Moreover, we improved the matching results by employing an autotuner to automatically discover pass sequences, for each pair of F & F′, by searching over the manually-identified 17 optimization passes. This approach reduces the false-alarm rate from 7% to 4%, and with fewer passes, on average 8 instead of 17.

Our validation framework is publicly available at [31].

The rest of this paper proceeds as follows. The next section gives a high-level overview of our approach, followed by information on building blocks used in our work (Section 3), description of our approach (formal single-instruction translation validation, Section 4 and full-program translation validation, Section 5), detailed experimental evaluations (Section 6), discussion on the limitations and avenues for future work (Section 7), details on related work (Section 8), and conclusion (Section 9).

2 Approach Overview

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

Applicability of our Approach. 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 [50], VEX IR [63] etc., as long as (a) formal semantics for both the ISA and the target languages are available, and (b) the target language can be transformed to a canonical representation through a series of semantics-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 [66]. Our canonicalizer is approximated using a subset of LLVM optimization passes. Other notable lifters [8, 34] 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 we aim to validate the lifted code and 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 typical case of compiler-generated binaries [14].

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

Single-Instruction Translation Validation The goal of single-instruction translation validation, as shown in Figure 1(a), is to formally validate the translation of individual instructions of P in isolation using the following steps: (i) Each x86-64 instruction I is lifted to an LLVM IR sequence S using the lifter D (McSema in our case), (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 x86-64 and IR, we perform symbolic execution to generate symbolic summaries for I and S, (iv) Lastly, we say S is the correct translation of I if the corresponding summaries are semantically equivalent. We employ the $\mathbb{Z}3$ [33] solver for the equivalence checks. If the two summaries mismatch, meaning we find a bug, which is then reported. Otherwise, we add the pair <I, S> to a database (called *Store*), keyed by I, allowing reuse of the validation result.

Program-Level Validation The program-level validation, as shown in Figure 1(b), aims to validate that the lifted LLVM program T, generated by a lifter D, is the correct translation of binary program P. The key idea behind the validation strategy is to propose an alternate LLVM program T′ as a reference translation to be compared against T. The translation T′ is generated using a tool we developed, called Compositional Lifter, by carefully composing the validated lifted IR sequences corresponding to the individual binary instructions of P. The validated IR sequences are provided by the single-instruction translation validation technique above. *The composition T′ preserves the data- & control-flow of the original binary-program P and, more importantly, is syntactically very close to the original lifted IR T.*

Next, we seek to compare T and T′ *one function at a time*. Towards that goal, we use a set of 17 manually discovered LLVM optimization passes to close the syntactic gap between every pair of corresponding functions, F & F′ of T & T′ resp., except for the names of virtual registers and the order of non-dependent instructions. We compare the data dependence graphs extracted from the optimized pair of functions by a Matcher based on graph isomorphism (refer to Section 5.2). The isomorphism of data dependence graphs, for each pair of optimized functions, implies that the original lifted IR T is semantically equivalent to the reference translation T′, and therefore T is the correct lifting of the input binary program. Otherwise, the automatic validation fails, and mismatch reported as a potential bug for further analysis. The reason that we can get false alarms (i.e., even if F and F′ are semantically equivalent, the data dependence graphs extracted from their optimized versions are not isomorphic) because the selected

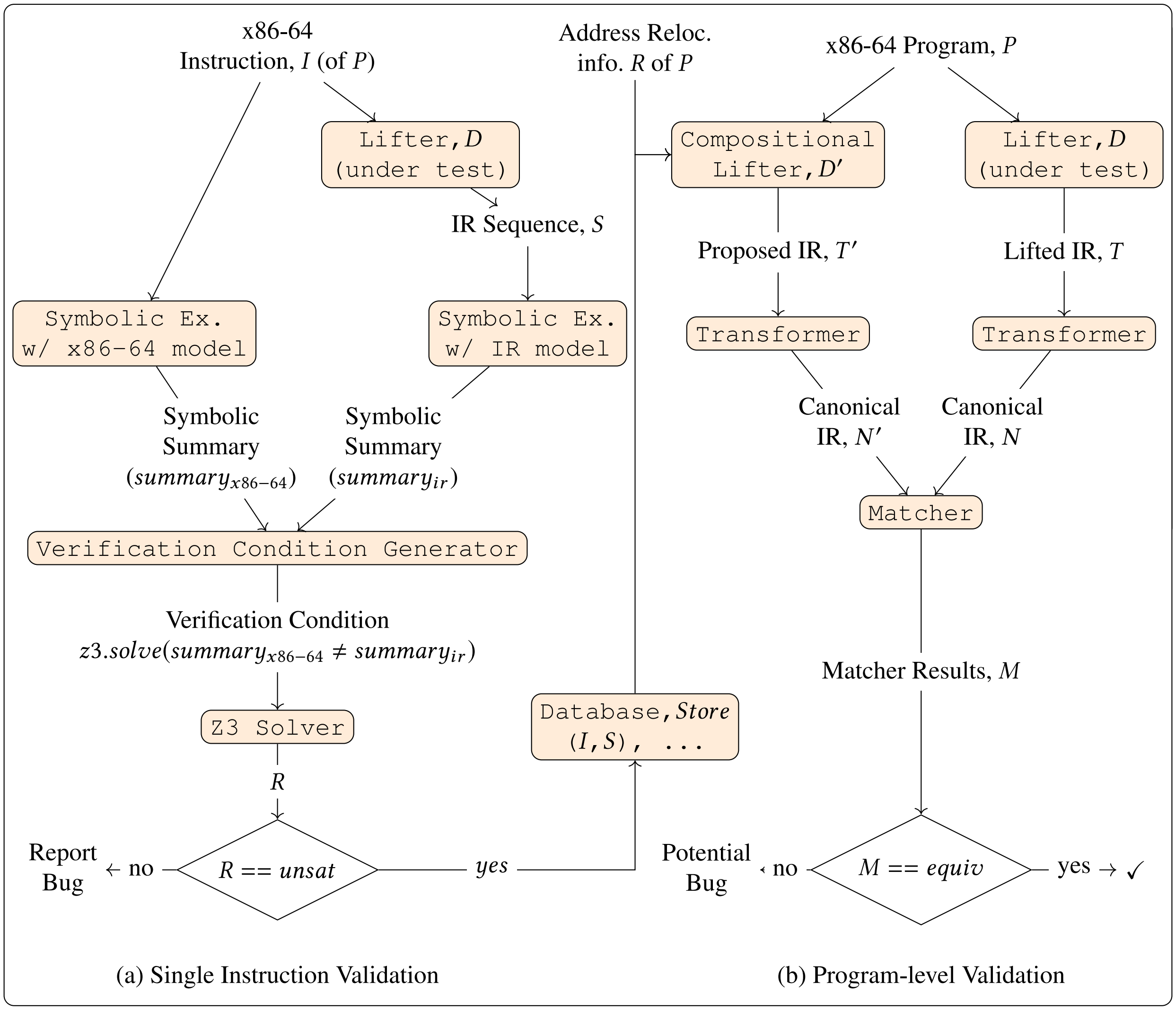

Figure 1. Overview diagram of the translation validation framework

LLVM passes may not be effective in reducing F and F′ to isomorphic graphs.

Composing the Techniques In essence, 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 translation validation. However, the ordering between the two techniques does not matter, i.e., single-instruction translation validation may be done offline; either 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 used in our work: (i) The binary lifter under test, McSema, (ii) The formal x86-64 semantics, and (iii) The formal LLVM IR semantics.

McSema. McSema [66] 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) front end, and (b) back end. The front end is responsible for parsing, loading, and disassembling a binary and exports an interface to the back end to query for the required information, e.g., the defined symbols, sizes of various binary sections, instruction listings, etc. The back end then uses this information and Remill [9] library to lift the individual instructions. McSema supports multiple different front ends with IDA Pro being the most robust, and supported option.

Conceptually, the implementation of McSema's back end is fairly straightforward: McSema exposes all of the architecture state, i.e., the program registers, conditional flags, and program memory, through an LLVM *struct*, aptly named *State*, which is passed as an argument to every lifted function. McSema simply scans through the disassembly of the binary and lifts each instruction one by one, emitting code to read and/or update the members of the struct based on the semantics of the lifted instruction. In essence, the code lifted by McSema simply encodes the operational semantics of the binary in LLVM IR.

x86-64 Formal Semantics. Our current work uses state-of-the-art x86-64 semantics, developed in our previous open-sourced work [32], which presented the most complete, 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. The specification covers 774 mnemonics, and each mnemonic admits several variants (3155 in total), depending on the types (i.e., register, memory, or constant) and the size (i.e., the bit-width) of operands. The semantics, defined in $\mathbb{K}$ [70], is executable (i.e. allows concrete execution), and comes with a symbolic execution engine automatically generated by the $\mathbb{K}$ framework[1].

LLVM IR Formal Semantics. We use the LLVM formal semantics [45], defined in $\mathbb{K}$, which models LLVM types (integers, composite arrays, structs and their corresponding pointers), the `getelementptr` instruction (used to compute the address of an element nested within a aggregate data-structure), 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. As a result, we cannot validate the translation validation of certain binary instructions whose lifted IR includes such unsupported constructs. This is a limitation of the available LLVM semantics and not a limitation of our work.

4 Single-Instruction Translation Validation

The single-instruction translation 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` and those of `S`. By input (resp., output) variables of an instruction we mean implicit and explicit register/memory/flags which are read (resp., written). By input (resp., output) variables of an lifted IR sequence `S` we mean the IR variables which are used to simulate the input (resp., output) variables of `I`. This information is valuable in setting up pre-conditions over corresponding input variables and post-conditions over output variables, thereby assisting the equivalence proofs between `I` and `S`.

As described in Section 3, McSema models the hardware architecture state using a *State* structure which holds all the simulated hardware registers at different offsets in the structure. 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 in lifted LLVM code.

```
// State structure type with irrelevant fields replaced
// with "...".
// The nested type "struct.GPR", at offset 6, models the
// general-purpose simulated registers. Similarly, the
// type "struct.ArithFlags", at offset 2, models the
// simulated status flags.
%struct.State ↦ type { %struct.ArchState, ...,
    %struct.ArithFlags,..., ..., ..., %struct.GPR, ...}

// Pointers to simulated registers (or flags) are
// computed using LLVM's getelementptr instruction.
// The constant operands m and n are offsets to index to
// the different nested elements of an object pointed to
// by a base pointer "%state" of the above type,
// denoting field n within the nested struct at
// field m of structure "%struct.State".
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 how the hardware registers or flags in the binary instruction corresponds to the simulated version of those in the corresponding lifted IR[2].

(2) Generating symbolic summaries: The $\mathbb{K}$ framework takes the $\mathbb{K}$-specification of x86-64 (resp., LLVM IR) as input and automatically generates a symbolic execution engine which we leverage to do symbolic execution of an x86-64 instruction (resp., the corresponding lifted LLVM IR sequence).

[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] Like McSema, fcd [8] has a similar approach to model the architectural state and infer variable correspondence. In case of Rev.Ng [34], the architecture registers are modeled as LLVM globals and variable correspondence refers to the mapping between x86-64 registers with those globals.

The result of symbolic execution on an x86-64 instruction (resp., the corresponding lifted IR) is a set of summaries capturing the output behaviors corresponding to each register, flag, and clobbered memory (resp., the simulated version of those in the lifted IR), expressed using $\mathbb{K}$ builtin operators such as `add`, `concat` and `extract`, over the symbolic values assigned to the input variables. For the running example of **`adcq %rax, %rbx`**, the following shows the symbolic summary corresponding to the output register `%rbx`[3] (summaries of other registers and flags are omitted).

```
// V_CF1, V_RAX64 and V_RBX64 are the symbolic values
// assigned to input variables. The subscript denotes
// the bit-width of the value. "extract" returns bits
// 1..64, where bit 0 is the most significant bit.
extract (
    add (
        (#if eq ( V_CF1 , 11 ) #then
            add ( concat ( 01 , V_RAX64 ) , 165 )
        #else
            concat ( 01 , V_RAX64 )
        #fi)
        , concat ( 01 , V_RBX64),
        )
    1 , 65 )
```

Similar symbolic summaries will be obtained for the simulated registers and flags in the lifter IR sequence, which is omitted as well for brevity.

Most x86-64 instructions require a bounded (and small) number of operations. However, 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 that the symbolic execution must handle. Conceptually, such loops can be realized using a *for* loop with index as the loop count decreasing by one in every iteration and the body consists of a *if* check which can break the loop if the indicated condition is met or the index reduces to zero. These loops are bounded by the maximum value the count register can hold and are simple as the index cannot change in any other ways; thus the x86-64 instruction will trivially terminate. In order to prove the equivalence of the translation of such an instruction, we first set up the precondition asserting that the register or memory value, corresponding to the loop trip count, and the corresponding simulated register in lifted IR are equivalent. Next, we symbolically execute the instruction and its corresponding lifted IR with a 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 co-inductive reasoning [68] to check the behavior of corresponding loops evolves in lock-step, and the fact that such loops are bounded by a constant thus must terminate [4].

(3) Generating & Solving the verification conditions: First, we convert the summaries written in $\mathbb{K}$ builtin operators to SMTLIB expressions. Given two symbolic summaries $\text{summary}_{x86-64}^{rbx}$ and $\text{summary}_{ir}^{rbx}$, for output x86-64 register `%rbx` and corresponding simulated register, we emit a satisfiability query as follows, to be solved by an SMT solver like $\mathbb{Z}3$,

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

Similar queries are generated for all registers, flags, and clobbered memory. Moreover, we add pre-conditions asserting the equivalence of input symbolic values assigned to the input variables of the binary instruction and its corresponding variables in the lifted IR. 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 symbolic input value).

The verification condition queries are then dispatched to the $\mathbb{Z}3$ solver to prove equivalence between corresponding summaries. When the query of non-equivalence is satisfiable, the solver generates an example which can be used as a test input to trigger the mismatch. Any such mismatches are regarded as bugs in McSema and reported along with the associated test inputs.

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. Note that the reuse of validation results is facilitated by the Store database (Figure 1).

Single-instruction translation validation of control-flow instructions: The single-instruction translation validation for control-flow instructions, e.g., jump (conditional/unconditional) and call, is critical in ensuring that the control-flow of the binary program is preserved in the McSema-lifted program. A conditional jump instruction, e.g., **`jcc rel-offset`** at program counter `pc`, evaluates the condition code `cc` as an appropriate expression over the status flags, and updates the `%rip` with either the address of the target instruction (`pc + rel-offset`) or of the fall-through instruction (`pc + sizeof(jcc)`). Such an instruction is lifted to LLVM IR code with three goals: (1) computing the condition code value, `%cond`, matching the value of `cc`; (2)

[3] All the values or addresses stored in registers, memory or flags are implemented as bit-vectors and presented in this paper as V_W to be interpreted as a bit-vector of size W and value V.

[4] We manually inspect that the symbolic summaries corresponding to the loop trip count decrements by one in every iteration when the indicated condition is not met. Also, the count register is not modified in any other way.

updating the value of the simulated register corresponding to `%rip`; and (3) transferring control-flow to the appropriate basic block, using an LLVM branch instruction (e.g., **`br i1 %cond, label <LT>, label <LF>`**), based on value of `%cond`.

The goals of single-instruction translation validation for the running example of **`jcc`** are twofold: (A) to ensure that the update of `%rip` by the binary instruction and of the corresponding simulated register in lifted LLVM IR are *equivalent*, and (B) the LLVM `br` instruction should preserve the control-flow semantics of the corresponding binary instruction, i.e., the **`jcc`** and **`br`** instructions should evaluate equivalent conditions (i.e., cc ≃ value of `%cond`) and based on it's evaluation the control should jump to *corresponding* targets, i.e., if `%cond` is true, then the basic block with label `LT` should begin with an instruction corresponding to the target instruction at `pc + rel-offset`, else the basic block with label `LF` should begin with an instruction corresponding to the target instruction at `pc + sizeof(jcc)`.

To ensure (A), we symbolically execute **`jcc`** (resp., corresponding lifted IR) with concrete `pc` assigned to the `%rip` (resp., corresponding simulated resister) and symbolic values assigned to the status flags (resp., corresponding simulated flags) affected by the condition code `cc` (resp., `%cond`). We compare the resulting symbolic summaries, for the register `%rip` and its simulated counterpart, for equivalence using a solver preconditioned on the equivalence of respective symbolic inputs.

To achieve (B), we exploit the observation that the lifted IR encodes the target addresses of **`jcc`** instruction (or the potential values of `%rip`) in the branch labels of the LLVM **`br`** instruction[5]. An example of label `LT` is `%block-4004b4`, where `4004b4` is the target address when the condition code `cc` is satisfied. Moreover, we define an auxiliary state in LLVM semantics which captures the embedded address of the current block label. The symbolic execution of the lifted IR, as mentioned above, also provides the summary of this special state, which is compared for equivalence with the summary of `%rip` with similar preconditions, as mentioned above.

The single-instruction translation validation of other control flow instructions (like unconditional jump and call) are handled similarly.

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 is then compared with `T` using the Matcher (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, mainly because it reuses the individual instruction translations performed by McSema. These are separately validated using single-instruction translation validation, as described in the previous section.

`P` is disassembled to identify function boundaries, and to decode instructions. If the decoded instruction I is already in Store, then its corresponding (validated) IR sequence is reused (line 13). Otherwise, I is lifted (using McSema) (line 5) to generate an LLVM IR sequence that is going to be validated using Phase 1 (lines 6-11). The validated IR sequences are then composed (line 15) following the data- and control-flow order of the binary program `P`.

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  $T' \leftarrow \phi$
2  foreach function $F$ in $P$ do
3      foreach instruction $I$ in $F$ do
4          if $I$ not in $Store$ then
5              $S \leftarrow$ McSema ($I$)
6              Perform Translation Validation of $I$ and $S$
                (Phase 1)
7              if Validation successful then
8                  Add $< I, S >$ to $Store$
9              else
10                 Report Bug
11             end
12         else
13             Extract $S$ from $Store$ for $I$
14         end
15         $T' \leftarrow$ Compose($T', S, R$)
16     end
17 end
18 return $T'$
```

The "Compose" step. Below we describe the step "Compose" (line 15), responsible for composing the IR sequences together, using a few example binary instructions.

[5] Similarly, for **`call`** instruction, the name of the lifted function call encodes the target address of the callee.

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 when lifted in isolation.

```
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 addr
 %RAX = getelementptr ... ; Compute simulated RAX addr
 %RBX = getelementptr ... ; Compute simulated RBX addr
 %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 ; instr. len=3 bytes
 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 sequence is then validated using single-instruction translation validation unless it is already validated. Next, the validated IR sequence is appended to the composed program as shown below.

```
define %struct.Mem* @composedFunc(%struct.State*,
        i64, %struct.Mem*)  {
    %MEM = alloca %struct.Mem*
    store %struct.Mem* %2, %struct.Mem** %MEM

    ; Code: adcq %rax, %rbx
    %loadMem = load %struct.Mem*, %struct.Mem** %MEM
    %retval = call %struct.Mem* @sub_adcq_rax_rbx(
        %struct.State* %0, i64 0, %struct.Mem* %loadMem)
    store %struct.Mem* %retval, %struct.Mem** %MEM

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

A similar composition happens for *all* the non control-flow instructions. For a control-flow instruction, like jump (resp., call), in addition to appending the (validated) IR sequence for the instruction, we need to generate the LLVM `br` (resp., `call`) instruction for the control-flow to jump to block(s) of code corresponding to the jump (resp., call) target address(es). The composition for instructions accessing data-section are handled differently and elaborated next.

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 because McSema does not have the full-program context to determine if the immediate operand is an integer or address. Depending on which section of the isolated binary executable the address belongs, it can be interpreted as a integer or an address[7]. However, the problem is the program-level validation may not use that lifting because the interpretations of the immediate operand, when lifted in isolation versus when lifted with full-program context, might be different. As a result, the composed IR, which consumes the translations of instructions in isolation, will be different from the one lifted by McSema. Upon optimization using LLVM passes, two such IRs will be optimized differently and eventually fail to match even when the translation of McSema is correct.

To aid in testing, we compile binaries with options to retain auxiliary information. To disambiguate between cases where an immediate operand 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. Every immediate operand that is a reference has a corresponding entry in the relocation table. We allow McSema *to incorrectly lift such instructions in isolation when invoked by algorithm 1*, and then we course-correct the lifted IR by consulting the relocation information, `R`.

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 correct the lifted output:

```
%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 (e.g., if the lifter is not able to correctly disambiguate an address from a integer) at testing

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

[7] During single-instruction translation validation, we validated the behavior of such instructions for both the possibilities of the constant operand by forcing the constant to belong to `.data` or `.text` section of the isolated binary.

that would otherwise be missed. During the use of Compositional Lifter in the field to validate the lifting of McSema on an unknown, blackbox binary, we can function without the 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 Transformer & 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 build on this observation to develop an inexpensive semantic equivalence checker that does not require heavyweight theorem provers, instead using graph isomorphism, assisted by semantics-preserving transformations (lines 2-3). The algorithm is realized by a tool we develop called the Matcher (line 4).

At first, the function pair (`F` & `F′`) is transformed to ($\mathtt{F}_N$, $\mathtt{F}'_N$), using LLVM optimization passes[8], to prune any syntactic differences except for the names of virtual registers and the order of non-dependent instructions. There is clearly some important relationship between the syntactic code differences in `T` and `T′` and the choice of optimization passes with the aim of exploiting those differences. As a few examples of syntactic differences: (1) Program counter updates like `%rip − C` (`C` being a positive constant) are lifted in `T′` using addition (`%rip + (−C)`) versus subtraction used in `T`, and (2) As an optimization, `T` hoists the address computations of simulated registers to the entry block which are then dereferenced at every use-site. On the other hand, in `T′`, such addresses are both recomputed and dereferenced at every use-site. Above syntactic differences are eliminated using (1) `-instcombine` (a peephole optimization pass on LLVM IR), and (2) `-early-cse` or `-licm` respectively.

Next, the Matcher algorithm works on data dependence graphs, $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$, generated from $\mathtt{F}_N$ & $\mathtt{F}'_N$. A vertex of the graph represents an LLVM instruction, and an edge between two vertices captures SSA def-use edges or memory dependence edges between LLVM load and store instructions, extracted from LLVM MemorySSA [1] analysis. If the Matcher fails to match `T` & `T′`, there *may* be a bug in the lifter.

Algorithm 2: Matcher Strategy

```
Inputs :T: McSema-lifted IR.
        T′: Compositional Lifter lifted IR.
Output :True ⟹ T & T′ semantically equivalent
        False ⟹ T & T′ may-be non-equivalent
1 foreach corresponding function pair (F,F′) in (T, T′) do
2     $F_N$ = Transformer (F)
3     $F'_N$ = Transformer (F′)
4     if !Matcher($F_N$, $F'_N$) then
          // A potential bug in McSema while lifting F
5         return false
6     end
7 end
8 return true
```

Checking Graph Isomorphism. Our algorithm to check the isomorphism of $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$ is built on a subgraph-isomorphism algorithm from Saltz et al. [67]. The algorithm, in general, first retrieves an initial potential-match set, 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 a vertex $\mathtt{I}_N$ in $\mathtt{G}_{\mathtt{F}_N}$ contains all the vertices in $\mathtt{G}_{\mathtt{F}'_N}$ which satisfy the following three criteria: (1) they have the same instruction opcode, (2) they have identical constant operands, if any, and (3) they have the same number of outgoing data dependence edges[9] as $\mathtt{I}_N$. 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. Our overall algorithm, *Matcher*, checks that the graphs $\mathtt{G}_{\mathtt{F}_N}$ and $\mathtt{G}_{\mathtt{F}'_N}$ are isomorphic and the instructions corresponding to the matching vertices are identical w.r.t. the instruction opcode and constant operands (note that all other operands are SSA variables, and so are validated by graph isomorphism).

Soundness of Equivalence via Graph Isomorphism. Our argument that isomorphism of $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$ implies semantic equivalence of the functions `F` and `F′` is based on the pioneering work by Horwitz et al., which proved that *if the program dependence graphs of two programs are isomorphic then the programs are "strongly" semantically equivalent* [43]. Our dependence graph representations $\mathtt{G}_{\mathtt{F}_N}$ and $\mathtt{G}_{\mathtt{F}'_N}$, which we check for isomorphism, only include the data dependences, and not the control dependences. Note, however, that the close structural similarity between the functions, `F` & `F′`, ensures the required equivalence of all control flow, as explained below. There is one minor exception to control-flow equivalence,

[8] The pass sequence (-mem2reg -licm -gvn -early-cse -globalopt -simplifycfg -basicaa -aa -memdep -dse -deadargelim -libcalls-shrinkwrap -tailcallelim -simplifycfg -basicaa -aa -instcombine) is determined by manually pruning the LLVM `-O3` sequence.

[9] Checking outgoing but not the incoming edges is a design choice. The later check will constrain the the average size of the sets even more which in turn improve the runtime of algorithm, but will not affect the soundness of the matcher in any way.

which introduces no semantic differences between the programs, and is addressed below. We first assume this exception does not occur, and informally prove semantic equivalence in three simple steps, as follows.

Let `PDG(f)` denote the program dependence graph of a function `f`.

(A) **The control flow graphs (CFGs) of `F` and `F′` are isomorphic:** `F` and `F′` are both obtained by lifting the same binary, via instruction-by-instruction lifting (using identical IR sequences for each one). For `F`, we check that the order of lifted IR sequences is the same as the order of binary instructions within each corresponding basic block. For `F′`, such an order is already preserved assuming the correctness of `D′`. The control-flow edges are verified to be identical by the single-instruction translation validation of control-flow instructions (Section 4). Together, these facts ensure isomorphism of the CFGs of `F` & `F′`. We note that the requirement to preserve the order is stricter than necessary because data-independent instructions can be reordered safely.

(B) **The control dependence graphs of `F` and `F′` are isomorphic**: This is straightforward to derive using (A) and the definitions of control flow and control dependence [12], and we omit the explanation.

(C) **If the data dependence graphs $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$ are isomorphic, then $\mathtt{PDG(F}_N\mathtt{)}$ and $\mathtt{PDG(F}'_N\mathtt{)}$ are isomorphic**: By definition, the nodes of $\mathtt{G}_{\mathtt{F}_N}$ are identical to the nodes of $\mathtt{PDG(F}_N\mathtt{)}$, and similarly for $\mathtt{G}_{\mathtt{F}'_N}$ and $\mathtt{PDG(F}'_N\mathtt{)}$. The edges of a PDG are simply the union of the control dependence edges and the data dependence edges. Combining (B) with the isomorphism of $\mathtt{G}_{\mathtt{F}_N}$ & $\mathtt{G}_{\mathtt{F}'_N}$, it follows directly that $\mathtt{PDG(F}_N\mathtt{)}$ and $\mathtt{PDG(F}'_N\mathtt{)}$ are isomorphic.

The one exception mentioned above is that, as a custom optimization, some address computations for simulated registers are hoisted to the entry block by McSema (i.e., in `F`), to be reused by later instructions throughout the function, whereas this hoisting does not happen in `F′`. The addresses are computed using LLVM's `getelementptr` instructions whose operands are immutable throughout the function in both `F` and `F′` (the *State* pointer (Section 4) and some constant arguments). As a result, the results of these computations are unaffected by their location in the code. One requirement is that these address computations must dominate their uses since their results are assigned to SSA values: this property is enforced by McSema by running the LLVM `verify` pass (which we also consider trusted). Together with the isomorphism of the data dependence graphs, this guarantees that the potential difference in locations of these instructions does not introduce any differences in any uses of those values, and thus no differences in the semantics of the two functions.

Note that the entire above argument (indeed, the theorem of Horwitz, et al. [43]) 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 alarms.

Autotuning-based Transformer. As per our matching strategy, in order to prove that two functions `F` & `F′` are semantically equivalent, they need to be reduced to isomorphic graphs via semantic preserving transformations. For transformations, we initially used a custom sequence of 17 LLVM optimization passes, discovered manually by pruning the LLVM `-O3` search space. Later experimentation revealed that (1) changing the order of passes improves the number of functions that are successfully proved isomorphic (the phase-ordering problem of optimization), and (2) not all of the 17 passes are needed for every pair of functions under equivalence check. These two observations motivate us to frame the problem of selecting optimal pass sequences, one for every pair of candidate functions, as an application of program autotuning.

We used the OpenTuner [15] framework to implement the autotuner. OpenTuner requires the client to specify a search space to explore, and an objective function to maximize. Our search space is all permutations of passes from the 17-length pass sequence. The objective function in our case is to maximize the fraction of nodes in $\mathtt{G}_{\mathtt{F}_N}$ (or $\mathtt{G}_{\mathtt{F}'_N}$) having non-empty initial potential-match sets. The framework then uses various heuristic search techniques to find the best configuration that maximizes the objective function, within a given resource budget (a fixed number of iterations). Such an autotuning-based Transformer addresses the phase-ordering problem, improving the Matcher results (refer to Section 6) by lowering the false-alarm rate, and also by using much fewer than 17 passes on average.

Comparison with LLVM-MD & Peggy. At this point, it is important to differentiate our approach to establish equivalence between two LLVM IR programs from existing, similar approaches for validating LLVM IR-to-IR optimization passes, e.g., LLVM-MD [81] and Peggy [79]. Like our approach, these tools eschew 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, the implementation of our Transformer is simpler, requires no additional implementation effort, re-uses existing, well-tested compiler passes, and still proves to be quite effective in reducing two semantically equivalent programs to isomorphic graphs, as demonstrated by our evaluations.

6 Evaluation

In this section, we present the experimental evaluation of single-instruction translation 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 lifting?

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. What is the runtime of our Compositional Lifter and Matcher-based approach?

Q4. Is program-level validation effective at finding additional real bugs in a complex lifter like McSema, beyond those found by single-instruction translation validation alone?

Usefulness of single-instruction translation 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 [32]. McSema supports 1922 instructions, all supported by [32]. We had to exclude 573 instruction variants because of limitations of the LLVM IR semantics [45], which does not support vector and floating-point types and associated operations, and various intrinsic functions[10]. 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), producing a counterexample for each failure, and 6 timed out. Except for timeouts, the solver found conclusive results in all the cases within a solver timeout of 30 secs. The solver time ranges from 0.25 − 29.89 secs with median 0.46 secs. The max time, recorded for `cmpxchgq %rcx, %rbx`, is because of the complex summary of the corresponding lifted IR.

Timeouts, declared based on a threshold of 24 hrs, correspond to `paddb`, `psubb`, and `mulq` family of instructions. On further investigation, we found that 4 out of 6 timeouts related to `paddb` and `psubb` are flaky: the $\mathbb{Z}3$ solver result toggled between *unknown* and *unsat* depending on the order in which other unrelated constraints are added (which is a known issue [5]). By removing the unrelated constraints[11], $\mathbb{Z}3$ concludes them to be equivalent. The remaining two cases (related to `mulq`) include solver constraints containing bit-vector multiplication, which the state-of-the-art SMT solvers are not very efficient at reasoning about. However, we manually inspected them to ensure that the generated code fragments are indeed semantically equivalent.

The 29 failures along with the test cases created from $\mathbb{Z}3$'s counterexamples were all reported and subsequently confirmed as bugs [10] by the McSema developers. The following are some brief examples of a few 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 [11] says the implementation should be `%xmm1` ← ∼`%xmm1` & `%xmm2`, whereas McSema interchanges the source operands.

Third, for `pmuludq %xmm2, %xmm1`, 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 (value of `%ah`), 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 hence could not be validated using single instruction validation. This brings us to a total of 2348 usable functions, which we compile using Clang[12] and feed the binaries to Compositional Lifter and McSema for lifting. The source LOC of the usable functions ranges from 1 − 1454, with median 18, whereas the LOC of the corresponding lifted IR (`*.ll` assembly files) after inlining, obtained by lifting the binaries compiled from the source functions, ranges from 86 − 32105 (median 656), with maximum LOC recorded for function "himenobmtxpa::jacobi".

The lifted function pairs are then optimized using the pass sequence (of length 17) and fed to the Matcher (Algorithm 2).

[10] However, we support an intrinsic called llvm.ctpop by implementing it in LLVM IR. This intrinsic is used pervasively in the lifted IR for updating the `%pf` flag.

[11] The unrelated constraints refer to the verification queries related to registers/flags other than the one under verification.

[12] We also used GCC-compiled binaries for the experiments, but most of them are not lifted by McSema due to unsupported instruction semantics.

Of the 2348 usable functions, the Matcher can prove the correctness of the translations for 2189 functions using graph isomorphism, i.e., a success rate of 93% (the inlined lifted IR ranges in size from $86 - 32105$, with the median as 611). We manually checked the remaining 159 and found them to be false alarms, with the following root causes:

- **Pass Selection & Phase-ordering problem (** 80% **of false alarms)**: The fixed-length pass sequence is not able to converge functions into isomorphic graphs because either it missing some key passes or the order of application of individual passes is not effective. We addressed the above problems using autotuning of the pass-sequence, as described below.
- **Difference in Lifting globals (** 20% **of false alarms)**: For data section addresses, McSema lifts a global with over-approximated size (as determined by IDA) which need not be equal to the actual source code size, whereas our Compositional Lifter determines the size as the width of the maximum access across all the instructions accessing that particular global. As a result, the lifted global sizes might be different from McSema. The memory dependence edges that we extract using LLVM IR `memory-ssa` analysis depend on the size of the globals, and hence the generated graphs will be different. A more accurate memory analysis might solve these issues.

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 lifter. We can further reduce this rate by addressing the phase-ordering problem using an autotuner, as described in Section 5.2. We leveraged the experience and effort put into custom-designing the fixed-length pass sequence by including the constituent passes in the search space for autotuning. We avoided crafting the search space using all the LLVM passes (e.g., 187 passes of Clang's $-O3$ pass sequence) because our experiments showed that such a large search space was *less effective* at avoiding false negatives in a fixed number of iterations.

For 2254 out of the total 2348 functions, the autotuner is able to find custom pass sequences that lead to successful matching. These matches include 65 previously reported false alarms (out of total 159), reducing total false alarms to 94 (or 4% of 2348). All previously positive cases remain positive with the autotuner, as well. The autotuner runtime ranges from 10.7 secs - 19.97 mins, with a median of 6.67 mins. The length of the generated pass sequence has distribution of [min:- 3, median:- 7, max:- 243, mean:- 8][13].

Judiciously adding LLVM passes to the search might help remove false alarms further. We leave this as future work.

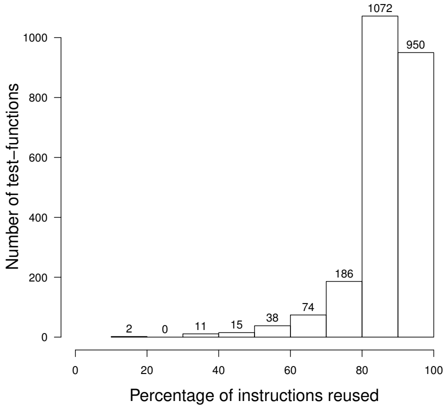

Figure 2. Distribution of reuse % by the Compositional Lifter in the Store database for an arbitrary execution sequence on 2348 test-functions.

Performance of Program-level validation: The performance of this phase is dominated by the time to run the Compositional Lifter and Matcher.

We chose to validate the individual instructions offline after program-level validation. The number of those instructions, to be validated offline, amounts to approx. 50% of the total 1349 variants we validated using single-instruction translation validation; implying that only a small subset of instructions are actually used in binary executables in practice.

The running time of the Compositional Lifter, on 2348 usable functions, ranges from $0.06s - 5.75s$, with a median of $0.63s$. Note that this performance depends heavily on the availability of instructions in the Store database for reuse, which in turn depends on the order in which the test functions are executed. For example, a large function with many commonly occurring instructions, if lifted first using the Compositional Lifter, will populate the Store sufficiently to create good reuse in later functions. Even with an arbitrary order of test execution, the Store reuse found is significant. Figure 2 shows the distribution of reuse percentage (both across & within functions) for an arbitrary order of lifter execution on 2348 test-functions.

The running time of Matcher, primarily the graph isomorphism algorithm, on 2348 usable functions ranges from $0.06s - 119.63s$, with a median of $4.91s$. We note that, for both Compositional Lifter & Matcher, the max time occurred for the largest function "himenobmtxpa::jacobi" (32105 LLVM assembly LOC).

Program-level validation: Effectiveness at finding bugs: In our experiments, all real bugs in McSema were caught by single-instruction translation validation and not program-level validation, which may be evidence that most of the complexity in lifting, by far, lies in lifting individual binary instructions to IR. Intuitively, this makes sense because of the large and diverse instruction set semantics, the simplicity of the compositional step, and the fact that every aspect of the composition logic is likely used hundreds of times per

[13] 43 out of 65 newly matched cases have an auto-tuned pass sequence of length greater than 17. For those cases, the search space, with 17 passes being ineffective, is composed differently out of multiple auto-tuned sequences derived from the other matching cases.

program. Nevertheless, bugs are possible in this aspect of the lifter.

We studied the effectiveness of program-level validation in finding bugs in McSema by artificially injecting bugs in the lifter's implementation. The injected bugs cover the following aspects of McSema's lifting: (1) *Instruction lifting*: McSema 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 constants*: McSema uses information from IDA [40] to know if an immediate operand used in a data-section access instruction is a constant or a memory address. The introduced bug forces McSema to take the wrong decision. (3) *Maintaining correct dependences among instructions*: The injected bug changes the order in which instructions are lifted, potentially violating data and control dependences between instructions.

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 mentioned before. All the injected bugs are correctly detected by the Matcher, establishing program-level validation as a complementary technique to single-instruction translation validation in finding bugs during lifting.

Note that only the first of these bugs would be caught by single-instruction translation validation: the binary instruction semantics would not match with the LLVM IR sequence semantics in that case. The second and third cases would (in general) produce equivalent semantics between each X86 instruction and the LLVM IR sequence, and so single-instruction translation validation would not detect the bug.

7 Discussion

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

Incomplete LLVM Semantics. The LLVM IR semantics [45] is currently under development and does not support all LLVM abstractions, e.g., vector and floating-point types and their associated operations, and various intrinsic 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 semantic rules are added, assuming $\mathbb{Z}3$ supports the requisite features.

Formally Verifying Transformation Passes. Our current implementation uses a small number of LLVM passes (17) to improve syntactic matching between the IR generated by McSema and by Compositional Lifter. For now, we trust the correctness of these passes to perform only semantics-preserving transformations. Formally proving correctness of arbitrary LLVM pass sequences is difficult. An alternative approach is to develop simple graph rewrites on SSA graphs that can be composed to mimic the transformations of LLVM passes and formally prove that these graph rewrites preserve program semantics. We leave this to future work.

Extending to Other Lifters. Our current work focuses on McSema, an open-source lifter from x86-64 to LLVM IR. Extending our work to support other lifters would not only improve trust in those lifters, but also help our system evolve to a generic framework to validate future lifters for (nearly) free. For lifters that are designed to translate binary instructions individually and compose the resulting LLVM IR, we believe that this could be done simply by customizing Compositional Lifter to capture the idiosyncrasies of each lifter.

8 Related Work

Traditionally, translation validation [64] uses compiler instrumentation to help generate a simulation relation to prove correctness of compiler optimizations [44, 65] or check the validity of compilation [71]. 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 lifters in their early development phase are updated and improved at a frantic pace. Without instrumentation, such simulation relations can be inferred, using symbolic execution, by collecting constraints from the input and output programs (as demonstrated in Necula's work [62]). First, in the context of translation validation of binary lifting, such inference is not straight-forward mainly because the two programs (x86-64 binary and lifted IR) are structurally very different with potentially different number of basic blocks[14]. (For example, Necula's approach cannot handle such cases because it requires branch equivalence.) Second, checking program equivalence, in general, is an undecidable problem, and using symbolic execution is very expensive. Hence, any solution which can avoid such overhead is of great importance in serving a practical validation approach. Consequently, we decided to move away from simulation-based validation approaches.

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

8.1 Testing-based Approaches

Martignoni et al. [57, 58] propose hardware-cosimulation based testing on QEMU [19] and Bochs [52]. 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 scalable and straightforward approach, it's effectiveness is limited because many semantics

[14] Instructions like `adcq` generate additional basic blocks upon lifting, which are not explicit in the binary program.

bugs in binary lifters are triggered upon a specific input and exercising all such corner inputs, using randomly generated test-cases, is impractical.

Chen *et al.* [26] proposed validating the static binary translator LLBT [73] and the hybrid binary translator [72], both of which translate ARM programs to x86 programs via an intermediate translation to LLVM IR. They validate this translation by running both programs on an input and comparing the architectural states after each instruction. The validator is evaluated using the ARM code compiled from EEMBC 1.1 benchmark. Like the 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.* [56] validate a "buggier and less complete" Lo-Fi emulator [19] by generating high-fidelity test-inputs creating using symbolic execution of a "faithful and more complete" (in terms of IA-32 ISA) Hi-Fi emulator[52] implementation of an instruction semantics. They execute each test instruction twice, once on a real hardware and next on the Lo-Fi emulator, and comparing the output states. However, the work [56] does not aim to validate the translation of full programs, which is one of our primary contributions. Note that an approach as above cannot scale naturally to binary function validation because 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

The closest work to ours in finding instruction-level bugs is MeanDiff [46], which proposed an N-version IR testing to validate three binary lifters, BAP [23], BINSEC [18], and PyVEX [3] by comparing their translation of a single binary instruction to BIL, DBA, and VEX IRs respectively. The IRs are first converted to some unified IR representations, one at a time. The resulting IRs are then symbolically executed to generate symbolic summaries for comparison using an SMT solver. First, we formally validate the translation of an instruction, using a thoroughly-tested semantics [32], as opposed to comparing the translation to other potentially incorrect translations. Second, the IRs they support are simpler than LLVM and so it is unclear whether the approach would be effective if LLVM had to be translated to the unified representation. Third, we perform program-level validation, which is not addressed by MeanDiff.

Interestingly, the MeanDiff paper [46] says that one motivation for relying on differential testing was that no formal specification of x86-64 ISA was available at the time. We do not have that limitation because we have developed a formal and thoroughly tested x86-64 ISA specification [32], and made it publicly available.

The work closest to ours, in terms of the goals, is the translation verifier, Reopt-vcg [42], which addresses verification challenges specific to the translator Reopt [4]. The verifier, which validates translations at basic-block level, is assisted by various manually written annotations, which are prone to errors. In future, they aim to generate such annotations automatically by instrumenting the lifter. Our approach does not need any such annotations, avoiding the overhead of maintaining instrumentation patches whenever the lifter is modified. Moreover, the validator uses the semantics of a small subset of x86-64, which limits its applicability to small programs. We incorporate a fairly complete x86-64 semantics [32], allowing translation validation for larger and more diverse programs.

Myreen et al. [60, 61] presented "decompilation into logic" — a framework for verified decompilation, where machine code is decompiled into tail-recursive functions defined in the language of the HOL4 theorem prover [75]. The decompiler proves a theorem stating that the function accurately describes the effect of the given machine code. Sewell at al. [71] proved correctness of compilation of the seL4 microkernel from C source down to ARM machine code by building on a formal model of ARM code generated by extending the above work of Myreen. Such a verified (de)compiler includes critical design decisions which need to be incorporated early in the design phase with the goal of verification in mind, and cannot easily be applied retroactively in existing (de)compilers.

9 Conclusion

In conclusion, we demonstrated that validation of lifters without instrumentation or heavyweight equivalence checking is feasible. The design is based on a simple insight: Formal translation validation of single machine instructions can be used as a building block for scalable full-program validation, achieving scalability by using symbolic execution and theorem provers only for the single-instruction case. Our experimental evaluation shows that single instruction validation is valuable in finding real bugs in McSema, a popular open-source lifter from x86-64 to LLVM IR. We construct an alternate lifter by composing validated single-instruction translation sequences, with a small amount of custom logic to handle control flow sequences and global data. To check the McSema translation, we compare the outputs of the two lifters, using semantics-preserving program transformations together with graph isomorphism of data dependence graphs. We believe our approach can be easily modified to support other lifters from x86-64 to LLVM that are designed to translate individual instructions, simply by modifying how the alternate compositional lifter is constructed.

Acknowledgments

We thank the $\mathbb{K}$ team, especially Daejun Park, for his extensive technical advice. We are grateful to Alastair Reid and other reviewers for their diligence in providing invaluable feedback. We thank our shepherd, James Bornholt, for his helpful guidance in responding to the reviewers' comments. This work was supported by the Office of Naval Research under contract number N00014-17-1-2996.

References

[1] [n.d.]. MemorySSA. https://llvm.org/docs/MemorySSA.html. Last accessed: April 12, 2020.

[2] 1996. ARM Architecture Reference Manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.architecture.reference/index.html. Last accessed: April 12, 2020.

[3] 2013. Python bindings for Valgrind's VEX IR. https://github.com/angr/pyvex. Last accessed: April 12, 2020.

[4] 2014. reopt: A tool for analyzing x86-64 binaries. https://github.com/GaloisInc/reopt. Last accessed: April 12, 2020.

[5] 2017. Z3's behavior seems to depend on the order in which formulas are asserted. https://github.com/Z3Prover/z3/issues/1106. Last accessed: April 12, 2020.

[6] 2018. Angr: A powerful and user-friendly binary analysis platform! http://angr.io/. Last accessed: April 12, 2020.

[7] 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: April 12, 2020.

[8] 2018. fcd: An optimizing decompiler. https://zneak.github.io/fcd/. Last accessed: April 12, 2020.

[9] 2018. Remill: Library for lifting of x86, amd64, and aarch64 machine code to LLVM bitcode. https://github.com/trailofbits/remill. Last accessed: April 12, 2020.

[10] 2019. A few discrepancies in x86-64 Instruction Semantics. https://github.com/lifting-bits/remill/issues/376. Last accessed: April 12, 2020.

[11] 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.

[12] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2^{nd} Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Sergi Alvarez. 2018. Radare2. https://rada.re/r/. Last accessed: April 12, 2020.

[14] 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

[15] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. Edmonton, Canada. http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

[16] 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* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 1083–1094. https://doi.org/10.1145/2568225.2568293

[17] 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

[18] 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* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 165–170. http://dl.acm.org/citation.cfm?id=2032305.2032318

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

[20] Ahmed Bougacha. 2017. Binary Translator to LLVM IR. https://github.com/repzret/dagger. Last accessed: April 12, 2020.

[21] 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* (San Francisco, California, USA) *(CGO '03)*. IEEE Computer Society, Washington, DC, USA, 265–275.

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

[23] 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* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 463–469. http://dl.acm.org/citation.cfm?id=2032305.2032342

[24] 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

[25] 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

[26] 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

[27] V. Chipounov and G. Candea. 2011. Enabling sophisticated analyses of x86 binaries with RevGen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 211–216. https://doi.org/10.1109/DSNW.2011.5958815

[28] 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* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[29] 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

[30] 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* (Alexandria, Virginia, USA) *(CCS '08)*. ACM, New York, NY, USA, 391–402. https://doi.org/10.1145/1455770.1455820

[31] Sandeep Dasgupta. 2020. A Scalable Validator for Binary Lifters. https://github.com/sdasgup3/validating-binary-decompilation. Last accessed: April 12, 2020.

[32] 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

[33] 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* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[34] 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* (Austin, TX, USA) *(CC 2017)*. ACM, New York, NY, USA, 131–141. https://doi.org/10.1145/3033019.3033028

[35] Draper-Laboratory. [n.d.]. An architecture-independent decompiler to LLVM IR. https://github.com/draperlaboratory/fracture. Last accessed: April 12, 2020.

[36] 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.

[37] 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* (Seattle, Washington) *(OSDI '06)*. USENIX Association, Berkeley, CA, USA, 75–88. http://dl.acm.org/citation.cfm?id=1298455.1298463

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

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

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

[41] 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

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

[43] S. Horwitz, J. Prins, and T. Reps. 1988. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL âĂŹ88)*. Association for Computing Machinery, New York, NY, USA, 146âĂŞ157. https://doi.org/10.1145/73560.73573

[44] 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

[45] Theodoros Kasampalis. 2020. Translation Validation for Compilation Verification. PhD thesis, University of Illinois at Urbana Champaign (to be published).

[46] 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* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 353–364. http://dl.acm.org/citation.cfm?id=3155562.3155609

[47] Kevin Kirchner and Stefan Rosenthaler. 2017. Bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation. In *Proceedings of the 12th International Conference on Availability, Reliability and Security* (Reggio Calabria, Italy) *(ARES âĂŹ17)*. Association for Computing Machinery, New York, NY, USA, Article 45, 7 pages. https://doi.org/10.1145/3098954.3103152

[48] 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

[49] J. Křoustek and P. Matula. 2018. RetDec: An Open-Source Machine-Code Decompiler. [talk]. Presented at Pass the SALT 2018, Lille, FR.

[50] 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.

[51] 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

[52] 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

[53] 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* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 83–93. https://doi.org/10.1145/1453101.1453114

[54] 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* (Chicago, IL, USA) *(PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[55] 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

[56] 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* (London, England, UK) *(ASPLOS XVII)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2150976.2151012

[57] 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* (Trento, Italy) *(ISSTA '10)*. ACM, New York, NY, USA, 171–182. https://doi.org/10.1145/1831708.1831730

[58] 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* (Chicago, IL, USA) *(ISSTA '09)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/1572272.1572303

[59] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbr&#252;cken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[60] Magnus O. Myreen, Michael J. C. Gordon, and Konrad Slind. 2008. Machine-code Verification for Multiple Architectures: An Application of Decompilation into Logic. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Portland, Oregon) *(FMCAD '08)*. IEEE Press, Piscataway, NJ, USA, Article 20, 8 pages. http://dl.acm.org/citation.cfm?id=1517424.1517444

[61] M. O. Myreen, M. J. C. Gordon, and K. Slind. 2012. Decompilation into logic — Improved. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*. 78–81.

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

[63] 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).

[64] 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

[65] 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* (Venice, Italy) *(POPL '04)*. ACM, New York, NY, USA, 1–13. https://doi.org/10.1145/964001.964002

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

[67] 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

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

[69] 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* (Washington, D.C.) *(SEC'13)*. USENIX Association, Berkeley, CA, USA, 353–368. http://dl.acm.org/citation.cfm?id=2534766.2534797

[70] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. [n.d.]. *The K Primer (version 3.2)*. Technical Report.

[71] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

[72] 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

[73] 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* (Tampere, Finland) *(CASES '12)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2380403.2380419

[74] 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).

[75] Konrad Slind and Michael Norrish. 2008. A Brief Overview of HOL4. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–32.

[76] Aaron Smith and S. Bharadwaj Yadavalli. 2018. LLVM Based Binary Raiser: llvm-mctoll. (2018). https://github.com/Microsoft/llvm-mctoll

[77] 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* (Hyderabad, India) *(ICISS '08)*. Springer-Verlag, Berlin, Heidelberg, 1–25. https://doi.org/10.1007/978-3-540-89862-7_1

[78] 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* (Orlando, Florida, USA) *(PLDI '94)*. ACM, New York, NY, USA, 196–205. https://doi.org/10.1145/178243.178260

[79] 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* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742. http://dl.acm.org/citation.cfm?id=2032305.2032364

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

[81] 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* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 295–305. https://doi.org/10.1145/1993498.1993533

[82] 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*.

[83] 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

[84] 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

[85] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security* (Washington, D.C.) *(SEC'13)*. USENIX Association, USA, 337–352.